

---

# METHODS & TOOLS

---

Practical knowledge for the software developer, tester and project manager

ISSN 1661-402X

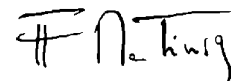
Winter 2007 (Volume 15 - number 4)

[www.methodsandtools.com](http://www.methodsandtools.com)

## The Toilet Paper Theory

The toilet is one of the most likely things that you share with your colleagues. Visiting the lavatories is also something that you will most likely to do every day. The toilet paper is an essential resource for this activity. Benjamin Franklin is quoted to have said "In this world nothing is certain but death and taxes." I would personally add a third thing: toilet paper rolls reaching the end of their existence. What follows could give you an interesting view on the organisation. In some companies, the toilet paper dispenser is locked. Only authorised persons can change rolls. Sometimes this is because the organisation does not trust its employees. Alternatively, it will be just another the sign of an outsourcing mentality that reached also some vital functions. You could also be in a "full service" company, where you are not expected to deal with these things. A maintenance employee will (or "should") regularly visit the toilet to check that everything is in place. The locked dispenser is often the sign of a tightly regulated environment and you should be already happy that you do not have to ask for approval before going to the lavatories. There is also the case where it is the "user" that should change the paper when the rolls end. The question is whether it will be the last or the next user that changes the roll. This reveals interesting insight about how developers care about the impact of their actions on their colleagues' life. Are they thinking about just what is necessary for them or do they get behind their own needs to make the small act that will make life easier for their colleagues?

There are many ways to use this theory. If you apply to a new job, the lavatories will give you some inside information about the culture of your prospective employer. If you are a project manager or a consultant, you could have a quick test on personal behaviour. Is this possible to create agile self-organising teams or to achieve high quality software when you find empty paper rolls in the lavatories? I doubt so. Small details can reveal a lot about the personality of a developer or an organisation. Nevertheless, a good attitude is the basis to the formation of good software development teams. This starts in some common places, as techniques are easier to teach and modify than behaviours. As we will soon flush the water on 2007, I would like to wish you all the best for 2008.



## Inside

Requirements for Outsourcing .....	page 3
Exploratory Testing: Finding the Music of Software Investigation.....	page 17
Four ways to a Practical Code Review .....	page 24

# Software Development Jobs

Methods & Tools introduces a new job board

**<http://www.softdevjobs.com/>**

Are you looking for an efficient, simple and cost effective tool to recruit experienced developers, Web designers, testers, database administrators or software project managers?

Your job post will reach an audience of 40'000 monthly Web site visitors and be mentioned in the future Methods & Tools issues that are delivered to a global community of more than 50'000 software development professionals. Facts: Asia, North America and Europe host each around 30% of our registered readers. They work as developers (37%), project managers (27%) and in quality assurance (20%).

**Post your job opening for free until  
January 15th 2008**

Methods & Tools readers will get a 50% discount for on normal price until March 21st 2008 with the discount code "MT01".

Job seeker? Get automatic alerts of new jobs in your country or speciality via e-mail or RSS

## **Requirements for Outsourcing**

Tom Gilb, <http://www.gilb.com/>

### **Abstract**

Outsourcing differs from other development because there is bound to be a contractual relationship, probably a geographic distance, a different sense of loyalty, language misunderstandings, cultural differences, reluctance to speak up to the client – and many other associated problems. Good requirements are always a problem, but outsourcing increases the problems, and makes even great demands on the requirements specification. The payoff for doing good requirements is greater, and the penalty for *not* doing them is more threatening.

I am going to argue that we need to make use of far more explicit background specification for each requirement, a page or more of specification for each requirement. I will argue that this is a necessary investment – because failure to do so will probably cost far more – sooner or later. I will argue that failure to be more detailed than normal will be counted in the clients disfavor in any legal proceedings trying to determine responsibility for failure of the project.

### **Outsourcing Requirements Principles**

Here is a set of principles for Requirements for Outsourcing:

1. If anything can be misunderstood, it probably will be.
2. Writers Are Responsible for Readers Wrong Renditions
3. Assume Nothing, Specify Everything
4. Too Much is Safer than Too Little
5. If They ask a question, document and integrate the answer
6. Quality Control before sending
7. Evolve Requirement Delivery
8. Quantify Quality
9. Constrain explicitly
10. Connect relationships

Let me explain them in more detail:

#### **If anything can be misunderstood, it probably will be.**

Every person has a strong tendency to interpret words slightly to largely differently from everybody else. When we ask 20-30 people to write down their interpretation of a short requirement statement, we always experience totally different, never identical answers from individuals working on the same project. We call this the ‘Ambiguity Test’ – and it really gets the point across to the whole group about how careful we must be when writing specifications that must be understood correctly.

### **Explicit Definition**

One simple tactic we recommend in ‘Planguage’ or The Planning Language [CE] is to take the trouble to explicitly define any term that could possibly cause misunderstanding, and then ‘Capitalize’ the term to signal that the reader must interpret it with the official definition.

For example:

Intuitiveness: Scale: Probability that a defined [User] can intuitively figure out how to do a defined [Task] correctly without any Errors Needing correction.

User: defined as: {Novice, Experienced, Expert}.

Task: defined as { All Tasks: {Data Entry, Screen Interpretation, Answering Call, <others undefined now> }.

Errors: defined as keyboard or mouse or touch screen inputs that are unacceptable, unintended, or incorrect.

Needing: defined as: objectively incorrect data, or data violating Corporate Data Standards.

Example: Capitalized Terms indicate formal definitions somewhere, locally or elsewhere in the project documentation, or 'later' in the definition process. Source CE, page 160 (line 1-3).

### Fuzzy Brackets

Another simple Planguage tactic, is to mark all 'dubious' terms and phrases with <fuzzy brackets>.

This means that we are not certain about whether the term could be misinterpreted. The early spec reader should be very careful in interpreting the term. The spec writer has declared the term to be potentially defective until rewritten or defined. People happily use this fuzzy brackets facility several times per sentence. It allows them to rapidly continue their flow of work, without getting into premature detailed discussions. But they have obliged themselves to get back later and fix it up, and failing that, they have at least warned the spec reader to be careful.

Performance Requirements: Ambition: <competitive> <breakthrough level of quality>. Reduce Product Cost. Improve Productivity [Engineering]. Improve timeliness of <engineering drawings>. Improve <drawing quality>. Reduce <drawing errors>. Others. Reduce <Engineering Process> timescales ('time to market'). Improve <Efficiency> [Manufacturing, Procurement]. Achieve <Growth>. Others
--

Example: The use of fuzzy brackets to mark all terms needing definition work, later. The text is taken from a Board level proposal for \$60 million, that we restructured and began to define. The Board was not willing to grant budget for this set of badly defined promises. Source: CE, page 73.

Advertisement – Agile Software Development Training - Click on ad to reach advertiser web site



# AGILE SOFTWARE DEVELOPMENT TRAINING

*Accelerate Your Career  
& Empower Your Team*

**BUILD-YOUR-OWN  
TRAINING WEEK**

Maximize the impact of your training by combining courses in one location to create a customized training week. Pair two courses and save up to \$300. For a complete list of courses available, visit [www.sqetraining.com](http://www.sqetraining.com) or call 888-268-8770 or 904-278-0524 for pairing discount options.

**SQE  
TRAINING**  
www.sqetraining.com

Improve your skills and help your organization increase its performance through targeted high-value training. Delivered by top software consultants, training through SQE Training is one of the best investments you can make to meet your business objectives.

### AGILE SOFTWARE DEVELOPMENT TRAINING WEEK LOCATIONS

February 25-29, 2008	Denver, CO
March 10-14, 2008	San Francisco, CA
April 14-18, 2008	Austin, TX
May 12-16, 2008	Boston, MA
June 2-6, 2008	Seattle, WA

### Choose from 4 specialized training courses:

#### THREE-DAY COURSES (Monday - Wednesday)

- Scrum Master Implementation Workshop
- Practical Test-Driven Development

#### TWO-DAY COURSES (Thursday - Friday)

- Design Patterns Explained
- User Stories and Estimation in Agile Development



Let SQE Training come to you. For more information about on-site training courses, contact SQE Training at 904-278-0524 x212 or 888-268-8770 or email [onsitetraining@sqe.com](mailto:onsitetraining@sqe.com).



<p>*** Soldier Friendliness ***</p> <p>Easy to use</p> <p>Not too heavy</p> <p>Always working</p> <p>*** Soldier Friendliness ***</p> <p>Gist: Everyone is supposed to use the tool without any pre-knowledge about it.</p> <p>*** Soldier Friendliness ***</p> <p>Average time for a "normal" soldier to learn how to use the 5 most important functions.</p> <p>*** Soldier Friendliness ***</p> <p>Light weight, small size (pocket), shock-, water-, temp- resistant</p> <p>Ease of use</p> <p>Independent of light conditions</p> <p>*** Soldier Friendliness ***</p> <p>Gist: Covers the soldiers need for fast and correct actions.</p> <p>*** Soldier Friendliness ***</p> <p>A reliable system that is easy to use in the dark.</p> <p>*** Soldier Friendliness ***</p> <p>Easy to handle without looking</p> <p>Easy to carry (whatever that implies)</p> <p>Rugged</p> <p>*** Soldier Friendliness ***</p> <p>Gist: Allows use in the field, by people without (higher) technical education, dressed in combat and carrying lot's of other equipment.</p> <p>*** Soldier Friendliness ***</p> <p>Gist: Could be carried by one soldier</p> <p>*** Soldier Friendliness ***</p> <p>Gist: Darkness</p> <p>Out in the nature</p> <p>*** Soldier Friendliness ***</p> <p>Easy to transport and &lt;use&gt; in a &lt;soldier environment&gt;</p> <p>*** Soldier Friendliness ***</p> <p>Usable in field</p> <p>*** Soldier Friendliness ***</p> <p>Light</p>	<p>Comfortable</p> <p>Dependable</p> <p>Accurate</p> <p>Ergonomic</p> <p>*** Soldier Friendliness ***</p> <p>Rugged</p> <p>Easy to use</p> <p>Light</p> <p>Fast user of</p> <p>*** Soldier Friendliness ***</p> <p>Gist: system accessibility under any conditions, with environmentalist approval.</p> <p>*** Soldier Friendliness ***</p> <p>Gist: the speed to figure out how to make a call with the system</p> <p>*** Soldier Friendliness ***</p> <p>Gist: user friendly for soldier</p> <p>*** Soldier Friendliness ***</p> <p>Gist: can be easy fixed outside the cloth</p> <p>Not heavy</p> <p>Easy to handle if with clothes on</p> <p>Easy to hear but should not be recognised by the enemy</p> <p>*** Soldier Friendliness ***</p> <p>Robust</p> <p>trustable</p> <p>*** Soldier Friendliness ***</p> <p>Waterproofness</p> <p>Long time power supply</p> <p>Solid</p> <p>Camouflage colour</p> <p>Can take a beating</p> <p>*** Soldier Friendliness ***</p> <p>Not traceable</p> <p>Not breakable</p> <p>Long standby-time</p> <p>Long Distance available</p> <p>*** Soldier Friendliness ***</p> <p>Ease of learning</p> <p>Ease of use</p> <p>Ease of maintaining</p> <p>Safety</p> <p>Extreme Availability</p> <p>*** Soldier Friendliness ***</p> <p>Easy to use in dark</p> <p>Water proof</p> <p>Impact proof</p> <p>Camouflage Coloured</p> <p>No "fancy" functions</p> <p>i.e.. Just call button</p>	<p>*** Soldier Friendliness ***</p> <p>The soldier shall be able to handle the phone after 50% instruction, compared to need by the old phones</p> <p>*** Soldier Friendliness ***</p> <p>Easy operable</p> <p>Idiot proof</p> <p>Easy cleanable / maintainable</p> <p>*** Soldier Friendliness ***</p> <p>No buttons to push pr function</p> <p>Hours to learn to use system for soldier</p> <p>Average <del>roof</del> means to perform a specific task</p> <p>*** Soldier Friendliness ***</p> <p>In what environment. And under what circumstances to be used</p> <p>*** Soldier Friendliness ***</p> <p>The soldiers can use the phone with one hand in the dark</p> <p>*** Soldier Friendliness ***</p> <p>Equipment can be operated with gloves on</p> <p>*** Soldier Friendliness ***</p> <p>System easily usable in the field, at night with extra light, with an <del>enc</del> very robust.</p> <p>*** Soldier Friendliness ***</p> <p>Gist: Time until &lt;new soldier&gt; can use &lt;item&gt; for &lt;task&gt;</p> <p>*** Soldier Friendliness ***</p> <p>Gist: Alla tjänster som tillhandahålls av ett icke-militärt system under "normala" förhållanden skall även tillhandahållas och användas under en krigssituation. (Swedish!)</p> <p>*** Soldier Friendliness ***</p> <p>Gist: a tool to trust in a crises situation. Reliable in all situations</p> <p>*** Soldier Friendliness ***</p> <p>Gist: usability, easy to use</p> <p>Easy to use during darkness</p> <p>Weight, not heavy</p> <p>*** Soldier Friendliness ***</p> <p>Can be used under stress.</p>
--	---	---

