Editor: **Christof Ebert**
Vector Consulting Services
christof.ebert@vector.com

Product Line Engineering

Klaus Schmid and Eduardo Santana de Almeida

Product line engineering (PLE) is one of the few industry-ready methods to manage reuse and variability in a defined way and thus bring software development maturity to a more advanced stage. The goal is to deliver specific product variants with fast cycle times at a manageable life-cycle cost with a defined quality level. Many IT and software organizations have started PLE but fail in industrializing the concepts and thus do not achieve sustainable benefits. Authors Klaus Schmid and Eduardo Santana de Almeida look at current technology for modeling and managing variation and thus facilitate PLE. I look forward to hearing from both readers and prospective column authors about this column and the technologies you want to know more about. —*Christof Ebert*

MAYBE YOU'VE HAD this experience with system development: been there, done that. This is a common problem because a new system's design can be rather similar to a previous one, yet there are many subtle differences. The software industry, like many industries before it, realized that it must provide customers with well-adapted solutions. This problem is by no means new, but it becomes more relevant as companies increasingly specialize in terms of the products they offer and as more variants must be built.

Many have attempted to address this challenge, especially from the perspective of maximizing reuse. Companies who face this problem often start with an approach where they copy the existing solution and adapt it as needed (the so-called clone-and-own model). But this approach doesn't scale well, which triggered product line engineering (PLE) approaches in the early 1990s. Industrial practice always played a strong role in PLE, and has even led to an industrial PLE applications hall of fame (http://splc.net/fame.html), which includes both large companies, such as Boeing, Philips, Lucent, and Toshiba, and many small companies.

So what is product line engineering? PLE consists of two life cycles:[1]

- *Domain engineering* creates customizable software and provides such assets to individual projects.
- *Application engineering* uses this software as the basis for developing a final product.

In principle, each life cycle covers the whole range of software development activities. In addition, scoping, a form of requirements management, is responsible for determining to which life cycle incoming requirements should go (see Figure 1). Of course, in practice, these activities are often not clearly differentiated, and a single developer might switch among them.

