TRIZ and Software - 40 Principle Analogies, Part 2

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

In this paper, I present the remaining 21 - 40 inventive principle analogies of TRIZ *in the context of software* and computing from my perspective; please see the September 2001 edition for the first twenty software analogies, located at:
http://www.triz-journal.com/archives/2001/09/e/index.htm

The software analogies presented here are by no means finite; these analogies attempt to stimulate the thinking process related to applying TRIZ to solve software problems. I am just connecting the dots.

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]:

21. Rushing through

a. Conduct a process or certain stages (e.g., destructible, harmful, or hazardous operations) at high speed.

Software Analogy: *Conduct the transfer of data in a burst mode just before a worst-case scenario.*

Software Example: 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 determining the admissible load at a link. This may lead to connections with low QOS requirements getting better service then 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. Convert harm into benefit

a. Eliminate the primary harmful action by adding it to another harmful action to resolve the problem.

Software Analogy: Inverse the role of the harmful process and redirect it back.

Software Example: 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

a. Introduce feedback (referring back, cross-checking) to improve a process or action.

Software Analogy: Introduce a feedback variable in a closed loop to improve subsequent iterations based on qualifiers.

Software Example: Rate-based feedback in an Asynchronous Transfer Mode (ATM) system. Closed-loop input rate regulation schemes have 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.

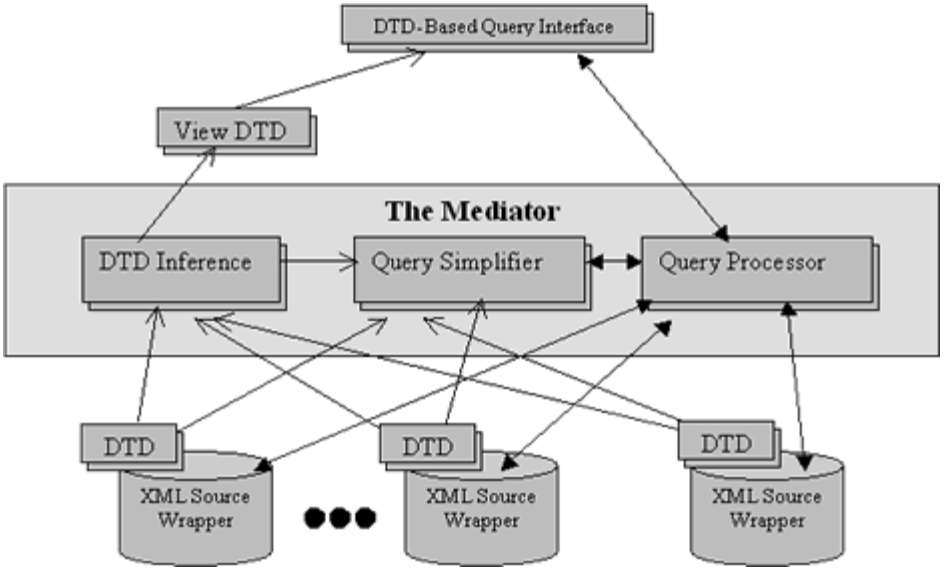
24. Mediator

a. Use an intermediary carrier article or intermediary process.

Software Analogy: Use a mediator to provide a view(s) of data to a process in the context of the processes application space.

Software Example: 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; lets say a user interface in query formulation and query processing more efficiently. The following is a picture of this example.



25. Self-service

a. Make an object serve itself by performing auxiliary helpful functions.

Software Analogy: Same as above.

Software Example: Symantec Update; this application periodically checks for updates 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

a. Instead of an unavailable, expensive, fragile object, use simpler and inexpensive copies.

Software Analogy: Instead of creating a new object that takes unnecessary resources perform a shallow copy.

Software Example: 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. Dispose (Inexpensive short-lived objects)

a. Replace an expensive object with multiple inexpensive objects, comprising certain qualities (such as service life, for instance).

Software Analogy: Same as above.

Software Example: Rather than developing a full application out of a prototype causing *expensive* cost overruns, use Throwaway (or rapid) prototypes. **Throwaway**

(or rapid) prototypes:

- are built as quickly as possible, without proper engineering,
- *implement only requirements that are poorly understood,*
- are used to learn which alleged requirements are real and which are not,
- are THROWN AWAY after the desired information is learned.

