This article is a short introduction to the forthcoming book 'TRIZ For Software Engineers'.

TRIZ For Software?

Darrell Mann
Systematic Innovation
Phone: +44 7980 864028
darrell.mann@systematic-innovation.com

Introduction

Can TRIZ be applied to the design and creation of software appears to be a question being asked by a growing number of companies. This article sets out to explore the field and make suggestions as to what a 'TRIZ for Software' toolkit might look like. Given that this author is set to publish a book on the subject before the end of the year, it seems likely to assume that the suggestion of the article will be, yes, TRIZ can be applied to software. Given some of the past discussions of TRIZ and software, however, it seems that the positive answer is not such an obvious one. Ikovenko spoke around the subject at the TRIZ Centrum conference in 2003 (Reference 1), and basically concluded that since software development was still at an early stage in its evolution ('software development is an art rather than a science') that it was not yet likely to be amenable to treatment by TRIZ. In his usual forthright way, Karasik (Reference 2) was more vehement in his dismissal of the possibilities of applying TRIZ to software problems ('being a computer scientist and a software engineer, I can assure Mr. Retseptor that 40 principles are not applicable to software engineering'). The spur to Karasik's argument was the earlier published list of 40 Inventive Principles for Software by Rea (Reference 3). Rea's analysis of the situation was that the 40 Inventive Principles – or most of them at least – could be observed in the design of software.

Observing the apparent use of Inventive Principles in already solved software problems, however, is markedly different from actually *using* them to solve a problem that has not yet been solved. As of today in fact, there are still no satisfactory published instances of a real software problem being successfully solved using either the Principles or TRIZ. So where does that leave us?

We have been teaching a 'TRIZ for Software' workshop to in-house clients willing to experiment with the possibilities of TRIZ for over 18 months now. Based on the experiences of these workshops and the ongoing extensive analysis of patents conducted during the programme of research to update of TRIZ, it is our firm belief that TRIZ *has* got something of significance to offer to software engineers. What we can already see, however, is that the bias and focus of the various TRIZ tools is markedly different in the software context than it is in other fields of application.

In order to explore that bias, however, we first need to explore the context and bounds of what is and what is not a 'software' problem. A good way to begin this exploration, then, is to examine the TRIZ (modified) Law Of System Completeness (Reference 4) and see how this might be applied to software systems:

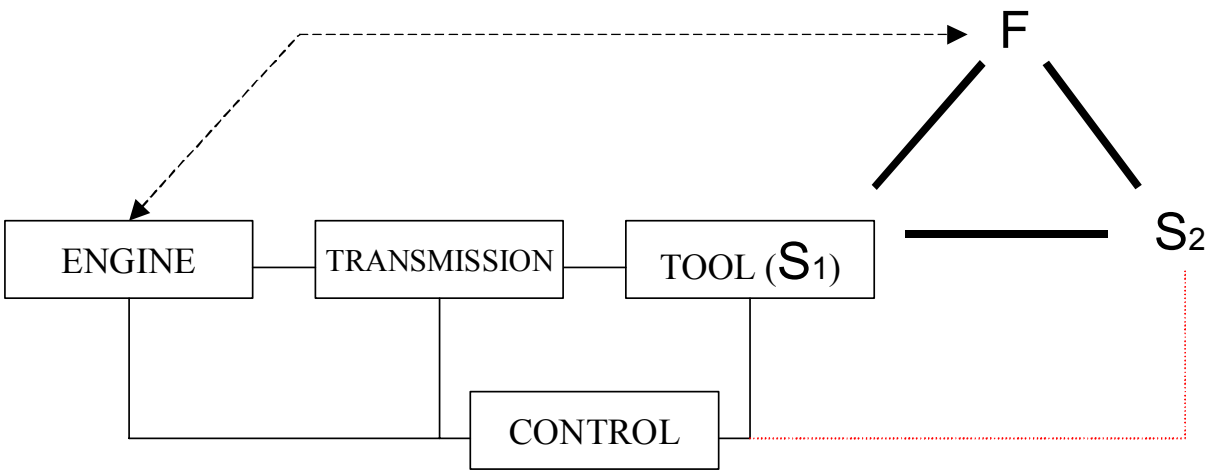

Figure 1: Law Of System Completeness (from Reference 4)

The first question that emerges after thinking about this model is 'is it applicable to software?' The answer is yes. As per the Stafford Beer Viable System Model (Reference 5), we can recursively apply the model at several hierarchical levels. At the software subroutine level, for example, we might think of the 'engine' as the algorithms contained in the routine; the transmission as the subroutine call; the tool as the outputs calculated by the routine and the control, the various lines of code within the software that make sure things happen in the right sequence and with the desired logic. The 'field' is a slightly bigger stretch, but if we think about it as a 'communication field' then the logic remains intact. The same analogies can be applied at progressively higher hierarchical levels in which we look at systems from the perspective of operating system, language, etc. At whatever level we look, ultimately the 'system' has to interface with something outside the system ('S2' in this case). This thing outside the system may be another routine or another program or the user.

