Anmerkungen zum Interdisziplinären Gespräch am 5.2.2021

Hans-Gert Gräbe, 11.02.2021

Bereits in der Ausschreibung zu unserem Interdisziplinären Gespräch wurde deutlich gemacht, dass wir uns weniger auf die Gültigkeit des einen oder anderen Entwicklungsgesetzes technischer Systeme fokussieren wollten, die in TRIZ-Theoriekorpus auf die eine oder andere Weise formuliert und priorisiert werden, sondern besser verstehen wollten, welche Stellung ein derartiger Komplex von Aussagen im TRIZ-Theoriekorpus hat und wie diese Stellung genauer zu begründen ist.

Das war bereits Thema unseres Seminars im zu Ende gegangenen Wintersemester, in dem wir Zugänge verschiedener TRIZ-Schulen zu den Entwicklungsgesetzen genauer studiert hatten, wobei uns insbesondere Unterschiede im Begriff „Gesetz“ bereits *innerhalb* der Wissenschaft – etwa zwischen Mathematikern und Physikern – aufgefallen sind. Auf dieses Thema ging *Nadine Schumann* noch einmal in ihrer Präsentation ein.

Ins Zentrum der Diskussion schob sich aber noch einmal (nach unserem Seminar im Wintersemester 2019/20) die grundsätzliche Frage, von welchem *Systembegriff* (im Singular oder im Plural) auszugehen ist, wenn zu Entwicklungsgesetzen von technischen und „nicht-technischen“ Systemen argumentiert wird. Das „nicht-technisch“ ist hier in Quotes gesetzt, da auch für Management-, kulturelle, ökonomische und sozio-ökologische Systeme gilt, dass diese nach ingenieur-technischen Prinzipien gesteuert werden sollen, in denen ein Realitätsabgleich in Feedbackschleifen zwischen „vernüftiger“ Modellierung als Beschreibungsform und praktischer Realisierung als Vollzugsform erfolgt.

Den Auftakt dazu gab die Präsentation von *Nikolay Shpakovsky*, der eine Hierarchie von Systembegriffen bereits in der TRIZ-Theorie selbst diagnostizierte – es sei deutlich zu unterscheiden zwischen

- A pure technical system (as a hammer) provides a function.
- A potentially workable system offers a service (design time).
- A really working system executes a process (run time)
- und einem System, das ein *Produkt* mit einem gewissen Gebrauchswert liefert (die Marktperspektive, welche von der Herstellerperspektive abstrahiert)

In der Diskussion dazu wurde deutlich, dass ein solcher konzeptioneller Zugang nur für Systeme bis zu einer gewissen Komplexität funktioniert. Ein solchere „Überschaubarkeitsaspekt“ ist ingenieur-technischen Zugängen allerdings einbeschrieben, denn Modellierungen für eine wirkmächtige Steuerung kommen ohne eine solche Komplexitätsreduktion nicht aus, auf welchem Level der Abstraktion sich diese auch immer bewegen mag.

Die Frage zunehmender Unübersichtlichkeit kann im Softwarebereich besonders gut studiert werden, worauf sich die Präsentationen von *Hans-Gert Gräbe* und *Stelian Brad* fokussierten. In den letzten 50 Jahren wechselten in diesem Bereich mehrfach die Ebenen, auf denen abstrahiert, modelliert und programmiert wurde.In den 1970er Jahren waren noch Lochkarten- und Lochbandsysteme mit einer direkten Adressierung von Speicheroperationen und prozessorspezifischen Assemblersprachen verbreitet. Das dort auftretende Problem zu großer Komplexität wurde durch die Einführung von Programmiersprachen auf höherem Abstraktionsniveau gelöst, wozu drei wesentliche konzeptionelle Zutaten erforderlich waren:

1. Ein allgemeines Hochsprachenkonzept, dessen Artefakte sich in Assemblerprogramm übersetzen lassen.
2. Werkzeuge (Interpreter und Compiler), welche diese Übersetzungen für verschiedene Hochsprachen praktisch ausführen.
3. Automatisierte Ablaufumgebungen (Betriebssysteme), in denen sich hardwarespezifisch diese *Digitalisate* ausführen lassen.