0740-7459/13/$31.00 © 2013 IEEE

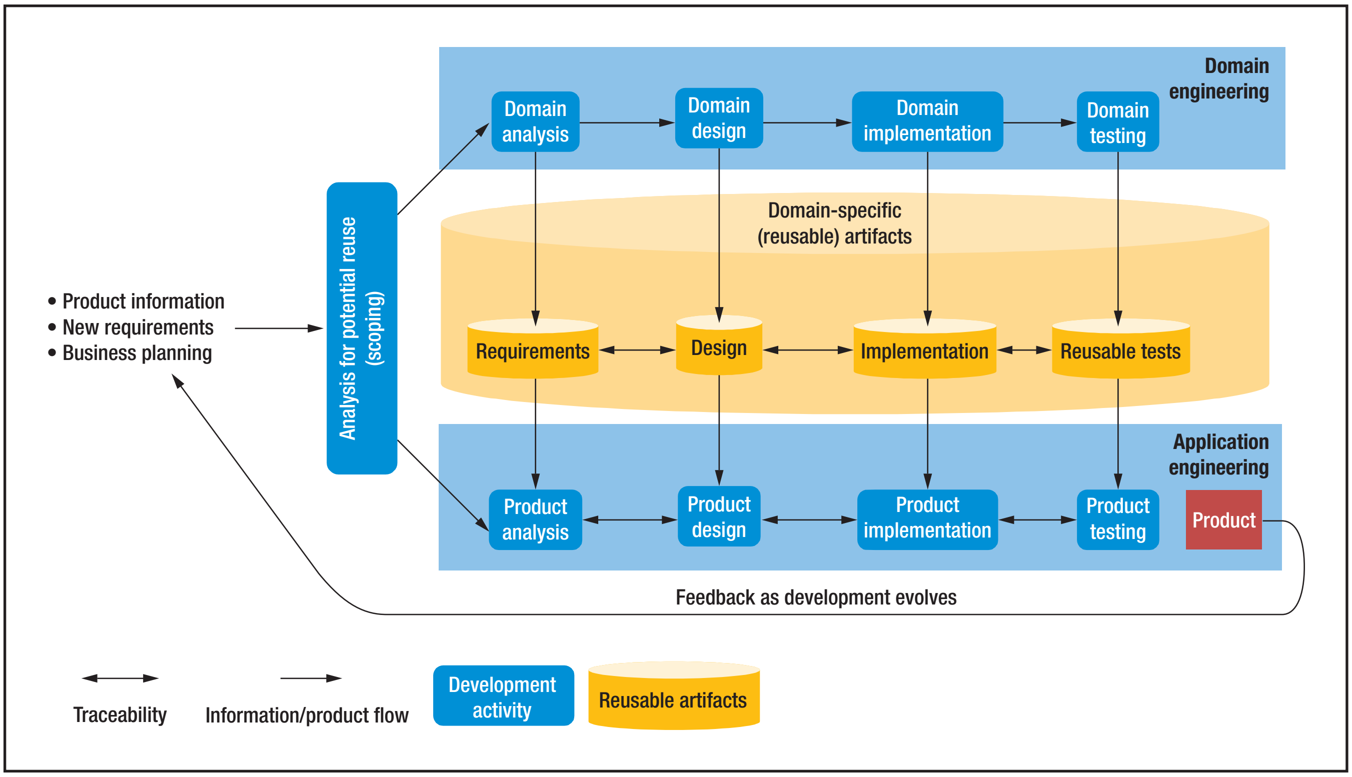


FIGURE 1. The two-life-cycle process model of product line engineering. The domain engineering life cycle creates assets that are reused in application engineering to build new systems.

As a whole, well-executed PLE impacts software development with processes, business strategies, organizational models, the main software engineering stages (as in Figure 1), and so on.

PLE's importance to industrial practice is emphasized by ongoing work on standards. At the International Organization for Standardization, a standardization effort is currently under way to provide guidelines for PLE at large,[2] and the Object Management Group is working on a standard for a variability modeling language that can be integrated with the UML (www.omgwiki.org/variability/doku.php).