The main reason for starting with this (admittedly abstract) model is that we soon notice something different about software relative to conventional technical systems. In a technical or indeed business system it is frequently the case that when we start asking ourselves whether the relationships between the different parts of a system are 'effective', we find ourselves answering that, no they are not. Insufficient, excessive and harmful actions are often to be found in abundance in most practical systems. In software, however, the idea of, say, an insufficient relationship between two subroutines is difficult to imagine. Attempts to construct a function analysis model of the insides of a piece of code and we are likely to end up with a picture in which there are no harmful, excessive or insufficient actions; in programming terms, we get what we write, and it either works or it doesn't; there is little or no middle ground. The only practical time, then, when we are likely to find ourselves identifying functional models containing negative relationships is when we reach the interface with the outside world. One piece of code does not interact 'inadequately' with another; but a piece of code very frequently interacts inadequately with a piece of hardware or with a user.

The main implication of this phenomenon is that the construction of function analysis models is of limited value in defining improvement opportunities (they will, of course, still be very helpful in enabling us to get the intra-block communication logic right in the first place). 'Problems' only start to occur when we examine links between the software and the outside world. This is likely to mean that we are going to be thinking about TRIZ as a problem solving tool at a somewhat different level from the detailed start point of a function analysis model.

Pillars

With this thought in mind, what then happens when we begin to explore some of the other pillars of TRIZ? Figure 2 illustrates the list of seven pillars found in the business version of TRIZ (Reference 6). (Note here that the increase relative to the five this author suggests exist in technical TRIZ emerges due to the increasing connection between TRIZ and other philosophies; we continue to call the method 'TRIZ' as a convenient label, but our meaning is somewhat evolved from what may be considered to be 'classical' TRIZ).

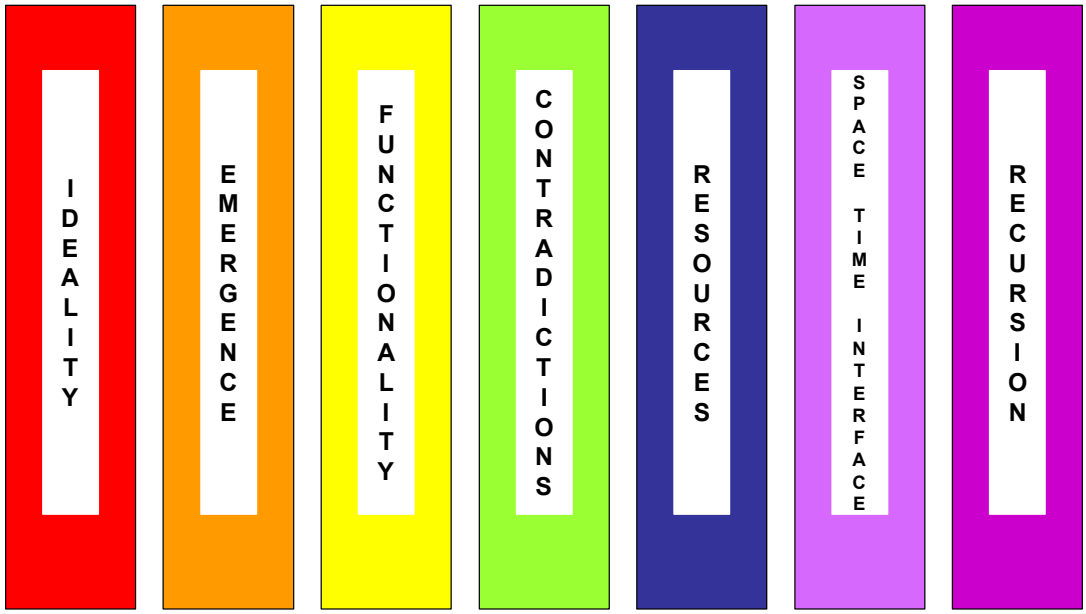

Figure 2: Law Seven Pillars Of 'TRIZ For Software'

Let us explore each of these pillars in the context of their possible relevance to a software context:

IDEALITY: all successful innovations evolve in a direction of increasing ideality – more benefits; less cost, less harm. Evolution towards an ideal final result occurs through a series of patterns that are repeated across different industries. Absolutely relevant to the design and evolution of software – many discontinuous evolution trends can be observed (see later Trends section), and the concept of 'self-x', as in software that 'writes itself' is a goal that many are actively working to achieve.

EMERGENCE: a pillar introduced into 'TRIZ' in the work of this author to make the method relevant to business situations. Just about all business problems involve complex systems and it is thus any appropriate solution generation strategy ought by rights to take complexity issues into consideration. While not all software systems are complex, their interfaces with the outside world increasingly are. The increasing prevalence of fuzzy-logic, genetic algorithms and agent-based software systems thus form a link to complex systems that cannot be ignored by any software problem solving method.

