Software and Inventive Ideation

Marlene Ross
260 Surrey Avenue
Ferndale, 2194, RSA
+27 84 811 3795
marlene.ross@sitwala.com

ABSTRACT
A plethora of creative thinking techniques and invention heuristics exist to guide problem solvers towards innovative solutions. One of the problems that one is faced with is the selection of the appropriate technique for a specific problem, since none of them covers the full spectrum of approaches towards problem solving. A recent model that attempts to capture the essence of all of these techniques and heuristics, facilitates a more generic approach. This model is leveraged to construct Ideation Domains for software development.

Keywords
Innovation, creative thinking techniques, invention heuristics, inventive principles, TRIZ

INTRODUCTION
Innovation is a concept that is important to software practitioners and researchers alike. For the former an innovative product could mean having the edge over competitors in the market, which usually translates into financial gain. For software researchers, innovation is what it is all about: finding a novel solution to a problem in order to make a contribution to the science of computing. *Inventive ideation* can be defined as creating a range of *innovative* ideas, alternatives and options that potentially meet the requirements of a problem.

What makes an idea innovative or creative? *Novelty* and *value* are recurring themes in the literature when evaluating ideas for their creativity. The idea or the approach that is followed must be new within its context. Boden captures it very eloquently: “Our surprise at a creative idea recognizes that the world has turned out differently not just from the way we thought it would, but even from the way we thought it could” [1]. However, without an application the idea is a mere eccentricity [1], therefore the idea must also have an impact or lasting influence, enhancing the quality of life in some way” [2].

The aim of this paper is to provide a short summary of the research activities in the field of creative thinking and invention heuristics in general, then to investigate related research in the software development domain and finally to present Ideation Domains for software development.

DELIBERATE CREATIVITY
Deliberate creativity, or the use of specific thinking techniques in order to improve the skills of people in creative problem-solving and invention, is a research topic that has received much attention over the past few decades, not only in academic circles, but also in popular literature.

Creative Thinking Techniques
Consider Edward de Bono’s books on lateral and parallel thinking. De Bono coined the term 'lateral thinking' in his book *The Use of Lateral Thinking*, published in 1967 [3]. De Bono defines lateral thinking as “methods of thinking concerned with changing concepts and perception”. The method encourages reasoning in way that is not immediately obvious and about ideas that may not be obtainable by using only traditional step-by-step logic. More recently he published a book on parallel thinking [4], which advocates applying multiple (possibly contradicting) trains of thought to the same problem. Arguments that are contradictory are not argued out, but are presented in parallel. The solution is based on the contributions of these multiple trains of thought. This approach to creative thinking has been popularised by De Bono’s Six Hats™ method [5].

Creative thinking methods can be divided into two broad categories, namely:

- *Linear or focusing techniques.* The problem space is explored incrementally (e.g. by using checklists or by sequentially tweaking individual parameters).

- *Random or intuitive techniques.* This is often done by generating remote or random analogies. These are viewed as intermediate stepping-stones (or ‘intermediate impossibles’) and the idea is that this step should be followed by a process of extracting a key principle, focusing on the differences or identifying any direct value of the random stimulation or provocation.

Invention Heuristics

An alternative approach towards creative problem-solving is to apply invention heuristics. These heuristics are based on experience, best practices and rules of thumb that have been acquired over several years. The most well-known methodology in this field is called TRIZ (pronounced ‘trees’). It was developed by Genrich Altshuller [6] and others between the late 1940's and 1980's. It is the Russian acronym for what can be translated as 'the theory of inventive problem-solving'. While working in the patent department of the Soviet navy, Altshuller surmised that it should be possible to derive some generic inventive principles by studying existing patents. His goal was to provide these generic inventive principles as a guide to find the ideas most likely to lead to innovative solutions.

Altshuller and his team studied over 40 000 patents to come up with 40 Inventive Principles (IPs) that may be used to manipulate 39 engineering parameters. One of the most prominent tools of the TRIZ methodology is a Contradiction Matrix (CM). It is based on the notion that there is often a trade-off between parameters. The matrix contains 39 rows and 39 columns representing the 39 engineering parameters mentioned earlier. Each row represents one of the engineering parameters to improve while the columns represent the parameters that could be adversely affected by improving that specific parameter. The entry in the cell at the intersection of the row and column contains the numbers of the Inventive Principles that could be applied to resolve the contradiction. Multiple IPs can be present in a cell and the order in which they appear indicates the frequency with which they have been identified in the patents that were studied. The original CM was based on mechanical engineering systems as is evident from the list of parameters in Table 1 [7].

1. Weight of moving object	21. Power
2. Weight of binding object	22. Waste of energy
3. Length of moving object	23.Waste of substance
4. Length of binding object	24. Loss of information
5. Area of moving object	25. Waste of time
6. Area of binding object	26. Amount of substance
7. Volume of moving object	27. Reliability
8. Volume of binding object	28. Accuracy of measurement
9. Speed	29. Accuracy of manufacturing
10. Force	30. Harmful factors acting on object
11. Tension, pressure	31. Harmful side-effects
12. Shape	32. Manufacturability
13. Stability of object	33. Convenience of use
14. Strength	34. Repairability
15. Durability of moving object	35. Adaptability
16. Durability of binding object	36. Complexity of system
17. Temperature	37. Complexity of control
18. Brightness	38. Level of automation
19. Energy spent by moving object	39. Productivity
20. Energy spent by binding object	-

Table 1: The 39 Parameters used in the TRIZ Contradiction Matrix

The Inventive Principles are given below in Table 2:

