"Best Practices" and "Context-Driven": Building a Bridge

International Conference on Software Testing, Analysis & Review
STAREast 2003
May 12-16, 2003 – Orlando, Florida, USA
Track session T15, Test Management

Neil Thompson
Thompson information Systems Consulting Ltd
23 Oast House Crescent,
Farnham, Surrey, England,
GU9 0NP, United Kingdom
www.TiSCL.com
+44 (0)7000 NeilTh (634584)

Abstract

The second basic principle of the Context-Driven school of software testing states that there are good practices in context, but there are no best practices. I believe that there are, although not the traditional ones. I propose a simple test by asking "Is there ever a time I wouldn't do or care about X?" If the answer is no, then X is an "always-good" practice. Practices such as risk management, prioritisation of work, using appropriate skills, choosing appropriate techniques, making informed decisions, and understanding our context, all seem to fit this definition. What varies is the degree of formality with which the practice is performed. In this paper, I explain:

- why the *concept* of "best practice" does not need to be inconsistent with a context-driven approach;
- how to evaluate your specific context; and
- how to choose specific good practices along the formal-to-informal continuum.

What this paper is:

- a genuine attempt to build a bridge between two apparently disagreeing factions, by:
  - (a) showing that the structure of "best practice" need not be as restrictive as many seem to think, and restating it in terms of "always-good" practices (principles, and constituent elements);
  - (b) applying additional structure to the insight and pragmatism of the Context-Driven school, and integrating that into the always-good structure;
- a new application of a process improvement framework, and a set of thinking and diagramming tools, which are already in the public domain (Goldratt's Theory of Constraints); and
- a set of options for applying these tools, ranging from the very basic to the comprehensive.

What this paper is not:

- an attempt to be controversial;
- a complex new theory;
- an all-or-nothing method;
- a rigid and precise application of Goldratt's Theory of Constraints; nor
- a "cookbook" of simple recipes.

Overview

Goldratt's Theory of Constraints is well known as a process improvement tool, especially in Dettmer's interpretation which concentrates on the five logical tools for integrated thinking. Part of it has also recently been applied by others to examine and resolve "superstitions" around how much documentation developers and testers should write. I now extend this concept to apply the wider Goldratt theory, and all of the thinking tools, to the choices necessary to determine what constitutes appropriate testing in a particular context. In process improvement, the theory concentrates on the weakest link, then the next-weakest, and so on. In a context-determining sense, the analogy is to concentrate on the key decisions in cause-effect sequence.

The five logical tools are diagramming techniques which in their process improvement form are:

- "what to change" (Current Reality),
- "what to change to" (Conflict Resolution then Future Reality), and
- "how to change" (Prerequisites then Transition).

I modify these diagrams to mean "context", "good practices", "what appropriate means", "questions to consider" and "choice categories and actions". Tables may be used in addition to the diagrams, to collate key information in a flexible way.

Contents

1. Introduction

Note: in this paper, "I" means the author and "you" means the reader.

1.1 Context-Driven and its seven principles

There is currently a schism (of variable depth and sometimes-surprising vehemence) between those advocating “Context-Driven” testing [refs 1, 2] and the adherents of "Best Practice", as exemplified by documentation standards, mandatory terminology, formalised processes, and certified qualifications.

The seven basic principles of the Context-Driven school are these:

1. The value of any practice depends on its context.
2. There are good practices in context, but there are no best practices.
3. People, working together, are the most important part of any project's context.
4. Projects unfold over time in ways that are often not predictable.
5. The product is a solution. If the problem isn't solved, the product doesn't work.
6. Good software testing is a challenging intellectual process.
7. Only through judgement and skill, exercised co-operatively throughout the entire project, are we able to do the right things at the right times to effectively test our products.

You may wonder to what extent the Best Practice faction actually disagrees with these principles. And I wonder to what extent is this schism imaginary, or merely cultural or geographic differences? Some of the principles are more debatable than others, that is for sure. Even if you claim to agree with *all* the principles, you may not qualify as a true Context-Driven member; some people merely pay lip-service, and this is not enough. But that discussion is for another place; it is principle number two which is the main subject of this paper.

1.2 If no best practices exist, what is "Best Practice"?

Several years ago, before Context-Driven became prominent, I tried to define "Best Practice" for myself, using:

- all the landmark testing text books[including, but not limited to, refs 3, 4, 5, 6, 7, 8];
- a symptom-driven approach to test process improvement [ref 9]; and
- my own practical experience in and around testing (25 years overall, in various test levels and doer / manager roles).

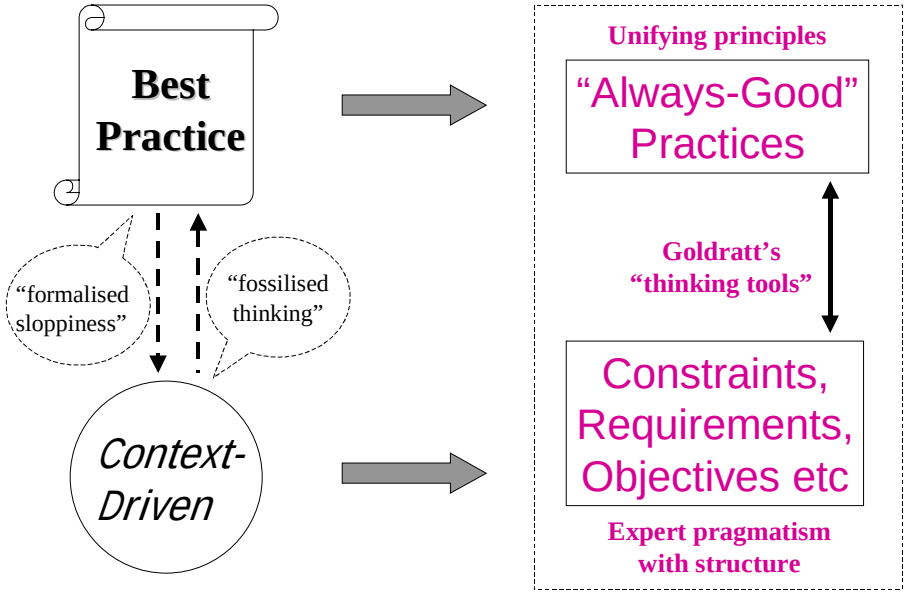
My answer turned out like a basic framework of things which I believe always apply to some degree, in some way. It consisted of a few basic principles, and elements within each principle. Some people say that is tantamount to "Best Practice", but it doesn't quite resemble the traditional view. Then I read more about Context-Driven, and it occurred to me that my principles have much about context in them, yet claim to be absolutes. The word "appropriate" features frequently. Perhaps if I were to call them "always-good" practices, they might build a bridge between Context-Driven principle number two and the traditional Best Practice viewpoint.

I then began to analyse variations in the degree of formality (rigour, detail, documentedness, "ceremony"[ref 10], etc.) with which each one may be applied, depending for example whether one is rushing out a shrink-wrapped PC product to hit a window of opportunity in the market, or whether one is developing safety-critical software for the government. The idea was that this analysis could then be used in a particular situation either to:

- adjust an existing methodology (real or merely used-in-theory) to be appropriate to the current project context, or
- to derive a methodology for a new project or department.

After working through a few examples, I came to believe that there are enough things, of sufficient specificity, which are inherently always good, that it is worth collecting them together, for example to structure our body of knowledge and help train testers. This seems quicker (though admittedly cruder) than offering a huge list of possible things which may or may not be good to do, then asking up to a dozen questions about the context before applying each of them[ref 2]. Instead, the more sophisticated Context-Driven thinking can be arranged around the always-good framework to help define in more detail what "appropriate" means. Diagram A below illustrates this unifying theme.

Diagram A: From conflict to an integrated framework



2. "Always-good" practices

My current list of "always good" practices is as follows. It does evolve slightly over time. The ones highlighted in bold are the ones which I have chosen to explain further in the sub-sections below, in terms of why they are always-good, and how the formality of their application may vary:

- Look at overall objectives of testing in terms of effectiveness and efficiency (both being desired, with effectiveness the primary, and having taken into account any mandatory standards).
- Make appropriate use of any highly recommended standards plus an effectiveness-efficiency-tailored selection from relevant optional standards. This should include consideration of integrity levels, which are used in some standards to allow different requirements which depend on the criticality of the product to which the standard is being applied[ref 11].