Case Study Example of Ambiguity Test: Dozens of different interpretations during an ambiguity test, for a multinational client, in London. We asked them to interpret 'Soldier Friendliness' of a military mobile phone.

### Writers Are Responsible for Readers Wrong Renditions

If the reader – the outsourcer - misunderstands requirements I hope we can agree that the fault is entirely the spec writer.

As with all communication the communicator should know their audience, and communicate so they can understand. Anything less is both arrogant and impractical.

We have a rule in Planguage that says:

R9: Clear: Specifications should be ‘clear enough to test’ and ‘unambiguous to their intended readers.’ <- CE, p 17
---

One client of ours took the full consequences of this rule by introducing another rule that the intended (types of) readers were to be listed on the front page of a requirements set.

We frequently ask people to count the number of serious-consequence violations of this rule in a sample of their own, often ‘approved’ requirements. The extrapolated estimate of the number of violations is on average about 100 per 300 words. Extrapolated means that since the quality control process we use is provably 1/3 effective, and a team finds about 33 ‘defects’ (rule violations) in a page, then we reckon (and can prove by further QC) that there are about 100. People are quite ‘shocked’ by this result – especially when they realize it is serious. We normally release requirements with a huge and unacceptable lack of ambiguity and lack of clarity. We don’t bother to conduct a quality control process to measure this; so we are unaware of it. But it should come as no surprise. We know that even dictionaries happily give us about many different definitions of the same word, and the definition I need, in a narrow technical area like systems engineering (for , for example architecture, is not usually one of those.

architecture   ärki tek ch əɾ   rkə t k(t) əɾ   k t kt ə   noun
--

1. the art or practice of designing and constructing buildings.
---

• the style in which a building is designed or constructed, esp. with regard to a specific period, place, or culture : Victorian architecture.
--

2. the complex or carefully designed structure of something : the chemical architecture of the human brain.
---

• the conceptual structure and logical organization of a computer or computer-based system : a client/server architecture.
--

Sample definition, not the one I need for systems engineering (of which there are many!)

Assume Nothing, Specify Everything.

You need to make all assumptions explicit. You need to do this in detail, in the context of each individual requirement.

R14: Assumptions: All known assumptions (and any relevant source(s) of any assumptions) should be explicitly stated.
--

<i>The ‘Assumption’ Planguage parameter can be used for this purpose. But there are also a number of alternative ways, such as {Risk, Source, Impacts, Depends On, Comment, Authority, [Qualifiers], If }. In fact, any reasonable device, suitable for the purpose, will do.</i>
---

The box above cites one Planguage rule regarding assumptions.

Here is a template one could use to remind people to state assumptions:

Requirement Specification Template (A Summary Template)

Tag: <Tag name for the system>.

Type: System.

===== Basic Information =====

Version: <Date or other version number>.

Status: <{Draft, SQC Exited, Approved, Rejected}>.

Quality Level: <Maximum remaining major defects/page, sample size, date>.

Owner: <Role/e-mail/name of the person responsible for changes and updates>.

Stakeholders: <Name any stakeholders (other than the Owner) with an interest in the system>.

Gist: <A brief description of the system>.

Description: <A full description of the system>.

Vision: <The overall aims and direction for the system>.

===== Relationships =====

Consists Of: Sub-System: <Tags for the immediate hierarchical sub-systems, if any, comprising this system>.

Linked To: <Other systems or programs that this system interfaces with>.

===== Function Requirements =====

Mission: <Mission statement or tag of the mission statement>.

Function Requirement:

<{Function Target, Function Constraint}>: <State tags of the function requirements>.

Note: 1. See Function Specification Template. 2. By default, 'Function Requirement' means 'Function Target'.

===== Performance Requirements =====

Performance Requirement:

<{Quality, Resource Saving, Workload Capacity}>: <State tags of the performance requirements>.

Note: See Scalar Requirement Template.

===== Resource Requirements =====

Resource Requirement:

<{Financial Resource, Time Resource, Headcount Resource, others}>: <State tags of the resource requirements>.

Note: See Scalar Requirement Template.

===== Design Constraints =====

Design Constraint: <State tags of any relevant design constraints>.

Note: See Design Specification Template.

===== Condition Constraints =====

Condition Constraint: <State tags of any relevant condition constraints or specify a list of condition constraints>.

===== Priority and Risk Management =====

Rationale: <What are the reasons supporting these requirements? >.

Value: <State the overall stakeholder value associated with these requirements>.

Assumptions: <Any assumptions that have been made>.

Dependencies: <Using text or tags, name any major system dependencies>.

Risks: <List or refer to tags of any major risks that could cause delay or negative impacts to the achieving the requirements>.

Priority: <Are there any known overall priority requirements? >.

Issues: <Unresolved concerns or problems in the specification or the system>.

===== Evolutionary Project Management Plan =====

Evo Plan: <State the tag of the Evo Plan>.

===== Potential Design Ideas =====

Design Ideas: <State tags of any suggested design ideas for this system, which are not in the Evo Plan>.

Example Template: In addition to the explicit parameter 'Assumptions' there are a number of other parameters that deal with some class of assumption such as 'Rationale, Risks, Priority, Design Constraints' and many more. Source CE page 77.



## **Too Much is Safer than Too Little**

When we teach and consult with companies, we quickly show that what was a short one-liner of a requirement, easily becomes a full page, with all the background information, such as assumptions and relationships is added to the requirement as background information. We would not recommend this if we did not believe, and if our clients do not believe, that it pays off.

One director of a telecom company defended this in front of his CEO and fellow directors (and this author) by pointing out that the traditional one line marketing requirement cost on average \$400,000 to implement. If the cost of building a full page of related data to that requirement, was the price the company needed to pay to protect that investment, then it was a small price indeed. This company has since won the battle with the reluctant (to specify requirements well) marketing people, which included firing two reluctant marketing directors. And included a founding director patiently working with marketing people to improve the quality of requirements. It also includes test and quality managers patiently working with engineers to increase the scope of requirement definitions (for several years).

Emergency Stop: Type: Function. Description: <Requirement detail>. Module Name: GEX.F124. Users: {Machine Operator, Run Planner}. Assumptions: The User Handbook describes this in detail for all <User Types>. User Handbook: Section 1.3.5 [Version 1.0]. Planned Implemented: Early Next Year, Before Release 1.0. Latest Implementation: Version 2.1. "Bug Correction: Bug XYZ." Test: FT.Emergency Stop. Test [System]: {FS.Normal Start, FS.Emergency Stop}. Hardware Components: {Emergency Stop Button, Others}. Owner: Carla.
--

Real example: some additional parameters to describe a requirement in excess of the 'Description'. Source CE, p.91.

## **If they ask a question, document and integrate the answer**

I have a fanatic habit. When I am listening to clients discuss requirements, I grab the answers to questions and document them in the requirement specification. I am fearful that:

- nobody heard the answer correctly
- if they did then the answer is unintelligible or incorrect but no one can be bothered to challenge it now
- people who were not in the room will have different information, but they will never know what was said
- conditions will change and this will not be true or relevant any more in the future

So, I figure it is good practice to document things, and hope this is more useful than no documenting it.

I also have a stringent practice of documenting exactly who gave the answer, quite publicly – right then and there.

Assumption: the CEO fully backs this requirement. <- John Jones.

It has the interesting effect of getting the person cited as the source to take a good look at what he is being quoted with (was it even captured accurately?), and make him wonder if he really wants to be stuck with having said it (is it really accurate, and will it make me seem foolish later?) Just healthy, I figure.

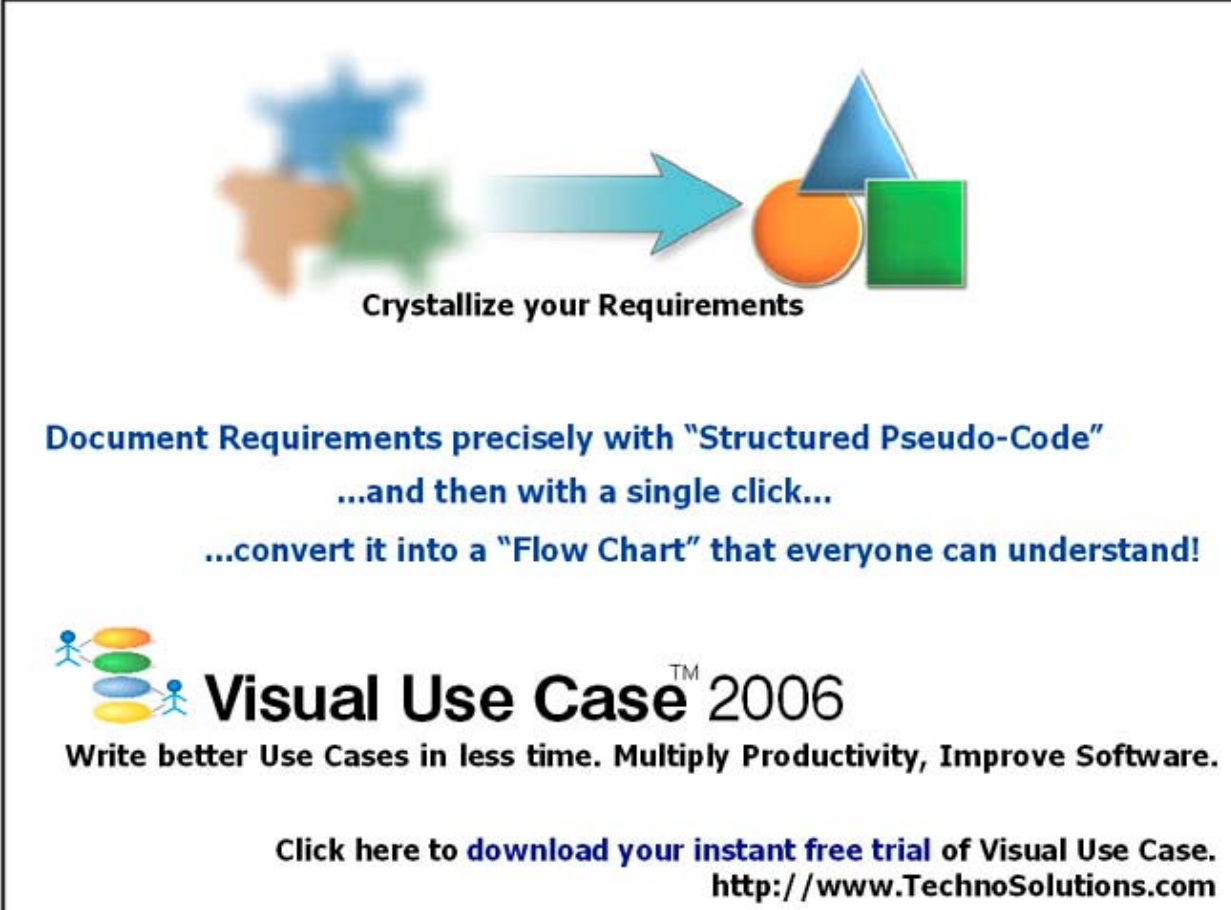
### Quality Control before sending

Can we agree that we should be responsible for NOT sending 100 potential misunderstandings per page to an outsourcing supplier? You need to set an exit level for your requirements process, for example “no more than 1.0 Major defects per page”.

