Separating Concerns in Software Logistics

Danny Greefhorst

Software Engineering Research Centre
PO Box 424, 3500 AK The Netherlands
greefhor@serc.nl

Software logistics deals with the storage, administration, distribution and installation of software artefacts, from a full product life-cycle perspective. Software logistics is an important issue in system family engineering where management of common and variable assets is of utmost importance. This implies the need for an integral approach to variability, reaching from problem space to solution space, and leading to a good separation of concerns. This paper proposes a feature-driven approach to software logistics, enabling multidimensional variation, abstraction and traceability.

Introduction

Software logistics [Florijn99, Florijn00] deals with the storage, administration, distribution and installation of software artefacts. While from the perspective of software development, software (as the coding of the solution to a problem) is seen as the goal, from the logistics perspective, software is seen as an object of manipulation, storage and transport. Software engineering issues closely related to software logistics include product management, version management, derivation management, release management and configuration management. Central concepts in software logistics are *products*, *parts* and their accompanying *specifications*. Raising the abstraction level from individual artefacts (files) to products and the parts they consist of, shields the various stakeholders from details they do not have to deal with. Warehouses form fences between life-cycle activities, and can be seen as gatekeepers of the core assets (see figure 1).

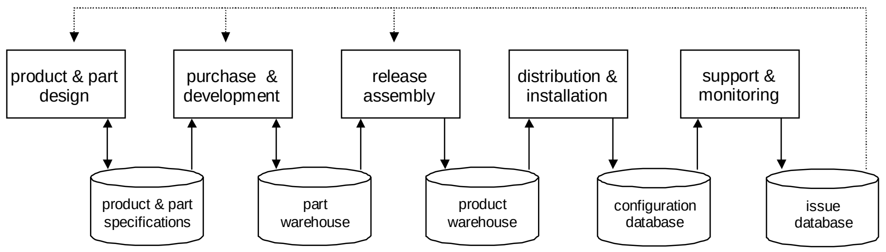

Figure 1 Software logistics processes and datastores

A *system family* [Jacobson97] is a collection of systems sharing a common set of features that address the specific needs of a defined domain. Commonality and variability are two important concepts when talking about system families. The first expresses the things all members in the system family have in common, while the second stresses their differences. The reason for developing system families is that it pays to develop all common aspects of highly related systems only once. These “core assets” consist of a domain model, a reference architecture and implementation components. To support these engineering processes all these assets must be stored and managed, implying the need for software logistics.

Commonality and variability can be expressed in terms of *features* (see [Kang90, Kang98]). A feature is a prominent or distinctive user-visible aspect, quality, or characteristic of a software system. Relationships can be discerned between features, such as composition, generalization and realisation. A feature diagram is a graphical representation of features and their relationships. Supplemented with additional information such as composition rules, issues, decisions, binding information, and priorities the feature diagram forms the feature model. Feature models can be defined at multiple levels of abstraction. At the requirements level the feature model can be seen as a high-level view on the requirements model. For manageability purposes, the feature model should only describe those features that the domain analyst deems important.

Feature-Driven Software Logistics

We propose a feature-driven approach to software-logistics, where the feature-model forms the central view on the system. Such an approach provides a number of benefits such as ease of communication, abstraction, traceability and multi-dimensional variation. Features abstract from requirements, which is important in system family engineering because by combining a number of systems in a family, the size and complexity of the requirements also increases, which severely hinders the detection of commonalities and variabilities in the family (also see [Bass99]). Traceability is provided by mapping features in the problem space to parts in the solution space. The most important reason for a feature-driven approach, is that it allows construction of systems by selecting from a potentially large space of features, also referred to as *multidimensional variation* [Conradi98].

An essential element in our feature-driven approach to software-logistics is the existence of a mapping from features to parts. Ideally, all features map one-to-one to parts, leading to a clean separation of concerns, which reduces software complexity and thus maintenance. The latter is not only due to a reduction in the number of locations in the software that have to be adapted, but also to a reduction of required rebuilds, redeployments and re-installations. To get some more grip on this mapping the mechanisms involved must be explored.

