Incremental Product-line Development

Kester Clegg & Tim Kelly & John McDermid
Rolls-Royce University Technology Centre
in Systems and Software Engineering,
Department of Computer Science,
University of York, York, United Kingdom
kester@cs.york.ac.uk, tpk@cs.york.ac.uk, jam@cs.york.ac.uk

October 14, 2002

Abstract

Fundamental to the success of a product-line strategy is having some means to attain the global architecture that all products will share. Migrating to the architecture is often perceived as a difficult part of implementing the strategy. However, the technique presented here permits a low-risk, incremental development of the architecture via a process of negotiation. In effect, small scale product-lines are set up to supply products of a particular type to other systems. Stakeholders of the systems come together to negotiate an interface to the product and define it in an abstract form. As these small scale product-lines increase in number to cover most systems' entities, their collective abstract interfaces start to define the product-line architecture.

1 Introduction

Using a software product-line means employing a 'shared architecture and a set of components' [2]. However, getting one project to share another's architecture can be difficult, particularly when a company has previously organised its software development on a project-by-project basis. While progressing towards a single architecture for all products, the implementation strategy can become entangled in the need for generic requirements to be defined as part of an 'upfront' analysis. There is also a belief that the new architecture must attempt to encompass all foreseeable variation in future products and that this is only achievable via a lengthy and expensive commonality / variability analysis. This may be true to a degree, but gives the perception that implementing a new product-line provides uncomfortable hurdles in terms of the amount of process re-organisation required.

This paper puts forward the opinion that a product-line strategy can be implemented with minimal impact to ongoing projects while still laying a good foundation for future product variation. It shows a low risk method to obtain a product-line architecture in an OO (object-orientated) environment, giving specific examples from C++ . The technique illustrated encourages the transmission of expertise across projects and allows the architecture to be developed over time. This incremental development of the product-line architecture means that metrics can be applied at an early stage to evaluate the strategy before substantial investments in process alteration are made. The technique also has the advantage of being easy to understand and apply. The fundamental principle is simply:

encapsulate and abstract an interface to entities which could serve in other systems.

This principle assumes the same basic entities (in our case, C++ classes) are common to the systems in question, even though they may vary greatly in their implementation. For example, a Maintenance system will always require some means of recording and reporting faults. By drawing a functional boundary around those areas, we can look at how these entities can be supplied to systems with a consistent interface.

The first step is to propose an abstract interface which all implementations will share. This abstract interface needs to be negotiated with teams working on other systems that may (or already) use a similar entity as part of their subsystem. The process of negotiation defines the interface to be shared by a mixture of common needs and consensus.

Products are the concrete implementations of points of variation within a product-line architecture. Prior to them being elevated to this position however, they are first proposed as points of variation in a class model for a particular project. Having abstracted their interface, the areas for negotiation with other projects are easy to identify. Cross-project negotiations over these shared interfaces can be planned during the modelling stage and budgeted as external to the current project.

The word *entity* is used throughout this paper to refer to a bounded element of functional decomposition. While we are in the main referring to C++ classes, such a functional 'block' could equally be a Matlab Simulink subsystem block. What is important is that the functional boundary that captures the level of reuse is broadly similar on both projects. One advantage with C++ is that the abstraction is part of the language and adherence to the interface can be enforced by the compiler.

1.1 Related Work

The ideas in this paper follow on from work on product-lines by Jan Bosch [1, 2, 15] and papers presented by David Sharp at the Software Technology Conference on the potential for design patterns to be used with product-lines [6, 10]. There has already been work done on the need for product-line architectures to manage product variation effectively [8, 9] and this paper acknowledges the importance others have placed on establishing a global product-line architecture [3]. Providing an 'upfront' analysis to give insight into the commonality and variability of products was addressed as part of a previous study in the domain [11]. The technique advocated here builds on recent work by the University Technology Centre (UTC) in Systems and Software Engineering here at the University of York [5, 12, 13].[1]

1.2 Problem Context

This paper takes as an example the Maintenance subsystem of the Rolls-Royce family of civil aerospace engines. A Maintenance subsystem is embedded software in a safety-critical, real-time environment. The company aims to move its engine Control System software development towards a product-line strategy in order that greater re-use is made of existing solutions for previous engines. Essentially, the Maintenance subsystems on these engines do not vary greatly in their functionality from engine to engine, and as such would seem a good basis for a software product-line.

Current projects are using a mixture of model-generated C and handwritten C++ code. Having OO allows us to use a variety of existing design patterns and one of these, the AbstractFactory pattern [7], is ideally suited to the implementation of product-lines. The example shows how to apply the pattern to existing entities in a class model of the subsystem. It gives guidelines on how to ensure a product-line approach is taken, rather than one which will be hampered by the demands of the current project.