If you do set such a requirement process exit level, seriously, then our client experience (the earliest serious large scale experience was at Douglas Aircraft, for engineering drawings, 1989) is that you will strongly motivate engineers to learn to reduce their defect injection – to write clearly enough to succeed and to ‘exit’ from the requirements process. (If you want to learn more about this process, see my paper Agile Specification Quality Control: Shifting emphasis from cleanup to sampling defects (INCOSE 2005) available at [www.gilb.com](http://www.gilb.com)). It takes several learning cycles, but individual engineers actually reduce their ‘defect injection’ in requirements by one order of magnitude in a few months and more in the longer term.


---

Advertisement – Visual Use Case - Click on ad to reach advertiser web site



**Crystallize your Requirements**

**Document Requirements precisely with "Structured Pseudo-Code"**  
**...and then with a single click...**  
**...convert it into a "Flow Chart" that everyone can understand!**

 **Visual Use Case<sup>TM</sup> 2006**  
**Write better Use Cases in less time. Multiply Productivity, Improve Software.**

Click here to **download your instant free trial** of Visual Use Case.  
<http://www.TechnoSolutions.com>

Before you waste both your outsourcers time and your own – make sure you quality control requirements in relation to a reasonable set of rules of specification. Have a reasonable requirement process exit level – not 100 major defects per page (by default – when you don't measure the level).

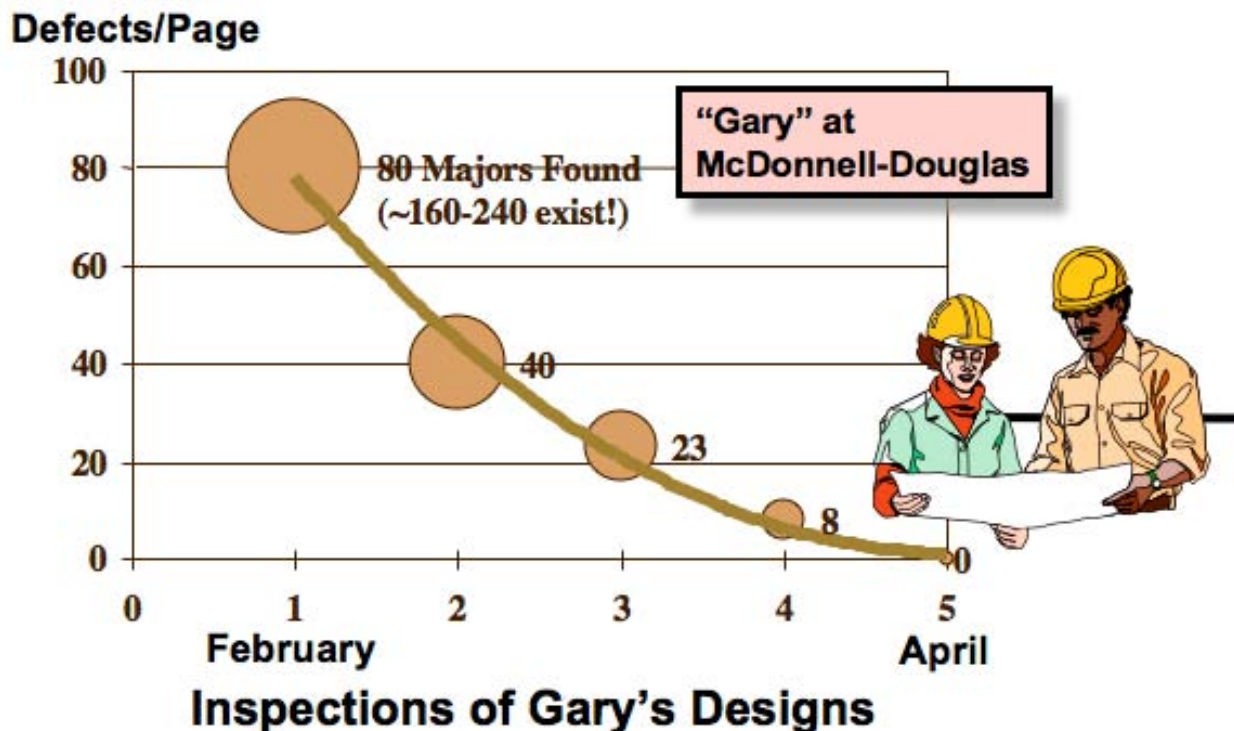


Illustration: the personal learning curve of one aircraft engineer, when subject to specification quality measurement and a numeric exit condition. Hundreds of engineers went through this defect injection reduction process. A multinational bank reported reduction from 80.4 Major defects/pages to 11.2 when comparing the requirements for several IT projects.

### **Evolve Requirement Delivery**

One way to find out if you have done a good enough job on requirements is to check the real thing. The evolutionary project management (Evo) method consciously divides up a project into about 50 increments. Each increment will attempt to deliver some of the requirements to some of the stakeholders.

If you start getting high value requirements back from the supplier, on a regular basis, then you must be communicating requirements well. If not, you can at least analyse your problems early and correct your process. It is not possible to have a large scale problem without you getting early and frequent warning signals.

You can even make an interesting 'no cure no pay' contractual relationship with your supplier/ Pay them for requirements delivered successfully – not work carried out.

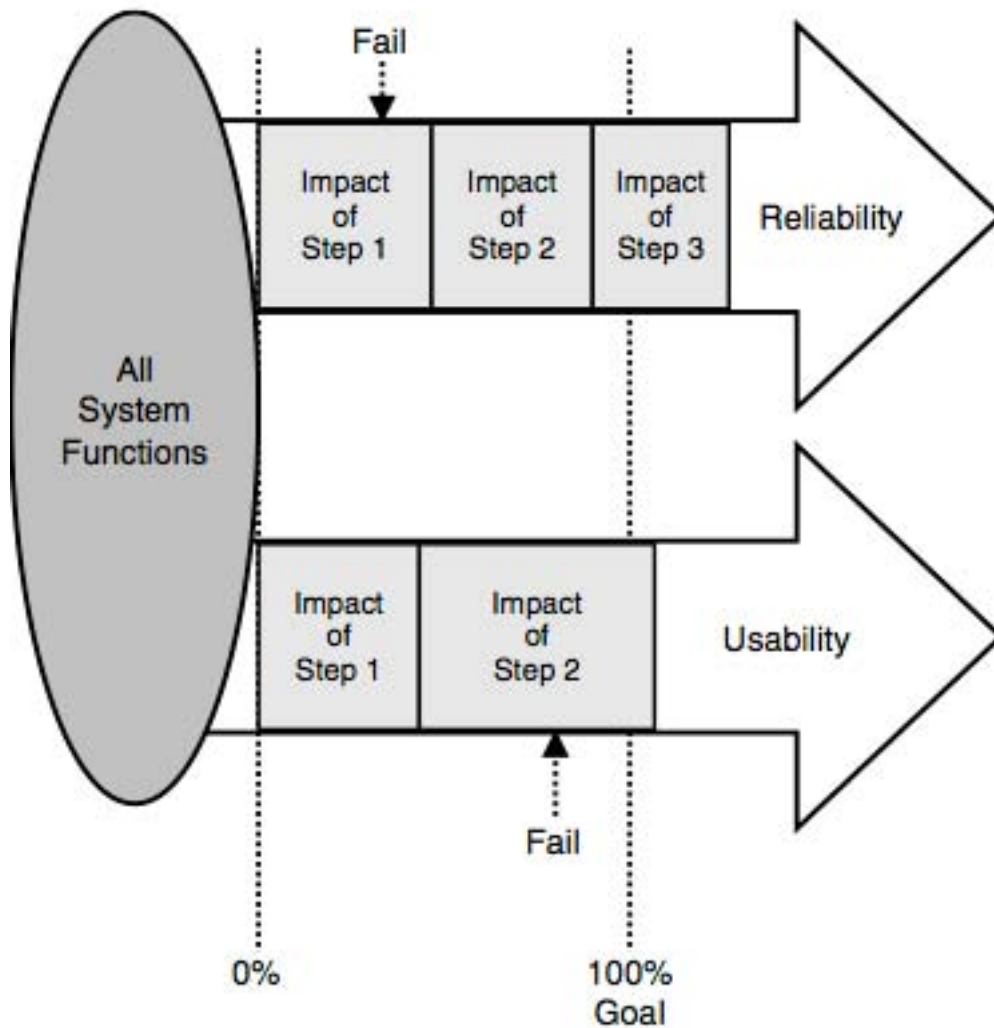


Figure. The gradual delivery of requirements at each Evo step (simplified) Source CE page 158.

### **Quantify Quality**

Quality requirements are a dominant reason for many projects. We want to replace an old system with one that has higher qualities.

Quality levels are a powerful driver of system costs. The nearer you get to perfect quality, the nearer you get to infinite costs.

For any serious outsourcing communication, you have to quantify the qualities you want.

Most people make the mistake of simply declaring the quality to be critical “Highest levels of security” ... “State of the Art Reliability”.

Or worse, they simply replace the quality requirement with a design, intended to deliver the requirement, “industry standard programming languages:... “Consistent user interfaces” are examples.



## Requirements

Here is an example of a requirement specification rule:

*Rule QQ: All system quality requirements must be defined quantitatively with a scale of measure and at least one numeric target or constraint level required.*

Quantification gives you about as clear and unambiguous communication of your quality requirement to the outsourcer as you will get. It also forces you to really think more deeply about what you really expect from the outsourcer. I observe that people do not have clear ideas about most of their quality requirements at all. They have no policy to quantify qualities. In fact they feel no responsibility to clarify qualities – in spite of often explicitly declaring them to be ‘primary motivation for this project’ (I saw that today at work, at the latest. I see it at least once a week).

In addition quantification means you can expect to measure both partial and complete progress towards the quality target levels. This means you can see if your supplier is really delivering at a rate consistent with meeting your deadlines.