- Think of effectiveness as being based on risk management and quality management (risk first, because it is easier to define than quality and tends to drive the earlier tests; risk reduction is part of quality).
- (2.1) Use **a V-model** (not "*the* V-model") so you know what you are testing against, can see specific business and technical risks, and can define an appropriate level of acceptance testing to build confidence in the system and measure residual risk (the lower levels of testing are more for detecting failures).
- (2.2) Actually, use an extension of the V-model called the **W-model**, which covers all of verification and validation, makes clear which parts are testing and which parts are quality assurance / control in an individual project, and caters explicitly for retesting and regression testing.
- Tailor risks and quality targets to factors appropriate to the project and product.
- List and evaluate risks (probability, impact, etc.).
- Prioritise tests based on risks and quality targets.
- (2.3) **Refine test specifications progressively** (testware development lifecycle):
  - plan based on priorities and constraints;
  - design flexible tests to fit specific available human and environmental resources;
  - allow for appropriate procedure and script formats (e.g. separating out generic overall guidelines into overall test execution and management procedures); and
  - use synthetic plus lifelike test data at the appropriate test levels.
- (2.4) **Define and measure test coverage**, and allow for coverage changes.
- (2.5) In anomaly and incident management, **distinguish problems from change requests**.
- (2.6) **Use both severity (business impact, importance) and priority (testing impact, urgency)**, and vary assessment appropriately as go-live approaches.
- (2.7) **Distinguish retesting from regression testing**.
- Measure test progress and significance of remaining problems, thereby quantifying the net benefit of going live at any point after subtracting residual risk.
- Target efficiency by the following:
  - define and agree roles and responsibilities;
  - use an appropriate skills mix across all testing;
  - (2.8) **use system and acceptance testers from outside the development team**;
  - rehearse acceptance tests alongside system tests, so actual acceptance runs more smoothly;
  - (2.9) **use appropriate techniques**; and
  - use appropriate tools.
- (2.10) Decide process targets (maturity, effectiveness and/or efficiency) and **improve processes over time**.
- Define and use metrics.
- Backtrack incidents through the error-fault-failure chain to feed into overall quality processes.

2.1 Not *the* V-model, but "*a*" V-model

A V-model in some form just *is*, because it is a fact that software specification goes through some process of stepwise refinement, e.g. from requirements to module specifications. The least formal situation is of programmers talking to customers, or imagining what customers might want, and writing code directly. There is then a separate specification in each programmer's brain; the more programmers, the more likely these specifications will differ. The term adaptive development[ref 12] has been used to refer to software which evolves to meet a perceived need (as distinct from transformational, which is built to a specification). But I argue that a V-model also applies to adaptive development, as the system still probably has (in addition to programmers) designers, users and operations staff, and there are hence several viewpoints to be tested.

In iterative lifecycles, there is some structure within each iteration, so there is still integration testing and there are modified versions of system and acceptance testing. There is always at least one level of specification against which to test; what varies is the number of levels, the quality of information at each level (it may be spoken and inferred rather than written, and written documents will probably be out of date and otherwise imperfect) and the mapping to test levels. A V-model does not necessarily imply paper documentation: ironically, large specs can be read electronically, and some agile methods prefer handwritten index cards. The V-model should not be taken to imply a waterfall development lifecycle; that is an over-specialisation of it.

In summary: it seems to me undeniable that:
- all systems are integrated from parts;
- all projects have different viewpoints involved (from programmer to user);
- testing at different integration and viewpoint levels mitigates different risks; and
- testers need some basis on which to recommend "test results verify/validate correctly, no further action" or "something's wrong, fix/enhance software and/or its spec".

2.2 W-model

The W-model is even better than the V-model, because all empirical evidence seems to indicate that reviews are more cost-efficient than dynamic testing in finding faults. You may not have the time, personnel or desire to do full inspections of all products, but the W-model shows you the opportunity, and the compromises you are making by not doing reviews. By the way, the W-model was already in the public domain by 1993[ref 13], so is not new as some seem to be claiming.

2.3 Testware development lifecycle

A similar principle applies to test specification; unless the tester sits at a keyboard and runs tests directly without any input information at all (the informal extreme), there is a progressive structure to test specification. Even exploratory testing is usually structured: for example, in the session-based approach[ref 14] it has time-boxed objectives, and may have documented outputs, e.g. enhanced test data. The key benefit of exploratory testing is that the most productive structuring, in choosing what to do next, comes during execution rather than before. But it still has a small amount of planning and design up front.

So, for test specification in general, there are gradations in the ratio of pre-planned to unplanned work, the format and rigour of the test plan, the level of detail of test procedures or their equivalent, and similar factors. What does not vary is that you do stepwise refinement of the test specification.

2.4 Define and measure test coverage

Test coverage is just for certification exams, safety-critical projects and idealistic naive project managers, right? No, think about it: as a tester, could you ever reply when asked what you have tested, "I don't know"? That would surely be absurd at best, negligent at worst. You must have some measure, even if it is an informal set of notes from an exploratory testing session. If you don't know what you are doing, expressible in some terms, you cannot justify being in your job.

2.5 Distinguish problems from change requests

That old contractual hang-up, only applicable to dinosaur projects using cynical software-house behemoths and huge specs to test against? No, it's more subtle than that, because the distinction is between what you're aiming for in this release and what you're aiming for in the next release. You may have the most agile micro-timeboxed approach on the planet, but if one programmer implements some refactoring now and another programmer leaves it until the next release, there'll be a problem. There must be a shared view of what the current spec is, even if it's only in everyone's heads.

2.6 Use both severity and priority

It should never be enough to record only one evaluation degree against an incident / anomaly during testing. The impact on the testing schedule needs to be assessed separately from the impact on the business / user organisation if the anomaly were not to be fixed before go-live. The two could be completely opposite in magnitude. The closer we are to go-live, the more relatively important is the business impact, (a) for the obvious reason that the imagined business impact will soon be real, and (b) because testing progressively mitigates risk so the later we are, the less testing is blocked.

2.7 Distinguish retesting from regression testing

Here's another point which may at first sight look over-fussy, but surely any tester who does not do this cannot be in control. To show that an anomaly has been fixed, the same conditions must be applied as originally exposed it. Any old regression test will not do, because it may misleadingly suggest the anomaly has gone (the bug may be remain hidden under those conditions).

2.8 Use independent system and acceptance testers

Suppose an individual develops and tests a simple website then sends it live. No separate system or acceptance testing here? Not really so. Most people would ask a few friends and colleagues to look it over before publicising the url. Could it ever be a good thing *not* to do that? The equivalent of system testing could be the developer taking a deep breath, switching his/her demeanour out of development mode for a while, and testing the system as a whole. This is about the most extreme case of informality; in most projects however, the people who pay for the software have some say in when they are willing to make that payment, and their decision is therefore independent of the developers. Sometimes there is no explicit acceptance testing, but it is still the users who advise the payers whether the system is acceptable or not. OK, perhaps I should have worded it a little more loosely, something like "use appropriate independence at higher testing levels".

2.9 Use appropriate techniques

This is rather broad, isn't it? Well yes, but it (a) focuses thinking that techniques are there to be used rather than merely quoted generically and lazily in the testing strategy (never to be referred to again!), and (b) acknowledges that we really should consider different techniques in different contexts and at different levels. Much of the literature states or implies that

the same set of techniques are applicable to all levels of testing, but I don't believe it's that simple. We should be clear when we're using black-box techniques, and when glass-box techniques are appropriate, and which specific ones, and why.

2.10 Decide process targets and improve over time

Single projects don't need process improvement? Hmm. They may think they don't. Or corporate management may (strangely) fail to connect projects together and pass on historic data. But if asked over a beer to discuss lessons learned, would you ever say "no comment, process improvement is not appropriate for me nor for you"?

And what about after go-live: maintenance testing doesn't use much of all this stuff? Yes it does; it's just that the risks change, with the emphasis much more on regression testing, and the other principles need interpreting in a special way. For example, we still seek a specification against which to test, but we may now have two of them: the original spec, and a separate spec of the change.

3. What is involved in building this bridge?

3.1 How deep is the schism?

It does seem to me that the views of Best Practice teachers are widening, though some are still indignantly opposed to anything which they see as empty posturing, seductive trendiness or merely statements of the "obvious". Conversely, the things which the Context-Driven school seems to dislike most are these (in no particular sequence):

- large volumes of documentation;
- unthinking adherence to bureaucratic rules;
- excessive attention to quality and process maturity concepts of arguably unproven value;
- promotion of standardisation over initiative, skill and judgement;
- potential disenfranchisement of testers who have not been through the "official" qualification courses and examinations, yet may be superb at what they do;
- undesired legal liability implications of certification; and
- statements dismissing their own carefully-considered approaches as sloppy or just plain wrong.

But need best practice really be all of these things? The vehement part of the debate centres on ethics: how much is each side promoting its views just to enhance the careers and financial incomes of the protagonists, whether it be selling expert consultancy or off-the-shelf training courses? Or is one side more “correct” than the other?

3.2 How strong are the bridge supports?

How well thought-through are my alleged always-good practices? Well, they aren't guaranteed, but they have survived some scrutiny from myself and a few others over the last few years. I'm happy to amend them over time, however, as I learn. There are things deliberately omitted because they're statements of the obvious (eg "get good test environments") or because I can't see any clear message. A good example of the latter is whether coding or testing should come first. I don't currently quote an "always-good" principle for this, because early test preparation is already included in the W model (and even some versions of the V-model). However, the arguments for not only writing acceptance, system and integration tests first, but extending this principle to unit testing – and including automated execution in parallel with coding – have reached a level of some sophistication. In Test-Driven Development[ref 15], Kent Beck recommends first of all running a test which doesn't work (using "red bar" patterns), then fixing the test and/or the code ("green bar") by quick and dirty means, then refactoring to a "finished" (for now) version. I think this means glass-box techniques don't then have their traditional meaning; the coder may be looking out of the box rather than into it.

Yet I can see situations where code-first would still look better to some. As 100% testing isn't possible, an extremist view of "write the tests first" might arguably never deliver any software! More seriously, a simple evolutionary development might make more progress if the users (on-site throughout coding, of course!) could see and get modified a nearly-working system *before* a complete automated test suite had been successfully run (justify tool, select tool, buy tool, configure tool etc). They may have to confront the prohibition of *any* manual tests (as is suggested by some eXtreme Programming advocates[ref 16]). In any case, the agile methods which emphasise comprehensive tests at or before code-writing have to say what those tests are based on. If there are no detailed written specs, then the tests become the de facto spec and need changing as the real or virtual actual spec changes!