2 Defining the Entities

The Maintenance subsystem of an engine receives and acts on being notified of faults by the operating system (OS). After checking and validating the fault, it writes the current inputs to an area of non-volatile memory as a snapshot and generates the appropriate message which may or may not be sent to the main on-board computer. The principal processing parts of the subsystem are in the application of combinatorial logic to validate the fault in the contexts of other faults, dispatchability (time allowed until a fault is declared valid) and suppression rules that determine whether a message will eventually reach the cockpit or not.

While there is considerable complexity in the Maintenance subsystem, the functionality can be encompassed in relatively few entities. According to OO principles, keeping objects as *lightweight* as possible (to contain minimal functionality) is better in terms of future maintainability. However, it can lead to a proliferation of classes in the design and this could have an impact on projects trying to share common interfaces. The more interfaces there are, the more restrictive the overall design becomes and this is a particular concern for product-line architectures.

Given existing built-in test equipment (BITE) specifications, the two obvious candidates to match real-world objects with classes were faults and messages. The management of objects created from these classes

[1] References [5, 12, 13] are only available from the UTC with the permission of Rolls-Royce (UK) plc.

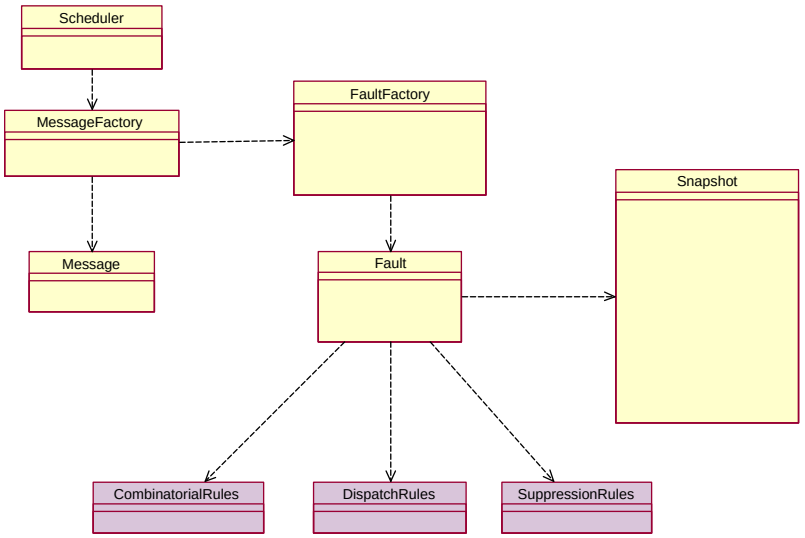

Figure 1: Simple maintenance architecture

could be done using the well-established pattern of *factory* classes following standard OO practice.[2] The resulting model of a somewhat simplified Maintenance subsystem is set out in Figure 1.

3 The AbstractFactory Pattern

The AbstractFactory pattern was envisaged as a means of enabling different implementations of the same functional requirements to be built within a common architecture. Importantly, it encouraged development towards abstract interfaces, rather than implementation specific routines that executed similar functionality.

The pattern allows for any number of products to be created. Which product is created depends on which concrete factory is instantiated previously by the client. However, by using abstract base classes with pure virtual functions, the same code can be used to create products of different types. The client simply calls the interface provided by the AbstractFactory. It does not need to worry which concrete factory will actually instantiate its own version of the product.

3.1 Applying the AbstractFactory Pattern

The AbstractFactory pattern can clearly find an application in a product-line approach to the Maintenance subsystem. Two things are needed for such an approach:

- A commitment to a single, shared architecture for future Maintenance subsystems;
- A belief that the same fundamental entities of a Maintenance subsystem will continue to be present in future designs.

The latter point is less important than the first, given as we shall see, that entities can be moved gradually into product-lines. However, without the first, a product line isn't possible. Without the second, there is no incentive to create a product-line. Our approach was simply to consider each of the principal entities in the class model to be candidate products or (product factories) in a Maintenance subsystem product-line. Essentially, applying the principle *abstract and encapsulate any point of variation* to each entity in the model will give the same result, the AbstractFactory pattern merely provides a recognised framework for generic code to compile different concrete implementations.

[2]Feedback from the company has suggested that 'factory' is not a good name for such classes in safety-critical applications as it implies objects can be created dynamically. As no dynamic memory allocation is allowed, their role here would be limited to the initial creation of objects and perhaps some means of providing access to those objects.