Inventive Principle	Description
1. Segmentation	1.1 Divide an object into separate independent parts or sections. 1.2 Make an object easy to put together and take apart. 1.3 Increase the degree of fragmentation or segmentation.
2. Taking out	2.1 Take out an undesired part or function of the object. 2.2 Take out the cause or carrier of an undesired property or function.
3. Local quality	3.1 Change the structure or environment of an object from uniform to non-uniform. 3.2 Make each part of an object function in conditions most suitable for its operation. 3.3 Make each part of an object fulfill a different and useful function.
4. Asymmetry	4.1 Change the shape from symmetrical to asymmetrical. 4.2 If an object is already asymmetrical, increase the degree of asymmetry.
5. Merging	5.1 Bring closer together identical or similar objects, assemble similar parts to perform parallel operations. 5.2 Make operations parallel, bring them together in time.
6. Universality	6.1 Make an object that performs multiple functions, thereby eliminating the need for multiple objects.
7. Nested doll	7.1 Put one object inside another. 7.2 Allow one object to pass through a cavity in the other (telescopic effect).
8. Anti-weight	8.1 To counter the weight of an object, merge it with others that provide lift. 8.2 To compensate for the weight of an object, make it interact with the environment to provide buoyancy, etc.
9. Preliminary anti-action	9.1 Where an action has both harmful and useful effects, replace it with anti-actions to control the harmful effects. 9.2 Create actions or stresses beforehand in an object that will oppose known undesirable actions or stresses later on.
10. Preliminary	10.1 Perform the required change of an object (either fully or partially) before it is needed. 10.2 Pre-arrange objects such that they can come into action at the most convenient place and not losing time for their delivery.
11. Beforehand cushioning	11.1 Prepare emergency means beforehand to compensate for the potentially low reliability of an object.
12. Equipotentiality	12.1 In a potential field, limit position changes.
13. ‘The other way round’	13.1 Use an opposite or inverse action to solve the problem. 13.2 Instead of the action dictated by the requirements, implement the opposite action. 13.3 Make movable objects fixed, and fixed objects movable.

Inventive Principle	Description
	13.4 Turn the object or process 'upside down'.
14. Spheriodality	14.1 Instead of rectilinear parts, surfaces or forms, use curvilinear ones. 14.2 Use rollers, balls, spirals, and domes. 14.3 Change linear motion to rotary motion, use centrifugal forces.
15. Dynamics	15.1 Allow or design characteristics of an object, environment or process to change to be optimal or find the optimal operating condition. 15.2 Divide an object into parts capable of moving relative to each other. 15.3 If an object or process is rigid, make it movable or adaptive.
16. Partial, satiated or excessive action	16.1 If 100% of the objective is hard to achieve using a given solution or method, use 'slightly less' or 'slightly more' of the same method.
17. Another dimension	17.1 Move an object in two or three-dimensional space. 17.2 Use a multi-storey arrangement rather than single-storey. 17.3 Tilt or re-orientate the object, lay it on its side. 17.4 Use another side of a given area.
18. Mechanical vibration	18.1 Cause an object to oscillate or vibrate. 18.2 Increase or change the frequency of vibration, or use its resonant frequency. 18.3 Use piezoelectric vibrators instead of mechanical ones.
19. Periodic action	19.1 Replace continuous actions with periodic or pulsating actions. 19.2 If an action is already periodic, change the magnitude or frequency of periodic actions.
20. Continuity of useful action	20.1 Make all parts work at full load, all the time. 20.2 Eliminate idle or intermittent actions or work.
21. Skipping	21.1 Conduct a process, or certain stages (e.g. harmful or hazardous operations) at very high speed.
22. 'Blessing in disguise'	22.1 Use harmful factors (e.g. harmful or hazardous operations) to achieve a positive effect. 22.2 Eliminate primary harmful action by adding another harmful action to resolve the problem. 22.3 Amplify a harmful factor to such an extent that it is no longer harmful.
23. Feedback	23.1 Introduce feedback to improve a process or action. 23.2 If feedback is already used, change its magnitude or influence in accordance with operating conditions.
24. Intermediary	24.1 Use an intermediary carrier article or process. 24.2 Merge one object temporarily with another (which can easily be removed).
25. Self-service	25.1 Make an object serve or organize itself by performing auxiliary helpful functions. 25.2 Make an object perform supplementary or repair operations. 25.3 Use waste resources, energy or substances.
26. Copying	26.1 Use simple and inexpensive copies instead of unavailable, expensive, fragile objects. 26.2 Replace an object or process with an optical copy.

Inventive Principle	Description
	26.3 If visible copies are used, move to infrared or ultraviolet copies.
27. Cheap short-living objects	27.1 Replace an expensive object with a multitude of inexpensive objects, compromising certain qualities (e.g. service life).
28. Mechanical substitution	28.1 Replace a mechanical means with a sensory (optical, acoustic, taste or smell) means. 28.2 Use electric, magnetic and electromagnetic fields to interact with the object. 28.3 Change from static to movable fields. 28.4 Use fields in conjunction with field-activated (e.g. ferromagnetic) particles.
29. Pneumatics and hydraulics	29.1 Use gases and liquids instead of solid parts. 29.2 Use Archimedes forces to reduce the weight of an object. 29.3 Use negative or atmospheric pressure. 29.4 A spume or foam can be used as a combination of liquid and gas properties.
30. Flexible shells and thin films	30.1 Use flexible shells and thin films instead of 3-D structures. 30.2 Use flexible and thin films to isolate an object from its environment.
31. Porous materials	31.1 Make an object porous or add porous elements. 31.2 If an object is already porous, use the pores to introduce a useful substance or function.
32. Colour changes	32.1 Change the color of an object or its external environment. 32.2 Change the transparency of an object or its environment. 32.3 In order to observe things that are difficult to see, use coloured additives, or luminescent tracers.
33. Homogeneity	33.1 Make objects interact with a given object of the same material (or identical properties).
34. Discarding and recovering	34.1 Make portions of an object that have fulfilled their functions go away (discard, dissolve, evaporate, etc.). 34.2 Conversely, restore consumable parts of an object directly in operation.
35. Parameter changes	35.1 Change an object’s physical state (e.g. to a gas, liquid or solid). 35.2 Change the concentration or consistency. 35.3 Change the degree of flexibility. 35.4 Change the temperature, pressure, etc.
36. Phase transitions	36.1 Use phenomena that occur during phase transitions (e.g. volume changes).
37. Thermal expansion	37.1 Use thermal expansion (or compression) of materials. 37.2 If thermal expansion is used, use multiple materials with different coefficients of thermal expansion.
38. Enriched atmosphere	38.1 Replace common air with oxygen-enriched air. 38.2 Replace enriched air with pure oxygen.
39. Inert atmosphere	39.1 Replace a normal environment with an inert one. 39.2 Add neutral parts, or inert additives to an object.
40. Composite materials	40.1 Change from uniform to composite (multiple)