But just going through the above discussion suggests that these might be valid points to add to the list. Other improvements recently suggested[ref 17] to my claimed always-good practices are to:

- extend the quality targets objective to include balancing with time, scope and cost;
- extend assessment of test coverage changes to include reassessing risk (from implicit to explicit);

- broaden the point about retesting and regression testing to say something like "assess consequence of each fix"; and
- map the always-good practices as a whole to the excellence criteria of the European Foundation for Quality Management: the feedback loop in my Diagram E (Section 5.2) already has a rough resemblance to that in the EFQM Excellence Model® (which has five enabling criteria leading to four result criteria, then innovation and learning[ref 18]).

3.3 How can pragmatism enrich a framework rather than replace it?

In this paper I use my framework of always-good practices as one input into a scheme of tables and diagrams, plus associated "thinking tools". This scheme can be used (in full or in selected parts) to either tune existing methodologies to be appropriate to the current project context, or to derive an appropriate methodology for a new project or department. The thinking tools are from Goldratt's Thory of Constraints, a method which is intended for process improvement[ref 19] but which I believe can be adapted to position oneself at the appropriate position on a continuum of informal-to-formal. I claim this to be an innovative application of Goldratt's theory, though my ideas were originally inspired by an application of Goldratt to measuring the effectiveness and efficiency of the test (and development) processes[ref 20]. Specifically, I build on a recent direct use of Goldratt's Conflict Resolution diagram in dispelling "superstitions" about documentation in the software development and testing process[ref 21].

My extensions of the above ideas cover all the conflicts between Context-Driven and Best Practice, not only the one about documentation, and incorporate all the thinking tools, not only one (though the user has discretion to omit some). In other words, the Context-Driven tool-kit gets a more structured, yet still flexible, “user guide”. Goldratt's Theory of Constraints is promoted by Goldratt himself and by other consultancies, with associated proprietary material, but many books and papers have been published and the application of the ideas is very much in the public domain, so they are readily usable.

3.4 What are the dimensions of “formality”?

I mentioned above the idea of a continuum from informal to formal. It isn't a simple continuum because the concept of formality is not an exact one. I consider it to have several dimensions, including (but not necessarily limited to):

- adherence to standards and/or proprietary methods;
- detail;
- amount of documentation;
- "scientific-ness";
- degree of control;
- consistency and repeatability;
- "contracted-ness";
- trainedness and certification of staff; and
- "ceremony", eg degree to which tests need to be witnessed, results audited, and progress reported.

So, what consitutes formal or informal is a different mixture of these dimensions for each of the always-good practices.

4. Who is Goldratt, what is his Theory of Constraints, and why here?

Eliyahu M Goldratt (Eli) is an author and consultant whose introduction of a computer system in 1979 disproved a scheduling myth and hence questioned the usefulness of cost accounting in manufacturing. His first book, "**The Goal**" (1984)[ref 22], was a treatise on factory process improvement, written in the style of a love story / thriller. This explained how thinking logically and consistently about problems gives better understanding of cause-and-effect relationships between actions and results, and leads to some basic principles for success. It sought to make process improvement more scientific, but with a view of "science" limited to an understanding of the way things happen and why (not some search for ultimate truth). The quest was for a minimum set of assumptions that can logically explain observed phenomena. The assumptions are not provable absolutely, and may be unexpectedly disproved, necessitating a better assumption. I can see analogies here with thinking on software testing!

Goldratt attempts to show that science, in this form, does not require exceptional brain power but it does require the courage to face inconsistencies and possibly-invalid preconceptions. This says something about education, and about epistemology, the theory of knowledge. Goldratt attempts to follow Socrates in the delivery of his message. He also claims at the outset that the principles are applicable in companies other than manufacturers, and indeed in personal / family life (though surprisingly little seems to have since been done with these extensions by Goldratt himself or by others so far).

The specific messages of The Goal are that:

- there is only one goal (in the case of manufacturing, it is to make money);
- throughput is the most important parameter in processes leading to that goal;
- a process is constrained by its least-throughput component;

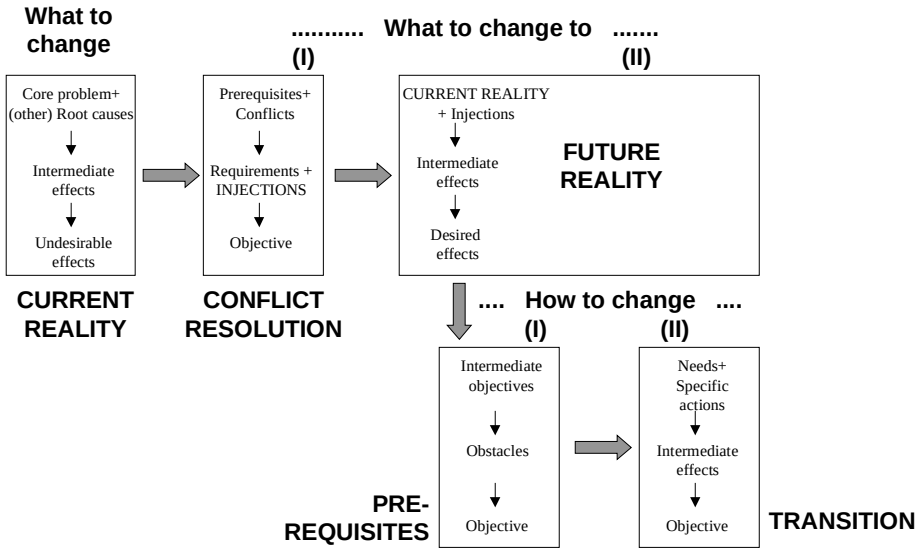
- a good way to manage throughput is to synchronise it (like marchers to a drum-beat, tied together with ropes);
- having improved the least-throughput component, the next step is to find the new "weakest link" and work on that;
- below that level of detail, continual juggling of priorities can be a false economy;
- queues at constraints should be managed as "buffers"; these are especially needed upstream of the tightest constraints, to allow for the unexpected, and especially changes in constraints;
- cause-and-effect relationships are not always as they first seem; and
- the basic questions to ask are "what to change", then "what to change to", and third "how to cause the change".

Goldratt's further books (some also with co-authors) include the following:

- "**The Race**" (1986)[ref 23] clarified the definitions and relationships of throughput, inventory and operating expense, and focussed on synchronisation (drum-beats) as the key to reducing inventory. It elaborated on the drum-buffer-rope theory; for example, slack is not always bad, the rope length is analagous to an inventory buffer, and the weakest marcher should be tied to the pace-setter. The overall message was to increase our standard of living, but purely by improving manufacturing.
- "What is this thing called **Theory of Constraints** and how should it be implemented?" (1990)[ref 24] addressed (a) the problems of continuous process improvement, requiring thinking processes that enable people to invent simple solutions to seemingly complicated situations, and (b) understanding the psychology of individuals and the psychology of organisations enough to overcome obstacles. The Socratic method is not enough, and to cause-effect considerations are added the precursor scientific steps of classification then correlation. Also, before deciding "what to change to", we may need an intermediate step of compromising between apparently conflicting "prerequisites": the evaporating cloud technique. And thirdly, tactics are introduced to induce the appropriate people to cause the required changes. The three basic questions had therefore at this point been expanded into a focussing process.
- "**The Haystack Syndrome**" (1990)[ref 25] distinguished information from data and applied Theory of Constraints (TOC) to decision-making.
- "**It's Not Luck**" (1994)[ref 26] extended TOC to marketing, sales and distribution, and consolidated the diagramming tools.
- "**Critical Chain**" (1997)[ref 27] applied TOC to project management.
- "**Necessary but not Sufficient**" (2000)[ref 28] looked at software provision and applied TOC thinking to: systems cost-justification, balancing feature-richness against supportability, using sophisticated algorithms to optimise throughput, changing processes to get the best out of technology, and considering those changes across the whole supply chain, replacing "push" of product with "pull" from customer demand.

As Goldratt tends to write novels with a business / educational message interwoven, other authors have distilled, extended and exemplified the material to be more directly usable. Particularly significant is Bill Dettmer, who has focussed on the five logical thinking tools for answering the three basic questions. Diagram B below summarises these, paraphrasing Dettmer's interpretation[ref 19].

Diagram B: Goldratt's five logical tools for process improv't



Although there is much in Goldratt's wider material which is applicable to software testing, it is these thinking tools which are the main subject of this paper. Dettmer and other summarisers[eg refs 29, 30] have also written on associated matters such as:

- sufficient causes and necessary conditions (the two thinking processes which the five tools support);
- categories of legitimate reservation (for checking validity of cause-effect logic);
- success measures;
- negative branches;
- spheres of influence and group dynamics;
- relationships with Deming's thinking, eg "profound knowledge"[ref 31]; and
- incorporation of an ideas-generating tool for inventive problem-solving[ref 32].

But these refinements are not essential to my current message. Although Goldratt now devotes his time to the wider goals of enhanced knowledge generation and dissemination, his original principles have arguably helped transform manufacturing from an art to a science. Perhaps they can help software testing along a similar path.