28. Replacement of Mechanical System

a. Replace a mechanical means with a sensory (optical, acoustic, taste, or olfactory) means.

Software Analogy: Same as above.

Software Example: This is a straightforward example, voice recognition alleviates the mechanical action of typing and mistyping and then backspacing like I am now.

29. Pneumatic or hydraulic construction

a. Use gas and liquid parts of an object instead of solid parts (e.g., inflatable, filled with liquids, air cushion, hydrostatic, hydro reactive).

Software Analogy: None at this time.

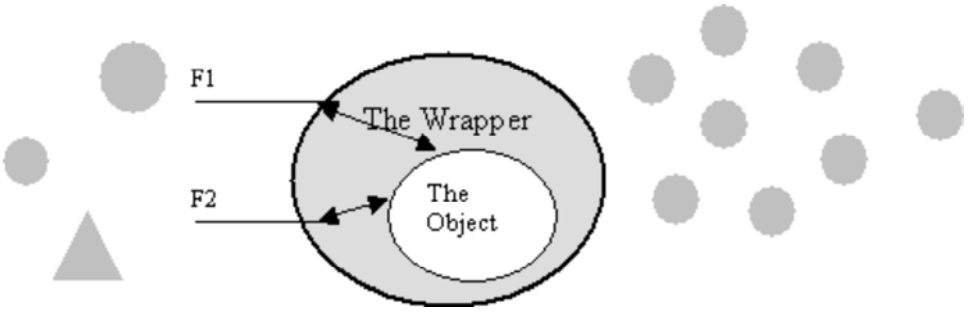
Software Example: NA

30. Flexible films or thin membranes

a. Isolate the object from the external environment using flexible shells and thin films.

Software Analogy: Isolate the object from the external environment using wrapper objects.

Software Example: 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. Porous materials

a. Make an object porous or add porous elements (inserts, coatings, etc.).

Software Analogy: None at this time.

Software Example: NA.

32. Changing the color

a. Change the transparency of an object or its external environment.

Software Analogy: Same as above.

Software Example: A transparency function in a photo or drawing program.

33. Homogeneity

a. Make objects interacting with a given object of the same material (or material with identical properties).

Software Analogy: Create pure objects of a certain type ensuring identical properties.

Software Example: The container data object such as an array. Each array element MUST be of the same type allowing for consistent write and read operations.

34. Rejecting and regenerating parts

a. Discard portions of an object that have fulfilled their functions or modify these directly during operation.

Software Analogy: Discard unused memory of an application.

Software Example: The garbage collector process in the Java programming language, periodically “cleans” up memory by discarding objects that have lived past their scope..

b. Conversely, restore consumable parts of an object directly in operation.

Software Analogy: Same as above.

Software Example: Backtracking in databases allows for an application to restore (or backtrack) to earlier transaction states.

35. Transformation properties

a. Change the degree or flexibility.

Software Analogy: Same as above.

Software Example: 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. Phase transition

a. Use phenomena that occur during phase transition (e.g., volume changes, loss or absorption of heat, etc.).

Software Analogy: None at this time.

Software Example: NA.

37. Thermal expansion

a. Use thermal expansion (or contraction) of materials.

Software Analogy: None at this time.

Software Example: NA.

38. Accelerated oxidation

a. Replace common air with oxygen-enriched air.

Software Analogy: None at this time

Software Example: NA.

39. Inert Environment

a. Replace a normal environment with an inert one.

Software Analogy: None at this time.

Software Example: NA.

40. Composite materials

a. Change from uniform to composite (multiple) materials.

Software Analogy: Change from uniform software abstractions to composite ones.

Software Example: 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. This paper presents examples of composite patterns, discusses an analysis and composition technique, and demonstrates that composite patterns extend the pattern idea from single problem

solutions to object-oriented frameworks.

Conclusion

The analogies presented are just some of many bridges that can be used while observing software problems with TRIZ in mind. While some of the physical principles may appear to be very distant to the application of software, one needs to remember that software can be viewed with a great deal of abstraction. Likewise, we may find hidden analogies as we apply TRIZ more frequently to software problems.

About the author: Kevin C. Rea is the principal consultant with Global Platforms Corporation, a company providing structured innovation training and specialized IT/computing and military solutions using TRIZ. He is also an 8-year veteran of Lucent Technologies - Bell Labs 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.