Variability Management

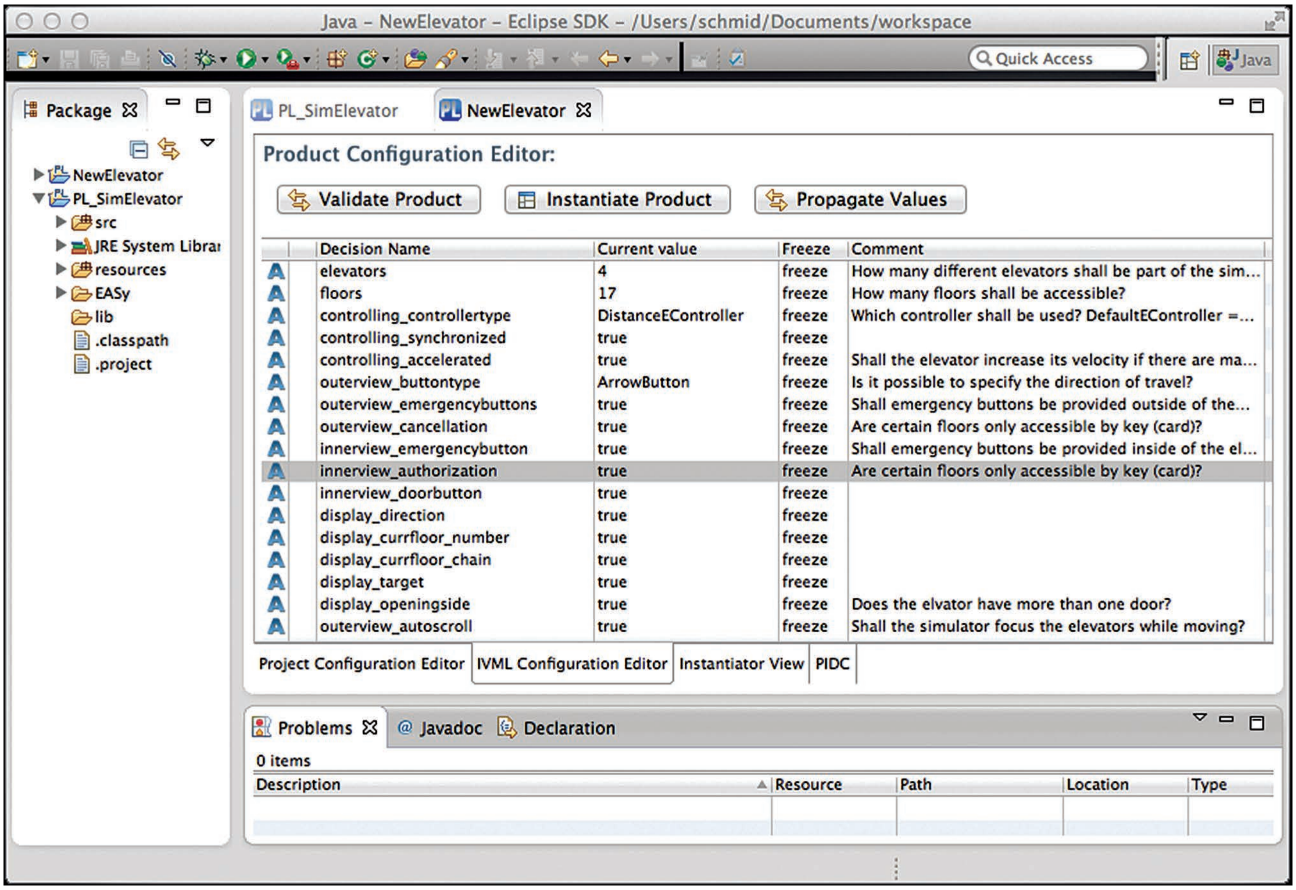
PLE tool support focuses almost exclusively on a single, cross-cutting aspect at the heart of PLE: variability management (VM), or making software and artifacts (such as requirements, tests, and documentation) configurable in a way that they can be jointly developed, while each product still receives its specifically adapted version. VM tools support four main activities:

- modeling variability,
- modeling the relationship between variability and a generic artifact,
- supporting configuration of generic artifacts, and
- deriving customized products.

The idea of VM is to identify and model characteristics that cause two products to differ (their *variabilities*). For example, for elevator systems, there might be differences in terms of how many stories a building has, whether multiple elevators can cooperate, whether the elevator call uses a single button or directional buttons, and so on. These characteristics—the basis of configuration decisions—are called *features* and are summarized in a variability model. Figure 2 shows an example of such a model created with EASy-Producer.

Typically, there are many dependencies among decisions, and not all configurations are valid—for example, our elevator example features constraints, such as the control strategy for cooperating elevators, which can only be used if there is more than one elevator. Constraints typically come from three main sources:

- Higher-level decisions constrain lower-level decisions; for example, freight elevators (business-level decision) use specific types of motors



Decision Name	Current value	Freeze	Comment
elevators	4	freeze	How many different elevators shall be part of the sim...
floors	17	freeze	How many floors shall be accessible?
controlling_controllertype	DistanceEController	freeze	Which controller shall be used? DefaultEController =...
controlling_synchronized	true	freeze	
controlling_accelerated	true	freeze	Shall the elevator increase its velocity if there are ma...
outerview_buttontype	ArrowButton	freeze	Is it possible to specify the direction of travel?
outerview_emergencybuttons	true	freeze	Shall emergency buttons be provided outside of the...
outerview_cancellation	true	freeze	Are certain floors only accessible by key (card)?
innerview_emergencybutton	true	freeze	Shall emergency buttons be provided inside of the el...
innerview_authorization	true	freeze	Are certain floors only accessible by key (card)?
innerview_doorbutton	true	freeze	
display_direction	true	freeze	
display_currfloor_number	true	freeze	
display_currfloor_chain	true	freeze	
display_target	true	freeze	
display_openingside	true	freeze	Does the elvator have more than one door?
outerview_autoscroll	true	freeze	Shall the simulator focus the elevators while moving?