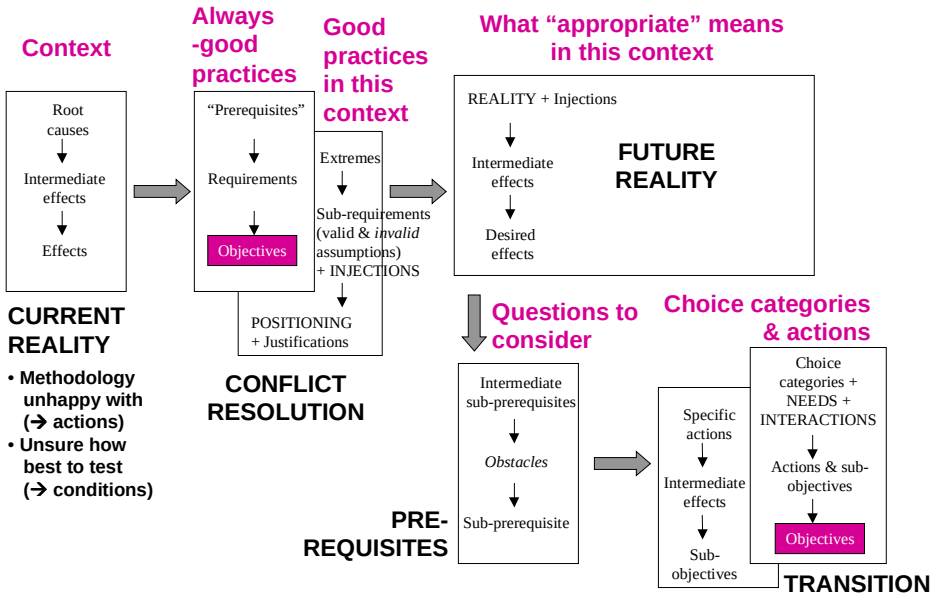
5. How to apply the thinking tools, in part or in full

There is insufficient space in this paper to describe the full Dettmer usage of intermediate effects, injections, obstacles etc. In any case, in Diagram B above I have simplified and adjusted the details very slightly. What I *will* try to explain here is the way I apply the five logical tools and their diagrams to software testing methods:

- "what to change to": Current Reality tree (**context of testing on this project**);
- "preliminaries to what to change to": Conflict Resolution diagram (which I have split into two: **always-good practices**, and **good practices in this context**);
- "what to change to": Future Reality tree (**what *appropriate* means in this context**);
- "preliminaries to how to change": Prerequisites tree (**questions to consider**); and
- "how to change": Transition tree (again I have split this into two: **choices to make**, then **choice categories and actions to implement**).

These and their inter-relationships are summarised in Diagram C below.

Diagram C: Goldratt's five logical tools applied to testing



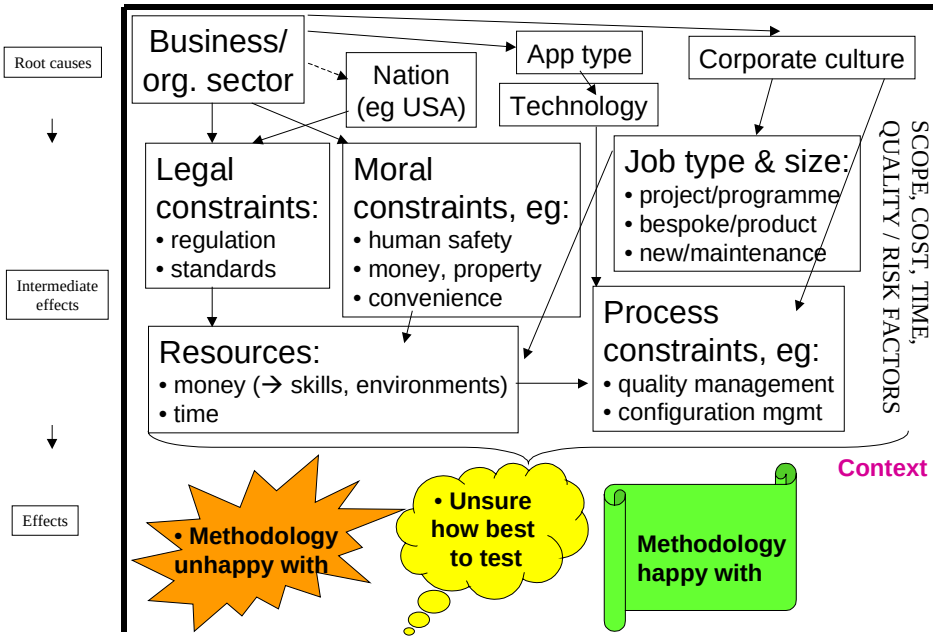
In the sub-sections below I give examples of each of these, in sequence. It is *not* necessary use all the diagrams, but it is logical to start at the Current Reality and continue as far as you feel necessary. I point out in the text where are the most likely stop points. Each diagram is given its Theory of Constraints title, and each sub-section heading explains the use of that diagram in software testing methodology. At the top or side of each diagram is a link to the TOC nomenclature for anyone who is interested, though you don't need to know that to use the method.

5.1 Context

An example of a **Current Reality tree** is shown in Diagram D below. It illustrates how business /organization sector, nation, corporate culture, application type and technology form a dependency chain which tends to determine what legal constraints

are on you, what standards you are meant to use, what quality and configuration management processes surround your testing, etc.

Diagram D: Current Reality

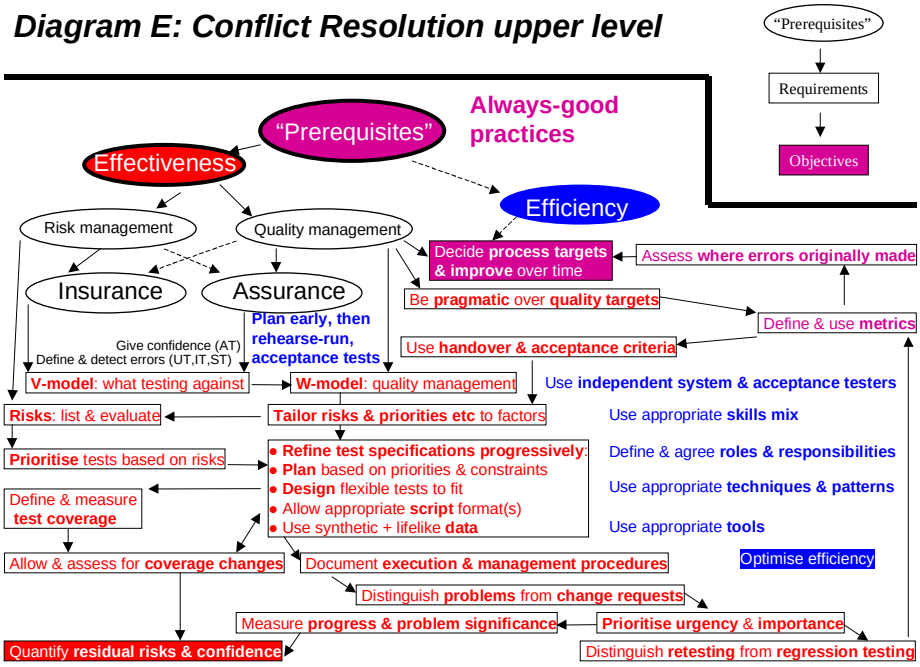


This may merely be a way of justifying the testing methodology you already use and are happy with, or it might reveal that your current methodology is not quite appropriate for your context. Or, it might start the process of replacing a testing methodology that you're unhappy with. Or perhaps you don't have a testing methodology yet, and just want some lightweight advice on how best to test in your current situation.

5.2 Good practices

I have taken the liberty of splitting the Conflict Resolution concept into two. My **first (upper) level Conflict Resolution diagram** is a dependency network of "prerequisites" (effectiveness and efficiency) via "requirements" (eg use an appropriate interpretation of the V-model) to "objectives" (eg quantify residual risks and confidence in project/product). See Diagram E below, and the subsequent explanation.

Diagram E: Conflict Resolution upper level



Implicit in the "prerequisites" ellipse is Goldratt's single goal: in your system it will probably be to help make money, or to help preserve / improve lives. I have then assumed that any sane person would desire both effectiveness and efficiency in pursuit of the goal: do the right things, and avoid wasting time and resources in the process.

Effectiveness and efficiency are then split into what I believe are their elements, and this diagram actually represents the full list of always-good practices from Section 2. But it enriches the list by visualising dependencies between the practices: the ellipses are a kind of top-down analysis of what should drive good testing, and the rectangles and other text are a kind of bottom-up distillation of the literature and my experiences. Effectiveness is shown in red, efficiency in blue, and magenta is where they come together. For those of you reading in black-and-white: the effectiveness requirements are in rectangular boxes, and the objectives are solid boxes. The dependencies between the boxes are roughly time-based: first they follow the testing lifecycle for a project, then they loop back for the next release, or for the next project, or for your next job. Efficiency requirements are not built into the dependency network because they tend to apply throughout.

This diagram in my method is fixed: that is, it claims to embody always-good practices for software testing. It is however a personal view; you could create your own, although this one has stood up well to a little external scrutiny so far.

In the simplest version of this advice, you could stop here. Just use this diagram of always-good practices as a crib-sheet to guide your testing: do these things, in some way, in roughly this sequence, and you should be OK.

