TRIZ And Software Innovation:
Historical Perspective And An Application Case Study

Darrell Mann
Director, Systematic Innovation Ltd, UK
Phone: +44 (1275) 337500
E-mail: darrell.mann@systematic-innovation.com

'If it can be specified, it can be programmed'.

Abstract

The paper focuses on the relevance and application of TRIZ ideas and strategies to the design of software systems. The paper is divided into two main sections. In the first section the focus is on a historical review of innovations in the software domain. In this section we examine the contributions of innovators from Charles Babbage to Alan Turing, Tim Berners-Lee to Martin Fowler in the context of the TRIZ trends of evolution and the discontinuous jumps their innovations represented. We then describe a number of key evolution software trends consistent with these and other software innovations and, using the currently untapped evolution potential, make predictions as to where the software science is likely to evolve in the future. In the second part of the paper we consider a real case study application of the software evolution trends. The focus of the case study is design of control systems for unmanned air vehicles (UAVs). UAVs represent a considerable number of design challenges for control engineers, many of which currently have no effective solution. The most serious of these problems in several applications is the inability of operators to obtain act upon local environmental effects data (cross-winds, foreign object ingestion, rain, etc) when the aircraft has unknowingly deviated from its planned mission. By examining the contradiction present in this situation and integrating some of the TRIZ predicted discontinuous evolution trends the paper proposes a number of control architecture innovations that look set to deliver significant operational capability enhancements in the UAV arena.

What Is Software Innovation?

It is appropriate to begin the paper with a discussion on what innovation actually means in the software context. In its most general sense, 'innovation' is usually taken to mean a new idea that has been successfully deployed onto the market; something that is satisfying a genuine market need and is generating a sustainable profit. We can apply the same definition in the software arena, but if we do, we quickly run into the problem that the software is often only a relatively minor part of a much bigger story. Thus we can ask, was Internet Explorer an innovation? Or the Internet? Or eXtreme Programming (XP)? None of these stories is easy to untangle. Internet Explorer, for example, certainly meets our definitions regarding market need and profit, but was the software idea itself

new? The Internet on the other hand never delivered a direct 'profit' (financial at least) to Tim Berners-Lee's, and so even though the software concept was novel, should it be called an innovation? Then XP, which continues to be popular amongst software developers, has little if any innovation in terms of software, but rather is all about novel ways of organising software engineers.

We can begin the process of untangling these and other stories by doing two things. The first is to re-define 'innovation' as a '*discontinuous jump towards a more ideal system*'. The key word in this definition is 'discontinuous'. It is there to give the clear message of a distinct shift from one way of doing something to another. The second thing we need to do is define the possible domains in and around the software world. Figure 1 is an attempt to do this. The 'world' here has been divided into a collection of four overlapping domains; software, technical, business and mathematics. They overlap because a) any kind of segmentation like this is drawing boundaries that don't exist in real life, and b) because often an innovation straddles several domains. Like for example, the computer mouse – a primarily software-domain innovation, but with elements of physical hardware.

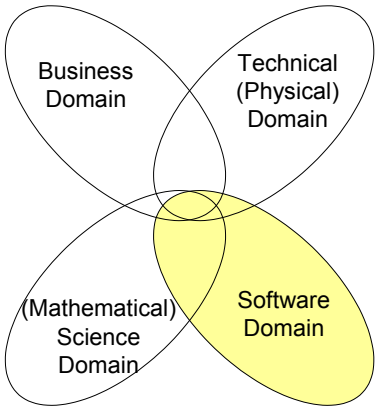

Figure 1: Different Innovation Types In And Around Software Domain

This picture should hopefully help us to see that XP, although it relates to software innovation, is actually a business domain jump. Internet Explorer similarly, is an innovation that is all about business – the jump here (and business success factor) being about bundling your navigation tool with other software products. With these definitions and boundaries in mind, and given the title of this paper, our concern from here on in is innovation in the shaded 'software' region of the Figure, and more specifically, discontinuous jumps that have occurred in the software domain. Our task in identifying and mapping software innovations is helped considerably by the work of David Wheeler (Reference 1). Wheeler has published excellent materials on the evolution of software systems. Although he never uses the 'discontinuity' definition, we see that each of the things he includes in his list meet our definition. Table 1 reproduces a modified and slightly expanded version of the data found in Wheeler's work.

The right-hand column of the Table describes the discontinuous jump made by each innovation in terms of the discontinuous software evolution trends uncovered as a result of our own extensive programme of research on software systems. We will discuss these trends in more detail in the next section.

