Summary

This position paper addresses some practical concerns in software process evolution, drawing primarily from experiences in the U.S. Department of Defense (DoD) and TRW. It focuses on the following three position statements:

1. The phrase "software process" is semantically overloaded. This causes a good deal of confusion and wheel-spinning. A top-level software process taxonomy is provided to help clarify the situation.

2. Evolution concerns vary by process subclass. Sources of variation include stimuli for evolution, and the mix of technical and cultural considerations involved in implementing evolution.

3. No process is an island. The most serious software process difficulties arise when a particular software process is developed and executed in isolation from concurrently evolving related processes. Some approaches for addressing heterogeneous process interactions are proposed.

1. Software Process Taxonomy

In attempting to coordinate software process research, development, and standardization within DoD, a number of slowdowns were encountered due to confusion about the classes of process being addressed (e.g., trying to mandate corporate-wide process improvements via a contractual instrument whose scope of enforcement covered only a single project). The following taxonomy of software process classes and subclasses was evolved as a way of distinguishing the various process initiatives and solutions.

I. Software processes
IA. Software project processes
IA1. Acquisition and life-cycle management practices
IA1a. Process incentives
IA1b. Regulations, specifications, and standards (also IA2a)
IA2. Process tailoring guidelines
IA2a. Regulations, specifications, and standards
IA2b. Process-oriented environments (also IA3a)
IA3. Project process models
IA3a. Process-oriented environments
IA3b. Project practice elements (also IB1a)
II. Corporate software processes
IB1. Institutionalized practices
IB1a. Project practice elements
IB1b. Corporate process training and standards
IB1c. Personnel practices
IB2. Process monitoring and improvement
IB2a. Total quality management, continuous process improvement
IB2b. Process metrics
IB3. Product line management practices
IB3a. Domain engineering and reuse practices
IB3b. Component brokerage practices

Thus, in a DoD context, one can see that the MIL-STD-SDD (Software Development and Documentation) standards effort falls in subclass IA1b/IA2a (Regulations, specifications, and standards), addressing software acquisition practices and software process tailoring guidelines. The SEI Capability Maturity Model focus is primarily in subclasses IA3b/IB1a (Project practice elements) and IB1b (Corporate process training and standards) for levels 2 and 3, and in the IB2 subclasses (TQM, CPI, and metrics) for levels 4 and 5.

2. Software Process Evolution Concerns by Subclass

Acquisition process incentives. A major evolution driver is the need to keep pace with the evolution of related processes. For example, the primary reason for the failure of DoD-STD-2167A to break away from the document-driven waterfall model in DoD-STD-2167 was the fact that contractual progress payments remained tied to the production of documents.

Regulations, specifications, and standards. The major evolution drivers here are the lessons learned from previous regulations; and incompatibilities across co-evolving regulations. For example, major drivers for the MIL-STD-SDD effort are the lessons learned from previous standards, and the need to co-evolve with DoD standards for system engineering, specification practices, configuration management, work breakdown structures, and quality assurance; as well as with initiatives to revise DoD's top-level acquisition strategies and policies.

Process tailoring guidelines. Major evolution drivers here are changes in regulations and changes in preferred process drivers (e.g., software reuse). For example, another contributing factor in the failure of DoD-STD-2167A to reduce dependence on the waterfall model was the lack of concurrently-developed tailoring guidelines for its application to non-waterfall processes.

Process-oriented environments. Major evolution drivers here are changes in preferred processes and changes in environment technology. For example, the TRW Software Productivity System environment phaseout was due largely to difficulties in evolving it away from screen (vs. window) oriented workstations and waterfall-model assumptions. Another class of environment evolution reflects the need for the environment to evolve across a project's life cycle, as the project's functions, roles, and organizational responsibilities evolve.

Project process models. Major evolution drivers here are lessons learned from previous process model experience. An example is the evolution of the waterfall model from SAGE in 1956 to DoD-STD-2167 in 1985.