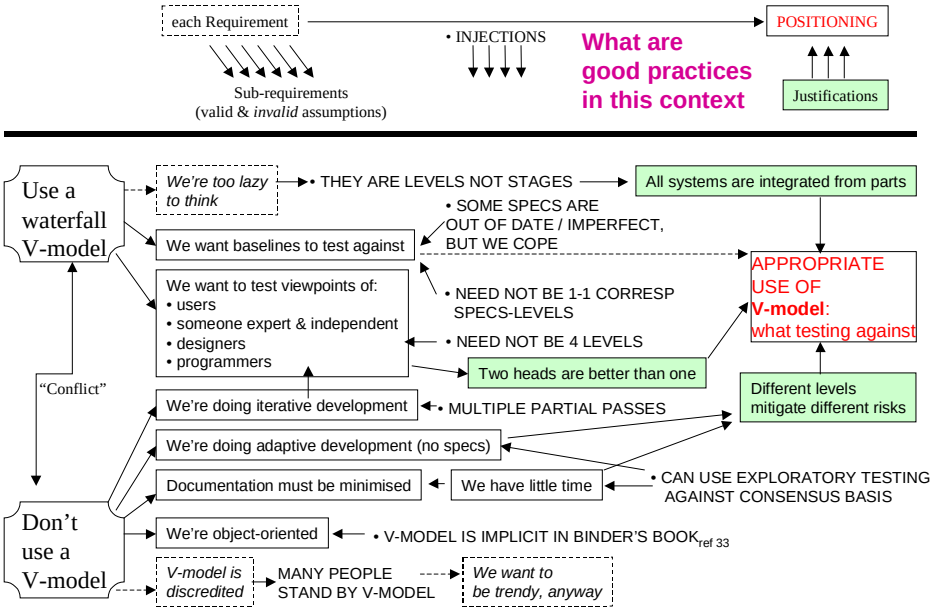
My **lower Conflict Resolution diagram** merges information from the Current Reality tree and the upper Conflict Resolution diagram. It takes each requirement from the "always-good practices" diagram and expands it into its own diagram, based on the current context, to begin to determine where on the formality scale “appropriate” sits for this requirement. There are two ways to do this: either go straight to your current context, or start with as wide a view as you can manage then zoom in on the factors appropriate to you. In this paper I take the wider approach, because:

- obviously I don't know your context, and it varies per reader; and
- it is tempting to accumulate a "complete" picture for all known contexts, as a body of knowledge (or rather, a body of thinking) and to distil out the always-good factors ("sub-objectives" in my TOC syntax).

The full version of this would have 25 lower-level Conflict Resolution diagrams and would need a book rather than a conference paper to document. But the Theory of Constraints says find the weakest link and strengthen that, before moving on to the next-weakest. In practice there will be a few links which jostle for position at the top of the hit-list. My analogy is that in a test process improvement or methodology clarification situation, you will be interested in a few factors in particular, so you may only require one, two or perhaps up to six of these diagrams.

Diagram F below illustrates one selected example: what having "*a V-model*" means on the scale between no V-model and a full, traditional, waterfall V-model.

Diagram F: Conflict Resolution lower level



The purpose of this diagram is to understand our positioning for "appropriate use of a V-model" between the two extremes of formal to informal which are shown in the "demonstrators' placards" on the left of the diagram. Clearly there is a conflict between them, but can we deduce a compromise? The next step in the thinking process is to split the requirement into sub-requirements (the white rectangles, such as "documentation must be minimised"). After you've brainstormed a number of these, you can start to challenge their validity.

Ways to make some "positive" statements sound questionable:	Ways to question directly ("negatively"):	Neutral questions:
• we must always...	• oh, really? who says?	• so what?
• it's absolutely impossible to...	• is that still true?	• why is that?
	• are there no exceptions at all?	

After this treatment, some of the sub-requirements start to look like invalid assumptions (eg the ones in italics on Diagram F above) and the surviving ones start to crystallise into a network of necessary conditions. To complete that network, the Conflict Resolution thinking tool uses the concept of "injections" (shown in upper case on Diagram F above): things which hadn't originally occurred to you but which now, as you've thought more deeply and with more structure, allow you to merge parts of the rationale for extreme one with parts of the rationale for extreme two. In other words, you have the beginnings of a specific compromise, or a "win-win situation". At the end of the chains of necessary conditions appear some justifications for the positioning you are about to decide.

So, in this example, the V-model need not necessarily imply a waterfall lifecycle, or a fixed set of testing levels, or perfect written specifications to test against, or a one-to-one correspondence of test levels to documents. But we can say that we want a V-model positioned as follows:

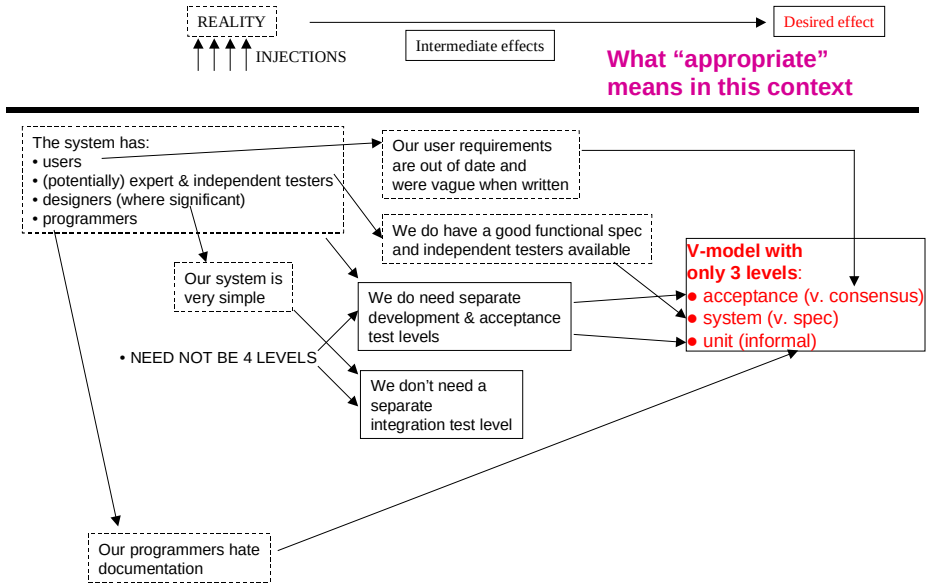
Justifications	**Positioning:** Use a V-model which...
Two heads are better than one	... recognises testing as needing separate people in addition to developers, and at least one distinct testing stage
Need some basis on which to test	... states what specifications you have and how they will be used
All systems are integrated from parts	...tests from the parts, via integration, to the whole
Different test levels mitigate different risks	... looks at the whole set of risks and addresses the most appropriate ones at the most appropriate levels of testing
Need some handover mechanism into live use	... defines where handovers are done, especially acceptance

The requirement remains to use a V-model, but you now know more about why this is good, and the positioning suggests how to go about defining appropriate use of a V-model in your context.

5.3 What “appropriate” means

It is possible to stop here, but if you want more detailed analysis you can go on to use the **Future Reality tree** to isolate each variable in the decision and determine exactly what reality factors and injections lead you to the most appropriate decision. Diagram G below continues the V-model example as illustration.

Diagram G: Future Reality



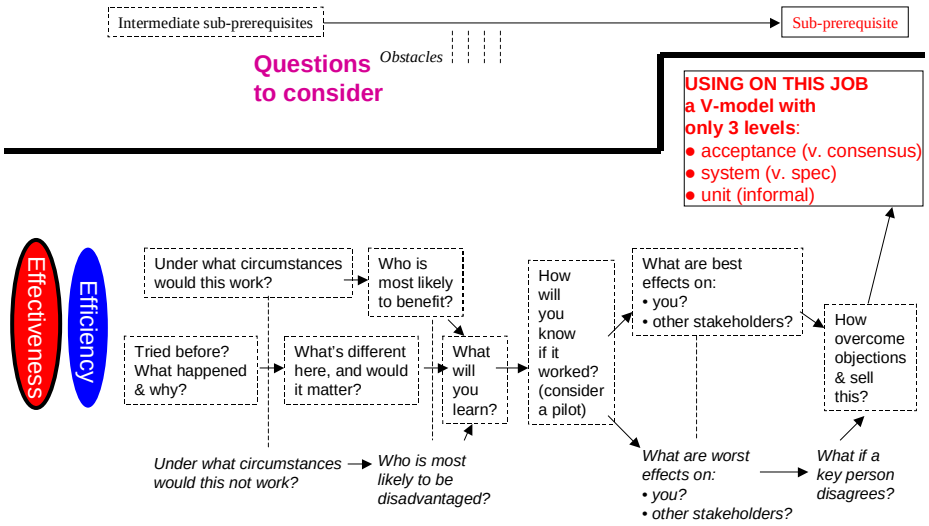
We have therefore used "realities" from a particular context (facts from the Current Reality tree) and injections from the Conflict Resolution diagram (eg a V-model need not have four levels) to determine some intermediate effects (eg we do need separate acceptance testing, we don't need separate integration testing) and hence a desired effect (eg have three levels of testing on this project).

When I say "isolate each variable", I assume that all the variables of interest can be accommodated on one diagram. It is convenient to have a one-to-one relationship between each lower-level Conflict Resolution diagram (our positioning for this requirement) and the Future Reality tree which translates that positioning into a specific "desired effect".