Table 1: Macro-Level Software System Innovations

Innovation	Source	Year	Breakthrough/Trend
Analytical Engine (software)	Charles Babbage	1837	Design Process Action Co-ordination
Boolean Algebra	George Boole	1845	Design Process Mono-Bi-Poly (V)
Turing Machines	Alan Turing	1936-7	Design Process Non-Linearities
Stored Programme	John von Neumann	1945	Action Co-ordination Mono-Bi-Poly (V)
Hypertext	Vannevar Bush	1945	Nesting - Down
Subroutines	Maurice Wilkes, Stanley Gill, David Wheeler	1951	Nesting – Down Mono-Bi-Poly (V)
Assemblers	Alick E. Glennie	1952	Human Involvement
Compilers	Grace Murray Hopper	1952	Human Involvement Nesting - Up
Human-like notation (FORTRAN)	John Backus	1954-7	Design Process
Stack Principle ("the operation postponed last is carried out first")	Frierich L. Bauer and Klaus Samelson	1955	Nesting – Down Action Co-ordination
Time-Sharing	John McCarthy	1957	Segmentation Mono-Bi-Poly (S)
List-Processing (LISP)	John McCarthy	1958-60	Non-Linearity Design Process
Survivable Packet-Switching Networks	Paul Baran	1960	Connections Dynamization
Word-Processing	(IBM)	1964	Human Involvement
Mouse-Based User Interface	Douglas C Englebart	1964	Human Involvement
Semaphores	E. W. Dijkstra	1965	Action Co-ordination Design Robustness
Hierarchical Directories (Multics)	Louis Pouzin	1965	Nesting – Down
Unification	J.A. Robinson	1965	Action Co-ordination Reducing Complexity
Structured Programming	Bohm & Jacopini	1966	Segmentation Nesting - Down
Spelling Checker	Les Earnest	1966	Feedback & Control Human Involvement
Object-Oriented Programming	Ole-Johan Dahl & Kristen Nygaard	1967	Nesting – Up Connections
Separating Text Content from Format	William Tunnicliffe	1967	Segmentation
Graphical User Interface (GUI)	J.C.R. Licklider	1968	Boundary Breakdown Human Involvement
Regular Expressions	Ken Thompson	1968	Reducing Complexity Boundary Breakdown
Standardized Generic Markup Language (SGML)	C.F. Goldfarb, Ed Mosher, & Ray Lorie	1969-70	Nesting – Down Mono-Bi-Poly (V)

Relational Model and Algebra (SQL)	E.F.Codd	1970	Connections Dimensionality
Distributed Network Email	Richard Watson	1971	Degrees Of Freedom
Modularity Criteria	David Parnas	1972	Segmentation Action Co-ordination
Screen-Oriented Word Processing	Lexitron and Linolex	1972	Mono-Bi-Poly (V) Action Co-ordination
Pipes	M. D. McIlroy	1972	Connections
B-Tree	Rudolf Bayer Edward M. McCreight	1972	Nesting – Down Segmentation
Portable Operating Systems (OS6,Unix)	J.E. Stoy & C. Strachy	1972-6	Nesting –Up Boundary Breakdown
Internetworking using Datagrams (TCP/IP)	(Cyclades Project) France	1972	Boundary Breakdown Segmentation
Font Generation Algorithms	Peter Karow	1973	Mono-Bi-Poly (S) Segmentation
Monitor	Hoare & Hansen	1974	Nesting –Up Action Co-ordination
Communicating Sequent-ial Processes (CSP)	C.A.R. Hoare	1975	Action Co-ordination Rhythm Co-ordination
Diffie-Hellman Security Algorithm	Diffie-Hellman	1977	Boundary Breakdown Dynamization
RSA security algorithm	Rivest, Shamir, and Adleman	1978	Boundary Breakdown Action Co-ordination
Spreadsheet	Dan Bricklin & Bob Frankston	1978	Design Point Segmentation Increasing Dimensions
Lamport Clocks	Leslie Lamport	1978	Action Co-ordination Nesting - Time
Distributed Newsgroups (USENET)	Tom Truscott, Jim Ellis, Steve Bellovin	1979	Segmentation Asymmetry
Model View Controller	(Xerox, PARC)	1980	Segmentation
Remote Procedure Call	(Xerox, PARC)	1981	Action Co-ordination
Distributing Naming (DNS)	-	1984	Segmentation Nesting-Down Mono-Bi-Poly (Inc-Diff)
Semantic Search	David A. Plaisted	~1985	Segmentation
Lockless version mgmt.	Dick Grune	1986	Dynamization
Distributed Hypertext via Simple Mechanisms (www)	Tim Berners-Lee	1989	Mono-Bi-Poly (V) Segmentation Connections
Design Patterns	Gamma, Helm, Johnson, Vlissides	1991	Design Process Knowledge
Secure Mobile Code (Java and Safe-Tcl)	(Sun)	1992	Design Robustness Mono-Bi-Poly (V)
Refactoring	W.F.Opdyke	1993	Design Process Nesting - Down
Web-Crawling Search Engines	(World-Wide-Web Worm)	1994	Connections Asymmetry

Two things are perhaps the most striking about this list. Firstly is that, according to Wheeler (and indeed our own research) over the history of software development there have not been that many innovations. Actually, we ought to qualify that statement; there have not been many innovations at this macro-scale.