FIGURE 2. An example of an elevator configuration using EASy-Producer. This demonstration system is rather simple and contains only a few decisions and constraints. Real-world systems can have many decisions (on the order of several thousands) that can also form complex hierarchies and can be governed by many constraints.

(technical decision).

- Domain properties might exclude certain combinations; for example, elevator team support isn't compatible with having a single elevator.
- The implementation might not support specific combinations, even if they're allowed in the domain.

Whatever the reason, constraint information is important because it encodes organizational knowledge about the domain and existing implementations. Either the developers or the PLE tools need to ensure that no misconfigurations will occur—they could lead to incorrect behavior, if the resulting system works at all. Here, configuration support can significantly improve development quality.

Configuration Support

Variability management tools automate the creation of product-specific artifacts from generic artifacts using the variability model and corresponding configuration information as input (see Figure 3). In Figure 2, we saw an example of such a variability model along with the corresponding configuration for a specific product (values were already entered). Generated artifacts can be requirements, code, tests, or even complete products.

Figure 4 shows an approach that applies this concept to IBM Rational DOORS to derive instantiated requirements, which are shown in Figure 5.[3] For illustration purposes, we chose an approach that describes the variability information directly within the requirements flow. For DOORS, this information is typically encoded as attributes. Of course, the important piece isn't the instantiation of a single artifact, but rather that a PLE tool is able to coordinate and execute all relevant instantiation activities across all different artifacts, from the requirements down to code and test (for example, see Figures 6 and 7). So, along with the product-specific requirements, we would get product-specific documentation, implementation, and tests. Of course, different kinds of underlying assets (such as different formats) require different customization technologies. Moreover, different approaches exist in terms of whether variable parts are contained as part of the overall generic artifact (as in Figures 4 and 6), configuration happens by removing some parts, or the variable parts are separate and mixed into common parts on demand (aspect-oriented techniques, for example, can be used for this). This shows, however, that in many industrial PLE cases, a large number of different techniques must be applied in a coordinated fashion. This brings about its own level of complexity, which corresponding tool support can address.

As Table 1 indicates, many tools support VM.[5] Traditionally, most tools were interactive; more recently, they've become increasingly textual, similar to special-purpose programming languages. To some extent, even multi-paradigm tools exist, which combine textual and interactive approaches. Another important distinction is the expressiveness of the language for configuration. The more expressive the language, the easier it is for the user to express all relevant concepts; however, this also leads to a more complex realization of the tool. From a practical perspective, it's important to know which artifacts a tool supports out of the box. Especially commercial tools provide a broad range of different instantiation mechanisms and connectors to other development tools.

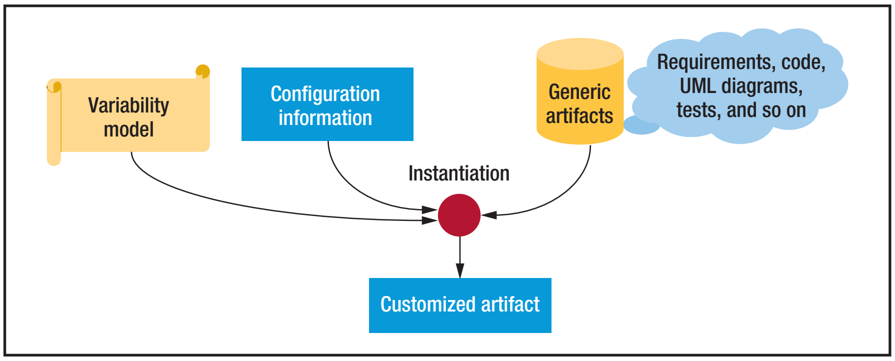
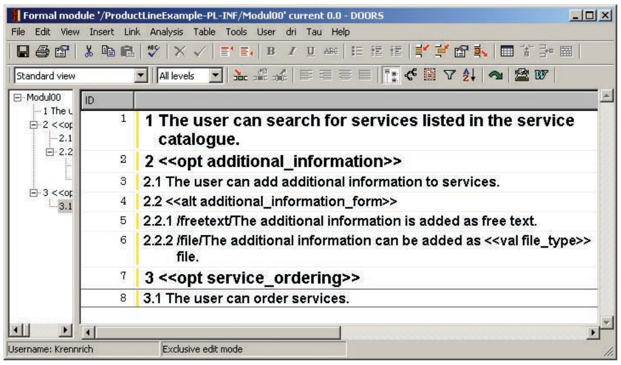