5.4 Questions to consider

Again, it is possible to stop at this point, because you now know exactly what you want to do. But the thinking tools allow for further sophistication, if desired. The **Prerequisite tree** can be used to assess possible obstacles to your provisional decisions. As an example, Diagram H below expands into flowchart form the eleven questions suggested by the best-known book on context-driven testing[ref 2] to evaluate each "lesson" in terms of its appropriateness to your situation.

Diagram H: Prerequisites



In terms of "Goldratt" syntax, the desired effect from the Future Reality tree has become a specific sub-prerequisite to our overall prerequisites of effectiveness and efficiency. This diagram helps us identify intermediate steps to get to this, and to analyse potential obstacles.

5.5 Choices, choice categories and actions

The final steps are also optional, but are interesting because they expose one of the main drivers for this method: a central principle of the Theory of Constraints (see below). Whether you are in a process improvement situation or you want to formalise a new methodology for your testing, you can use the Transition tree diagrams to:

- integrate up your requirements into specific choices for appropriate testing (having considered and mitigated the obstacles from the Prerequisites tree); and
- summarise those choices into convenient categories and plan any necessary actions to get your method implemented or improved.

This categorisation may be linked to proprietary process-based or symptom-based improvement methods if desired. And this is the point about the central principle. You may be asking: "why not just *start* with a proprietary process improvement method?". Well, there are several possible answers, which I will discuss in the next sub-section. But the main one is that according to Goldratt, you don't know what you should improve until you've done some analysis. This paper is about such analysis.

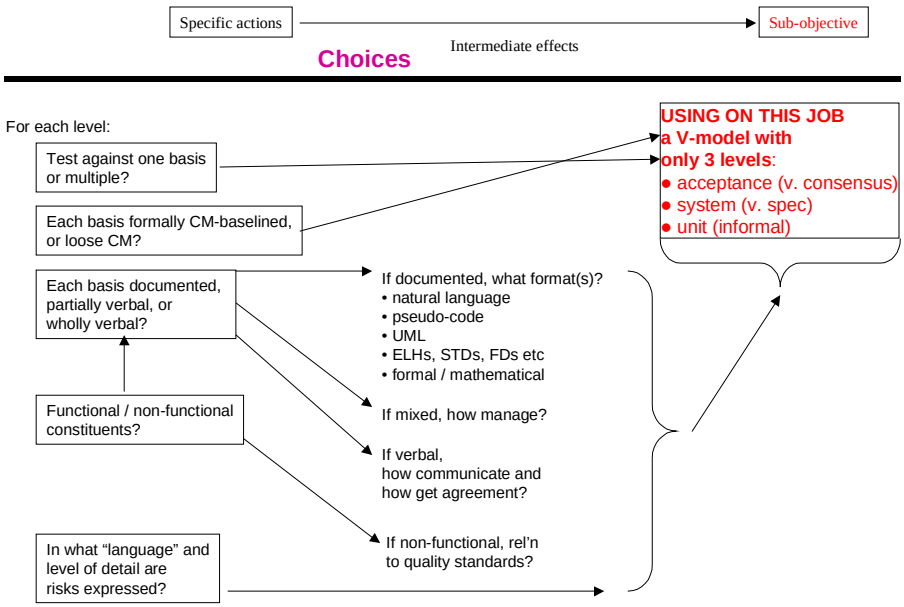
Again, I have seen fit to split a thinking tool into two:

- first, a lower level of the Transition tree which lets us plot specific actions to achieve each sub-objective – or, in testing terms, the detailed choices we have made for what constitutes "appropriate" in our context; and

- second, an upper level which in effect brings us back to the top level at which we started (a kind of meta-V-model!) but now crystallises all that we were forced to do, thought we might do, positioned ourselves to do, made specific, troubleshot and refined, into choice categories so that we can see exactly how we are meeting our original objectives.

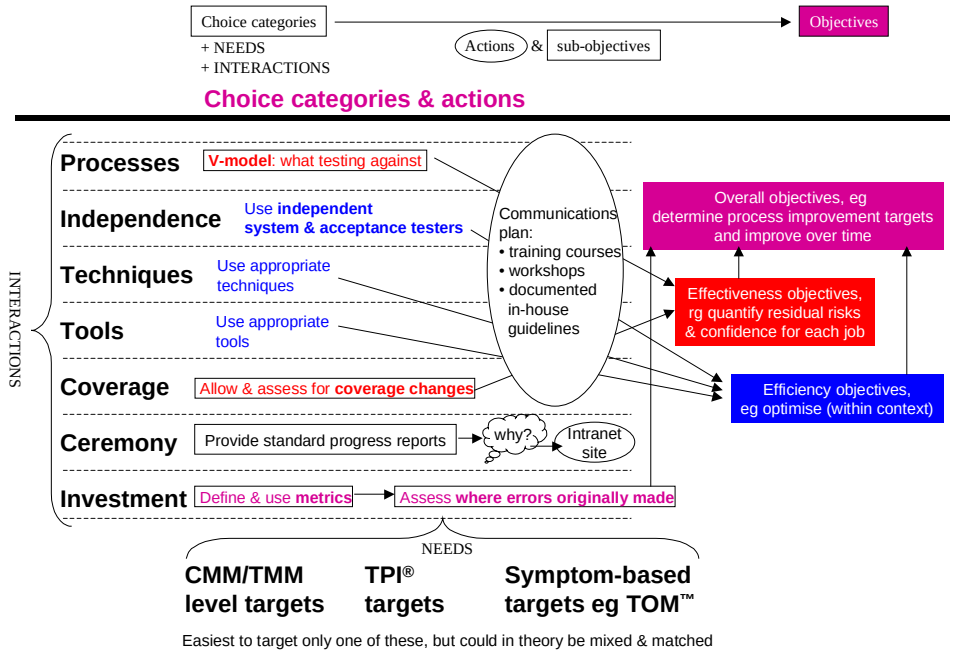
Diagram I below shows an example of the **lower-level Transition tree**, in which we look in detail at how we will determine the basis for what testing against.

Diagram I: Transition lower level



Moving on to the **upper level Transition tree**, Diagram J below shows an example of how the various choices we have made can be categorised, if desired, into headings such as processes, independence, techniques, tools, coverage, ceremony and investment.

Diagram J: Transition upper level



This list of categories is not fixed; it just seemed to sum up for me the main elements in a methodology. You may want to fit your testing method to a proprietary process improvement methodology, and in the next section I will look at why and how this might be done.

5.6 So why not just start with a process improvement method?

That's a good question: one of the best one could possibly ask about this material. Possible answers include:

- "we're not ready for process improvement, we just want a structure to apply context-driven";
- "there is some controversy in the field, and we want a way of choosing a process improvement methodology"; or
- "we don't seem to fit the process improvement methods we've seen".

But the most relevant answer for this paper is, as I have already mentioned, that it requires some analysis before you know where best to apply your efforts, either in process improvement or deciding on a method for a particular context.

Public-domain and proprietary test process improvement methodologies seem to fit into two main groups:

- those based on the Capability Maturity Model in some way, eg TMM[ref 34]; and
- those which ask about your symptoms (what you and/or your customers think is wrong with your testing) and then suggest possible remedies, eg TOM™[ref 9].

It is arguable that TPI®[ref 7] combines perceived advantages from both groups, in that it:

- is based around a test maturity matrix with built-in dependencies between different key areas as maturity overall increases; but
- it works in steps (roughly, determine target and area of consideration, determine current situation, determine required situation, then implement changes).

However, there is disagreement in the industry over which approach is best. And none of these methodologies seem to me to embody Goldratt-type thinking; they all make some assumptions about what sequence improvements should be done in, or what symptoms indicate what remedies. The framework and tools in this paper are more general in two ways:

- they will fit a context to a formal-informal scale for an individual project, not just improve processes over time; and
- they let you make your own connections between cause and effect.

But I am not recommending we ignore that useful (and still-refining) body of knowledge, hence this step of building in the test process improvement method(s) of your choice. There are several ways this could be done, for example:

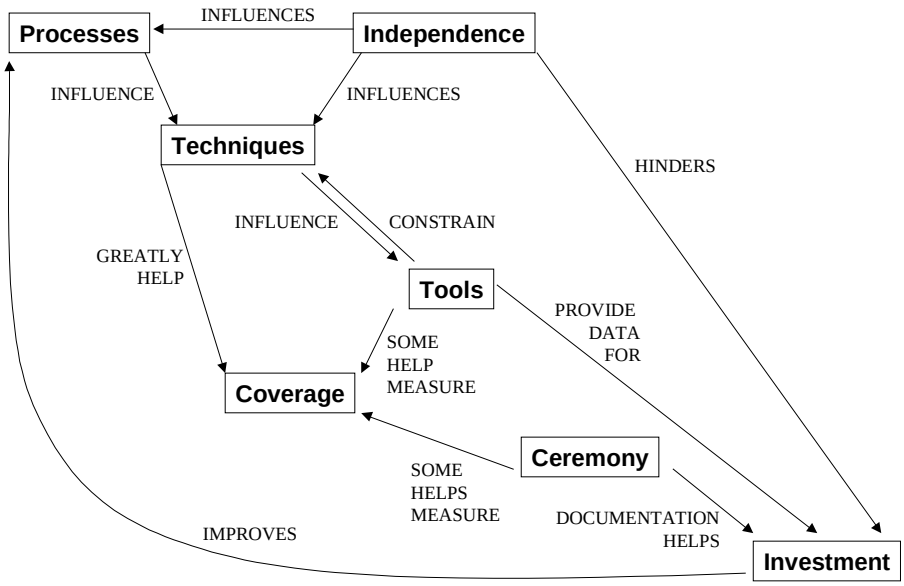
- read several methods and pick out the most apparently-useful advice for your situation;
- aim for TMM-type improvement but use the Goldratt thinking tools to focus your efforts; or
- take a symptom-based approach but use this Goldratt framework to structure your thoughts and sequence the remedies.