FUNCTION: customers primarily buy functions (benefits), therefore producers should focus on the function delivered by the products and services they deliver and not just the product itself. If customers find a better way of achieving a function, they will stop buying your product or service. Also relevant in the software context; software exists to deliver a useful function, either to other pieces of software or to a piece of hardware or a human.

CONTRADICTION: systems evolve in the direction of increasing ideality through the successive emergence and resolution of conflicts and contradictions. Evolution is

therefore fundamentally discontinuous in nature. The contradiction-eliminating strategies of others have been mapped and can be used to accelerate the evolution of any system. We may also observe the presence of contradictions and contradiction elimination as a primary driver of software evolution. Contradictions tend to occur at the interfaces between software and the outside world rather than wholly 'within' the software.

RESOURCES: anything in or around a system that is not being used to its maximum potential is a resource. Seen to be relevant in the context of relationships between the software and the outside world, but somewhat less meaningful as a concept when we think about the software itself. Especially in a world in which Moore's Law continues to apply. Beyond the initial overhead associated with actually creating the software, the recurring costs are as close to zero as can be imagined, and so the importance of effective use of resources becomes less and less important. We note, however, that programmers take up considerable amounts of unnecessary memory space when constructing a piece of software simply because memory is cheap and it is easier to write a bulky working algorithm than a compact elegant one. There is, in other words, considerable untapped potential in almost all software systems. See also the evolution potential discussion later in this article.

SPACE/TIME/INTERFACE: the human brain is subject to an effect known as psychological inertia; it fools itself into looking at situations from one specific angle. When we are looking to improve a system, we need to be able to change our perspective of it. Perspective shifts can involve physical (or virtual) space, temporal issues, or the way in which different elements of a system interface and relate to one another. This pillar is as important in software design as in the physical world. Perhaps more so since there is a strong tendency for many software engineers to operate in a closed virtual world largely divorced from the interface with actual users. Tools to help software engineers see their world from different perspectives (e.g. through the 9-Windows tool) are considered to be extremely relevant.

RECURSION: another pillar not in classical TRIZ, but found in 'business TRIZ' (Reference 6 again). Recursion is an idea emerging from Stafford Beer's cybernetics research and elsewhere, and relates to the phenomenon whereby certain system phenomenon repeat at different hierarchical levels. The afore-mentioned law of system completeness is one such system that follows the idea of recursiveness.

All of these topics are covered in more detail in the forthcoming 'TRIZ For Software Engineers' book (Reference 7). Given the rather more limited scope of this article, let us switch now from the abstract to the specific and begin to look at some of the problem definition and solution generation tools that might be found in a TRIZ for software toolkit. We will begin with an examination of the possibilities of a contradiction elimination methodology:

Solving Software Contradictions

The classical TRIZ Contradiction Matrix has been the subject of much updating in recent times, through first the publishing of 'Matrix 2003' (Reference 8) and then more recently a matrix for business problems (Reference 6). Despite moving several steps towards making it relevant for the problems faced by software engineers and finding many software patents that had something to contribute to the construction of the tool, Matrix 2003 contains much that was considered off-putting to software users. Hence in 2003 we decided that it was necessary to create a bespoke matrix for software problems. This

matrix has now been completed, and will become publicly available for the first time when the 'TRIZ for Software Engineers' book is published. By way of a taster, however, Figure 3 illustrates the parameters that make up the sides of the new Matrix.

Size (Static)
Size (Dynamic)
Amount of Data
Interface
Speed
Accuracy
Stability
Ability to Detect/Measure
Loss of Time
Loss of Data
Harmful Effects Generated By System
Adaptability/Versatility
Compatibility/Connectability
Ease Of Use
Reliability/Robustness
Security
Aesthetics/Appearance
Harmful Effects On System
System Complexity
Control Complexity
Automation

Figure 3: 21 Parameters Of The New Software Matrix

The method of operation of the new Matrix is exactly the same as that used in the other Matrices; the user has to first identify what they wish to improve in their system, then what is it that is preventing them from making the improvement. From here it is then necessary to translate the specifics of the conflict pair into a pair from the Matrix parameter list that most closely matches those specifics. From there, the Matrix will reveal the most likely Inventive Principles used by others to successfully challenge that conflict pair. The new Matrix represents the outcome of studying around 40,000 software patents and design solutions. The number is limited by the fact that currently only the US is granting software patents (many of which, when we examine them, appear to have a very low degree of novelty), and that the methods used in non-patented breakthrough designs are usually hidden inside source code that is invisible to the outsider.