Inventive Principle	Description
	materials.

Table 2: Altshuller's 40 Inventive Principles

The TRIZ methodology has since been adapted to suit many other fields.

A Generic Model for Inventive Ideation

The problem with the plethora of creative thinking mechanisms and invention heuristics is exactly that: there are so many to choose from, and even when a particular technique is selected it is debatable whether it is the optimum selection for the specific problem. For example, if the TRIZ methodology is used, the random stimulation technique is not applied, because the methodology is based on the concept of tweaking individual parameters in an incremental fashion. If a random stimulation technique is used, one is left to wonder whether better results would not have been obtained if a more structured technique had been employed. In [2] a model is proposed that attempts to capture the essence of all of these techniques and heuristics in order to facilitate a more generic approach. It is demonstrated that the 10 mechanisms listed below underpin most of the creative thinking techniques and invention heuristics found in the literature.

Theme	Mechanism	Description
Separate	1. Segment	Increase the modularity of the object or make it segmentable. The resultant parts remain in the same place.
	2. Re-move-ment	This mechanism covers the concepts of removing and movement. Move a part away from the object, either temporarily (movement) or permanently (removing).
Change	3. Adjust	Incrementally change problem attributes.
	4. Distort	Deliberately change parameters outside their normal range to provoke thinking. The purpose is to create 'intermediate impossibles'. (This mechanism is particularly popular in lateral thinking.)
Copy	5. Associate	Produce an analogy that can be copied or borrowed from in order to solve the problem. Consider concepts that are associated with the features of the object.
	6. Random Stimulation	Random stimulation can be generated for example by randomly selecting words or pictures. It is generally more effective in 'fuzzy' problem areas that are broadly defined.
Combine	7. Re-arrange	Form new combinations or change traditional relationships between objects.
	8. Add	Add new features or functions.
Convert	9. Other – Use	Remove the object from its normal environment to fulfill a different function. Introduce another object to provide the same function.
	10. Transform	Transform the problem or its elements to a different domain in which novel insights may be gained to solve the problem.

Table 3: The Generic Mechanisms of Inventive Ideation

The diagram below visually conveys the following three features of the model [2]:

- *Frequency.* Some mechanisms are applied more often than others in the TRIZ methodology and other creative thinking techniques. This is indicated by the clockwise arrangement of the mechanisms, with the Adjust/Distort mechanisms being used most often and the Other-Use / Transform mechanisms being used least often.
- *Types of problems.* The top half of the model is predominantly concerned with temporal, physical and sensory attributes of the objects (e.g. colour, action, function and size). The lower half of the model is generally applied to problems that involve objects and their parts or their environment.
- *Metaphorical distance.* The distance of the mechanism from the centre of the model represents the metaphorical distance that the mechanism removes the thinking from the problem. For example when using the Adjust mechanism, attributes are tweaked

incrementally. When applying the Distort mechanism, the attributes are modified beyond their normal ranges.