And this leads to another reason for taking your own view on causes and effects; there are interactions between the choice categories. In the case of TPI®, the basic categories are[ref 7]:

Cornerstones	**Lifecycle** of test activities related to the development cycle	Good **organization**	The right **infrastructure and tools**	Usable **techniques**
Interactions between cornerstones	"...within each test process some degree of attention must be given to each cornerstone. For a balanced test process, the substantiation of the cornerstones should be in balance"			
Key areas	• **test strategy** • **lifecycle model** • **moment of involvement**	• **commitment and motivation** • **test functions and training** • **scope of methodology** • **communication** • **reporting** • **defect management** • **testware management** • **test process management**	• **test tools** • **test environment** • **office environment**	• **estimating and planning** • **test specification techniques** • **static test techniques** • **metrics**
(all cornerstones)	• **evaluation** • **low-level testing**			
Interactions between key areas	Specific dependencies between specific key areas (crossing between cornerstones) between levels A, B, C & D (where each level attains a specific scale number dependent on key area). Scale numbers: • 1 to 5 are primarily aimed at **control** of the test process • 6 to 10 focus more on the **efficiency** of the test process • 11 to 13 are characterised by increasing **optimization** of the test process.			

For the choice categories I am using in this paper, more general interactions such as the following may be mapped out (see Diagram K below).

Diagram K: Interactions between choice categories

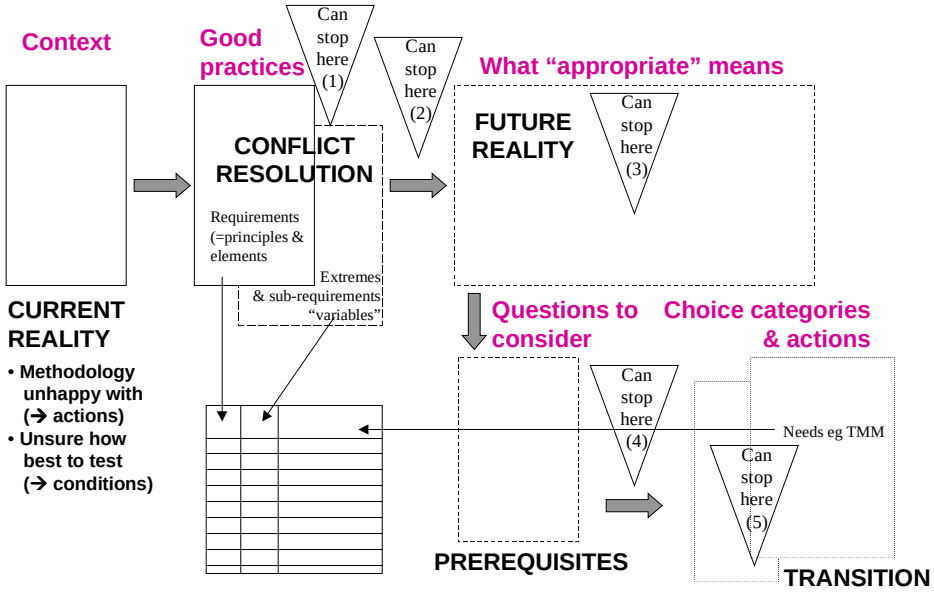


But these categories are my own choice on the particular occasion I wrote this paper; you may choose your own.

5.7 Can apply fully, partially or very simply

To reiterate: there are several ways to apply this series of thinking tools. Diagram L below summarises the most likely points at which users of the framework may feel they have done enough.

Diagram L: Use of tables, and summary of framework



What the above diagram also shows is that one or more tables may be used as:

- a thought-gathering process before attempting diagrams; or
- (for people who don't like diagrams) a possible substitute.

Not to use diagrams is really to miss the point of the method, but it would still be possible to derive some benefit by using a table to list the requirements you have for a testing method and to identify the variables for each requirement, on the formal-informal continuum.

Both of the above possible uses of tables are illustrated in the next sub-section.

5.8 Use of tables

The following table starts the thought-gathering process by collecting requirements and variables. The third column could be added to keep track of where, if anywhere, those variables feature in the diagrams you use.

Requirements	Variables	Diagrams
Use a V-model: what testing against	Documented-ness culture	Current Reality tree
	Degree to which you want a baseline	Conflict resolution diagram - lower level
	Written-ness of baseline	
	SDLC: iterative-ness	
	SDLC: integration and increment methods	
	Up-to-dateness of baseline	
	Configuration management on baseline	Transition tree - lower level
	(etc)	
W-model	Existence, sophistication and power of Quality department separate from Testing	Current Reality tree
	Definitions and relative emphases of Quality Assurance and Quality Control	
	How early system specs are available	Conflict resolution diagram- lower level
	Time and resources available for improving system specs on demand	
	Availability of users for validation tasks	
	Sophistication of regression test planning	Transition tree - lower level
	(etc)	

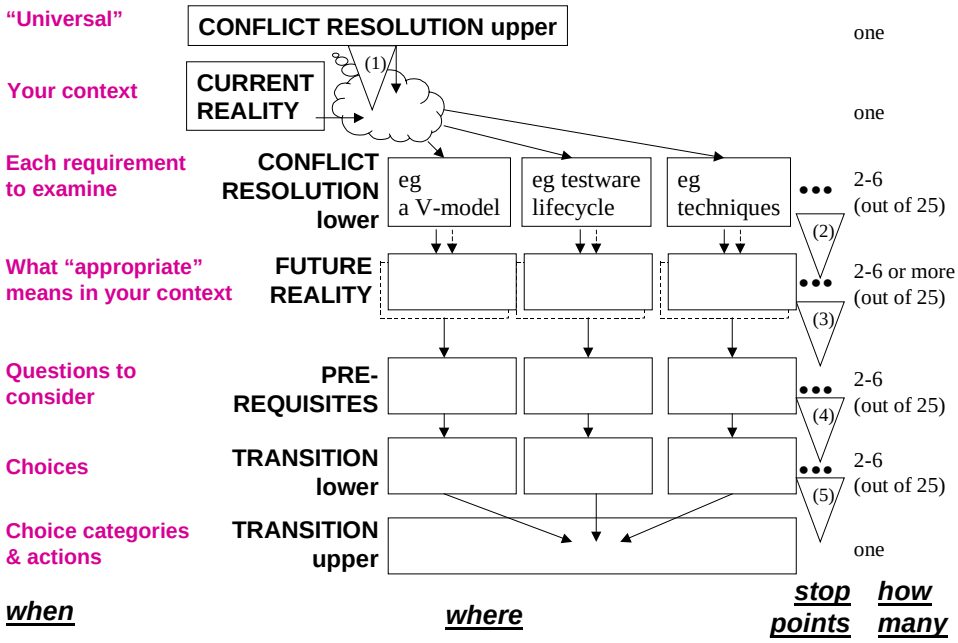
Such a table could be expanded by adding columns such as:

- occurrence in TMM, TOM™ and/or TPI®; and/or
- relevance to your desired choice categories (such as methods, independence, techniques, tools, coverage, ceremony, investment).

5.9 Summary

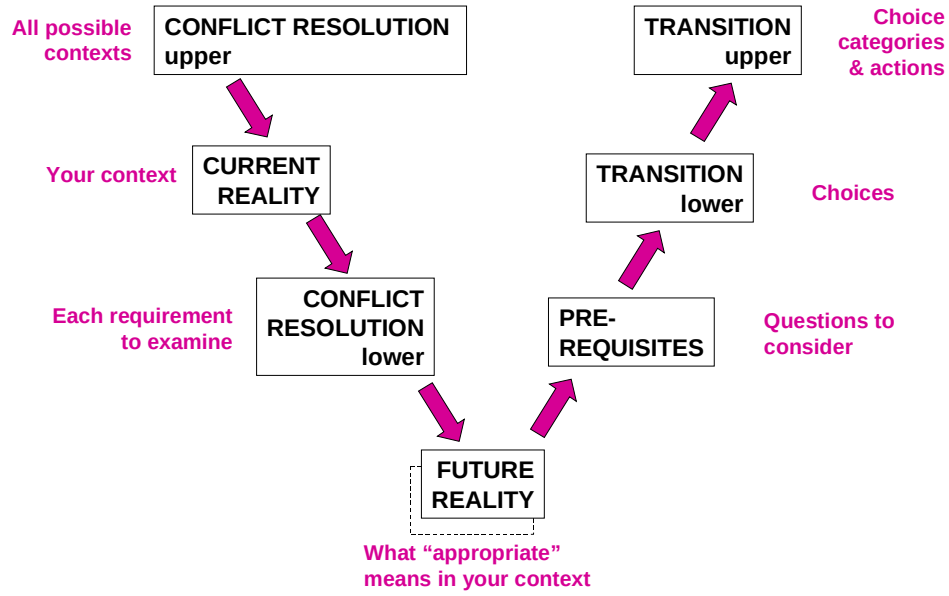
Taking a step back: the entire framework is summarised in Diagram M below.