A pair of simple examples, however, should serve the dual purpose of demonstrating how the Matrix was compiled and how a user might make use of the tool:

US6,785,819

This patent was granted to Mitsubishi Denki Kabushki Kaisha on August 31 2004 (actually some time after the Matrix work was completed). The background to the patent as described in the disclosure is as follows:-

"Recently, a computer system that uses LAN is commonly being adopted in organizations. Commonly, a plurality of LANs located in various locations in an organization's interoffice network are connected altogether to form an intranet. Extending further, an extranet which includes the organization's allied companies to form a network altogether is also becoming widespread. There are various ways to connect a plurality of LANs located in various locations. To give one

example, there is a case of using a low-cost internet instead of a leased line. In this case, access from outside should be regulated, so a firewall is generally set at a boundary of outside and inside of the network. This helps to increase the safety factor inside the LANs. The firewall is a technique which only permits access from outside to a specific location or to a specific application of the LAN. An example of this technique is disclosed in Japanese unexamined patent publication HEI 7-87122. Specifically, the firewall is mostly used in a system which only allows SMTP (simple mail transfer protocol), which is an electronic mail transfer protocol, to pass through. In this case, only an electronic mail message can pass through the firewall. As other examples, there are a system which allows HTTP (hyper text transfer protocol) to pass through, which is a data communication protocol of WWW (world wide web), a system which allows a CORBA (common object request broker architecture) communication protocol IIOP (internet inter-ORB protocol) to pass through, and a system which allows a communication protocol such as RMI prepared by JAVA processing system to pass through. In a network computer system, services under a LAN environment such as file sharing, printing to common printer, or use of CPU server cannot be adopted because of the firewall. Accordingly, in cases when one wishes to obtain a certain data or a program from other location, then the one can only rely on someone at the other location to transmit a required data or the program using an independent channel, or the one can only rely on a method of mailing media such as a tape".

The problem thus identified by the inventors is a conflict between parallel desires to improve data transfer capability and maintain security. Translating these specific requirements into the terms of the software Matrix will then reveal the conflict pair illustrated in Figure 4.

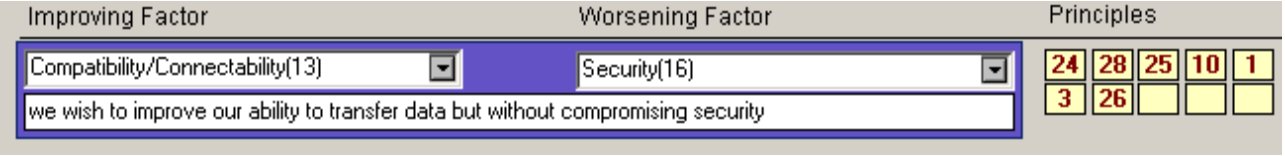

Figure 4: US6,785,819 Conflict Translated Into The Terms Of The Software Matrix
(picture from CREAX Innovation Suite software)

The numbers at the right hand side of the figure relate to the Inventive Principles used by other software engineers to successfully challenge the connectability-security conflict. It is now up to us to translate these generic solution directions into something that might help us to solve our specific problem. The inventors of US6,785,819, for example used a software agent to achieve a solution:

According to one aspect of the present invention, an agent method for transferring an agent inside a network system including a first computer system having an access control unit which allows access in case of meeting a pre-determined communication condition and a second computer system, comprises steps of: authenticating the second computer system for transmitting the agent, and transmitting the pre-determined communication condition of the first computer system to the authenticated second computer system; receiving and storing the pre-determined communication condition, creating the agent, and transmitting the agent according to the pre-determined communication condition by the second computer system; and receiving the agent via the access control unit and executing the agent by the first computer system.

In other words, the inventors used a combination of Principle 24 ('Intermediary') and 25 ('Self-Service' - Make an object serve or organise itself by performing auxiliary helpful functions').

US6,789,097
This patent ('Real-time method for bit-reversal of large size arrays') was granted to Tropic Networks Inc on September 7 2004. Hence it too was granted after work on the new

Matrix was completed. The invention disclosure abstract does a good job of describing both the conflict resolved by the inventors and the strategy they used to achieve the solution:-

A digital signal processor DSP for bit-reversal of a large data array of a size has a direct memory access (DMA) controller for performing in-place bit reversal routines on an external memory during a first stage, and a central processing unit (CPU) for swapping small sub-arrays on internal memory in a second stage. The two stage method according to the invention importantly reduces the real-time implementation for sorting large size data arrays on uni-processor DSP platforms, by extensively using the external memory and avoiding a random access to the internal memory. As well, the invention provides for improved dense integration and reduced costs when used in dense wavelength division multiplexing (DWDM) systems.

The basic conflict under consideration here involves time and amount of memory. If we map these onto the Matrix we will get the following:

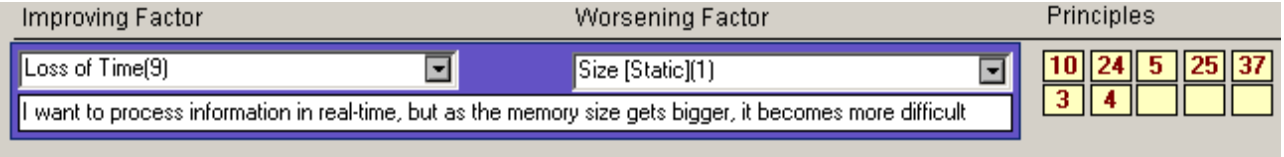

Figure 5: US6,785,819 Conflict Translated Into The Terms Of The Software Matrix

Again we see a good match between the recommendations made by the Matrix and the strategy used by the inventor – in this case specifically the use of a Prior Action (Principle 10) and an external memory (Principle 24).