Secondly, and leading on from this statement, is the observation that nesting and recursion appear to have played a significant role in the evolution story. Again, more on this subject later, but in the meantime, it is useful to note a pattern of evolutions to other levels (higher or lower) followed by segmentations, Mono-Bi-Poly and Co-ordination jumps. Thus, over time, the software domain has gradually extended. It has extended periodically by nesting jumps either downwards into the sub-system and sub-sub-system, etc levels or upwards into super-system or super-super-system, etc. Thus what we see today is the emergence of highly hierarchical architectures. We can see this schematically in Figure 2.

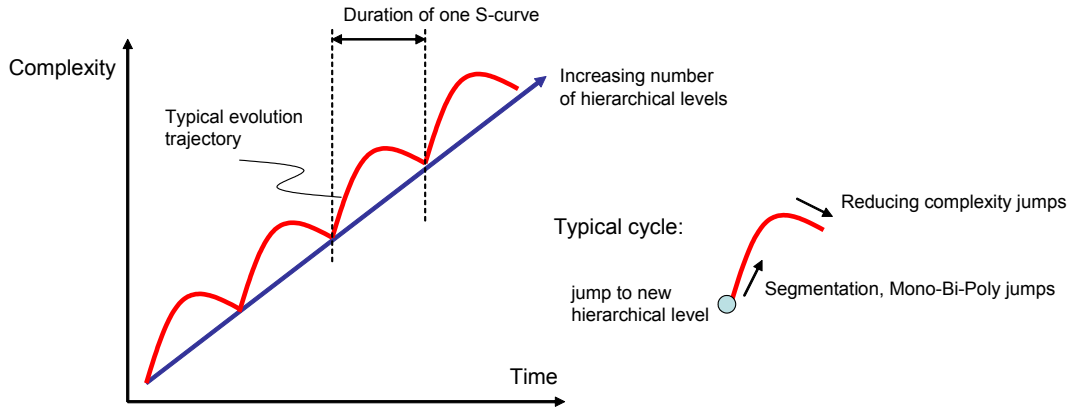

Figure 2: Recursion In Software Systems

This is important for us to keep in mind when looking for other examples of software innovation. What Wheeler has done is brought together macro-level system innovations that have had a clear and visible impact on the world. We could equally well, however, observe similar discontinuities at the sub-system and lower levels. These will tend to be less visible to the public at large, but they are nevertheless still valid 'innovations' by our discontinuity definition, and thus – if they are shown to fit into an overall pattern – they can teach us much about generating our own innovations systematically. The word 'pattern' here is a useful one to think about since the 'Design Patterns' are classed as one of Wheeler's macro-level innovations. The Design Patterns (Reference 2) are in essence a mini-version of the TRIZ story. These patterns are all about uncovering 'good' design practice and making it available in abstracted form to others working in the domain. So, the emergence of Design Patterns in the macro sense can be viewed as a significant discontinuous jump in the direction of the more ideal system. When an individual software engineer uses one of the Patterns in the construction of a sub-sub-system subroutine or DLL, it may not be so visible to the outside world, but it would nevertheless count as an innovation because a discontinuous jump has occurred.

A good analogy to keep in mind here is the Toyota innovation strategy (Reference 3). Toyota pride themselves on the large number of small-scale innovations they succeed in introducing into their vehicles (the Reference quotes a million per year). They make little

or no attempt to classify those innovations into ‘big’ or ‘small’ categories, but rather employ the idea that the overall job is to move in the direction of the ‘ideal’ system. We recommend a similar approach when thinking about software innovation; the hierarchical level in the system where the discontinuous jump towards the ideal is not nearly so important as actually making the jump.

In many ways, the Toyota ‘success by a million jumps’ strategy is even more important in the software context than it is in automotive. Introducing a discontinuous change into a physical object like a car or a windshield wiper takes far more effort (and money) than changing a few lines of code. That fact coupled with the Linux/open-source/sharing culture means that things can and ought to be able to evolve much faster in most parts of the software world.