Impact Estimation Table: Reportal codename "Hyggen"											
Current Status			Improvements			Reportal - E-SAT features					
Units	Units	%	Past	Tolerable	Goal	Units	Units	%	Past	Tolerable	Goal
75,0	25,0	62,5	50	75	90	83,0	48,0	80,0	40	85	95
			Usability.Intuitiveness (%)			0,0	67,0	100,0	67	0	0
14,0	14,0	100,0	Usability.Consistency.Visual (Elements)	0	11	4,0	59,0	100,0	63	8	4
15,0	15,0	107,1	Usability.Consistency.Interaction (Components)	0	11	10,0	397,0	100,0	407	100	10
5,0	75,0	96,2	Usability.Productivity (minutes)	5	2	94,0	2290,0	103,9	2384	500	180
5,0	45,0	95,7	Usability.Flexibility.OfflineReport.ExportFormats	5	1	10,0	10,0	13,3	0	100	100
3,0	2,0	66,7	Usability.Robustness (errors)	3	4	774,0	507,0	51,7	1281	600	300
1,0	22,0	95,7	Usability.Replacability (nr of features)	1	0	5,0	3,0	60,0	2	5	7
4,0	5,0	100,0	Usability.ResponseTime.ExportReport (minutes)	5	3	0,0	0,0	0,0	?	?	?
1,0	12,0	150,0	Usability.ResponseTime.ViewReport (seconds)	13	5	3,0	35,0	97,2	38	3	2
1,0	14,0	100,0	Development resources	15	3	0,0	800,0	100,0	800	0	0
203,0					191	1350,0	1100,0	146,7	150	500	1000
						64,0			0		84

Current Status			Improvements			Survey Engine .NET					
Units	Units	%	Past	Tolerable	Goal	Units	Units	%	Past	Tolerable	Goal
1,0	1,0	50,0	Usability.Replacability (feature count)	14	13	7,0	9,0	81,8	16	10	5
20,0	45,0	112,5	Usability.Productivity (minutes)	65	35	17,0	8,0	53,3	25	15	10
4,4	4,4	36,7	Usability.ClientAcceptance (features count)	0	4	943,0	-186,0	#####	170	60	30
101,0			Development resources	0		5,0	10,0	95,2	15	7,5	4,5
					86	2,0			0		48

Current Status			Improvements			XML Web Services					
Units	Units	%	Past	Tolerable	Goal	Units	Units	%	Past	Tolerable	Goal
1,0	1,0	50,0	Usability.Replacability (feature count)	14	13	7,0	9,0	81,8	16	10	5
20,0	45,0	112,5	Usability.Productivity (minutes)	65	35	17,0	8,0	53,3	25	15	10
4,4	4,4	36,7	Usability.ClientAcceptance (features count)	0	4	943,0	-186,0	#####	170	60	30
101,0			Development resources	0		5,0	10,0	95,2	15	7,5	4,5
					86	2,0			0		48

Case Study. Real (our client FIRM) tracking of 25 quality levels in product development, in the 9<sup>th</sup> of a 12 week cycle before customer delivery. In the improvement % column is the degree of progress towards 100% of the planned target levels. It is clear that this project with 25 engineers working in 4 parallel teams is on track for meeting the targets on time. They would have to average 75% to be on track in general.

### Constrain Explicitly

We have a tendency to be explicit about what we want. But we may fail to be equally explicit about what we don't want. If we fail to define all relevant constraints, together with our target requirements.

It is not enough to say how warm you want the room temperature to be. You need to specify the upper and lower limits too.

It is not enough to assume the outsourcer will know what your national laws are, or that your system must respect them.

You need to make a long explicit list of those constraints, and take nothing for granted.

Here is a template for thorough specification of scalar requirements (variables like performance, quality, costs). Source CE, page 135.

Elementary scalar requirement template <with hints>

Tag: <Tag name of the elementary scalar requirement>.

Type: <{Performance Requirement: {Quality Requirement, Resource Saving Requirement, Workload Capacity Requirement}, Resource Requirement: {Financial Requirement, Time Requirement, Headcount Requirement, others}}>.

===== Basic Information =====

Version: <Date or other version number>.

Status: <{Draft, SQC Exited, Approved, Rejected}>.

Quality Level: <Maximum remaining major defects/page, sample size, date>.

Owner: <Role/e-mail/name of the person responsible for this specification>.

Stakeholders: <Name any stakeholders with an interest in this specification>.

Gist: <Brief description, capturing the essential meaning of the requirement>.

Description: <Optional, full description of the requirement>.

Ambition: <Summarize the ambition level of only the targets below. Give the overall real ambition level in 5–20 words>.

===== Scale of Measure =====

Scale: <Scale of measure for the requirement (States the units of measure for all the targets, constraints and benchmarks) and the scale qualifiers>.

===== Measurement =====

Meter: <The method to be used to obtain measurements on the defined Scale>.

===== Benchmarks ===== "Past Numeric Values" =====

Past [<when, where, if>]: <Past or current level. State if it is an estimate> <- <Source>.

Record [<when, where, if>]: <State-of-the-art level> <- <Source>.

Trend [<when, where, if>]: <Prediction of rate of change or future state-of-the-art level> <- <Source>.

===== Targets ===== "Future Numeric Values" =====

Goal/Budget [<when, where, if>]: <Planned target level> <- <Source>.

Stretch [<when, where, if>]: <Motivating ambition level> <- <Source>.

Wish [<when, where, if>]: <Dream level (unbudgeted)> <- <Source>.

===== Constraints ===== "Specific Restrictions" =====

Fail [<when, where, if>]: <Failure level> <- <Source>.

Survival [<when, where, if>]: <Survival level> <- <Source>.

===== Relationships =====

Is Part Of: <Refer to the tags of any supra-requirements (complex requirements) that this requirement is part of. A hierarchy of tags (For example, A.B.C) is preferable>.

Is Impacted By: <Refer to the tags of any design ideas that impact this requirement> <- <Source>.

Impacts: <Name any requirements or designs or plans that are impacted significantly by this>.

===== Priority and Risk Management =====

Rationale: <Justify why this requirement exists>.

Value: <Name [stakeholder, time, place, event]: Quantify, or express in words, the value claimed as a result of delivering the requirement>.

Assumptions: <State any assumptions made in connection with this requirement> <- <Source>.

Dependencies: <State anything that achieving the planned requirement level is dependent on> <- <Source>.

Risks: <List or refer to tags of anything that could cause delay or negative impact> <- <Source>.

Priority: <List the tags of any system elements that must be implemented before or after this requirement>.

Issues: <State any known issues>.

In bold, I have highlighted the aspects of the requirement specification that give information about constraints with respect to this particular requirement.

### Connect relationships

You need to help your outsourcers understand relationships between requirements and a long list of other things.

Here are some specific principles about this: (from another recent paper on this called "Requirement Relationships: A Theory, some Principles, and a Practical Approach" ([www.gilb.com](http://www.gilb.com)))

#### The 'Requirement Relationship' Principles:

1. THE CLIENT STAKEHOLDER PRINCIPLE: A requirement specification that has no identified *client* stakeholder, is not a *valid* requirement . *Because - we cannot ascertain its usefulness or value to a given stakeholder.*
2. THE SERVER STAKEHOLDER PRINCIPLE: A requirement specification that has no specified, or implied, *server* stakeholder(s) is not yet *seriously* planned for real implementation. *Thus we cannot understand who will deliver it, when, or how efficiently*
3. THE REQUIREMENT RELATIONSHIPS PRINCIPLE: A single requirement can have *any useful number* and *types* of relationships that are worth specifying. *The total costs of specification should be less than the expected benefits in the long term for the system.*
4. THE EARLY RELATIONSHIP PRINCIPLE: Failure to deal with requirement relationships in the *requirement* specifications *themselves* will have the effect of increasing development and maintenance costs. *Because the relationships will then more likely be sensed, and dealt with, downstream, in design, testing and operation or even decommissioning.*
5. THE DYNAMIC RELATIONSHIP PRINCIPLE: Requirement Relationships are not static, nor are they are all *determinable* initially. *Consequently we need to track them as they emerge and change; we need to verify them, and we need to analyze the consequences of any change in requirement relationships.*
6. THE RELATIONSHIPS ARE 'CRITICAL KNOWLEDGE' PRINCIPLE: The requirement *relationship* knowledge is *itself* far more valuable and critical than the requirement alone. *This is because it potentially helps us to impact greater value and scope, earlier and better than we otherwise would be aware of, or would deal with. The requirement itself might change but most relationships might remain as useful facts*
7. THE 'REQUIREMENT REVIEW BASIS' PRINCIPLE: All requirement review processes are dependent on the quality and quantity of requirement relationship information available. *Otherwise we risk approving requirements in ignorance of critical facts.*
8. THE RISK MANAGEMENT PRINCIPLE: The Risk Management process is continuously dependent on the quality of requirement relationship information. *All requirement relationship specifications help us to identify and manage risks.*
9. THE 'BUTTERFLY EFFECT' PRINCIPLE: Even *one single* fault in a requirement relationship specification can be the root cause of project or system failure. *It is impossible to be sure that even a single missing or incorrect requirement relationship specification will be unable to severely or critically damage your engineering effort.*
10. THE DESIGN RELATIONSHIP PRINCIPLE: All *architecture and design* specifications must follow the same relationship specification principles, as their 'near cousins', requirements. *This is because, all 'solutions, means, designs, architectures, and strategies' are themselves also requirements, as viewed by other stakeholders.*

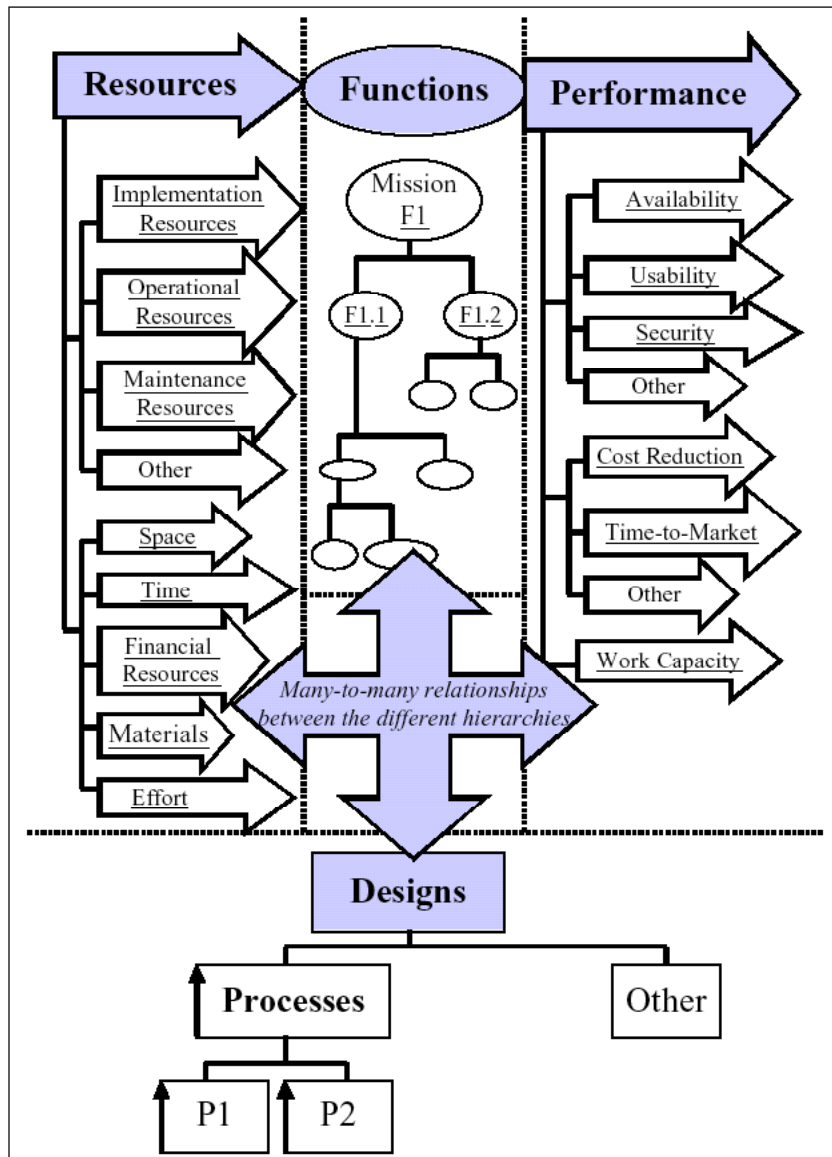


Illustration: This shows the four main system attribute types: resource, function, performance and design. It also shows the processes, which implement the functions. Using Planguage, the complex relationships amongst these four different types can be specified. For example, a specific performance level might apply only to a handful of functions; rather than the entire system. Or, a function might be implemented by several processes. Or, different resources can be specifically allocated to different functions. [source: CE 2005, Figure 3.3]:

### Summary

Much better quality of requirements is a current necessity for most of us. But outsourcing places demands on the requirements process that are unusually high because of physical and cultural differences. This paper has tried to give some specific and practical advice on what to do to specify better requirements. It is hard work, but it is a lot less work than dealing with the misunderstandings caused by bad requirements. Copyright © 2007 by Tom Gilb

### References

CE: Gilb, Tom, Competitive Engineering, A Handbook For Systems Engineering, Requirements Engineering, and Software Engineering Using Planguage, ISBN 0750665076, 2005, Publisher: Elsevier Butterworth-Heinemann. A sample chapter will be found at Gilb.com.