Reverse engineering an existing solution is, of course, not much of a demonstration of the capabilities of the Inventive Principles. The only real answer to this problem is to actually begin to use the Principles to work on a real live conflict. Such a task is beyond the scope of this article, since our aim is rather to overview the broad applicability of TRIZ to software problems. In doing this, then, it is perhaps important to next move on to look at the Inventive Principles in a little more detail.

Inventive Principles For Software

The first thing to say on the subject of the Inventive Principles is that our research has confirmed the existence of the same 40 Principles as are found in classical TRIZ. As was the case with the conversion from technical to management Principles, we have had to alter some of the terminology of the Principle descriptions to better suit the software context, but beyond that, they have stayed the same. No software engineers according to our analysis have identified inventive strategies that do not fit into the existing Principles framework. This is in itself a useful finding. The list of Principles featuring the modified titles is illustrated in Figure 6 below:

The TRIZ for Software research has generated a comprehensive description of each Principle along with a cluster of software examples for each. The main aim of such lists, as with Rea's list (Reference 3) and other lists of Principles in other domains, is to provide a list of examples to explain the meaning of each Principle, in the hope that once users can begin to understand what a Principle is about they will begin to be able to connect it to the problem they are working with. As ever with TRIZ, in the final analysis the jump from the generic solutions offered by the Inventive Principles to a specific solution to a specific software problem is going to require some serious thinking. In software as in other fields, 'systematic' does not necessarily mean 'easy'.

Also worth noting in this regard is the additional finding from our research that the majority of software problem solutions appear to merge ideas from several Principles. Hence a solution generation process in which solutions are first generated from individual Principle triggers, and then subsequently combined to form stronger solutions is at the very least 'highly recommended'; very few breakthrough software solutions, in our experience to date, look as though they will come from one idea generated from one single Inventive Principle.

1. Segmentation
2. Extraction
3. Local Quality
4. Asymmetry
5. Combination
6. Universality
7. 'Nested Doll'
8. Counterbalance
9. Prior Counter-Action
10. Prior Action
11. Prior Cushioning
12. Remove Tension
13. 'The Other Way Round'
14. Loop
15. Dynamics
16. Slightly Less/Slightly More
17. Another Dimension
18. Vibration
19. Periodic Action
20. Continuity of Useful Action
21. Hurrying
22. 'Blessing in Disguise'
23. Feedback
24. Intermediary
25. Self-Service
26. Copying
27. Cheap/Short Living
28. Another Sense
29. Fluidity
30. Thin & Flexible
31. Holes
32. Colour Changes
33. Homogeneity
34. Discarding and Recovering
35. Parameter Changes
36. Paradigm Shift
37. Relative Change
38. Enrich
39. Calm
40. Composite Structures

Figure 6: 40 Inventive Software Principles
(green titles highlight text changes relative to original list)

Looking beyond the basic Principles, then, is the possibility of analyzing which Principles are being used more than others:

Frequency Sequence Of The Software Inventive Principles

The analysis described here is based on the same 40,000 software solutions used to compile the software Matrix. The method of presenting our findings is exactly the same as was recently used for the updated technical and new business Matrices. The software Principle frequency sequence, then is as follows:

	1st	2nd	3rd	4th	5th	6th	7th	8th	9th	10th
0	**10**	**3**	**24**	**13**	**2**	**25**	**35**	**7**	**1**	**4**
+10	**37**	**15**	**28**	**17**	**19**	**5**	**32**	**40**	**6**	**23**
+20	**26**	**14**	**9**	**12**	**34**	**31**	**11**	**22**	**29**	**16**
+30	**39**	**21**	**27**	**33**	**20**	**18**	**30**	**8**	**36**	**38**

Figure 7: 40 Inventive Software Principle Frequency Sequence

Hence, we see that Inventive Principle 10 is the most commonly used; Principle 3, the second most common, and so on through to Principle 38 down in 40th place.

The list is also reproduced in a slightly different format in the fourth column of Table 1 below. The Table also reproduces the ranking of Principles from Matrix 2003 (see

Reference 9 for a more detailed analysis of Matrix 2003 versus classical placings), and then the relative changes between business and technical Principles sequences.