Diagram M: How many diagrams, where and when



I mentioned earlier that the framework itself resembles a meta-V-model; this is illustrated in Diagram N below.

Diagram N: The framework is a "meta-V-model"



6. Conclusions

There is definitely a difference between "Best Practices" and the "Context-Driven" approach, but"...:

- the differences are arguably not as great as some Context-driven or Best Practice enthusiasts state (or wish?);
- actually, there are several particular differences of opinion: importance of specific standards from standards bodies; best way to teach new testers; importance of standardised terminology; necessity for reproducible tests; and even the ethics of each approach;
- perhaps it is partly an American-European cultural difference: meritocratic pragmatism versus institutional formalism;
- this is not the whole story however; there are European CD-ers and American BP-ers;
- the context-driven school is more prevalent in the Commercial Off-The-Shelf, web and other competitive commercial software markets, but...
- context-driven does not exclude safety-critical or other high-integrity applications; those are contexts, and it cautions against applying the disciplines of those to *all* projects; so...
- each side *attempts* to be all-inclusive, but from different prevalent backgrounds, so the other side still sees the other as exclusive;
- the position is complicated by some leading-edge thinkers actually sounding very dogmatic, for example that there should be *no* manual tests (anywhere in XP);
- it seems that now, just as the CD-ers have battled hitherto not to be misunderstood, the BP-ers are modifying, or at least better explaining, their own position, e.g. by building some Context-Driven material into best-practice-style training courses, even as far as allowing deviations from (or at least interpretation of) IEEE 829![ref 35;]
- in itself it should become part of Best Practice to determine, and heed appropriately, the context of the customer, the project and its business and technical domains – some people claim it is already there;
- it remains to be seen how this possibility will develop, within and outside the standards community, over the coming years; but in the meantime I offer this paper as a way of bridging between the two extremes by providing a framework which integrates the two; and
- in science and in philosophy, theories tend to become more general and inclusive over time, so the omens are good (the same does not seem to apply to the arts or some religions, so there may be a message there!).

7. Lessons learned, and way forward

Reactions (paraphrased here) received from various people to early drafts of this material included:

- "but this looks very like our "Best Practice" training course syllabus";
- "this may be too controversial";
- "I don't understand what you're on about – and in any case, why bother to analyse what are merely statements of the obvious?"; and
- "we should be able to come up with something more specific / detailed / quantitative / prescriptive than this".

Hmm... no common theme there. So the lessons learned for me so far have been that:

- some people seem to prefer disagreement to agreement (and that may not be a bad thing); and
- what some people regard as progress, others may regard as merely obvious and yet others may not comprehend at all.

Is this a revision or rival to the risk-based approach in which I have taken part?[ref 36] Not really, because that message is only a part of the overall testing debate (and plays a prominent role in the always-good practices here). But over time I am happy to have traditional views challenged, and this paper goes wider than the risk-based elements, by considering everything which has to be done, even the "routine".

It is possible that Agile methods devotees will get me to moderate some of my always-good practices even more. Certainly, much of Goldratt's thinking is inextricably linked with lean manufacturing, and for software that means Agile methods. The arguments are increasingly persuasive[ref 37] yet can still be challenged[ref 38] without appearing too reactionary. But the thinking tools have wider applicability, and even Agile evangelists admit that some organisations are unable to embrace the degree of change required. I want to be general and all-inclusive.

The concepts in this paper have been input to discussions by members of the testing community in the UK[ref 17], under the working title “Appropriate Testing”. But this is only one of the approaches under consideration, and alternatives (or at least refinements) may emerge from that forum.

One area for my further research is the extent to which other parts of Goldratt's thinking (especially the core constraints ideas and the concept of throughput) are really applicable to software development and testing. Most of the applications have been in manufacturing, though the application to project management appears to have some credibility. Successful applications are reported also in health care, finance and construction[ref37]. There have been some attempts to relate it to software development, but generally so far tentative or with caveats.

Returning to context-driven methods: on a much wider scale than software testing, the concept of a "Methodology per Project"[ref 39] has been proposed by Alistair Cockburn and refined as part of the Agile methods movement. This does not use the same principles as suggested in this paper, though there are some similarities in the matters considered, for example:

- how many methodologies? is one "better" than another? how to know which elements to adopt?
- main variables are said to be staff size and system criticality;
- other variables include scope of concerns, standards, people and their beliefs;
- some basic principles are applied, e.g. a more populous project necessitates a larger methodology; a critical project requires more publicly visible correctness, a greater methodology weight requires greatly more cost;
- methodologies are distilled to basic elements (in Cockburn's case roles, skills, activities, techniques, tools, teams, deliverables, standards, quality measures and project values); and
- people vary, therefore rigid and highly-disciplined methodologies do not work in reality as well as intended.

So, this paper is not a simple cookbook to be followed faithfully. It is a new application of a set of thinking tools which can be used in several ways, from the simple to the comprehensive. In the words of Eli Goldratt: "Our textbooks should not present us with a series of end results but rather a plot that enables the reader to go through the deduction process himself [or herself]".

References

Note: after some thought, I decided to include many references, for example so that the interested reader may get the full picture. But for the particularly-busy or less-interested reader, I have highlighted the most pertinent references in bold.

1. www.contextdriventesting.com
2. Kaner, C., J. Bach & B. Pettichord (2002), *Lessons learned in software testing: a Context-Driven approach*, Wiley
3. Myers, G.J. (1979), *The art of software testing*, Wiley
4. Hetzel, W.C. (1988), *The complete guide to software testing*, 2nd edition Wiley
5. Beizer, B. (1990), Software testing techniques, 2nd edition Thomson
6. Kit, E. (1995), *Software testing in the real world*, ACM / Addison-Wesley
7. Pol, M. & T. Koomen (1999) *Test Process Improvement*, Addison-Wesley
8. Kaner, C., J. Falk & H.Q. Nguyen (1999), *Testing Computer Software*, Second edition Wiley
9. www.evolutif.co.uk/cgi-bin/tomoverview.asp
10. Herzlich, P. “Graduating ceremony”, www.evolutif.co.uk/default.asp?page=articles\ceremony.html
11. Reid, S. "Testing and standards" in *The Testing Practitioner* (Van Veenendaal et al. 2002, Uitgeverij Tutein Nolthenius)
12. Mellis, W. “Process and product orientation in software development and their effect on software quality management” in *Software Quality* (editors Wieczorek & Meyerhoff 2001, Springer)

13. Herzlich, P. "The politics of testing" in *EuroSTAR 1993* (proceedings, BCS / SQE)
14. Lyndsay, J. "Adventures in session-based testing" in *STAREast 2002* (proceedings, SQE)
15. Beck, K. (2003), *Test-Driven Development - by example*, Addison Wesley
16. Crispin, L. & T. House (2003), *Testing Extreme Programming*, Addison Wesley
17. Software Testing Retreat (informal UK conference, personal communications, 2003)
18. EFQM (2003) *Introducing excellence, European Foundation for Quality Management*, www.efqm.org
19. Dettmer, H.W. (1997), *Goldratt's theory of constraints: a systems approach to continuous improvement*, American Society for Quality
20. Pas, J. "Software testing metrics" in *EuroSTAR 1998* (proceedings, aimware)
21. Daich, G. "Software documentation superstitions" in *STAR East 2002* (proceedings, SQE)
22. Goldratt, E.M. (1984; 2nd ed. 1992 with J. Cox), *The Goal*, North River Press
23. Goldratt, E.M. & R.E. Fox (1986), *The Race*, North River Press
24. Goldratt, E.M. (1990), *What is this thing called Theory of Constraints and how should it be implemented?*, North River Press
25. Goldratt, E.M. (1990), *The Haystack Syndrome: sifting information out of the data ocean*, North River Press
26. Goldratt, E.M. (1994), *It's Not Luck*, North River Press
27. Goldratt, E.M. (1997), *Critical Chain*, North River Press
28. Goldratt, E.M., with E. Schragenheim & C.A. Ptak (2000), *Necessary but not sufficient*, North River Press
29. Dettmer, H.W. (1998) *Breaking the constraints to world-class performance*, American Society for Quality
30. Scheinkopf, L.J. (1999) *Thinking for a change: putting the TOC thinking processes to use*, St. Lucie / APICS
31. Lepore, D. & O. Cohen (2000) *Deming & Goldratt: the TOC & the system of profound knowledge*, North River Press
32. Domb, E. & H.W. Dettmer, (1999), *Breakthrough innovation in conflict resolution: marrying TRIZ and the Thinking Process*, www.goalsys.com
33. Binder, R.V. (2000), *Testing Object-Oriented systems: models, patterns and tools*, Addison Wesley
34. Burnstein, I., T. Suwannasart & C.R. Carlson, "Developing a Testing Maturity Model", www.iit.edu
35. Evans, I. "Testing fundamentals" in *The Testing Practitioner* (Van Veenendaal et al. 2002, Uitgeverij Tutein Nolthenius)
36. Gerrard, P. & N. Thompson (2002), *Risk-based e-business testing*, Artech House
37. Poppendieck, M. (2003) Lean Development: an Agile toolkit, Addison Wesley planned publication; advance extracts www.poppendieck.com
38. McBreen, P. (2003), *Questioning Extreme Programming*, Addison-Wesley
39. Cockburn, A. "A methodology per project", www.crystalmethodologies.org

v1.0 Neil Thompson 02 Mar 2003