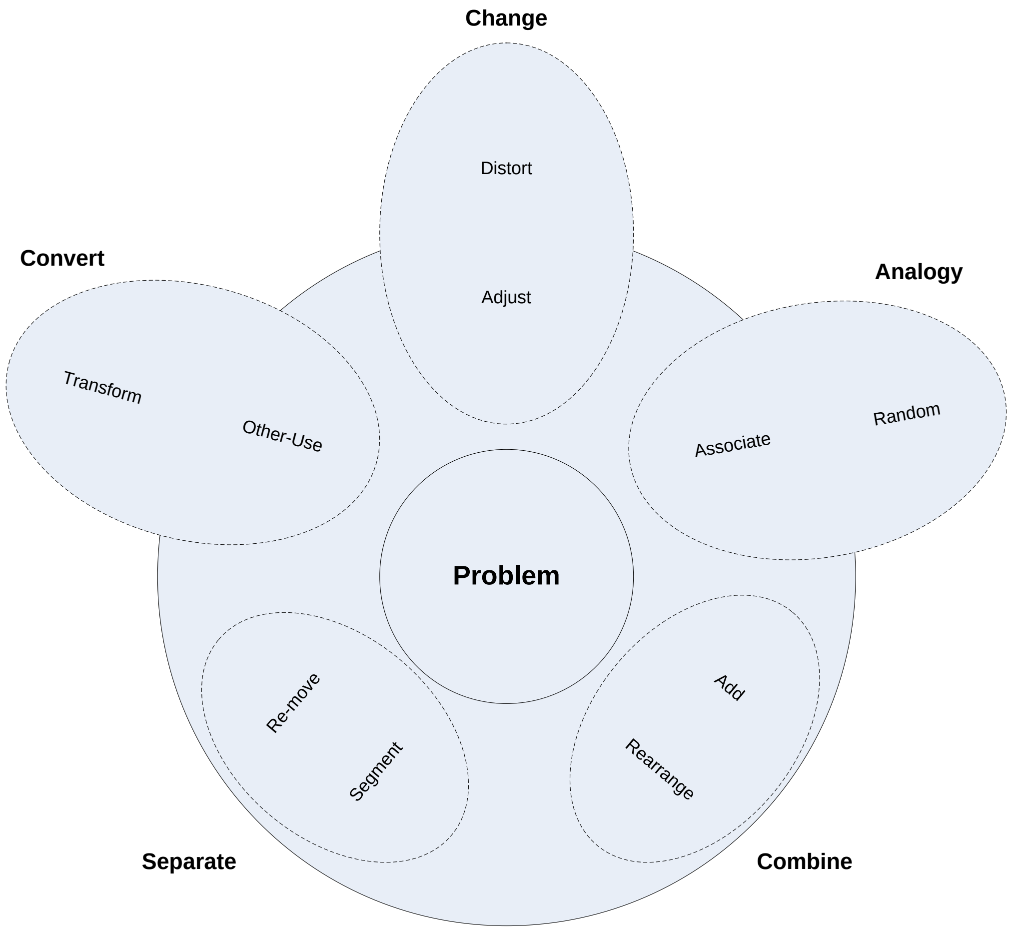

Figure 1: Generic Model for Inventive Ideation

The above generic model of mechanisms of inventive ideation has been integrated with a system model for physico-mechanical systems [2], i.e. it has been shown how these mechanisms could be applied to physico-mechanical systems.

SOFTWARE AND INVENTIVE IDEATION

This section discusses some of the research that has been done regarding inventive ideation in software systems.

Software Inventions

This topic begs the question: What is a software invention? One would think that the obvious way to find an answer is to analyse software patents. After all, the TRIZ methodology is based on the study of patents. (It is estimated that over the years more than two million patents in many industries and locations have been studied to amount to an effort of about 35 000 man-years![2]). However, there are a number of reasons why software patents do not provide a true representation of software innovation:

- Many software innovations occurred long before software could be patented [8].
- There is a large lobby against patenting of software, in particular those who engage in writing open source [9].
- The huge gap between the time that software programs came into existence and the time when software patents became available has caused in a lack of material at the Patent Offices, resulting in the granting of many patents that are

not truly novel [10]. Many examples of trivial software patents and ones that are not truly inventive can be found in [11].

Wheeler [8] offers the following definition of a software innovation: “it has to be a technological innovation that impacts how computers are programmed (e.g., an approach to programming or an innovative way to use a computer) “.
Many examples of software innovations can be found in [8], such as the Turing machine, the first assembler, the first compiler, the stack principle, semaphores, structured programming, object-oriented programming, spreadsheets, the use of the mouse, Graphical User Interfaces, etc.

Software and Invention Heuristics

In recent years a great deal of research has been done on how to apply the TRIZ methodology to software [12, 13, 14, 15, 16]. Instead of Altshuller's 39 parameters, Mann [14] identified 21 parameters for software, as presented in Table 4.

1. Size (static)	12. Adaptability / Versatility
2. Size (Dynamic)	13. Compatibility / Connectability
3. Amount of data	14. Ease of use
4. Interface	15. Reliability / Robustness
5. Speed	16. Security
6. Accuracy	17. Aesthetics / Appearance
7. Stability	18. Harmful effects on system
8. Ability to detect / measure	19. System complexity
9. Loss of time	20. Control complexity
10. Loss of Data	21. Automation
11. Harmful effects generated by system	-

Table 4: The 21 Parameters that constitute the sides of the Contradiction Matrix for Software [14]

Software analogies for the 40 IPs in TRIZ have been described in [12, 13, 14, 15]. Table 5 gives descriptions and examples of each principle. In the cases where the original principles were very system-specific, the new principles have been shown in italics.

Name [14]	Description [12, 13]	Examples [12, 13]
1. Segmentation	1.1 Divide a system into autonomous components. 1.2 Separate similar functions and properties into self-contained program elements (modules). 1.3 Increase the level of granularity until a known atomic threshold is reached. (The atomic threshold is the smallest structural unit of an object or component; e.g. bits can be thought of as atomic in the context of an encoding scheme.)	1.1 Intelligent agents can operate independently of each other, achieving a common goal. 1.2 C++ templates provide a means to containerize code so as to make the runtime execution of this code modular. 1.3 Fragmentation of Confidential Objects. This idea, based on object fragmentation at design time, is to reduce processing in confidential objects; the more non-confidential objects that can be produced at design time, the more application objects can be processed on un-trusted shared computers. The atomic threshold is where the confidential object is segmented to the point where it is no longer valid *as a confidential object.*
2. Extraction	2. Given a language, define a	2. Extraction of Text in Images. A