Variability model
Configuration information
Generic artifacts
Requirements, code, UML diagrams, tests, and so on
Instantiation
Customized artifact

FIGURE 3. The instantiation process. In product line engineering, generic artifacts that cover the demands of different configurations must be specialized to the needs described by a specific configuration; this is called *instantiation*.

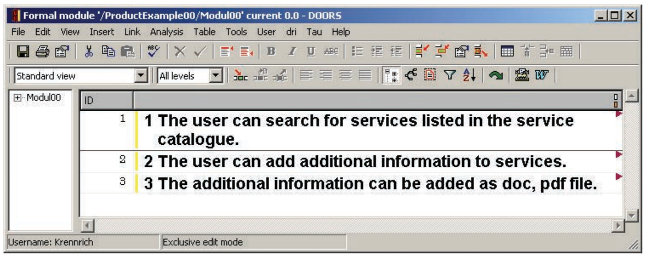

File Edit View Insert Link Analysis Table Tools User dri Tau Help
Standard view
All levels
Modul00
1 The u
2 <<op
2.1
2.2
3 <<op
3.1

Formal module '/ProductLineExample-PL-INF/Modul00' current 0.0 - DOORS

ID	
1	1 The user can search for services listed in the service catalogue.
2	2 <<opt additional_information>>
3	2.1 The user can add additional information to services.
4	2.2 <<alt additional_information_form>>
5	2.2.1 /freetext/The additional information is added as free text.
6	2.2.2 /file/The additional information can be added as <<val file_type>> file.
7	3 <<opt service_ordering>>
8	3.1 The user can order services.

Username: Krennrich Exclusive edit mode

FIGURE 4. An example of variability representation in DOORS.[4] There are several ways of mapping variability to requirements, including mappings to DOORS. The mapping shown here directly represents variability as requirements objects. (Others represent it, for example, in the form of requirements attributes.)


File Edit View Insert Link Analysis Table Tools User dri Tau Help
Standard view
All levels
Modul00

Formal module '/ProductExample00/Modul00' current 0.0 - DOORS

ID	
1	1 The user can search for services listed in the service catalogue.
2	2 The user can add additional information to services.
3	3 The additional information can be added as doc, pdf file.

Username: Krennrich Exclusive edit mode

FIGURE 5. An example of instantiated product requirements in DOORS.[4] This shows the instance of the generic requirements model given in Figure 4.

Modeling Complex Product Line Situations

Often, it's not sufficient to look at a single product line, but several infrastructures must be combined to derive the final products. Such situations are called *multi-software product lines*. Recently, even more complex supply chains came into focus. These can either be employed within a single company (typically in large companies) or across multiple companies, leading to the notion

```
Target target = new Target(self.getId(), (Integer) self.getValue());
/*
 * #if($controlling_synchronized == true)
 */
ElevatorSimulator.getInstance().getSuperController().addTarget(target, iElevatorIndex, false);
MainWindow.getInstance().highlightFloorButtons(id, true, iComp);
MainWindow.getInstance().highlightFloorButtons(id, false, Math.abs(iComp - 1));
/*
 *#else
 */
ElevatorSimulator.getInstance().getController(iElevatorIndex).addFloor(target);
setHighlight(icomp, true);
setHighlight(Math.abs(iComp - 1), false);
/*
 * #end
 */
```

