TRIZ and Software - 40 Principle Analogies, Part 1

Kevin C. Rea, Principle Consultant
REA Consulting
E-mail: kcronline@gmail.com
Web site: http://kevincrea.com/

Over the last few years, I have contemplated many ways that TRIZ could be used in Computer Science & Information Technology http://www.triz-journal.com/archives/1999/08/d/index.htm . Since then I have successfully used TRIZ concepts to solve interesting problems and have generated 13 patent submissions. Applying TRIZ to software problems still has a way to go, however, I hope to accelerate the application of this powerful methodology to information technology and other "software-related" problems through a series of articles for the TRIZ Journal.

In this paper, I present 20 of the 40 inventive principle analogies of TRIZ *in the context of software* and computing from my perspective; in other words, these analogies are works in progress and may be updated or appended at a later date (the remaining 20 will be published in a later issue of the TRIZ-Journal).

Background:

Genrich Altshuller developed the 40 Principles more than 20 years ago. He and his team of associates reviewed thousands of worldwide patents selected specifically from leading industries for the inventive nature of their solutions to technical contradictions. In particular, Altshuller was interested in investigating contradictions that were resolved *without* compromise.

Altshuller found that utilizing principles previously used to solve similar problems in other inventive solutions could solve technical problems. For example: a "wearing problem" in the manufacturing of an abrasive product, and a "wearing problem" with the cutting edge of a back-hoe bucket, were both solved utilizing the principle of **"Segmentation."** After discovering this correlation, Altshuller was able to identify 40 such principles from his analysis of successful inventions.

Altshuller's Inventive Principles [1] [2]**:**

1. Segmentation

a. Dividing an object into independent parts.

Software Analogy: *Divide a system into autonomous components.*

Software Example: Intelligent Agents. Intelligent agents can operate independently of each other, achieving a common goal.

b. Make an object modular.

Software Analogy: *Separate similar functions and properties into self-contained program elements (modules).*

Software Example: C++ templates. C++ templates provide a means to containerize code so as to make the runtime execution of this code - modular.

c. Increase the degree of fragmentation or segmentation.

Software Analogy*: 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).*

Software Example: 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 no longer valid, *as a confidential object.*

2. Extraction

a. Separate (extract) an interfering part or property from a technical system, or single out the only necessary part (or property).

Software Analogy: *Given a language, define a representation for its grammar along with an interpreter that uses the representation to extract/interpret sentences in the language.*

Software Example1: Extraction of Text in Images. A 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.

Software Example2: Parser. Parsing data refers to the process by which programming data input is broken into smaller, more distinct chunks of information that can be more easily interpreted and acted upon.

3. Local Quality

a. Change a technical system's structure from uniform (homogenous) to non-uniform; change an external environment (or external influence) from uniform to non-uniform.

Software Analogy: *Change an object's classification in a technical system from a homogenous hierarchy to a heterogeneous hierarchy.*

Software Example1: A non-uniform sampling method for character recognition. The method determines feature extraction as sampling features one-dimensionally in a direction perpendicular to a given line orientation. It samples features at non-uniform intervals so as to avoid misidentifying two lines in close proximity as a single line.

Software Example2: 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 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.

b. Make each part of a technical system fulfill a different and useful function.

Software Analogy: *Same as above.*

Software Example: Promote a data object to higher levels in a single index tree. For this example, I choose the context of spatial data in processing technology and data management. Non-uniformity in data extents is a general characteristic of spatial data. Indexing such non-uniform data using conventional spatial index structures such as R-trees is inefficient for two reasons: (1) the non-uniformity increases the likelihood of overlapping index entries, and, (2) clustering of non-uniform data is likely to index more dead space than clustering of uniform data. Using the TRIZ-way, we must look in our existing environment for fulfillment of useful functionality. To make the impact of these anomalies more “useful”, we invent a new scheme that promotes data objects to higher levels in tree-based index structures; these object then fulfill different functions based on the positional context in the index tree.

4. Asymmetry

a. Change the shape of a technical system from symmetrical to asymmetrical.

Software Analogy: *Change the asymmetry of a technical system in order to non-uniformly affect a desired result of a computation.*

Software Example: 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 that the average.

5. Consolidation

a. Make operations contiguous or parallel; bring them together in time.

Software Analogy: *Make processes run in parallel.*

Software Example: Synchronize threads of execution in time. The synchronized primitive, the monitor, “consolidates” threads of different priority into a master arbitrator that determined which thread gets the processor and

when.

6. Universality

a. Make a part or object perform multiple functions; eliminate the need for other parts.

Software Analogy: *Make a technical system support multiple and dynamic classifications based on context.*

Software Example: Based on a user's login preferences a context exists as the 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. Nesting (Matrioshka)

a. Place one object into another; place each object, in turn, inside the other.

Software Analogy: *Inherit functionality of other objects by "nesting" their respective classes inside a base class.*

Software Example: 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. Counterweight

a. To counter the weight of a system, merge it with other objects that provide lift.

Software Analogy: *Use sharing to support large numbers of fine-grained objects efficiently to counter dynamic loads on a technical system.*

Software Example: A shared object that can be used in multiple contexts simultaneously; it acts as an independent object in each context - it's indistinguishable from an instance of the object that's not shared.