Name [14]	Description [12, 13]	Examples [12, 13]
	representation for its grammar along with an interpreter that uses the representation to extract/interpret sentences in the language.	text segmentation technique that is useful in locating and extracting text blocks in images. The algorithm works without prior knowledge of the text orientation, size or font. It is designed to eliminate background image information and to highlight or identify the regions of the image that contain text.
3. Local quality	3. Change an object's classification in a technical system from a homogenous hierarchy to a heterogeneous hierarchy.	3. Non-uniform access algorithms. In a wireless environment, information is broadcast on communication channels to clients using powerful, battery-operated palmtops. To conserve the usage of energy, the information to be broadcast must be organized so that the client can selectively tune in at the desirable portion of the broadcast. Most of the existing work focuses on uniform broadcast. However, very often, a small amount of information is more frequently accessed by a large number of clients while the remainder is less in demand. Using the local quality principal, non-uniform algorithms can be developed that predict the suitable access behavior for a particular operation.
4. Asymmetry	4. Change the asymmetry of a technical system in order to non-uniformly affect a desired result of a computation.	4. Suppose we have balls and bins processes related to randomized load balancing, dynamic resource allocation, or hashing. Suppose n balls have to be assigned to n bins, where each ball has to be placed without knowledge about the distribution of previous places balls. The goal of the algorithm is to achieve an allocation that is as even as possible so that no bin gets much more balls than the average.
5. Combination	5. Make processes run in parallel.	5. Synchronize threads of execution in time. The synchronized primitive, the monitor, "consolidates" threads of different priority into a master arbitrator that determines which thread gets the processor and when.
6. Universality	6. Make a technical system support multiple and dynamic	6. Based on a user's login preferences a context exists as the

Name [14]	Description [12, 13]	Examples [12, 13]
	classifications based on context.	result of a need to make behavior specific. Depending on the situation or context, the technical system will show a characteristic identity, with contextual properties (or in general, contextual behavior).
7. 'Nested doll'	7. Inherit functionality of other objects by “nesting” their respective classes inside a base class.	7. Nested objects in object-oriented system. Objects reside inside other objects to enhance services and functionality; this takes place by “nesting” classes inside other classes at design time.
8. Counter-*balance*	8. Use sharing to support large numbers of fine-grained objects efficiently to counter dynamic loads on a technical system.	8. A shared object that can be used in multiple contexts simultaneously; it acts as an independent object in each context – it is indistinguishable from an instance of the object that’s not shared.
9. Prior counter-action	9. Perform preliminary processor actions in system that will improve a later computation.	9. Reverse lines of text before matching line-breaks to increase match pattern efficiency**.**
10. Prior action	10. Same as above.	10. The Java Virtual Machine prepares textual “code” into an intermediate form before executing it and/or compiling it to a machine-specific binary.
11. Prior cushioning	11. Use an algorithm that handles worst-case harmful effects and maintains global invariance.	11. Fair scheduling in wireless packet networks.
12. *Remove tension*	12. Change the operational conditions of an algorithm so as to control the flow of data into and out of a process.	12. A transparent persistent object store.
13. 'The other way round'	13. Store transactions in reverse order for backing out.	13. Recovery and backtracking systems (database).
14. *Loop*	14. Replace linear data types with circular abstract data types.	14. The bounded buffer data structure provides an unlimited storage mechanism for storing digital information such as program variables. This circular structure is similar to Altshuller’s circular runway analogy (except that his planes will get off the runway or get run over, likewise the programmer needs to ensure that valid data are used before the processor completes a write to the same location in the bounded buffer).
15. Dynamics	15. Same as above.	15. Dynamic Linked Libraries (DLLs).
16. *Slightly less / slightly more*	16. Increasing the performance of	16. When performance

Name [14]	Description [12, 13]	Examples [12, 13]
	measurable and deterministic computations by perturbation analysis.	measurements are made of program operations, actual execution behavior can be perturbed. For example, in synchronization, the measurement and subsequent analysis of synchronization operations (e.g., barrier, semaphore, and advance/await synchronization) can produce accurate approximations to actual performance behavior. Therefore, by using perturbation analysis, we can do slightly more or less to affect the performance output of our computation.
17. Another dimension	17. Use a multi-layered assembly of class objects instead of a single layer.	17. Aggregation of inherited objects towards a new arrangement of functionality.
18. *Vibration*	18. Change the rate of an algorithm execution in the context of time until the desired outcome is achieved.	18. This requires a visual analogy of periodically changing the rate of an algorithm on an object that in turn resonates the overall system to an ideal state.
19. Periodic action	19. Instead of performing a task continually, determine the time boundaries and perform that task periodically.	19. Scheduling algorithms (e.g., alert mechanisms, cron-jobs, replication events).
20. Continuity of useful action	20.1 Develop a fine-grained solution that utilizes the processor at full load. 20.2 Develop a fine-grained concurrent solution that eliminates all blocking processes and/or threads of execution	20.1 Near video-on-demand (NVoD) scheduling of movies of different popularities for maximum throughput and the lowest average phase offset. Continuity of video based on using buffering (e.g., Real Player or Windows Media Player). 20.2 Barrier synchronization solutions; read and write database transaction algorithms.
21. *Hurrying*	21. Conduct the transfer of data in a burst mode just before a worst-case scenario.	21. Using a burst-level priority scheme for bursty traffic in Asynchronous Transfer Mode (ATM) networks. Statistical gain is achieved in ATM networks by making bursty connections share resources stochastically. When connections with different Quality of Services (QOS) requirements share the same resources, the highest requirements would typically be the limiting factor in

Name [14]	Description [12, 13]	Examples [12, 13]
		determining the admissible load at a link. This may lead to connections with low QOS requirements getting better service than they require, leading to an underutilization of the resources. To alleviate this problem we need “rush-through” using a burst-level priority scheme. This scheme handles related cells in a network on a burst-by-burst basis. Bandwidth is allocated to bursts on-the-fly according to their priorities.
22. 'Blessing in disguise'	22. Inverse the role of the harmful process and redirect it back.	22. Defeating Distributed Denial of Service (DDoS) attacks. A DDoS attack saturates a network. It simply overwhelms the target server with an immense volume of traffic that prevents normal users from accessing the server. In contrast to other types of DoS attacks that operate on an individual basis, these distributed attacks rely on recruiting a fleet of “zombie” computers that unwittingly join forces to flood the victim server. The critical harm is because of the attack’s distributed nature. Attackers can exploit the Internet’s insecure and readily accessible channels to aggregate an enormous traffic volume that doesn’t infiltrate but effectively jams the secure channels. So in applying TRIZ we can convert harm (overloading of computers) into a benefit (decreasing the zombie’s effectiveness) by creating bottleneck processes on the zombie computers, limiting the attack ability; this could be done by requiring the attacking computer to correctly solve a small puzzle before establishing a connection. Solving the puzzle consumes some computational power, limiting the attacker in the number of connection requests it can make at the same time.
23. Feedback	23. Introduce a feedback variable in a closed loop to improve subsequent iterations based on qualifiers.	23. Rate-based feedback in an Asynchronous Transfer Mode (ATM) system. Closed-loop input rate regulation schemes have