If this appears not to have happened (and a few minutes scanning through some of the thousands of really poor software patents – see Reference 1 again for Wheeler’s discussion on the subject) it is likely to be as much as anything because the emphasis in the software world in since its inception has most frequently been to find enough people to write enough competent code to satisfy the market need. The key word here is ‘competent’. The word ‘innovation’ has only really entered the software world vocabulary because the growth curve is beginning to flatten and competition between software writing companies is starting to hot up. So now let’s not just talk about ‘innovation’, but ‘systematic innovation’ in the software context:

Discontinuous Software Trends

Having discussed the idea of discontinuous jumps (at whatever hierarchical level) as the basis of innovation and made the distinction between what is software and what is not, over the course of the last seven years our team of researchers has been systematically looking for and reverse engineering all forms of software discontinuities.

Guided in part by the original discontinuous trend patterns uncovered in technical and business systems, we have so far uncovered 26 patterns relevant in the software context. As shown in Figure 3, these 26 discontinuous evolution trends appear to fall into three basic groups – physical, temporal and interfacial. In turn these three clusters map very well onto the three important aspects of the Ideal Final Result – Free, Perfect and Now (Reference 4).

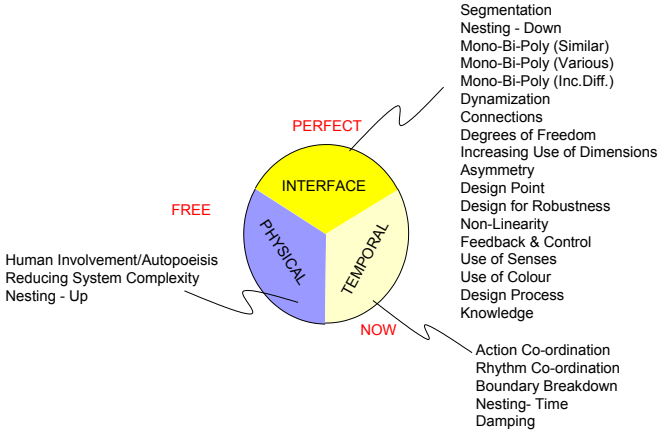

Figure 3: Discontinuous Software Evolution Trends In Free/Perfect/Now Clusters

As presented in Reference 5, each of these trends can be distinguished by a number of distinct stages. Crudely speaking, each of these stages can be thought of as an s-curve. Thus the jump from one stage to another represents a discontinuous shift from one s-curve to another. In this way, the software innovations mapped in Table 1 each represent a jump along at least one of the 26 trends. Not all of the 26 trends shown in Figure 3 are present in Table 1 since the Table is about macro-level jumps. The trends not featured in Table 1 are thus generally found when we uncover innovations at sub-system and sub-sub-system levels.

Many of these Trends will be familiar to TRIZ advocates, at least in terms of their titles. That fact plus the limited space available here forces the discussion to examine just one or two of the trends in more detail. In line with earlier discussions, the Nesting trends appear to be important and therefore worthy of more detailed investigation. As first discussed in Reference 6, there are two versions of this trend pattern; one where the system evolves to a new level of detail at a lower level; and one where a system is nested into a higher level system. Both are believed to be a significant driver for the recursive evolution trajectory plotted in Figure 2.

Figure 4 illustrates the first of these two types; what we now call the 'Nest –Down' trend.

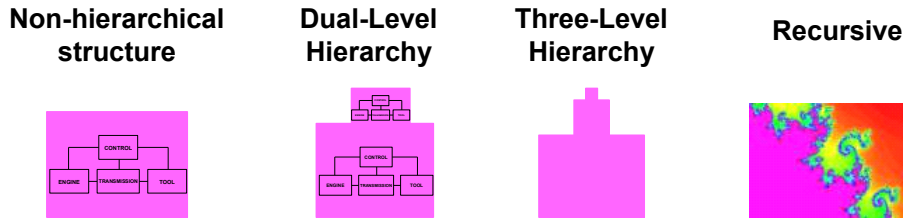

Figure 4: 'Nest-Down' Discontinuous Evolution Trend

In many ways this Trend is like the Mono-Bi-Poly trend. The significant difference – and the reason we now feature it as a distinct entity in its own right – is that the key to successful use of the trend is that users think specifically about adding more hierarchical levels to the present system. A further similarity to Mono-Bi-Poly is that the trend is open-ended in that it should always be asking us to think about whether there is an advantage in adding a *further* level. A difference is that with Mono-Bi-Poly we will eventually reach a point where we can see no further advantage in adding something else to the system. With the Nesting trends, so far at least and consistent with Figure 2 (and evolution in natural systems), we have not observed this 'disappearing advantage' phenomenon.

