

# Position paper for APSEC '96, Panel on Reuse

## Title: Common Sense Reuse

Stan Jarzabek

Department of Information Systems and Computer Science  
National University of Singapore

Whether we believe in reuse-based approach to software development or not, we end up reusing anyway. In each project, we reuse application domain knowledge, solutions to problems, programming expertise, programming methodologies and languages, operating systems and compilers. If we do not believe that reuse is important, then why we pay good money to get skilled people on software projects? We hope they will bring all kinds of expertise to the project: problem solving and analytical skills (talent inherited in genes and developed by training), programming expertise, knowledge of methods and tools, etc. Though software development may still start with a blank sheet of paper, by reusing the skill and knowledge of those people, we hope to build a successful software product. The question, then, is not whether to reuse or not. The question is whether a conscious effort to extent the scope of reuse beyond an informal level can bring measurable improvements in software productivity and quality. Many companies that made such an effort tell us that the answer is yes.

Most people will agree that a company that has an inventory of its resources and assets can work more efficiently than a company that does not have such an inventory. Common sense tells us that it is better to know what we have than not to know. Having an asset inventory cannot harm but can bring benefits. Classification and documentation of a company's assets is a natural first step in extending the scope of reuse beyond the intuitive, uncontrolled and informal level of reuse. This stage of institutionalizing reuse may address all company's assets, not only software. Software assets may include project experiences, development methods, software tools as well as requirements and design specifications, test plans and code modules. A reuse framework can be built to allow the staff to examine and share the company's assets.

In my view, a common sense reuse of software inevitably leads to the concept of program families [1], also known as Domain-Specific Software Architectures (DSSA) [2]. Though reuse of isolated software components may be possible and beneficial (especially if components are reasonably big), most often we find out that it is difficult to integrate modules that were not designed to work together. The DSSA approach avoids this problem. Prior to software development, we analyze stable and variant characteristics of software systems we wish to build in a given application domain. Then, we build a common core (called a reference architecture [2]) that we can reuse across software systems in that domain. In the DSSA approach, we never start from scratch, but rather build software systems by extending and refining the reference architecture. We avoid integration problems, as major design decisions underlying the family of software systems are encoded into the reference architecture. We do not try to put together components that do not fit. Examples of successful DSSA solutions exist in a number of application domains such as compiler-compilers, user interfaces and database applications. Programming techniques exist to apply the DSSA approach in other well understood domains.

To adopt the DSSA approach, companies must invest in domain analysis and must change the software process. Will they decide to do that? Only a strong business argument can substantiate the DSSA approach. The effort must focus on a product line that is important to a company, that will last and in which a company plans to develop many similar systems. Under these conditions, the savings due to the DSSA approach will attract attention of decision makers and the effort will receive their support.

[1] Parnas, D. Designing Software for Ease of Extension and Contraction, IEEE Trans. on Software Engineering, March 1979, pp. 128-137.

[2] Tracz, W. DSSA: Pedagogical Example, ACM Software Engineering Notes, July 1995, pp. 49-62.