Name [14]	Description [12, 13]	Examples [12, 13]
		come to play an important role in the transport of the Available Bit Rate (ABR) traffic service category for ATM. By modeling the feedback system as a finite Quasi-Birth-Death (QBD) process, the performance of a delayed feedback system with one congested node and multiple connections can be achieved.
24. Intermediary	24. Use a mediator to provide a view(s) of data to a process in the context of the processes application space.	24. Using mediators in conjunction with the eXtensible Markup Language (XML) to enhance semi-structured data. Mediation can be an important part of XML. In conjunction with a Document Type Definition (DTD), a mediator can assist another process; let's say a user interface in query formulation and query processing more efficiently.
25. Self-service	25. Same as above.	25. Symantec Update; this application periodically checks for updates of its applications; if there are new artifacts that need to be updated, a dependency graph is implemented and executed thus servicing the application.
26. Copying	26. Instead of creating a new object that takes unnecessary resources perform a shallow copy.	26. A shallow copy constructs a new compound object and then (to the extent possible) inserts references into it to the objects found in the original.
27. Cheap / short living	27. Same as above.	27. Rather than developing a full application out of a prototype causing *expensive* cost overruns, use Throwaway (or rapid) prototypes.
28. *Another sense*	28. Same as above.	28. Voice recognition alleviates the mechanical action of typing and mistyping and then backspacing.
29. *Fluidity*	29. -	29. -
30. *Thin and flexible*	30. Isolate the object from the external environment using wrapper objects.	30. A wrapper or adapter object isolates and object from its external environment by maintaining a fixed interface between the inner-object and the outer object (the wrapper object).
31. *Holes*	31. -	31. -
32. Colour changes	32. -	32. -
33. Homogeneity	33. Create pure objects of a certain type ensuring identical properties.	33. The container data object such as an array. Each array element MUST be of the same type

Name [14]	Description [12, 13]	Examples [12, 13]
		allowing for consistent write and read operations.
34. Discarding and recovering	34. Discard unused memory of an application.	34. The garbage collector process in the Java programming language periodically "cleans" up memory by discarding objects that have lived past their scope.
35. Parameter changes	35. Same as above.	35. A software application can be transformed to provide a different service based on properties changing dynamically. This flexibility allows for more multi-role objects in an application.
36. *Paradigm shift*	36. -	36. -
37. *Relative change*	37. -	37. -
38. *Enrich*	38. -	38. -
39. *Calm*	39. -	39. -
40. *Composite structures*	40. Change from uniform software abstractions to composite ones.	40. Software design patterns are the core abstractions behind successful recurring problem solutions in software design. Composite design patterns are the core abstractions behind successful recurring frameworks. A composite design pattern is best described as a set of patterns the integration of which shows a synergy that makes the composition more than just the sum of its parts.

Table 5: Interpretations and Examples of the 40 Inventive Principles as applied to Software Systems

APPLYING THE GENERIC MODEL OF INVENTIVE IDEATION TO SOFTWARE

The previous section has shown how the parameters and Inventive Principles used in the TRIZ methodology can be modified to suit software systems. As illustrated in the Table 5, in some cases there is no natural analogy in the software field. We suggest that instead of trying to force an analogy where there is no true match, one should rather try and find a more appropriate model. When considering the descriptions of the mechanisms in the generic model for inventive ideation given earlier, there is no need for any artificial manipulation to make it suit software problems. It can be used as is. It has the added advantage that it covers the whole spectrum of creative thinking techniques and invention heuristics. It now remains to be shown how this model can be applied to software systems.

Since the generic model is based on a set of 10 mechanisms that should be applied on the attributes of the system, one should first establish what the relevant generic attributes of software systems are. Once these attributes have been identified, a set of Ideation Domains are created. An Ideation Domain consists of a mechanism, the attributes on which the mechanism can be applied and the inventive principle(s) or creative thinking technique(s) that are suggested.

In order to determine the relevant attributes for a target system, one needs to create a system model containing the key system descriptors [2]. A software system model is derived as follows: Typically a software system has external interfaces towards its environment. At these interfaces, the software system could receive input (stimuli), resulting in actions being performed by the system (this could include output from the system towards its environment in various forms, e.g. messages sent out on a network, information displayed on a screen, etc.). The system itself could consist of multiple interacting objects that could in turn perform actions as a result of certain stimuli. There are also certain constraints within which the system and its objects have to operate. Our software system model therefore has the following key system descriptors and attributes:

Key system descriptors	Attributes	Notes
System	Robustness	The ability of the system to remain operational even when abnormal stimuli or abnormal sequences of stimuli are received.
	Availability	The availability of a system is the percentage of time that the system is operational. The Mean Time To Repair (MTTR) for a software system can be computed as the time taken to reboot or to switch over to a redundant system after a software fault is detected.
	Security	This includes access control, authentication and encryption aspects.
	Scalability	This covers the ease with which the system would be able to handle a larger number of users, more data, etc.
	Ease of use / "Look and feel"	This includes everything related to the user's interaction with the system. This could be visual, audio and tactile.
	Architectural complexity	This covers the structure of the system. It includes the interfaces of the subsystems.
	Control complexity	All aspects related to locus of control are covered here. For example, it includes algorithm design, the way in which concurrent modules interact, etc.
Environment	Type	The type of environment, which could include aspects such as the operating system and the development environment (e.g. compiler).
Stimulus	Order	This includes aspects such a synchronous/asynchronous operation.
	Duration	The stimuli can be periodic, continuous or once-off.
	Type	Various types of stimuli can be received e.g. messages (data), interrupts, etc.
Action	Accuracy / Functionality	This covers ways in which correctness of data is ensured (e.g. when dealing with concurrent systems), how the system meets functional requirements, the accuracy of timers, etc.
	Duration	The action performed could be periodic, continuous or once-off. An example of how this attribute can be tweaked is by processing only a subset of the elements of a large list at a time and sending a stimulus to itself to process the next subset. Instead of processing the complete list at once and preventing other interactions with the system during that time, the once-off duration is changed to a number or periodic durations.
	Order	This includes the order of any output of the system, as well as the order of execution of steps (the algorithm itself).
Object	Interface	The pre- and post-conditions for each type of interaction offered by the object are covered here.
	Extensibility	This covers the ease with which the object can be modified for added functionality.
	Compatibility / Interoperability	This includes ways to ensure interoperability or backwards compatibility with other objects.
	Symmetry	Aspects such as master / slave, client / server models, etc. are covered here.
Constraint	Space	A software system usually has to operate within certain hardware constraints regarding permanent storage, memory and /or bandwidth. It is therefore important to minimise static and dynamic memory utilisation, the amount of data transmitted / stored, etc.
	Time	A certain reaction time is usually expected of a software system. Furthermore, efficiency in terms of speed should be maximised.

Table 6: Key System Descriptors and Attributes of Software Systems

In order to construct the 10 Ideation Domains for software systems, the above attributes have to be considered for each mechanism in the generic model and suitable invention heuristics should be suggested. These invention heuristics are not necessarily Inventive Principles, but could also be creative thinking techniques. A separate table should be created for each Ideation Domain. In the interest of brevity, Table 7 below shows the parameters and the different types of mechanisms with the corresponding invention heuristics all in one table. Note that the heuristics in the table below are not exhaustive, but serve as an example only. They are also given at a high level only.

Key system descriptors	Attributes	Mechanism	Invention heuristic
System	Robustness	Remove	Ignore any out of sequence or invalid stimulus.
	Availability	Add	Add redundant modules to improve availability.
	Security	Distort, Add	Change parameters outside their normal range (the concept of a public key). Add information to secure the system, e.g. passwords.
	Scalability	Rearrange, Add	Restructure the system to remove bottlenecks. It may be required to add modules in order to achieve that.
	Ease of use / "Look and feel"	Add, Distort Associate	Add audio or tactile output to visual output, e.g. add audio indications of what is on the screen for visually impaired users. Distort the output, e.g. when the pointing device moves over an item it is enlarged and not just highlighted. Use analogies from other environments to come up with new user interfaces (see example in Table 8).
	Architectural complexity	Re-arrange	Use different criteria to structure the system.
	Control complexity	Remove, Add	Remove the concept of a program counter. Add conditions under which a statement will execute infinitely often.
Environment	Type	Adjust	Change the type of the environment, e.g. sequential to concurrent, or from single-user to multi-user.
Stimulus	Order	Adjust	Change the communication from synchronous to asynchronous or vice versa.
	Duration	Adjust	Change a continuous or once-off stimulus to periodic or vice versa.
	Type	Adjust	Adjust the contents of the stimulus so that it multiplexes information from multiple sources or vice versa if it is already multiplexed.
Action	Accuracy / Functionality	Remove	Remove redundant information to ensure data integrity.
	Duration	Adjust	Instead of performing a task continually or once-off, perform it periodically.
	Order	Rearrange	Store transactions in reverse order for backing out. Replace linear data types with circular abstract data types. Change the order of the output of the system if that will allow the user to make an early decision on whether to abort the remaining output.
Object	Interface	Other-Use	Use the same interface, but apply a different meaning (polymorphism).
	Extensibility	Add, Adjust	Extend an object by adding methods or overriding existing methods.

Key system descriptors	Attributes	Mechanism	Invention heuristic
	Compatibility / Interoperability	Add, Remove	Add a wrapper to emulate an existing interface for backwards compatibility / interoperability purposes. Ignore (remove) any elements in a received message that are not known (typically elements are added inside Protocol Data Units as the protocols evolve) to ensure that earlier versions will be compatible with later versions.
	Symmetry	Adjust	Change from a symmetrical to an asymmetrical architecture (e.g. master/ slave to autonomous objects)
Constraint	Space	Rearrange	Restructure data in order to minimise memory usage or reduce the amount of data transmitted over the network (messages going to the same destination can be multiplexed, saving on overhead).
	Time	Segment	Change the atomicity of execution in order to improve reaction time.

Table 7: Sample of Software Ideation Domains

For some mechanisms there might be Inventive Principles for many of the individual attributes, as demonstrated in Table 7.

For other mechanisms, such as the Associate mechanism, there is a basic technique that applies to all attributes. For this mechanism, the software designer always needs to find an analogy between the attribute and a similar attribute in other environments.

Word associations could be used to find analogies. For example, words such as 'stop', 'go' could lead to analogies like traffic lights and railroad signals. Table 8 shows examples of important software innovations of the past where the Association mechanism was evident.