Inventive Principle	**Classical TRIZ Ranking**	**Matrix 2003 Ranking**	**Software Matrix Ranking**	**Change** (Classical-Software)	**Change** (Matrix 2003-Software)
1	3	7	9	-6	-2
2	5	5	5	-	-
3	12	2	2	+10	-
4	24	10	10	+14	-
5	33	12	16	+17	-4
6	20	27	19	+1	+8
7	34	17	8	+26	+9
8	32	37	38	-6	-1
9	39	24	23	+16	+1
10	2	8	1	+1	+7
11	29	39	27	+2	+12
12	37	19	24	+13	-5
13	10	3	4	+6	-1
14	21	15	22	-1	-7
15	6	14	12	-6	+2
16	16	28	30	-14	-2
17	19	9	14	+5	-5
18	8	25	36	-28	-11
19	7	11	15	-8	-4
20	40	40	35	+5	+5
21	35	32	32	+3	-
22	22	36	28	-6	+8
23	36	33	20	+16	+13
24	18	6	3	+15	+3
25	28	13	6	+22	+7
26	11	23	21	-10	+2
27	13	35	33	-20	+2
28	4	4	13	-9	-9
29	14	26	29	-15	-3
30	25	22	37	-12	-15
31	30	16	26	+4	-10
32	9	21	17	-8	+4
33	38	38	34	+4	+4
34	15	31	25	-10	+6
35	1	1	7	-6	-6
36	27	30	39	-12	-9
37	26	20	11	+15	+9
38	31	34	40	-9	-6
39	23	29	31	-8	-2
40	17	18	18	-1	-

Table 1: Comparison Of Classical, Matrix 2003 and New Software Matrix

The ‘change’ columns indicates that there have been some quite significant shifts that have taken place between the classical matrix and the new Software Matrix. To a lesser extent there have also been some shifts relative to Matrix 2003. The biggest ‘risers’ up the sequence list perhaps offer the most interesting insight into the differences between conflict resolution strategies for software and technical problems: The rises in the use of Principles 11 (‘Beforehand Cushioning’) and 23 (‘Feedback’) are both an indication of an

industry still in the early phases of its development. The rises seen for Principles 37 ('Relative Change' in our software application) and 7 ('Nested Doll') are more indicative of the ease with which these two strategies may be deployed in a software context. Nesting in particular can be something that adds considerable complexity in a physical system, but in a piece of software use of the strategy merely requires the creation of additional lines of code. The rise of the Relative Change Principle is likewise due to the ease with which this strategy can be deployed in a digital environment.

Regarding the Principles falling down the list, it seems clear that there are certain of the Principles that are more difficult to interpret in the software context. Big fallers include Principles 18 ('Vibration'), 30 ('Flexible Shells And Thin Films'), 31 ('Porous Materials or 'Holes') and 36 ('Phase Transition'). Whilst software examples all of these Principles can be found, their existence is rare and sometimes the imagination stretch required is quite high. Detailed analysis of the priority sequence illustrated in Figure 7 will, in fact, show that there is a big gap in frequency of use of the top 16 Principles compared to the 24 below them.

Ideality/Trends

Moving on from Contradictions, another important tool in the software context appears to be the Ideality and Trends parts of TRIZ. A big part of the research underlying the TRIZ for Software work has examined patterns that exist in software evolution as systems jump discontinuously from one way of doing things to another. This is because we hope that knowledge of such trends will allow us to considerably accelerate the evolution of existing systems in the same way that is beginning to happen with the technical systems we are involved with.

To date, we have uncovered 24 discontinuous trends. As with the technical and business trends, we have found a broad split of these trends into physical, temporal and interfacial categories – Figure 8. a Glance at the trend titles shown in the figure will indicate a fair degree of connection with the technical and business trends. Detailed analysis of these trends, however, reveals some interesting differences (Reference 7).