Project practice elements. Major evolution drivers here are the development of new project practice elements (e.g., unit development folders, Cleanroom testing) and changes in process models. For example, "preparing for Preliminary Design Review (PDR)" is considerably different for a document-oriented PDR than it is for a prototype-, executing-kernel-, and document-oriented PDR. Project practice elements also need to evolve across
a project's life-cycle. For example, change control works best if deeply delegated during development and provided with a flexibility option during integration and test.

**Corporate process training and standards.** Major evolution drivers here are changes in preferred processes, changes in external standards, and changes in training technology. Effective changes in corporate standards require a complementary training program, to ensure that the standards aren't just published and put on people's shelves. For example, TRW's establishment of a waterfall-oriented set of corporate software development policies in 1976-77 involved an integral two-phase training program (phase 1 was to train department heads to be able to train their department personnel), and a mandatory examination at the end.

**Personnel practices.** Major evolution drivers here are lessons learned from previous personnel practices, and new roles created by evolving corporate processes and practices (e.g., total quality management (TQM) and performer empowerment). Examples include evolution to dual or multiple career paths for software personnel, software reuse incentives, and flowdown of contract award fees to project performers.

**Process monitoring and improvement.** This is the heart and soul of process evolution for all classes. Major evolution drivers are new monitoring and improvement methods (e.g., TQM and corporate benchmarking) and new metrics drivers. Examples of the latter are software reuse, prototyping, risk management, and new process models: "percentage of requirements specifications incomplete" is a good progress metric for waterfall processes, but a poor one for evolutionary development.

**Product line management practices.** These are corporate processes involving product line definition, domain engineering, software reuse, and component brokerage, including component certification, cataloging, version control, and access control. Major evolution drivers are changes in corporate product lines, domain architectures, commercial component and brokerage capabilities, and product line technology. An example of the latter has been the need to evolve signal processing domain-specific software architectures through the introduction of vector processors, application specific integrated circuits, and massively parallel processors.

3. **No Process Is an Island**

A number of the software process evolution concerns in Section 2 involved mismatches between the directions of evolution of two or more process subclasses. At least as many software problems arise from mismatches in the evolution of software processes and related non-software processes: e.g., hardware/system life-cycle processes, and the organizational processes of software customers and users. A notable example was the Strategic Air Command (SAC) 465L command and control system, which colocated a number of users with the developer in New Jersey to ensure the system's user-responsiveness. Unfortunately, by the time the development was complete, the command and control concepts of operation at SAC Headquarters in Omaha had evolved to such an extent that 95% of the delivered software had to be scrapped.

A related point is that an organization's processes are conditioned on the organization's evolving culture. Thus, one encounters the process counterpart of Conway's Law for software products: "The structure of an organization's software-related processes reflects the structure of its overall organizational processes." A striking example is the difference in software processes between a company's entrepreneurial contract
This implies that, since organizations' cultures are heterogeneous, the need for process reconciliation between heterogeneous software-related processes is inevitable, much as we might like a world in which all organizations used the same software processes. Thus, support mechanisms for heterogeneous process reconciliation are very important. Example mechanisms include improved technology similar to heterogeneous software-component and database reconciliation technology: process frameworks with open interfaces, process wrappers and mediators, and process interface formalisms. They also include cultural support mechanisms such as TQM (aligning developer processes with customer and supplier processes) and negotiation processes; along with mixed technical-cultural support mechanisms such as collaborative process definition and execution capabilities.

Other mechanisms for facilitating co-evolution of multiple software and non-software processes include the process analogue of Parnas' information-hiding technique: to anticipate sources of change and to encapsulate these within one's process architecture. Another mechanism is a process context monitoring capability, to highlight changes in related processes which will impact the effectiveness of current software processes (a much-desired capability for systems like SAC's 465L). Our most fundamental need, though, is for a continual awareness -- especially when we get deeply immersed in software process engineering practice -- that our software processes are not islands, entire of themselves.