## **Exploratory Testing: Finding the Music of Software Investigation**

Jonathan Kohl [jonathan@kohl.ca](mailto:jonathan@kohl.ca)  
Kohl Concepts Inc., [www.kohl.ca](http://www.kohl.ca)

My friend Steve is an exceptional classical guitarist. Watching him perform is inspiring – he has a rare mastery over the instrument and has spent years developing his craft. Steve can also explain the techniques he is using while he is playing, to teach and demonstrate how a student can learn and improve their own skills. Steve can make a guitar sing, and says that music is about tension and resolution. If music is all tension, you get uncomfortable as a listener. If it only resolves, it is boring, tedious repetition. Steve extends this concept to the actual physical actions that a guitarist employs to create certain sounds. For example, if you play with a lot of tension, you will limit your ability to do certain tasks. To make music, you need to find a balance between tension and resolution, and to find this balance, you need a mix of knowledge, skill and creativity.

Like Steve, my friend James Bach is also exceptionally skilled. James isn't a guitarist, he is a software tester. James is also inspiring to watch while he practices his craft. He is a master of skilled exploratory testing: simultaneous test design, execution and learning [1]. James can also explain the testing techniques he uses while he is testing, to instruct testing students. The first time I saw him test software, I was reminded of my friend Steve. This time the tension and resolution wasn't related to music composition or the execution of techniques on a musical instrument. Instead, the tension and resolution revolved around ideas. James would simultaneously design and execute tests based on his curiosity about the application. He would get feedback on a test, learn from it and design a new test. The tension was generated by the questioning nature of his tests, and the resolution emerged from the results of those tests. There was something almost musical in this interplay between the mind of the tester and the application being tested. This shouldn't be surprising; as a software tester, James has a well-developed mix of knowledge, skill and creativity.

As a software testing consultant and musician, I meet a lot of skilled testers who do amazing work. Through experience and a lot of trial and error, they have developed skills they can't easily explain. Unfortunately, with software testing, there aren't as many obvious avenues for skill development as there are for musicians. Many software testers don't realize that there are learnable exploratory testing skills they can develop to help them become even more valuable to software development teams.

When I work in a new organization, it doesn't take long for me to get introduced to the testers who are highly valued. Often, when I ask testers about the times they did their best work, they apologize for breaking the rules: "I didn't follow any pre-scripted test cases, and I didn't really follow the regular testing process." As they describe how they came through for the team on a crucial bug discover, they outline activities that I identify with skilled exploratory testing. Little do they know that this is often precisely why they are so effective as testers. They have developed analysis and investigative skills that give them the confidence to use the most powerful testing tool we have at our disposal: the human mind. Exploratory testing is something that testers naturally engage in, but because it is the opposite of scripted testing, it is misunderstood and sometimes discouraged. In an industry that encourages pre-scripted testing processes, many testers aren't aware that there are other ways of testing other than writing and following test scripts. Both software testing and music can be interpreted and performed in a variety of ways.

Western classical music is often highly scripted in the form of sheet music. Compositions are written in a language that performers can interpret with their voices or instruments. Despite a detailed well-disseminated shared “language” for printed music, it is difficult to perform music exactly the way the composer intended, particularly with musical pieces that have been around for centuries, because we don’t have the composer around anymore to consult. The opposite of playing from sheet music is improvisation, creating unrehearsed, unscripted music. A continuum exists between these two styles, because a composition is open to at least some interpretation by the performer, and some performers embellish more than others. Software that plays music is very precise in repeating what is input from sheet music, but is rarely as pleasant to listen to as a real performer. Music can be boring and tedious when played by a computer program, and full of life when played by a musician. At the other end of the spectrum, successful improvisation requires skill, and top performers study to develop a large breadth and depth of musical theory and technical proficiency on their instruments in order to successfully and creatively improvise.

In testing, test scripts that are written down are also open to interpretation by the test executor. Automating these tests is the only way to guarantee that they will be repeated exactly the same way, but like automating music, the lack of interpretation in execution can limit the results. A computer can only find the problems we predict and program it to find. Repeating scripted tests over and over can get boring, tedious, and may only feel like idea resolution, without the vital tension created by curiosity. At the other end of the spectrum, there is improvisational testing: exploratory testing. Pure exploratory testing means that my next test is completely shaped by my current ideas, without any preconceptions. Pure scripted testing and pure exploratory testing are on opposite ends of a continuum.

This analogy of music and software testing isn’t perfect however. Music is performed for entertainment purposes or as practice for musicians who are developing their skills. The end goal is entertainment for listeners, skill development, and the enjoyment of the musician. Software testing on the other hand isn’t generally done for entertainment, instead it is used to discover information.

---

Advertisement – On-Demand Business Applications - Click on ad to reach advertiser web site

---



## Build, Sell and Deliver On-Demand Business Applications

You	We
You know exactly what needs to be done.	We know how to do it.
You configure business rules.	We ensure application productivity and responsiveness.
You support your customers.	We deliver an infrastructure for remote setup and customer support automation.
You are billing your customers according to your pricing model.	We charge you royalty fee.
You sell.	We are offering a sales force infrastructure for leads management and accounting.

**Voice mail:** +1-847-465-3930 | **E-mail:** [info@dbflex.net](mailto:info@dbflex.net) | **Website:** [www.dbflex.net](http://www.dbflex.net)

As Cem Kaner says, software testing is an investigative activity to provide quality-related information about software [2]. To gather different kinds of information, we want to be open to different interpretations, and to be able to look at a problem in many different ways. In music, improvisation can have negative effects when used at an inappropriate time or in an inappropriate manner. (When a musician plays a wrong note, we really notice it.) In software testing, exploring and improvisation, even when done wrong, can often lead to wonderful sources of new information. Inappropriate interpretations can be a hazard in musical performances, but on software projects, accidents, or “playing the wrong notes”, can lead to important discoveries. Furthermore, software projects are faced with risk, and exploratory testing allows for us to instantaneously adjust to new risks.

What does skilled exploratory testing look like? Here is scripted testing and exploratory testing in action. In one test effort, I came across a manual test script and its automated counterpart which had been written several releases ago. They were for an application I was unfamiliar with, using technology I was barely acquainted with. I had never run these tests before, so I ran the automated test first to try to learn more about what was being tested. It passed, but the test execution and results logging didn’t provide much information other than “test passed.” To me, this is the equivalent of the emails I get that say: “Congratulations! You may already be a winner!” Statements like that on their own, without some sort of corroboration mean very little.

I didn’t learn much from my initial effort: running the automated test didn’t reveal more information about the application or the technology. Since learning is an important part of testing work, I delved more deeply. I moved on to the manual test script, and followed each step. When I got to the end, I checked for the expected results, and sure enough, the actual result I observed matched what was predicted in the script. Time to pass the test and move on, right? I still didn’t understand exactly what was going on with the test and I couldn’t take responsibility for those test results completely on blind faith. That violates my purpose as a tester; if I believed everything worked as advertised, why test at all? Furthermore, experience has taught me that tests can be wrong, particularly as they get out of date. Re-running the scripted tests provided no new information, so it was time leave the scripted tests behind.

One potential landmine in providing quality-related software information is tunnel vision. Scripted tests have a side effect of creating blinders - narrowing your observation space. To widen my observation possibilities, I began to transition from scripted testing to exploratory testing. I began creating new tests by adding variability to the existing manual test, and I was able to get a better idea of what worked and what caused failures. I didn’t want to write these tests down because I wanted to adjust them on the fly so I could quickly learn more. Writing them down would interrupt the flow of discovery, and I wasn’t sure what tests I wanted to repeat later.

I ran another test, and without the scripted tests to limit my observation, noticed something that raised my suspicions: the application became sluggish. Knowing that letting time pass in a system can cause some problems to intensify, I decided to try a different kind of test. I would follow the last part of the manual test script, wait a few minutes, and then thoroughly inspect the system. I ran this new test, and the system felt even more sluggish than before. The application messaging showed me the system was working properly, but the sluggish behaviour was a symptom of a larger problem not exposed by the original tests I had performed.

Investigating behind the user interface, I found that the application was silently failing; while it was recording database transactions as completing successfully, it was actually deleting the data. We had actually been losing data ever since I ran the first tests. Even though the tests appeared to pass, the application was failing in a serious manner. If I had relied only on the

scripted manual and automated tests, this would have gone undetected, resulting in a catastrophic failure in production. Furthermore, if I had taken the time to write down the tests first, and then execute them, I would most likely have missed this window of opportunity that allowed me to find the source of the problem. Merely running the scripted tests only felt like repeating an idea resolution, and didn't lead to any interesting discoveries. On the other hand, the interplay between the tension and resolution of exploratory testing ideas quickly led to a very important discovery. Due to results like this, I don't tend to use many procedural, pre-scripted manual test cases in my own personal work.

So how did I find a problem that was waiting like a time-bomb for a customer to stumble upon? I treated the test scripts for what they were: imperfect sources of information that could severely limit my abilities to observe useful information about the application. Before, during and after test execution, I designed and re-designed tests based on my observations. I also had a bad feeling when I ran the test. I've learned to investigate those unsettled feelings rather than suppress them because feelings of tension don't fit into a script or process; often, this rapid investigation leads to important discoveries. I didn't let the scripts dictate to me what to test, or what success meant. I had confidence that skilled exploratory testing would confirm or deny the answers supplied by the scripted tests.

Testers who have learned to use their creativity and intelligence when testing come up with ways to manage their testing thought processes. Skilled exploratory testers use mental tricks to help keep their thinking sharp and consistent. Two tricks testers use to kick start their brains are heuristics (problem-solving approaches) and mnemonics (memory aids) [3].

---

Advertisement – MKS Integrity - Click on ad to reach advertiser web site

---

**I AM** A PROJECT MANAGER.  
**I MANAGE** A TEAM ACROSS FOUR SITES.  
**I NEED MKS.**



My team needs real-time collaboration across platforms and locations.