Variability mechanisms

Various mechanisms exist that help manage variability in the problem space and solution space. [Jacobson97] mentions inheritance, uses, extensions, parameterization, configuration and module interconnection-languages, and generation. This list can be extended with other mechanisms such as installation, reflection, naming and directory services, branching, and conditional compilation. Although not exactly a flexible approach, even traditional (hard-) coding can be seen as a variability mechanism. The disadvantage of most of these approaches is that features are not defined in one place, but instead affect (cross-cut) many parts of the system, leading to a management nightmare.

More recently, a new generation of aspect-oriented mechanisms has been introduced [Harrison93, Aksit96, Kiczales97, Prehofer97, Tar99]. These mechanisms solve the cross-cutting problem of features and other concerns by describing software parts such that they map one-to-one onto these concerns. Thus in these approaches, separation of concerns in the solution space enables multidimensional variation in the problem space.

What is missing from all this is a framework that relates all these mechanisms to one another, and puts them into perspective; not all mechanisms are appropriate for all situations. For that purpose it helps to reason about the variability activities these mechanisms support. In particular we distinguish between *addition*, *reduction* and *binding* of variability in both the problem space and the solution space.

Selecting a specific feature, and its accompanying realization, is commonly referred to as *binding*. Binding can occur at various moments in time such as design-time, construction-time, compile-time, installation-time, load-time and run-time. A distinction between static binding, changeable binding and dynamic binding is made in [Czarnecki00], indicating the moment at which the binding is made, and the easy with which it can be changed afterwards. Before variability can be bound, variants must first be *added* to the system. Finally, it is also possible to *reduce* the number of variability that was added earlier, by selectively excluding variants from the system. These activities can be used to reason about the aforementioned mechanisms, resulting in the following table which shows the applicability of activities to some of the mentioned mechanisms, and the moments in time at which they apply.

	Addition	Reduction	Binding
Uses	analysis-time		
Inheritance	design-time		
Traditional coding	construction-time		construction-time
Branching	construction-time		
Configuration Languages		assembly-time	assembly-time
Static parameterization			construction-time
Conditional compilation		compile-time	compile-time
Installation		installation-time	
Naming and directory services			run-time
Reflection			run-time
Aspect Oriented Programming	design-time		design-time

Table 1 Assessing variability mechanisms for variability activities

The given information helps understand the goal of the various mechanisms, and the moments in time at which they apply. It also helps to understand how they could be combined; mechanisms for addition can be supplemented with mechanisms for reduction and binding. Consider for example a system that defines variability at design-time using inheritance, reduces the number of variants using conditional compilation at compile-time, and finally binds the variant at run-time using reflection. The actual selection of the mechanisms can further depend on their specific advantages and disadvantages. Considerations could be the availability of tooling for the users environment, the required granularity of variability, the amount of memory a mechanism requires, or the performance penalty the approach incurs. In general, the later the mechanisms are applied, the more memory and performance it costs. These can be important considerations in embedded, possibly hard real-time environments. Advantages of later binding are the increased flexibility offered to the user; systems can even be composed at run-time [Schmerl97].

Conclusion

Using a system family approach to systems development can increase time-to-market through reuse of a great number of artefacts. Management of these artefacts throughout the entire lifecycle can however become quite complex, due to the potentially large amounts of systems, artefacts, and variability. We propose a feature-driven solution to software-logistics to support system family engineering, enabling multidimensional variation, abstraction and traceability. Various mechanisms for managing variability exist, and can be categorized according to the activities they support. The selection of mechanisms should be part of the architecture definition process, and should weigh the pros and cons of the various mechanisms. From a software-logistic perspective the mechanism should enable localization of a feature to one part, leading to a clean separation of concerns. Although aspect-oriented approaches exist, they may not be appropriate in all situations. Most importantly, having a framework for reasoning about variability gives us a chance to make well-considered decisions.