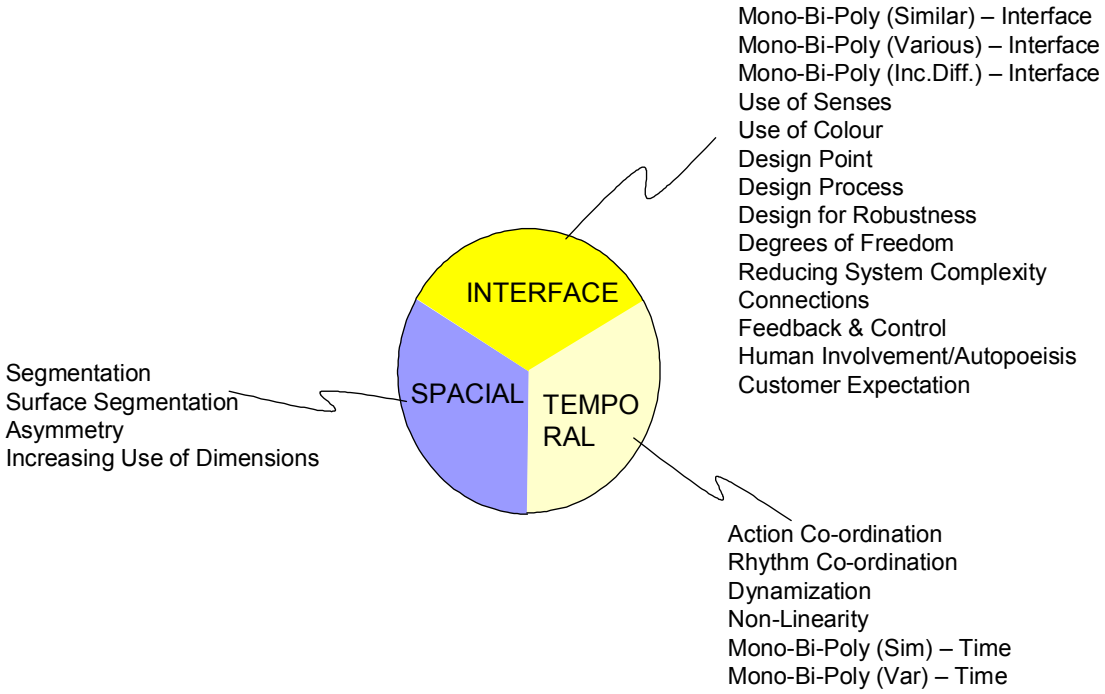

Figure 8: Current List Of Discontinuous Software Evolution Trends

Also present in the list are trends that have little or no corresponding equivalent in the technical or business worlds. Amongst these is a trend we have labeled 'Design Process'. The basic form of this trend is illustrated in Figure 9 below. In many ways, this trend is connected to the Capability Maturity Model (CMM, Reference 10) used by many as a standard in the software industry.

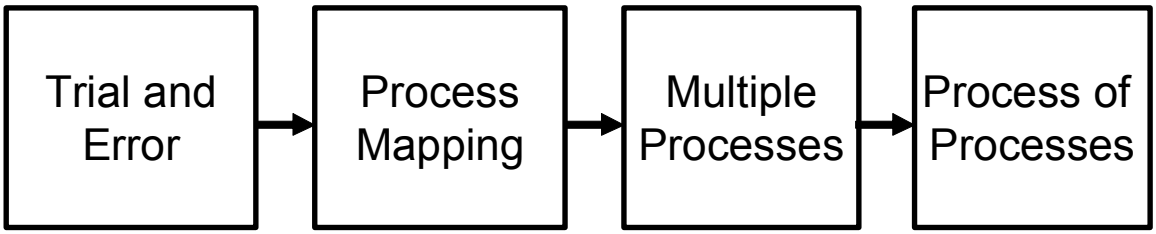

Figure 9: Typical Software Evolution Trend – Design Process

As with our technical and business trends, we have found it useful to also employ the evolution potential concept (References 4, 6 and 11) as a means of comparing a given software system against a global maturity scale. Figure 10 illustrates a typical example.

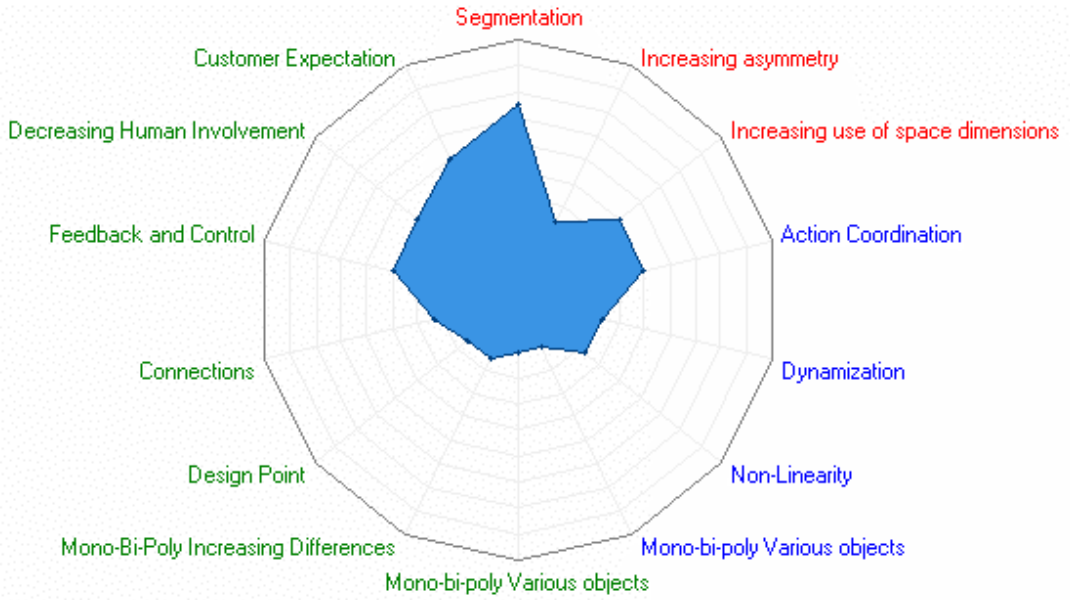

Figure 10: Typical Evolution Potential Radar Plot For Software Systems

Again, in common with technical and business systems, we see certain sectors of the industry are more advanced in certain trend directions than others. Hence we see a time where there are a number of 'low-hanging fruit' benefits to be obtained by the first users of the TRIZ trends, by taking known advances in one part of the industry and deploying them in another.

Self-X

Already we see many of the discontinuous software trends shown in Figure 8 clearly heading towards extensive use of 'self-x' (Reference 12). Autopoeisis in software systems – i.e. self-repairing, self-updating, self-reproducing – is already common in several sectors of the industry. The speed at which software systems continue to evolve relative to the technical systems bound by the limitations of the physical world means that consideration of the 'idea final result solution' is an essential in the vast majority of all software problem definition exercises.