FIGURE 6. An example of code with variability. One technique would be to use an **if-then-else** preprocessor construct in Java code. Here, velocity is used as the preprocessor (http://velocity.apache.org).

```
} else {
    Target target = new Target(self.getId(), (Integer) self.getValue());
    /*
     *                    /*
    ElevatorSimulator.getInstance().getSuperController().addTarget(target, iElevatorIndex, false);
    MainWindow.getInstance().highlightFloorButtons(id, true, iComp);
    MainWindow.getInstance().highlightFloorButtons(id, false, Math.abs( iComp - 1));
    /*
     *                    /*
}
```

FIGURE 7. An example of code after instantiation. The preprocessor directives have been resolved. Apart from instantiation, the code corresponds to the code in Figure 6.

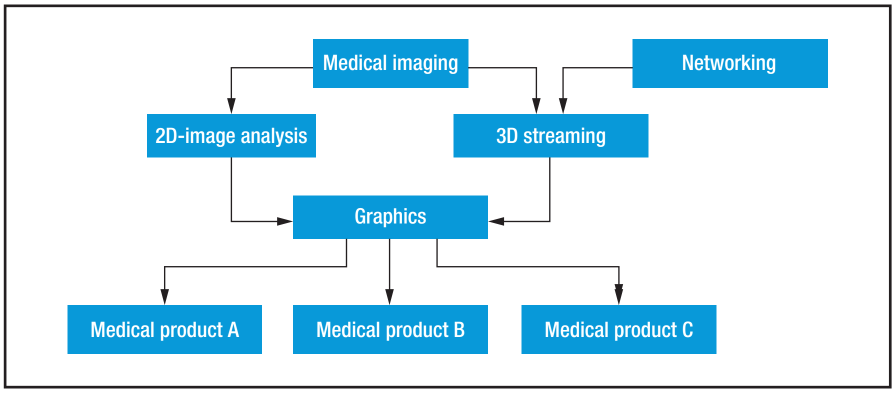


FIGURE 8. A complex product line/ecosystem situation. Different product line infrastructures can either be provided by different organizations within a single company or created by different companies.

of *software ecosystems*. Figure 8 shows such a situation for a hypothetical medical product line. Every box denotes a different product line infrastructure, and the results from one product line directly provide input to another. In this case, rather complex networks can occur in practice through a combination of product line composition and specialization. A consequence of such a situation is that all individual product lines have their own variability model, but these could also be dependent on each other. Here, tool support greatly enhances software production because all dependency management and production activities can be automated.

Although ecosystems are a generalization of PLE, few PLE tools address the ecosystems situation explicitly by providing product line composition and specialization mechanisms. One of these is EASy-Producer. This tool generalizes the existing notion of product lines in two important ways:

- Reusable infrastructure and final products are subsumed by the more

TABLE 1

An overview of some product line engineering tools.

Name	URL	I\T*	Type of configuration information handled	Type of artifacts supported	Multi-product line support	Commercial support	Comments
Dopler	http://ase.jku.at/dopler	I	Boolean, numbers, strings, sets, lists, compositions; constraints can be arbitrary Java code	Textual (arbitrary): Spring XML, Eclipse Plug-in Framework, Microsoft Word	Yes	No	Evolution support, infrastructure for tool integration, specialized tools for company-specific situations
EASy-Producer	www.uni-hildesheim.de/index.php?id=8035	I/T	Boolean, number, strings, sets, lists, compositions, references; constraints include quantifiers, explicit type system for variability, support of defaults, modeling and configuration integrated	Textual (arbitrary): several company-specific artifacts	Yes	No	Support of ecosystems, partial instantiation of artifacts, infrastructure for tool integration
FeatureIDE	wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide	I	Boolean; constraints are propositional formulas	Textual (arbitrary): AspectJ, C, C#, Java 1.5, Haskell, XML	No	No	Integration with several special frameworks and research prototypes
Feature Modeller (FM)	www.metadoc.de/anforderungsmanagement/werkzeuge/fm	I	Boolean; constraints are propositional formulas	Requirements information: DOORS integration	No	Yes	
GEARS	www.biglever.com/overview/software_product_lines.html	I	Boolean, strings, enumerations, numbers, sets, records; constraints are formulas	Textual, with many third-party connectors: Visual Studio, Eclipse, DOORS, Rational Rhapsody, Enterprise Architect, various CM systems	Yes	Yes	Support for multiproduct lines, infrastructure for tool integration
pure::variants	www.pure-systems.com/pure_variants.49.0.html	I	Boolean, strings, enumerations, numbers, sets, records, many special data types such as dates; constraints are prolog formulas	Textual, with many third-party connectors: DOORS, Rational Rhapsody, Enterprise Architect, Simulink, AUTOSAR, some CM systems	Yes	Yes	Eclipse integration, infrastructure for tool integration
Compositional Variability Management	www.cvm-framework.org	I/T	Boolean, numbers, strings, sets (cardinality); constraints with numerical relations	Textual (arbitrary): program code, component modeling, DOORS	Yes	No	