We need clear traceability throughout the project lifecycle, starting with requirements.

I need real-time updates on the status of my projects. I want project metrics in a dashboard view.

**We need ONE application lifecycle management platform to manage our projects from end to end. We need MKS.**

**With MKS, we are one.**

New in MKS Integrity 2007:

- ✓ Test Management
- ✓ Requirements Reuse
- ✓ SAP and PeopleSoft Change and Release Management

Call us: 519 884 2251 or toll free 1 800 613 7535 | [www.mks.com](http://www.mks.com)

**MKS**  
APPLICATION LIFECYCLE MANAGEMENT  
FOR THE ENTERPRISE



Musicians use similar techniques, and may recognize “the circle of fifths” as a heuristic to follow if they get lost in an improvised performance. (This isn’t a guarantee though, a heuristic may or may not work for you. When a heuristic is inappropriate, you simply try another.) Musicians tend to have large toolboxes of heuristics, and also use mnemonics as well. One example is “Every Good Boy Does Fine” which is used to remember the notes “EGBDF” on the lines of a staff. Skilled testers use similar tools to remember testing ideas and techniques.

I’m sometimes called in as an outside tester to test an application that is nearing completion. If the software product is new, a technique I might use is the “First Time User” heuristic. With very little application information, and using only the information available to the first time user, I begin testing. It’s important for me to know as little as possible about the application at this time, because once I know too much, I can’t really test the way a first-time user would.

To start testing in these situations, I often use a mnemonic I developed called “MUTII”. (A composer named Nicolo Mutii helps me remember it.) This mnemonic helps me maintain consistency in the way I think about testing. Expanding the mnemonic:

**Market**—The targeted constituency of users this software is intended for. For example, “the finance department”, or “medium sized accounting firms”.

**Users**—The actual users who will use the software. Who are the users? What do they do? What are their motivations for using our software?

**Tasks**—What are the tasks that the users will use this software for? What are some typical tasks in their work?

**Information**—What does the product tell me about the tasks it automates, and how I can perform them?

**Implementation**—Is the software easy to use as a first time user? Is it reliable? Can I easily implement the tasks given the information and design of the product?

Before I start testing the application, I gather information about the market and the users from the business. This helps frame the kinds of tests I will develop when I use the software for the first time. If I’m not familiar with the market and users, I will also ask for typical tasks engaged in by the users.

When I start testing, I open my notebook to take notes of my observations and thoughts, and any bugs I find. (See Figure 1.) I begin designing a test in my mind, execute it with the software, and observe the results. I keep repeating this process, changing tests, and referring back to my MUTII mnemonic.

Not only does each letter of the mnemonic help me frame my testing, but the acronym helps me quickly design and execute many tests under each section as I go. I may also use other heuristics and mnemonics as I test, if I find areas to explore differently or more deeply.

As I work through the application, I may find mismatches between the application and the market it is intended for, and the information supplied to users. If I have trouble figuring out the software’s purpose and how to use it, so will end users. This is important usability information that I write down in my notes. If the software isn’t usable, it isn’t going to sell. I invariably find bugs in this first testing session.

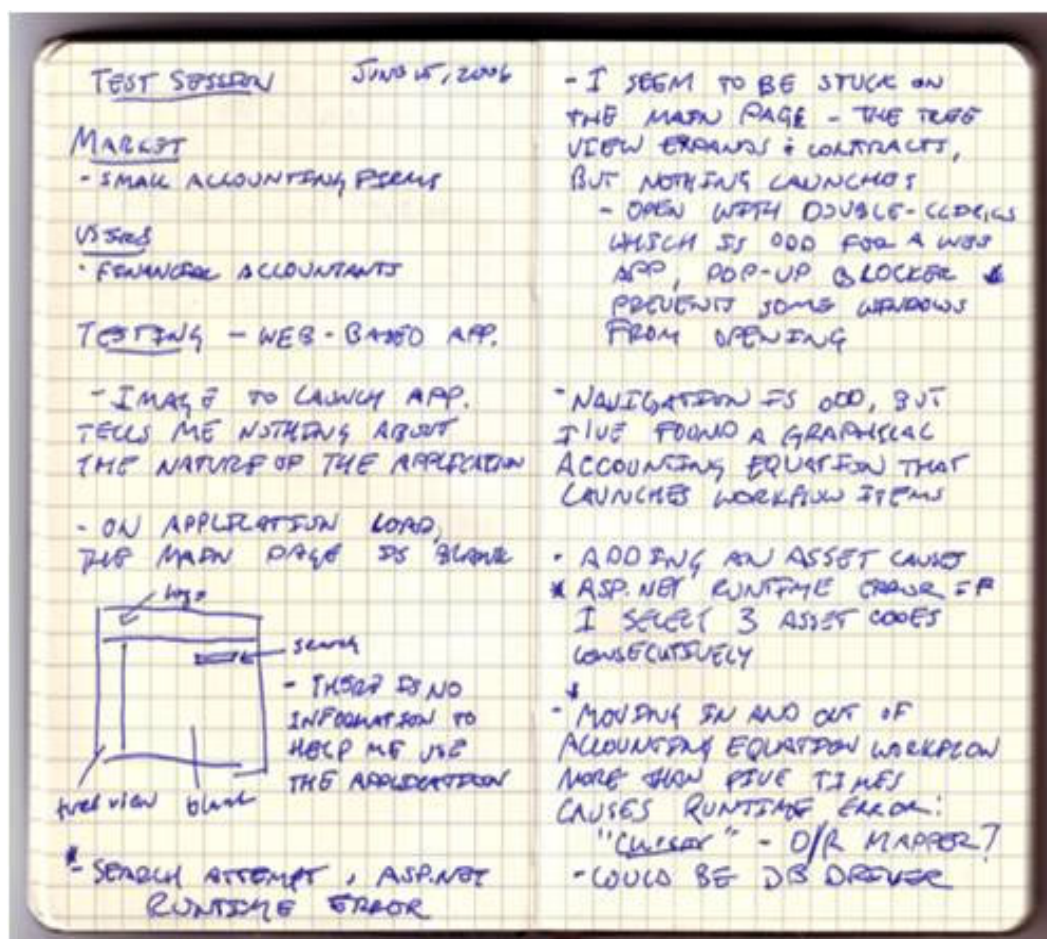


Figure 1: Excerpt from explanatory testing session notes

I explore them, take notes so I can report them later, and design new tests around those bugs. After the session is over, I will have bugs to report and usability questions to ask. I will now have a model developed in my mind for testing this software. My brain will work on this model constantly, even when I'm not testing, and other testing activities will help build this and other models of the software.

Using heuristics and mnemonics helps me be consistent when testing, but I don't let them rule my testing actions. If I observe something suspicious, I explore it. If something feels wrong, I investigate, and confirm or deny that feeling with defensible facts. It's common to switch from heuristics and mnemonics to pure free-form improvisation and back again, or to improvise around highly structured tests. Exploratory testing—like improvising—helps me adapt my thinking and my actions based on what the software is telling me. This is a powerful concept. You can seize upon opportunities as soon as you observe them. Furthermore, you can adapt quickly to project risks, and discover and explore new ones. By developing skills to manage your thinking about testing, you no longer have to wait for spontaneous discoveries to appear out of thin air, and not be able to explain why you found a particular problem, or repeat it.

Developing exploratory testing skill puts you in charge of your testing approach. Skilled software testing, like skilled musicianship, is often referred to as "magic", simply because it is misunderstood. Music follows a set of patterns, heuristics and techniques. If you know a handful, you can make music quite readily. Getting there by trial and error takes a lot longer, and you'll have a hard time explaining how you got there once you arrive. Testing using pure observation and trial and error can be effective, but can be effective much more quickly if there is a system to frame it.

Skilled exploratory testing can be a powerful way of thinking about testing. However, it is often misunderstood, feared and discouraged. When we dictate that all tests must be scripted, we discourage the wonderful tension and resolution in testing, driven by a curious thinker. We limit the possibilities of our software testing leading to new, important discoveries. We also hamper our ability to identify and adapt to emerging project risks. In environments that are dominated by technology, it shouldn't be surprising that we constantly look to tools and processes for solutions. But tools and processes on their own are stupid things. They still require human intelligence behind them. In the right hands, software tools and processes, much like musical instruments, enable us to realize the results we seek. There are many ways to perform music, and there are many ways to test software. Skilled exploratory testing is another effective thinking tool to add to the testing repertoire.

### References

1. Bach, James. (2003) Exploratory Testing Explained  
<http://www.satisfice.com/articles/et-article.pdf>
2. Kaner, Cem. (2004). The Ongoing Revolution in Software Testing. Presented at Software Test & Performance Conference, December, 2004, Baltimore, MD  
<http://www.kaner.com/pdfs/TheOngoingRevolution.pdf>
3. Bach, James. (1999, November) Heuristic Risk-Based Testing. Software Testing and Quality Engineering Magazine <http://www.satisfice.com/articles/hrbt.pdf>

Jonathan Kohl, Kohl Concepts Inc. © 2007 First published by Kohl Concepts Inc. in July 2007

---

Advertisement – Software Development Tools Directory - Click on ad to reach advertiser web site

---

Are you looking for the right development tools for your task? A new directory of tools related to software development has been launched. It covers all software development activities: programming (java, .net, php, xml, c/c++, ajax, etc), testing, configuration management, databases, project management, modeling, etc.

**Search and reference software development tools on  
[www.softdevtools.com](http://www.softdevtools.com)**

## **Four ways to a Practical Code Review**

Jason Cohen,  
Smart Bear Software, <http://www.smartbearsoftware.com/>

### **How to almost get kicked out of a meeting**

Two years ago I was *not invited* to a meeting with the CTO of a billion-dollar software development shop, but I didn't know that until I walked in the room. I had been asked by the head of Software Process and Metrics to come and talk about a new type of lightweight code review that we had some successes with.

But the CTO made it clear that my presence was Not Appreciated.

"You see," he explained, "we already do code inspections. Michael Fagan invented inspections in 1976 and his company is teaching us how to do it." His face completed the silent conclusion: "And you sir, are no Michael Fagan."

"Currently 1% of our code is inspected," offered the process/metrics advocate. "We believe by the end of the year we can get it up to 7%." Here Mr. Metrics stopped and shot a glance over to Mr. CTO. The latter's face fell. Whatever was coming, they obviously had had this discussion before.

"The problem is we can't inspect more than that. Given the number of hours it takes to complete a Fagan inspection, we don't have the time to inspect more than 7% of the new code we write."

My next question was obvious: "What are you going to do about the other 93%?" Their stares were equally obvious – my role here was to convince the CTO that we had the answer.

This story has a happy ending, but before we get there I have to explain what it means to "inspect" code because this is what most developers, managers, and process engineers think of when they hear "code review." It's the reason this company couldn't review 93% of their code and why developers *hate* the idea. And changing this notion of what it means to "review code" means liberating developers so they can get the benefits of code review without the heavy-weight process of a formal inspection.

### **Michael Fagan – father of a legacy**

If you've ever read anything on peer code review you know that Michael Fagan is credited with the first published, formalized system of code review. His technique, developed at IBM in the mid-1970's, demonstrably removed defects from any kind of document from design specs to OS/370 assembly code. To this day, any technique resembling his carries his moniker of "code inspection."