Related work

The feature-oriented domain analysis method FODA [Kang90] is extended in FORM [Kang98] to cover the entire spectrum of domain and application engineering, including the development of reusable architectures and code components. They promote general engineering principles such as separation of concerns, information hiding, localization of function, data, and control, parameterization of artefacts with features, and synthesis of design components based on feature selection.

A more elaborate discussion on feature modeling is given in [Czarnecki00]. Feature graphs are extended, and elaborate terminology on different kinds of features is presented. Generative programming is presented as a powerful variability mechanism.

A combination of the principles described in [Jacobson97], commonly referred to as the Reuse-Driven Software Engineering Business (RSEB), and FODA [Kang90] is described in [Griss98]. They propose refinements to the feature model, and position it as the central view on the architecture of the system.

A general introduction to feature-engineering is given in [Turner99]. The paper discusses the impact on different phases of the life cycle and provides ideas on how these phases can be improved by a feature-driven approach.

References

[Aksit96] M. Aksit, *Composition and Separation of Concerns in the Object-Oriented Model*, ACM Computing Surveys 28A(4), December 1996,

[Bass99] L. Bass, G. Campbell, P. Clements, L. Northrop, D. Smith, *Third Product Line Practice Workshop Report*, CMU/SEI-99-TR-003, 1999.

[Conradi98] R. Conradi, B. Westfechtel, *Version models for software configuration management*, ACM Computing Surveys, Vol. 30, No. 2, pages 232-282, June 1998.

[Czarnecki00] K. Czarnecki, U.W. Eisenecker, *Generative programming, Methods, Tools, and Applications*, Addison-Wesley, 2000.

[Florijn99] G. Florijn, *Software Logistics White Paper*, Software Engineering Research Centre, Utrecht, The Netherlands, 1999.

[Florijn00] G. Florijn, D. Greefhorst, H. Boer: *Softwarelogistiek, Levenscyclus software meer dan éénmalig ontwikkeltraject*, Software Release Magazine, Array Publications, June 2000.

[Griss98] M.L. Griss, J. Favaro, M. d'Allessandro, *Integrating Feature Modeling with the RSEB*, International conference on Software Reuse, Victoria, Canada, June 1998.

[Harrison93] W. Harrison, H. Ossher, *Subject-Oriented Programming (a critique of pure objects)*, Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), 1993.

[Jacobson97] I. Jacobson, M. Griss, P. Jonsson, *Software Reuse: Architecture, Process, and Organization for Business Success*, Addison-Wesley-Longman, May 1997.

[Kang90] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, A.S. Peterson, *Feature-Oriented Domain Analysis, Feasability Study*, SEI Technical Report CMU/SEI-90-TR-21, November 1990.

[Kang98] K.C.Kang, S.J.Kim, J.J.Lee, K.J.Kim, E.Shin, *FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures*, Annals of Software Engineering, Vol. 5, 1998.

[Kiczales97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopez, J.M. Loingtier, J. Irwin, *Aspect Oriented Programming*, European Conference on Object Oriented Programming (ECOOP), Springer-Verlag, LNCS 1241, Berlin, Germany, 1997.

[Prehofer97] C. Prehofer, *Feature-Oriented Programming*, European Conference on Object Oriented Programming (ECOOP), Springer-Verlag, LNCS 1241, Berlin, Germany, 1997.

[Schmerl97] B.R. Schmerl, C.D. Marlin, *Versioning and consistency for dynamically composed configurations*, 7th International Workshop on Configuration Management, Boston, May 1997.

[Tarr99] P. Tarr, H. Ossher, W. Harrison, S.M. Sutton, *N degrees of separation: Multi-dimensional separation of concerns*. In proceedings of the 1999 International Conference on Object-Oriented Programming, p. 107-119, May 1999.

[Turner99] C. R. Turner, A. Fuggetta, L. Lavazza, A.L. Wolf , *A Conceptual Basis for Feature Engineering*, Journal of Systems and Software, Vol. 49, No. 1, December 1999, pp. 3-15.