** Support for modeling variability can be interactive (I) or textual (T).*

general concept of a project with variability; thus, there's no longer a strong distinction between a product and the reuse infrastructure as basically as in all other tools. A product becomes the special case of a project with variability where all development time variability is resolved.
- EASy-Producer supports partial instantiation both for configurations and for artifacts.

For example, in Figure 8, 3D streaming could be derived from graphics by binding some, but not all, variability and, perhaps, adding some further functionality. Medical imaging would then be derived by composing (and instantiating) the two existing infrastructures, 2D-image analysis and 3D streaming. So, in this example, three different products are derived. The challenge is that the tool needs to keep track of all relevant artifacts throughout the whole ecosystem.

In principle, this can then be iterated to model rather complex supplier networks. The approach is inspired from industrial experience and is currently evaluated in several industrial projects. We hypothesize that tools will increasingly pick up these concepts.

Because the need to develop not only single products but whole sets of customized systems is an increasingly important need of companies, PLE continues to receive more attention. Industrial experience shows that related approaches can drastically reduce costs, time to market, defect density, and maintenance costs.[1] With increasing maturity in the area of software development, we expect that more and more companies will find themselves in situations where they produce variants of products that satisfy needs of specialized markets and customers. Thus, PLE, along with its corresponding tools, will become the norm in software engineering because it better fits the business strategies of modern software industry than more traditional project-focused development.

Acknowledgments
The Brazilian National Council for Scientific and Technological Development (CNPq) partially supported Klaus Schmid during the course of this work. The National Institute of Science and Technology for Software Engineering (INES), funded by CNPq and FACEPE, and FAPESB, also partially supported this work.

References
1. F. van der Linden, K. Schmid, and E. Rommes, *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*, Springer, 2007.
2. T. Käkölä, "Standards Initiatives for Software Product Line Engineering and Management within the International Organization for Standardization," *Proc. 43rd Hawaii Int'l Conf. System Sciences* (HICSS 10), IEEE CS, 2010, pp. 1–10.
3. K. Schmid, K. Krennrich, and M. Eisenbarth, "Requirements Management for Product Lines: Extending Professional Tools," *Proc. 10th Int'l Software Product Line Conf.*, IEEE, 2006, pp. 113–122.
4. K. Schmid and M. Schank, "PuLSE-BEAT: A Decision Support Tool for Scoping Product Lines," *Software Architectures for Product Families*, LNCS 1951, Springer, 2000, pp. 64–74.
5. L.B. Lisboa et al., "A Systematic Review of Domain Analysis Tools," *Information and Software Technology J.*, vol. 52, no. 1, 2010, pp. 1–13.

KLAUS SCHMID is a professor in software engineering and head of the Computer Science Institute at the University of Hildesheim. Contact him at schmid@sse.uni-hildesheim.de.

EDUARDO SANTANA DE ALMEIDA is an assistant professor in software engineering at the Federal University of Bahia and head of engineering at the Fraunhofer Project Center for Software and Systems Engineering. Contact him at esa@dcc.ufba.br.