### **Take a deep breath...**

I'm going to describe a "code inspection" in brief, but brace yourself. This is heavyweight process at its finest, so bear with me. It will all be over soon, I promise. A code inspection consists of seven phases. In the Planning Phase the author gathers Materials, ensures that they meet the pre-defined Entry Criteria, and determines who will participate in the inspection. There are four participants with four distinct roles:



The Author, the Moderator, the Reviewer, and the Reader. Sometimes there is an Observer. All participants need to be invited to the first of several meetings, and this meeting must be scheduled with the various participants. This first meeting kicks off the Introduction Phase where the Author explains the background, motivation, and goals for the review. All participants get printed copies of the Materials. (This important — it's not a Fagan Inspection unless it's printed out.) The participants schedule the next meeting and leave.

This starts the Reading Phase where each person reads the Materials, but each role reads for a different purpose and — this is very important — no one identifies defects. When the next meeting convenes this starts the Inspection Phase. The Moderator sets the pace of this meeting and makes sure everyone is performing their role and not ruining anything with personal attacks. The Reader presents the Materials because it was his job to "read for comprehension" since often someone else's misunderstanding indicates a fault in the Materials.

During the meeting a Defect Log is kept so the Author will know what needs to be fixed. Before the meeting ends, they complete a rubric that will help with later process improvement. If defects were found the inspection enters the Rework Phase where the Author fixes the problems, and later there will be a Verification Phase to make sure the fixes were appropriate and didn't open new defects. Finally the inspection can enter the Completed Phase.

---

Advertisement – SPA Conference - Click on ad to reach advertiser web site



## **Software Practice Advancement 2008** **16 - 19 March 2008**

The Robinson Centre, Bedfordshire, UK

**Technology... Practice... Process... People... Participation**



SPA Conferences bring together experts and practitioners across a variety of industries, from around the world, to share the latest thinking in software development. At SPA we don't follow the latest fads, instead we help you identify the real advances that will enable you to build better software.

SPA2008 will provide you with a unique high-energy learning experience. This is the conference for software professionals seeking the latest practices in software development. You will have the opportunity to explore a broad range of pioneering software development and deployment practices and hear about the latest ideas in software management techniques to help you better manage your projects and teams.

SPA2008 is the conference for participants. It is about sharing experiences, working together to address the hot topics, and asking the tough questions. Your participation is the key. You don't need to know the answers to join in, you just need the enthusiasm.

### **Breaking news...**

- Early Bird Discount now available until 31st January 2008
- Michael Feathers and Peter Deutsch confirmed as keynote speakers

**To book a place or find out more visit [www.spaconference.org](http://www.spaconference.org)**  
or call +44 (0) 870 760 6863





### A Typical Formal Inspection Process

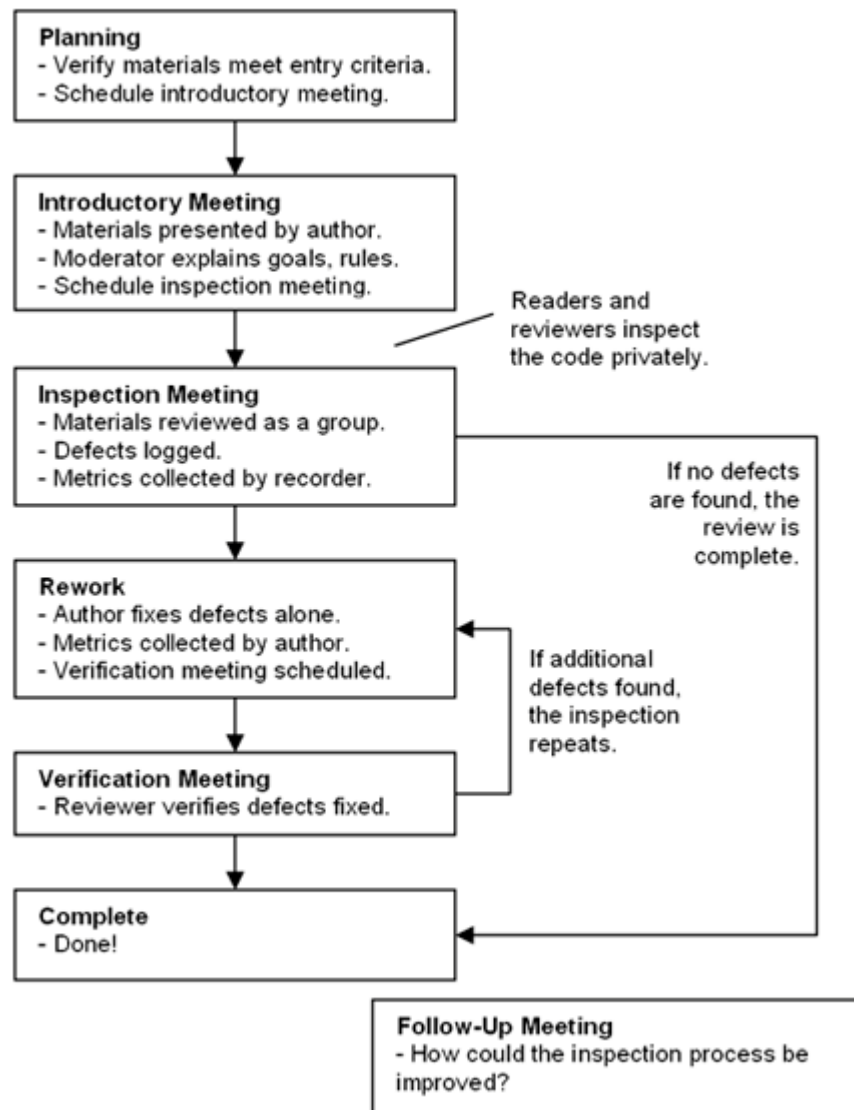


Figure1: Typical workflow for a "formal" inspection. Not shown are the artifacts created by the review: The defect log, meeting notes, and metrics log. Some inspections also have a closing questionnaire used in the follow-up meeting

#### ...you can let it out now

The good news is, this works. It uncovers defects, it helps when training new hires, and the whole process can be measured for process insight and improvement. If you have extra money laying around in your budget, Mr. Fagan himself will even come show you how to do it. The bad news should be obvious in this day of Agile Methodologies. Studies show that the average inspection takes 9 man-hours per 200 lines of code, so of *course* Mr. CTO couldn't do this for every code change in the company.

Over the years there have been experiments, case studies, and books on this subject, almost always using some form of "code inspection" as the basis. In our survey of published case studies and experiments in the past 20 years, we found that 95% of them tried inspections only in small pilot groups, and that in *no case* were they able to apply the technique to all their software development projects.

## **If "Agile" can do it, why can't we?**

But surely there is another way. Fagan inspections were designed in the days when business logic was written in assembly language and "computer science" wasn't a major and dinosaurs roamed the earth.

Have we learned nothing since then? Don't we need different techniques when reading object-oriented code in a 3-tier application? Don't the challenges of off-shore development require new processes? Hasn't the rise of Agile Methodologies shown us that we can have process and metrics and measurement and improvement and happy developers all at the same time?

So finish the story already!

By now you can guess how the story ends. Using arguments not unlike those above, Mr. Metrics and I convinced Mr. CTO to at least try our lightweight code review technique in a pilot program with a one development group that was already hopelessly opposed to Fagan inspections. The metrics that came out of that group demonstrated the effectiveness of the lightweight system, and within 18 months Code Collaborator was deployed across the entire organization.

## **What does "lightweight" mean?**

Assuming you've bought into the argument that code review is good but heavyweight inspection process is not practical, the next question is: How do we make reviews practical?

We'll explore four lightweight techniques:

1. **Over-the-shoulder:** One developer looks over the author's shoulder as the latter walks through the code.
2. **Email pass-around:** The author (or SCM system) emails code to reviewers
3. **Pair Programming:** Two authors develop code together at the same workstation.
4. **Tool-assisted:** Authors and reviewers use specialized tools designed for peer code review.

## **Over-the-shoulder reviews**

This is the most common and informal (and easiest!) of code review. An "over-the-shoulder" review is just that — a developer standing over the author's workstation while the author walks the reviewer through a set of code changes.

Typically, the author "drives" the review by sitting at the keyboard and mouse, opening various files, pointing out the changes and explaining what he did. The author can present the changes using various tools and even go back and forth between changes and other files in the project. If the reviewer sees something amiss, they can engage in a little "spot pair-programming" as the author writes the fix while the reviewer hovers. Bigger changes where the reviewer doesn't need to be involved are taken off-line.

With modern desktop-sharing software a so-called "over-the-shoulder" review can be made to work over long distances, although this can complicate the process because you need to schedule these sharing meetings and communicate over the phone.

### Over-the-Shoulder Review Process

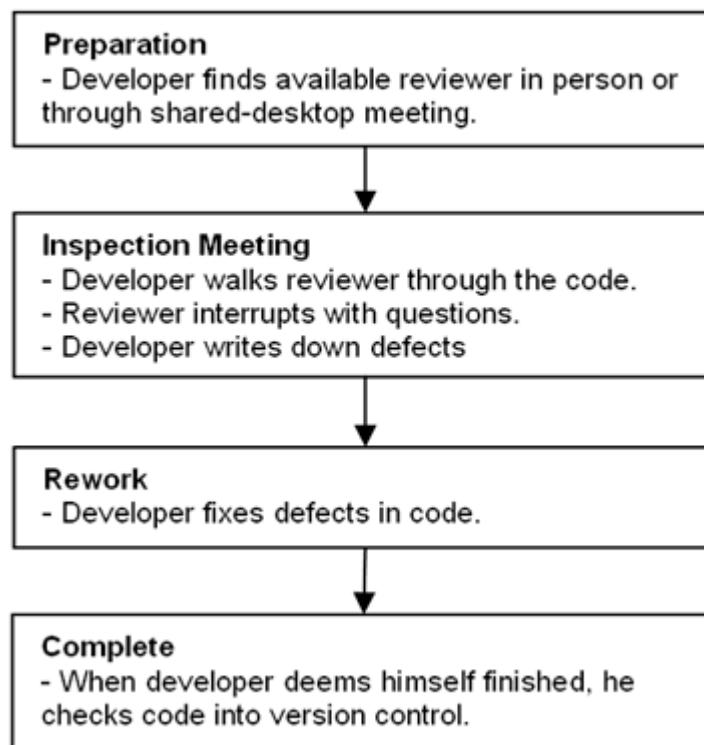


Figure 2: A typical Over-the-shoulder code walk-through process. Typically, no review artifacts are created.

The most obvious advantage of over-the-shoulder reviews is simplicity in execution. Anyone can do it, any time, without training. It can also be deployed whenever you need it most — an especially complicated change or an alteration to a "stable" code branch.

Before I list out the pros and cons, I'd like you to consider a certain effect that only this type of review exhibits. Because the author is controlling the pace of the review, often the reviewer doesn't get a chance to do a good job. The reviewer might not be given enough time to ponder a complex portion of code. The reviewer doesn't get a chance to poke around other source files to check for side-effects or verify that API's are being used correctly.

The author might explain something that clarifies the code to the reviewer, but the next developer who reads that code won't have the advantage of that explanation unless it is encoded as a comment in the code. It's difficult for a reviewer to be objective and aware of these issues while being driven through the code with an expectant developer peering up at him.

So:

- Pro: Easy to implement
- Pro: Fast to complete
- Pro: Might work remotely with desktop-sharing and conference calls
- Con: Reviewer led through code at author's pace
- Con: Usually no verification that defects are really fixed
- Con: Easy to accidentally skip over a changed file

- Con: Impossible to enforce the process
- Con: No metrics or process measurement/improvement

### Email pass-around reviews

This is the second-most common form of lightweight code review, and the technique preferred by most open-source projects. Here, whole files or changes are packaged up by the author and sent to reviewers via email. Reviewers examine the files, ask questions and discuss with the author and other developers, and suggest changes.