Mit der *Java Virtual Machine* wurde Mitte der 1990er Jahre ein weiteres Komplexitätsproblem gelöst. Die Komplexität der bis dahin erforderlichen Vielfalt hardwarespezifischer Werkzeuge wurde durch die Einführung einer weiteren sprachlichen Abstraktionsebene zwischen Assembler und Hochsprache, den *Java Bytecode*, vereinfacht. Die umfangreichen Entwicklungswerkzeuge für Java und die entwickelten Standardbibliotheken arbeiten auf dieser „virtuellen Maschine“ zusammen, die hardwarespezifischen Werkzeuge verarbeiten nur noch den Bytecode.

Diese Standardisierungen im Gebrauch computersprachlicher Werkzeuge sind Ergebnis eines umfassenden Austausch- und Diskussionsprozesses einer Open Source Community, deren Wurzeln bis in die 1980er Jahre zurückgehen. Open Source ist hier nicht aus ideologischen Gründen wichtig, sondern aus rein funktionalen – die Erfahrungen anderer müssen studiert, ausgewertet und verallgemeinert werden, um das Puzzle zusammenzuetzen. Die Entwicklungserfordernisse sind inhärent kooperativer Natur, die mit Geheimhaltung und IPR verbundenen Aus- und Abgrenzungen wirken sich komplett kontraproduktiv aus. Nach 2000 hatten sich diese ingenieur-technischen Bedürfnisse bis in Managementkreise herumgesprochen, und Henry Chesbrough wird mit dem Titel des gleichnamigen Buchs 2003 die „Erfindung“ des Begriffs „Open Innovation“ zugeschrieben. Der Versuch der Überetragung jenes kulturellen Paradigmas in die kulturell vollkommen anders geprägte unternehmerische „Welt“ ist allerdings bisher nicht wirklich gelungen wie die schnell wechselnden Konzepte einer Innovation 2.0 bis 4.0 (Stelian Brad, Folie 16) zeigen.

Jene Welt gemeinsamer Programmierstandards hat allerdings nach 2000 zu einer neuen Welt von Unübersichtlichkeit geführt – lohnt es, jedes Programm selbst zu schreiben oder gibt es Optionen der Nachnutzung von schon Vorhandenem? Die neue Unübersichtlichkeit bezog sich nicht nur auf die Frage, was es alles schon gibt, sondern auch auf die Frage, wie einfach es ist, Dinge zusammenzufügen, die ursprünglich für dieses Zusammenfügen nicht gedacht waren. Dieses Problem von Software-Komponenten-Architekturen und Wege zu dessen Lösung stellte *Hans-Gert Gräbe* in seiner Präsentation vor. Er ging dabei auf Wege der Formung von Komponentenwelten ein, die sich um konkrete Kompositionskonzepte herum konstituieren (als Beispiel wurde auf CORBA eingegangen), sparte aber aus, dass sich dabei die Welt der Programmierer weiter parzelliert. Waren es vorher die Welten von in verschiedenen Programmiersprachen geschriebener Komponenten, die wenig kompatibel untereinander waren, so sind es nun die Welten verschiedener Plattformkonzepte innerhalb ein und derselben Programmiersprache. Höhe der Abstraktion und damit Beherrschung eines höheren Grads von Komplexität wird durch Spezialisierung erreicht – allerdings nicht durch domänenzentrierte Spezialisierung, sondern durch weitere Verfeinerung der Konzepte und Werkzeuge.

Mit dem .NET-Konzept gibt es inzwischen gewichtige Ansätze, diese auseinander driftenden Welten wieder enger zusammenzuführen. Die Beschreibung damit verbundener Management- und Prozesserfahrungen greift dabei immer stärker auf das Konzept eines (technischen, energetischen, software-technischen usw.) *Ökosystems* zurück. Hier treffen sich die Vision einer „culture of polycentric agile strategic alliances“ (Stelian Brad, Folie 17) mit dem „Triumph des Anarchismus“ und dem „Tod des Copyrights“, wie es Eben Moglen schon 1999 vorausgesagt hat.