9. Prior counteraction

a. Preload counter tension to an object to compensate excessive and undesirable stress.

Software Analogy: *Perform preliminary processor actions in system that will improve a later computation.*

Software Example: Reverse lines of text before matching line-breaks to increase match pattern efficiency**.**

10. Prior action

a. Perform, before necessary, a required change of an object (either fully or partially). Carry out all or part of the required action in advance.

Software Analogy: *Same as above.*

Software Example: The Java Virtual Machine prepares textual "code" into an intermediate form before executing it and/or compiling it to a machine-specific binary.

11. Cushion in advance

a. Prepare emergency means beforehand to compensate the relatively low reliability of an object.

Software Analogy: *Use an algorithm that handles worst-case harmful effects and maintains global invariance.*

Software Example: Fair scheduling in wireless packet networks.

12. Equipotentiality

a. In a potential field, limit position changes.

Software Analogy: *Change the operational conditions of an algorithm so as to control the flow of data into and out of a process.*

Software Example: *A transparent persistent object store* (Voltage is potential energy; data is potential information).

13. Do it in reverse

a. Invert the actions used to solve a problem (e.g., instead of cooling an object, heat it).

Software Analogy: *Store transactions in reverse order for backing out.*

Software Example: Recovery and backtracking systems (database).

14. Spheroidality

a. Replace linear parts with curved parts, flat surfaces with spherical surfaces, and cube shapes with ball shapes.

Software Analogy: *Replace linear data types with circular abstract data types.*

Software Example: *Bounded buffer*. 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 ran 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. Dynamicity

a. Allow or design the characteristics of an object, external environment, or process to change to be optimal or to find an optimal operating condition.

Software Analogy: *Same as above.*

Software Example: Dynamic Linked Libraries (DLLs).

16. Partial or excessive action

a. If 100 percent of a system is hard to achieve using a given solution method, the problem may be considerably easier to solve by using "slightly less" or "slightly more" of the same method.

Software Analogy: *Increasing the performance of measurable and deterministic computations by perturbation analysis.*

Software Example: When performance 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. Transition into new dimension

a. Difficulties involved in moving or relocating an object along a line are removed if the object acquires the ability to move in two dimensions (along a plane). Accordingly, problems connected with movement or relocation of an object on one plane is removed by switching to a three-dimensional space.

Software Analogy: *Use a multi-layered assembly of class objects instead of a single layer.*

Software Example: Aggregation of inherited objects towards a new arrangement of functionality.

18. Mechanical Vibration

a. Utilize oscillation.

Software Analogy: *Change the rate of an algorithm execution in the context of time until the desired outcome is achieved.*

Software Example: 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

a. Instead of continuous action, use periodic or pulsating actions.

Software Analogy: *Instead of performing a task continually, determine the time boundaries and perform that task periodically.*

Software Example: Scheduling algorithms (e.g., alert mechanisms, cron-jobs, replication events).

20. Continuity of useful action

a. Continue on actions; make all parts of an object perform UF and/or NF at full load, all the time (Flywheel stores energy when a vehicle stops, so the motor can keep running at optimum power).

Software Analogy: *Develop a fine-grained solution that utilizes the processor at full load.*

Software Example: Near video-on-demand (NVoD) scheduling of movies of different popularities for maximum throughput and the lowest average phase offset. Continuity of video based using buffering (e.g., Real Player or Windows Media Player).

b. Eliminate all idle or intermittent actions.

Software Analogy: *Develop a fine-grained concurrent solution that eliminates all blocking processes and/or threads of execution* [3].

Software Example: Barrier synchronization solutions; read and write database transaction algorithms.

Conclusion

I believe that we are only scratching the surface of what TRIZ can do for software problems. Along these lines, I have committed to start a multi-part series on applying TRIZ to software problems - this will begin in the November issue of the TRIZ Journal. Each series will address a particular problem area - starting first with information security and looking at the recent Code Red Worm <http://www.cert.org/advisories/CA-2001-19.html> that wreaked havoc on the Internet and how TRIZ can be used to prepare for the next inevitable worm.

About the author: Kevin C. Rea is the principal consultant with Global Platforms <http://globalplatforms.com/> Corporation, a company providing training and specialized IT/computing problem solving using TRIZ. He is also an 8-year veteran of -Lucent Technologies Bell Labs <http://www.bell-labs.com/> where he is a computer scientist for third generation wireless product development (UMTS). He holds a B.S. in Electronic Engineering, a M.S. in Computer Science and is in pursuit of a PhD in Computer Science - applying TRIZ to advanced computing problems.

Part One of Two: *TRIZ and Software - 40 Principle Analogies.*

Endnotes:

1. The Innovation Algorithm. Dr. Genrich Altshuller, Technical Innovation Center, Inc. July 2000.

2. Engineering of Creativity: Introduction to TRIZ Methodology of Inventive Problem Solving. Semyon D. Savransky, TRIZ_SDS@hotmail.com. CRC Press LLC 2000, ISBN: 0-8493-2255-3.
3. Using TRIZ in Computer Science Concurrency <http://www.triz-journal.com/archives/1999/08/d/>. Kevin C. Rea, The TRIZ Journal - August 1999.