#### E-Mail Pass-Around Process: Post Check-In Review

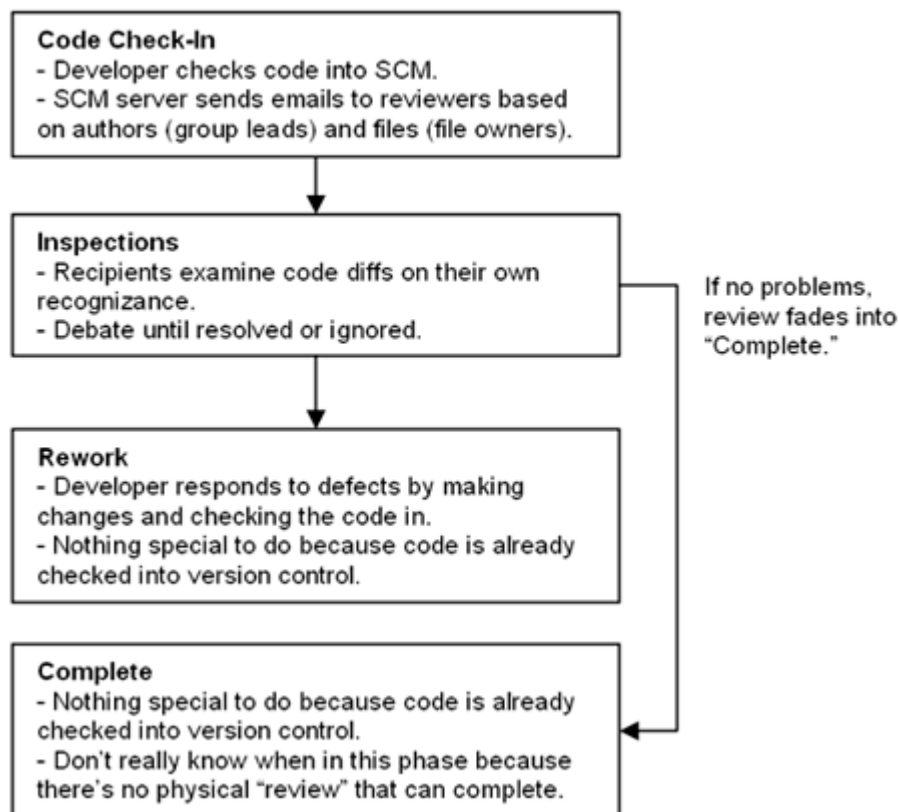


Figure 3: Typical process for an e-mail pass-around review for code already checked into a version control system. These phases are not this distinct in reality because there's no tangible "review" object.

The hardest part of the email pass-around is in finding and collecting the files under review. On the author's end, he has to figure out how to gather the files together. For example, if this is a review of changes being proposed to check into version control, the user has to identify all the files added, deleted, and modified, copy them somewhere, then download the previous versions of those files (so reviewers can see what was changed), and organize the files so the reviewers know which files should be compared with which others. On the reviewing end, reviewers have to extract those files from the email and generate differences between each.

The version control system can assist the process by sending the emails out automatically. The automation is helpful, but for many code review processes you want to require reviews *before* check-in, not *after*. Like over-the-shoulder reviews, email pass-arounds are fairly easy to implement. They also work just as well across the hall or across an ocean.

A unique advantage of email-based review is the ease in which other people can be brought into conversations, whether for expert advice or complete deferral. And unlike over-the-shoulder, emails don't break developers out of "the zone" as they are working; reviews can be done whenever the reviewer has a chance.

The biggest drawback to email-based reviews is that they can quickly become an unreadable mass of comments, replies, and code snippets, especially when others are invited to talk and with several discussions in different parts of the code. It's also hard to manage multiple reviews at the same time. Imagine a developer in Hyderabad opening Outlook to discover 25 emails from different people discussing aspects of three different code changes he's made over the last few days. It will take a while just to dig through that before any real work can begin.

Another problem is that there's no indication that the review is "done." Emails can fly around for any length of time. The review is done when everyone stops talking.

So:

- Pro: Fairly easy to implement
- Pro: Works with remote developers
- Pro: SCM system can initiate reviews automatically
- Pro: Easy to involve other people
- Pro: Doesn't interrupt reviewers
- Con: Usually no verification that defects are really fixed
- Con: How do you know when the review is "complete?"
- Con: Impossible to know if reviewers are just deleting those emails
- Con: No metrics or process measurement/improvement

### **Pair-programming (review)**

Most people associate pair-programming with XP and agile development in general. Among other things, it's a development process that incorporates continuous code review. Pair-programming is two developers writing code at a single workstation with only one developer typing at a time and continuous free-form discussion and review.

Studies of pair-programming have shown it to be very effective at both finding bugs and promoting knowledge transfer. And some developers really enjoy doing it. (Or did you forget that making your developers happy is important?)

There's a controversial issue about whether pair-programming reviews are better, worse, or complementary to more standard reviews. The reviewing developer is deeply involved in the code, giving great thought to the issues and consequences arising from different implementations. On the one hand, this gives the reviewer lots of inspection time and a deep insight into the problem at hand, so perhaps this means the review is more effective. On the other hand, this closeness is exactly what you don't want in a reviewer; just as no author can see all typos in his own writing, a reviewer too close to the code cannot step back and critique it from a fresh and unbiased position. Some people suggest using both techniques — pair-programming for the deep review and a follow-up standard review for fresh eyes. Although this takes a lot of developer time to implement, it would seem that this technique would find the greatest number of defects. We've never seen anyone do this in practice.



The single biggest complaint about pair-programming is that it takes too much time. Rather than having a reviewer spend 15-30 minutes reviewing a change that took one developer a few days to make, in pair-programming you have two developers on the task the entire time.

Of course pair-programming has other benefits, but a full discussion of this is beyond the scope of this article.

So:

- Pro: Shown to be effective at finding bugs and promoting knowledge-transfer
- Pro: Reviewer is "up close" to the code so can provide detailed review
- Pro: Some developers like it
- Con: Some developers don't like it
- Con: Reviewer is "too close" to the code to step back and see problems
- Con: Consumes a lot of up-front time
- Con: Doesn't work with remote developers
- Con: No metrics or process measurement/improvement

### Tool-assisted review

This refers to any process where specialized tools are used in all aspects of the review: collecting files, transmitting and displaying files, commentary, and defects among all participants, collecting metrics, and giving product managers and administrators some control over the workflow.

"Tool-assisted" can refer to open-source projects, commercial software, or home-grown scripts. Either way, this means money — you're either paying for the tool or paying your own folks to create and maintain it. Plus you have to make sure the tool matches your desired workflow, and not the other way around.

Therefore, the tool had better provide many advantages if it is to be worthwhile. Specifically, it needs to fix the major problems of the foregoing types of review with:

**Automated File-Gathering:** As we discussed in email pass-around, developers shouldn't be wasting their time collecting "files I've changed" and all the differences. Ideally, the tool should be able to collect changes *before* they are checked into version control *or after*.

**Combined Display: Differences, Comments, Defects:** One of the biggest time-sinks with any type of review is in reviewers and developers having to associate each sub-conversation with a particular file and line number. The tool must be able to display files and before/after file differences in such a manner that conversations are threaded and no one has to spend time cross-referencing comments, defects, and source code.

**Automated Metrics Collection:** On one hand, accurate metrics are the only way to understand your process and the only way to measure the changes that occur when you change the process. On the other hand, no developer wants to review code while holding a stopwatch and wielding line-counting tools. A tool that automates the collection of key metrics is the only way to keep developers happy (i.e., no extra work for them) and get meaningful metrics on your process. A

full discussion of review metrics and what they mean (and don't mean) will appear in another article, but your tool should at least collect these three things: kLOC/hour (inspection rate), defects/hour (defect rate), and defects/kLOC (defect density).

**Workflow Enforcement:** Almost all other types of review suffer from the problem of product managers not knowing whether developers are reviewing all code changes or whether reviewers are verifying that defects are indeed fixed and didn't cause new defects. A tool should be able to enforce this workflow at least at a reporting level (for passive workflow enforcement) and at best at the version control level (with server-side triggers).

**Clients and Integration:** Some developers like command-line tools. Others need integration with IDE's and version control GUI clients. Administrators like zero-installation web clients and Web Services API's. It's important that a tool supports many ways to read and write data in the system.

If your tool satisfies this list of requirements, you'll have the benefits of email pass-around reviews (works with multiple, possibly-remote developers, minimizes interruptions) but without the problems of no workflow enforcement, no metrics, and wasting time with file/difference packaging, delivery, and inspection.

It's impossible to give a proper list of pros and cons for tool-assisted reviews because it depends on the tool's features. But if the tool satisfies all the requirements above, it should be able to combat all the "cons" above.

### So what do I do?

All of the techniques above are useful and will result in better code than you would otherwise have.

To pick the right one for you, start with the top of the list and work your way down. The first few are the simplest, so if you're willing to live with the downsides, stop there. Tool-assisted review has the most potential to remove downside, but you'll have to commit to a trial period, competitive analysis, and possibly some budget allocation.

No matter what you pick, your developers will find that code review is a great way to find bugs, mentor new hires, and share information. Just make sure you implement a technique that doesn't aggravate them so much that they revolt.

**Independent Report by Forrester Research, Courtesy of MKS:**  
"Selecting the Right Requirements Management Tool - Or Maybe None Whatsoever". Forrester Research recently evaluated the requirements management tools landscape across four criteria: Baselineing, Linking & Tracing, MS Word Support, and Workflow. MKS is the only vendor in the evaluation to fully satisfy all of the critical features in Forrester's evaluation. Download your copy of the independent report, courtesy of MKS:

<http://www.mks.com/mtdecforresterreport>

---

**New Agile Software Development Portal.** This site is a repository for resources concerning agile software development approaches (Extreme Programming (XP), Scrum, Test Driven Development (TDD), Feature Driven Development (FDD), DSDM, Lean Software) and practices (refactoring, pair programming, continuous integration, user stories). The content consists of articles, news, press releases, quotations, books reviews and links to articles, Web sites, tools, blogs, conferences and other elements concerning agile software development. Feel free to contribute with your own articles, links or press releases.

<http://www.devagile.com/>

---

While the principles of the Manifesto for Agile Software Development may look appealing for inexperienced developers, serious professionals know that the real world is not similar to the "Little House on the Prairie". Look at this humorous mirror site to the Manifesto for Agile Software Development

<http://www.waterfallmanifesto.org>

---

Advertising for a new Web development tool? Looking to recruit software developers? Promoting a conference or a book? Organizing software development training? This clasified section is waiting for you at the price of US \$ 30 each line. Reach more than 50'000 web-savvy software developers and project managers worldwide with a classified advertisement in Methods & Tools. Without counting the 1000s that download the issue each month without being registered and the 40'000 visitors/month of our web sites! To advertise in this section or to place a page ad simply <http://www.methodsandtools.com/advertise.php>

---

---

**METHODS & TOOLS** is published by **Martinig & Associates**, Rue des Marronniers 25,  
CH-1800 Vevey, Switzerland Tel. +41 21 922 13 00 Fax +41 21 921 23 53 [www.martinig.ch](http://www.martinig.ch)  
Editor: Franco Martinig ISSN 1661-402X  
Free subscription on : <http://www.methodsandtools.com/forms/submt.php>  
The content of this publication cannot be reproduced without prior written consent of the publisher  
**Copyright © 2007, Martinig & Associates**

---