Other Tools

Looking beyond contradictions and trends, examination of the remaining TRIZ tools and testing their applicability in software applications reveals a number of possibles. Rea has, for example, speculated on the applicability of the S-Field tool in software applications (Reference 13). Our own experience with this tool, on the other hand, given the earlier discussion about the presence or otherwise of insufficient, excessive or harmful relationships between different parts of a software system, is that it is a step too far for most software engineers. While we believe that the Inventive Standards have something to offer in the software context, we consider that it is better to integrate these solution generation triggers into the other tools. Thus, to take a single example, an Inventive Standard recommending the incorporation of an 'intermediary substance or field' overlaps considerably with Inventive Principle 24, and so as long as the Principle 24 definition is modified appropriately we can effectively make the stand-alone Inventive Standard redundant.

The discussion of other tools, however, cannot end without some discussion of the Subversion Analysis tool. Alongside the Contradictions tool, we believe this is probably the most important TRIZ tool in the software engineers' armoury at this point in time. Put simply, there are still a very large proportion of software systems in the world that are anything but 'robust'. At least from the layman users' point of view. Designing robustness into increasingly complex software systems is an undoubtedly significant and growing challenge to software engineers. Although the form of the subversion analysis tool has had to change somewhat from the traditional technical form, the essential elements remain intact when we look to use the method in the software context. The key Subversion Analysis question, 'how can we destroy this system?' for example remains as if not more relevant in a software application than it does for any technical system. Reference 7 pays particular attention to the development and use of subversion analysis techniques in the software context.

Conclusions

Contrary to the view of Ikovenko (Reference 1 again), we believe the evidence from the patent database and elsewhere would suggest that the world of software is more science than art-form. Like Karasik (Reference 2), we believe that while classical TRIZ principles apply, they form a necessary but insufficient part of a full 'systematic innovation for software' story. At the very least, the findings of the research underlying 'TRIZ for Software' have revealed the need to incorporate ideas from complexity theory and cybernetics into the basic philosophy. It is probably still early days in the overall evolution of the full story.

Nevertheless, we also believe that it is possible to create tools capable of helping software engineers to do a better job. Specifically, we see the emergence of a definition/solution methodology comprising the following tools:

Definition:	1) Ideal Final Result
	2) Problem Explorer (Reference 4, 6 and 7)
	3) Subversion Analysis
	4) Contradiction Matrix
Solution Generation:	1) Inventive Principles
	2) Trends/Evolution Potential
	3) 'Self- X'

The early experience of Beta users of this 'TRIZ for Software' toolkit appears to show considerable benefit. Clearly it is still early days, but the ability to offer software engineers at least the foundations of a systematic innovation capability is probably enough for most to want to give it a try. The book 'TRIZ For Software Engineers' will be published shortly for those that might be interested in taking the story further.

References

1) Ikovenko, S., 'Laws Of Engineering System Evolution In Software Development', keynote paper at TRIZ Centrum, 3rd European TRIZ-Conference, ETH Zurich, March 2003.
2) Karasik, Y., editorial comment, Anti-TRIZ Journal, Vol.2, No.2, and private email correspondence, 8 September 2004.
3) Rea, K., 'TRIZ and Software - 40 Principle Analogies, Parts 1 and 2', TRIZ Journal, September and November 2001.
4) Mann, D.L., 'Hands-On Systematic Innovation', CREAX Press, June 2002.
5) Espejo, R., Harnden, R., 'The Viable System Model', John Wiley & Sons, New York, 1989.
6) Mann, D.L., 'Hands-On Systematic Innovation For Business & Management', IFR Press, August 2004.
7) Mann, D.L., 'TRIZ For Software Engineers', IFR Press, October 2004.
8) Mann, D.L., Dewulf, S., Zlotin, B., Zusman, A., 'Matrix 2003: Updating The TRIZ Contradiction Matrix', CREAX Press, Belgium, July 2003.
9) Mann, D.L., 'Comparing The Classical And New Contradiction Matrix, Part 2 – Zooming In', TRIZ Journal, July 2004.
10) Paulk, M.C., Weber, C.V., Garcia, S.M., Chrissis, M.B., Bush, M., 'Key Practices Of The Software Maturity Model', Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania.
11) Mann, D.L., Dewulf, S., 'Evolutionary Potential In Technical And Business Systems', TRIZ Journal, June 2002.
12) Mann, D.L., 'Ideality And Self-X', paper presented at 1st ETRIA TRIZ Future Conference, Bath, November 2001.
13) Rea, K., 'Applying TRIZ to Software Problems', TRIZ Journal, October 2002.

About The Author

Darrell Mann entered the world of software development in the days of punch-cards and Fortran 4. His first job at Rolls-Royce, after graduating with an Engineering degree, was the design and creation of computational fluid dynamic software tools. He continues to act as architect and problem solving consultant on many software-related projects to the present day, taking great pride in knowing as little of the industry jargon as is possible.