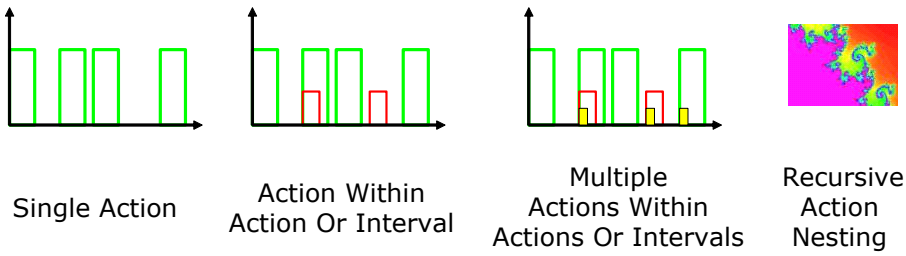

Figure 5: Temporal Interpretation Of 'Nest-Down' Discontinuous Evolution Trend

The Nest-Down trend is applicable in terms of system architecture and also temporally. Figure 5 illustrates the Nesting concept in terms of time – and the insertion of actions within other actions. As with the architecture interpretation, as yet we have not observed the 'disappearing advantage' characteristic yet.

Figure 6 illustrates the other main version of this trend, this time where nesting occurs in the opposite direction and one system (or its function) migrates into a higher level of a system hierarchy. This is what we have called the 'Nest-Up' trend.

Figure 6: 'Nest-Up' Discontinuous Evolution Trend

What this Trend is trying to illustrate are the stages in one upward nesting cycle. As with 'Nest – Down' we have not yet observed a limit to the number of times that this cycle can repeat. It does, however, happen less frequently. The trend has been constructed in the way that it has because the most common way of utilising the trends. This involves an Evolution Potential assessment of a given system (Reference 7), where users are looking to explore where that particular system is likely to evolve in the future. As far as 'Nest – up' is concerned the key question and direction the user is being asked to examine is 'is there a higher level system into which yours can be usefully integrated?'

With this thought in mind, we can use all of the uncovered trends to give us clues as to the likely future evolution directions of software systems. Our usual way of doing this is to use the Evolution Potential framework:

Untapped Potential

Given the earlier message that increasing hierarchy in software systems is an important evolution driver, it will immediately feel crude and naïve to simply construct one radar plot to describe the whole of the domain. Nevertheless, given the macro-level focus in Table 1, such a plot may be instructive in obtaining an indication of future macro-level evolution jumps. This should be okay, so long as we keep in mind the idea that if we were doing a serious analysis of any given software system we would typically construct plots for each of the levels and each of the elements within each element. Figure 7 illustrates this idea and the composite macro-level plot in more detail.

One of the difficulties in drawing this macro-level radar plot is in knowing whether systems have evolved to the ends of the Nesting, Segmentation and Mono-Bi-Poly trends. As stated earlier, the purpose of these Trends is to provoke a question. In the Figure 7 plot, we have gauged that for each of these trends there remains untapped potential. We believe this is the case because the Nesting trend doesn't yet appear to be subject to the law of diminishing benefits. Then, because the emergence of a new hierarchical level opens up new Segmentation and Mono-Bi-Poly opportunities, even

though the benefits in additional Segmentation or Mono-Bi-Poly advance may have been reached at one hierarchical level, they will not have been exploited at the new level.