Key system descriptors	Attributes	Examples of innovations resulting from the application of the Associate mechanism
System	Ease of use / "Look and feel"	• When Doug Engelbart read about the development of the computer, his exposure to radar screens during his time as a radar technician triggered the idea of having people sitting in front of cathode-ray-tube displays, "flying around" in an information space [17]. • Dan Bricklin was sitting in an MBA lecture and daydreamed about having a device where he could have a virtual image in front of him like in a fighter plane. He thought how convenient it would be if he could use a mouse to move around on the image to enter a few numbers, circle the relevant ones on which he wanted to do some calculations and then get the results. (He had seen a demonstration of a mouse some time before that.) This is how the idea of a spreadsheet was born [18]
	Architectural complexity	• Design patterns for software, published by Gamma, Helm, Johnson and Vlissides, are analogous to design patterns in architecture [19].
	Control complexity	• LISP was born when McCarthy realized that a program could itself be represented as a list [8]. • E. W. Dijkstra defined semaphores for coordinating multiple processes. The term derives from railroad signals, which in a similar way coordinate trains on railroad tracks [8].
Environment	Type	• The first Fortran implementation was completed in 1957. There were a few compilers before this point, but Fortran used notation far more similar to human notation [8].
Action	Accuracy / Functionality	• Ken Thompson embedded regular expressions (a concept which had been studied in mathematics) in the text editor *ed* to implement a simple way to define text search patterns [8].
Object	Compatibility / Interoperability	• The idea of standards for software occurred to people when they realised that compatibility / interoperability problems in other disciplines were solved by defining standards. One of the first standards to be defined was the ASCII code to represent characters as numbers [8].

Table 8: Examples of Software Innovations where the Associate Mechanism was evident

CONCLUSIONS AND FUTURE RESEARCH

This paper has explored possible ways in which creative thinking techniques and invention heuristics could be applied to software systems. Much of the current research in this area is being devoted to the adaptation of the TRIZ methodology to software systems. As was shown earlier, some of the Inventive Principles that have been defined for software systems seem rather forced. A more generic approach might therefore be more appropriate.

A second shortcoming of the TRIZ for software methodology is that it does not take advantage of the more radical creative thinking techniques such as random stimulation and lateral thinking.

The generic model for inventive ideation was found to be a very promising vehicle for facilitating a structured approach towards problem-solving without losing the benefits of creative thinking techniques such as random stimulation.

A small sample of software Ideation Domains was presented in the previous section to demonstrate how the mechanisms of the generic model can be applied to the attributes of software systems and result in useful invention heuristics. No artificial manipulation of the invention heuristics was necessary to make it suit software systems. This followed naturally because the basic model itself is generic.

Note that the work presented here represented only a small sample of the full spectrum of software Ideation Domains. More research is necessary to populate these domains completely. This will require an analysis of well-known and important software innovations and patents.

Applying the generic model for inventive ideation to software systems also addressed the second shortcoming of the TRIZ for software methodology. When examples of important software innovations from the past few decades were analysed, it was interesting that the associate mechanism was repeatedly identified as the mechanism that triggered the innovation. This confirmed the notion that the more novel ideas tend to be generated by the application of mechanisms that are outside the scope of the TRIZ methodology.

Another area that requires further research is the success rate of generating innovative ideas using this model. It is concluded in [2] that the application of the generic model of inventive ideation to physico-mechanical systems does indeed yield positive results. In [16] it is claimed that the use of the TRIZ methodology in a software case study also successfully produced innovative ideas. It would be very interesting to see whether empirical data on this topic would confirm these findings for the generic model when applied to software systems.

REFERENCES

1. Boden, M.A. The creative mind: Myths and mechanisms. Abacus Books, London, 1992.
2. Ross, V.E. A model for inventive ideation. Ph. D thesis, University of Pretoria, 2006.
3. De Bono, E. The use of lateral thinking. Jonathan Cape, London, 1967.
4. De Bono, E. Parallel thinking: from Socratic to De Bono thinking. Viking, London, 1994.
5. De Bono, E. Serious Creativity. Harper-Collins, New York, 1993.
6. Altshuller, G. S. To find an idea: Introduction to the theory of solving Problems of inventions. Nauka, Novosibirsk, USSR, 1986.
7. Savransky, S.D. Engineering of creativity: Introduction to TRIZ methodology of inventive problem solving. CRC Press LLC, Florida, 2000.
8. Wheeler, D.A. The most important software inventions. Available at http://www.dwheeler.com/innovation/, April 2008.
9. Tiller, R. Red Hat asks Federal Court to limit patents on software. Available at http://www.press.redhat.com/2008/04/07/red-hat-asks-federal-court-to-limit-patents-on-software/.
10. Webber, D.B. Software patents, in *Proceedings of Software Engineering Conference* (Sydney, Australia, 29 September – 2 October1997).
11. Foundation for a Free Information Infrastructure (FFII). European software patent horror gallery. Available at http://www.swpat.ffii.org/patents/samples/
12. Rea, K.C. TRIZ and software – 40 principle analogies, Part 1. Available at http://www.triz-journal.com/archives/, September 2001.
13. Rea, K.C. TRIZ and software – 40 principle analogies, Part 2. Available at http://www.triz-journal.com/archives/, November 2001.
14. Mann, D. TRIZ for software. Available at http://www.triz-journal.com/archives/, October 2004.
15. Mishra, U. The revised 40 principles for software inventions. Available at http://www.trizsite.com, July 2006.
16. Bhushan, N. Case study – Use of TRIZ in software design. Available at http://www.triz-journal.com/archives/, June 2008.
17. Engelbart, C. Alifetime pursuit. Available at http://www.bootstrap.org/chronicle/.
18. Bricklin, D. The idea. Available at http://www.bricklin.com/history/.
19. Gamma, E., Helm, R., Johnson, R. and Vlissides, J. Design patterns: Elements of re-usable object-oriented software. Addison-Wesley Publishing Company, 1995.