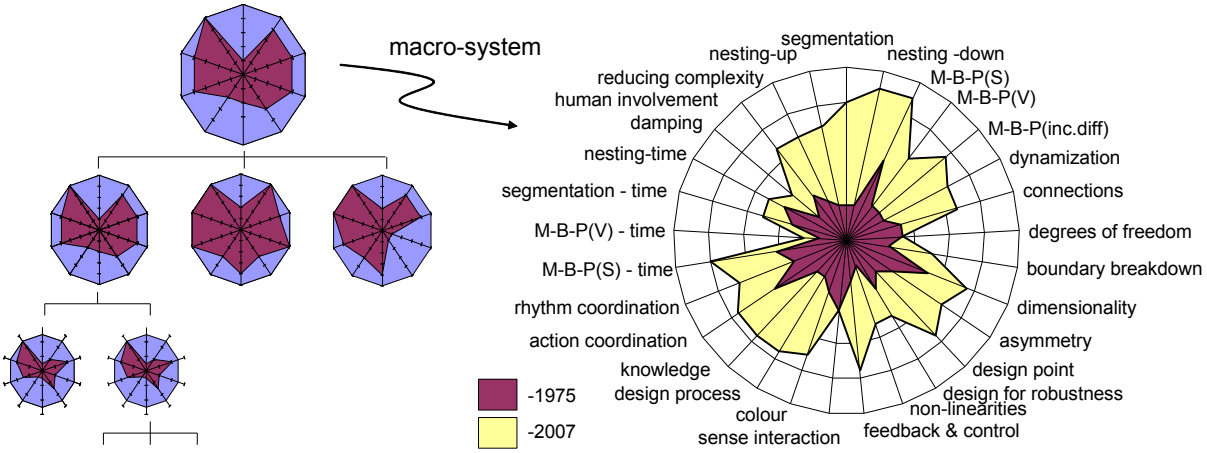

Figure 7: Macro-Level Software Evolution Potential History

The plot suggests to us that despite the rapid evolution of software systems, there continues to be large amounts of untapped potential. Nesting looks set to continue to be important (in this regard we note that the evolution of software is following a similar trajectory to that of the human brain – another hierarchically organised control system, where, according to Reference 8 the brain currently features at least seven hierarchical levels – suggesting that software (four or five levels depending on your perspective) still has significant untapped potential. Perhaps because architecture is more readily visualised than temporal issues, we speculate that there is considerably more opportunity for nesting of actions.

Also indicated by the plot is the probably obvious (to Microsoft users at least!) untapped potential in terms of system robustness evolution. Related but less obvious is the considerable untapped potential in terms of software systems capable of handling non-linearities. If this too sounds like an ‘obvious’ direction, the Dynamization and Connections trends go some way towards indicating likely solutions to achieve a non-linear capability in that both indicate shifts towards software architectures with dynamically switchable connections and links.

Perhaps also falling into the ‘non-obvious’ category of predictions is an interpretation of the Mono-Bi-Poly trend to the binary foundation of current systems. Although difficult to predict when it will happen, it feels clear to us that doing things using zeros and ones will ultimately hit a limit (either in terms of processing speed, or more likely, inability to handle fuzzy and non-linear situations) that will in turn provoke an evolution to a non-binary computing platform.

A Real Problem – UAV control

One of the big problems with this kind of high-level analyses of domains as broad as ‘software’ is that the outcomes and suggestions are at a similarly high level of abstraction. It is one thing to suggest that software systems will continue their evolution

towards increasing numbers of hierarchical levels, but quite another to work out why that might be useful in a particular situation. To at least begin to rectify this problem, this final section of the paper examines a real software problem.

One of the big shifts taking place in the aerospace world at the present time is the replacement of pilots with remote-controlled unmanned air vehicles (UAVs). This is happening for a number of reasons, not least of which are the desire to keep humans away from dangerous situations, and the fact that pilot physiology is an increasingly dominant factor preventing improvements in aircraft performance. Figure 8 illustrates a typical small UAV of the type used for reconnaissance purposes, usually over hostile territory.

Figure 8: RQ-2 Reconnaissance UAV

Software systems play a crucial role in the control of UAVs. The current state of the art in UAV control places much of the mission responsibility with the remote human operator. The aircraft is fitted with a number of sensors, not least of which is the camera system tasked with delivering the reconnaissance information back to the base station. One of the key problems facing UAV control system designers is ensuring the ability of the aircraft to conduct its intended mission in environments that have the potential to change rapidly and unpredictably. Typical problem scenarios include such things as local environmental shifts (air turbulence, sudden cross-winds, etc), foreign object damage to key aircraft systems and faults occurring within the aircraft. Current control systems are unable to do much to mitigate against any of these problems.

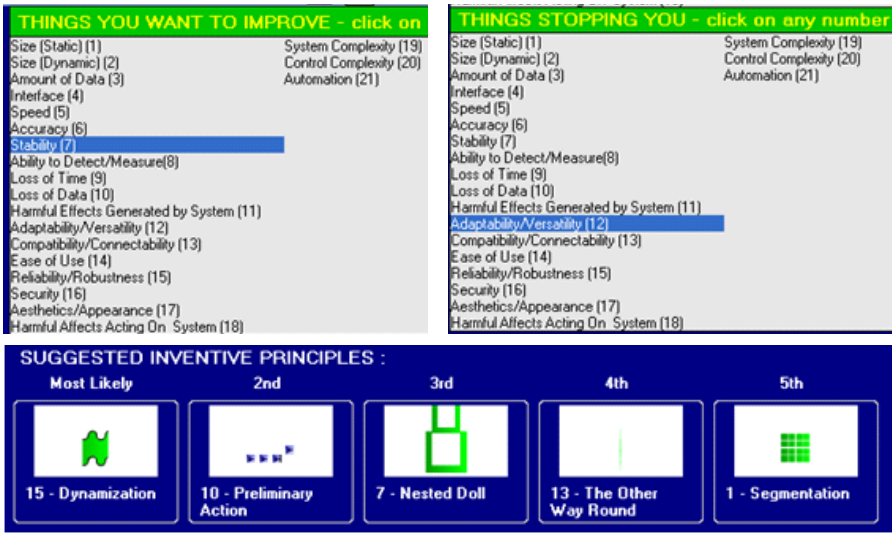

Figure 9: Mapping The UAV Autonomy Problem As A Contradiction

As is so often the case, the current system may be seen to have hit a contradiction. The conflict in this case is centres around the desire for predictable behaviour when the local environment is subject to considerable variation. Figure 9 illustrates the outcome of mapping this conflict pair onto the Software Contradiction Matrix (Reference 8)

Given the high degree of common ground between solving contradictions and discontinuous trend jumps, Figure 10 illustrates a system-level Evolution Potential plot for the UAV control system.

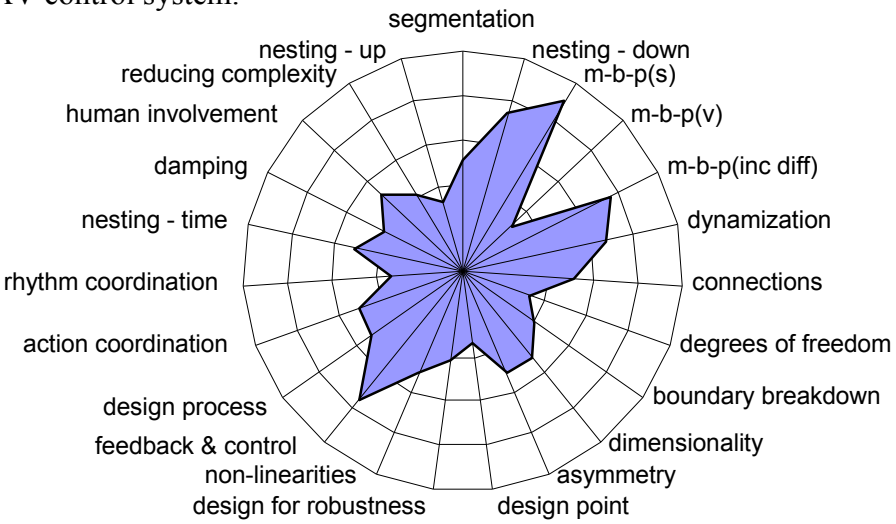

Figure 10: Reconnaissance UAV Macro-Level Evolution Potential

Combining the output from the Contradiction analysis and the Evolution Potential analysis shows a strong emphasis on Nesting as a strategy used by others in similar conflict situations. We also see it as an area of un-exploited potential in the current system. Neither the presence of Inventive Principle 7, Nested Doll from the contradiction analysis nor the untapped potential in each of the Nesting trends, however, tells us whether our UAV control problem is more likely to be solved by nesting either upwards into the super-system or downwards into the sub-system. The domain specialists in this situation ought to look in both directions.

In this particular case, the most useful solution direction emerged when the specialists looked in the Nest-Up direction. The start of the conceptual solution came about by thinking about how different UAVs flying the same or similar missions could somehow be nested together, and information from one UAV could thus be nested into the databanks of others. The basic concept here is that if each UAV knew what was happening to other ones in the proximity it would be possible to share information and allow each aircraft to ‘learn’ much faster about changes to the environment and itself.

Thus, for example, if all of the UAVs in a flight experienced a similar shift in performance at around the same time, that would be indicative of a perturbation in the environment. By combining mission trajectory information and GPS position information, it further becomes possible to locate where the environment shifts in order to potentially then allow other UAVs flying in the vicinity to avoid that location.

If, on the other hand, just one aircraft experienced a shift in performance then that was far more likely to be a fault on the aircraft. That fault could further be diagnosed by then examining the rate of change of performance – blocked fuel filter, to take one extreme, causing a far gentler performance shift than one caused by an impact from a bullet.

Picking up more solution cues from both the Contradiction and from the untapped Evolution Potential, this basic Nest-Up solution concept was further expanded to include elements of Dynamization (adjacent UAVs only needed to talk to each other sporadically when a perturbation occurs), Preliminary Action (control systems could be 'trained' and cross-calibrated first in non-threatening environments, and then later in controlled 'seeded-fault' situations. Without wishing to get too far into detail, the main features of the eventually chosen solution can be seen in Figure 11, alongside the before-and-after Evolution Potential radar plot.

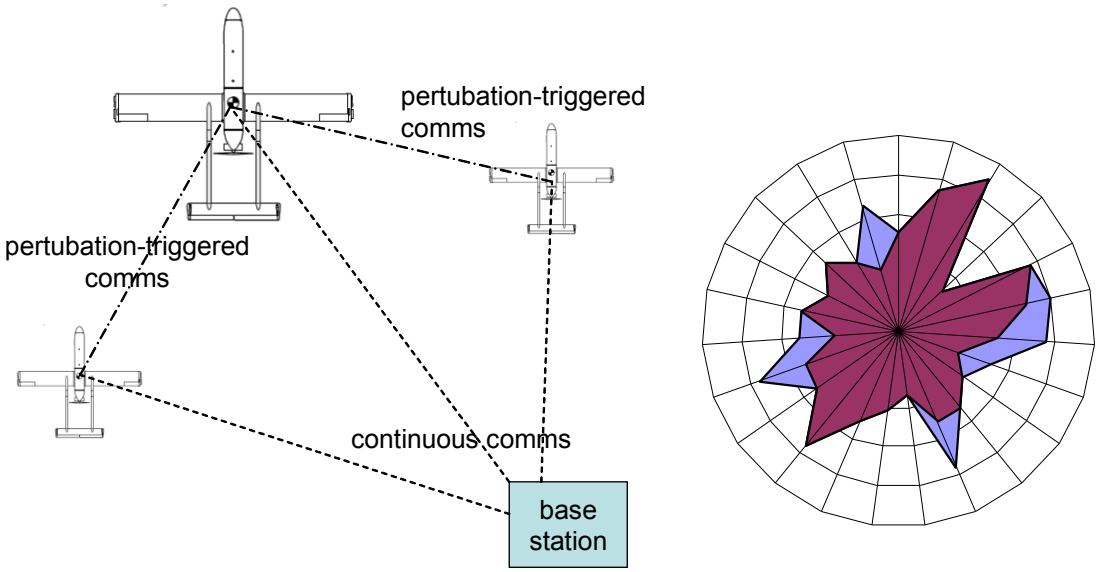

Figure 11: Schematic Solution For UAV Autonomy Problem

Summary & Conclusions

Software innovation has been defined as discontinuous jump in the direction of a more ideal system.

Based on the findings from the analysis of many thousands of software innovations meeting this definition, there are – so far – 26 patterns of discontinuous evolution jumps. These Trends have been designed to act as signposts pointing towards the more ideal system.

A historical analysis of macro-scale software innovations reveals a pattern of increasingly hierarchical systems, with increasing-decreasing complexity cycles occurring within each hierarchical level. There is no evidence to suggest that this evolution trajectory will not continue in the future.

Despite the rapid evolution of software systems, there remains considerable untapped potential when compared to the trend patterns uncovered during the ongoing research.

A specific software problem has been described and conceptual solutions have been developed using the software tools emerging from research into successful innovations in other software areas.

References

1) Wheeler, D.A., 'The Most Important Software Innovations', http://www.dwheeler.com/innovation/innovation.html, August 2001, revised January 2007.
2) Gamma, E., Helm, R., Johnson, R., Vissides, J., 'Design Patterns : Elements Of Reusable Object-Oriented Software', Addison Wesley, 1995.
3) May, M., 'The Elegant Solution: Toyota's Formula For Mastering Innovation', Simon & Schuster, 2007.
4) Systematic Innovation E-Zine, 'Space/Time/InterFace and Free/Perfect/Now', Issue 50, May 2006.
5) Mann, D.L., 'Systematic (Software) Innovation', IFR Press, 2007.
6) Systematic Innovation E-Zine, 'New Trends – 'Nest-Up' And 'Nest-Down'', Issue 51, June 2006.
7) Matrix+ Software, www.systematic-innovation.com.
8) Hawkins, J., Blakeslee, S., 'On Intelligence', Times Books, 2004.

About The Author

Darrell Mann actively researches in the development and exploitation of systematic innovation methods for both technical, software and business/social systems. After a career in Rolls-Royce leading R&D strategy programmes involving large, multi-disciplinary teams, he now consults for Fortune 500 companies all over the world in every industry sector. He has overseen strategic management implementation programmes in a number of companies and leads a team of researchers who's task is to be continually finding and integrating new best practice strategies into a coherent innovation culture framework. He is the author of over 600 journal papers, patents and patent applications as well as several books on systematic innovation. One day the book 'Systematic (Software) Innovation may actually get published.