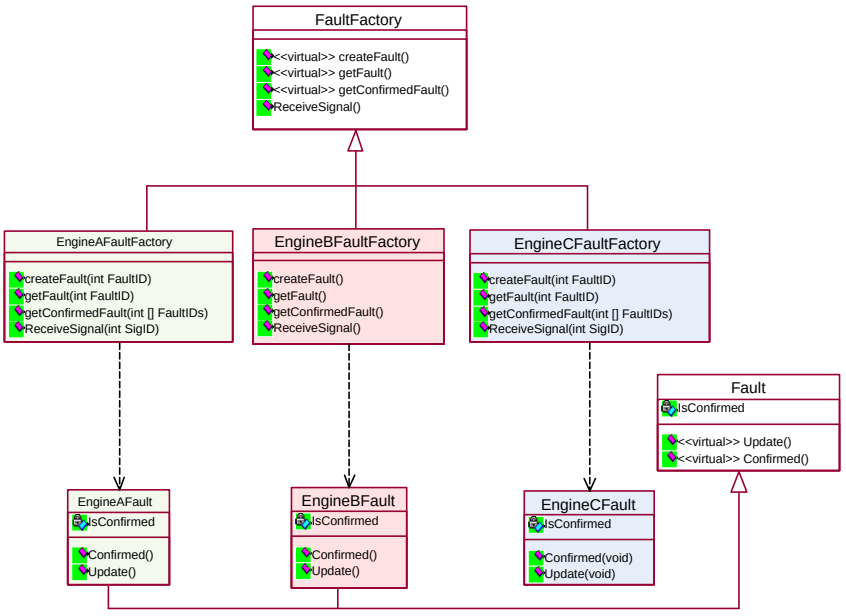

Figure 2: FaultFactory and fault classes, showing implementations for other engines.

3.2 Electing Suitable Entities

If we assume the next Maintenance subsystem will be virtually unchanged from Figure 1, other than being specific to that engine in its implementation, we can assume these same entities have potential for re-use. An attractive aspect of the AbstractFactory pattern is that it can be applied to a single entity in a subsystem design. That entity can form part of 'mini product-line'; in effect, a product-line that supplies *part* of a larger subsystem with a product. As the design matures, the abstract interfaces to the small scale product-lines gradually define the wider product-line architecture. However, by keeping them small and applying the abstraction to single entities at a time, the architecture can be kept flexible during its early stages.

The transparent classes in Figure 2 are abstract base classes, with the *capability* of the AbstractFactory class (i.e. the products this product-line can produce) defined as pure virtual functions.[3] These abstract interfaces must be negotiated with other project teams who may make use of the product-line to supply them with the entity concerned. The negotiation process is covered in greater detail in §4.2.

As the number of products increases and more engine variants are brought in, the complexity (and rigidity) of the model also increases. However, the more products that are defined in the product-line, the more reuse is achieved. Some tangential benefits include:

- greater robustness of the architecture as it matures to accommodate more engines
- greater awareness of commonality in requirements as expertise is shared amongst developers
- adherence to the product-line interface means many design considerations are already approved

4 The Specification of Abstract Interfaces

Much of the success of product-line implementations depends on the ability to foresee points of variation. The anticipation of likely variation points has always been a difficult design issue and designing for potential change has been the motivating force behind the introduction of OO techniques. However, OO techniques

[3]The (virtual) function calls illustrated in the models are not taken from any existing code at Rolls-Royce and should not be considered representative.

have tended to concentrate on achieving flexibility and reuse at a localised, low level in design. The product-line approach can be seen simply as a means to produce different but related products. Thus flexibility and reuse are considered at much higher levels than traditionally. However, previous product-line studies have emphasised the need for a detailed commonality and variability analysis of potential products[2]. Carrying out such a study can be expensive and time consuming. The need for an incremental product-line strategy stems from companies being unable to commit resources to a full-scale, upfront analysis of their product families. Where resources are not available, the incremental approach provides a low risk and relatively inexpensive alternative to implementing a product-line from the base-line of an existing project.

Our view equates products with variation points in the product-line architecture. The initial focus on starting a new project is therefore towards a breakdown of the basic constituents of a subsystem. Once the entities are roughly defined, each entity has the potential to become part of a mini product-line that supplies variants of the entity to the Maintenance subsystems of future engines. It is the responsibility of the new project to decide whether a particular entity can find a place in future systems. If the engineers believe so, they raise it with a product-line co-ordinator (whose role is described later) who can take the proposal forward with other projects.

This approach means that product-lines can be thought of as small scale, lightweight (in terms of process) and introduced as convenient to the project concerned. It is important that deadlines and ongoing work for the new project are not disrupted by the project having to bear the brunt of the re-organisation necessary to implement the product-line. The Maintenance subsystem of the next new engine might only agree to share the message handling or fault handling products of the Maintenance product-line, as the remainder of the generic subsystem is too different to be accommodated. There is no requirement for a new engine to be included in an 'all or nothing' approach. Inclusion can be partial, but the benefits of improved process and cross-project understanding for that part of the domain will still be valuable.

4.1 Specification of a New Product

Once the basic entities of a subsystem have been defined, thought should be given to how these entities could be supplied as products from a product-line. Using the AbstractFactory pattern, the abstract interface specification declares the product's interface in the form of pure virtual functions which a client can request. The product itself should have an abstract interface to allow it to be classed as a type so that the client can instantiate it without worrying about its specific nature.

It is not recommended that every detail of an existing implemented specification is brought to the negotiation table to discuss its abstract specification with other projects. A product-line needs to be able to encompass the variability of the products it supplies and an excessively detailed interface specification would prohibit this. The recommended stages of the process are:

1. A short proposal suggesting the entity's functionality could be common to other subsystems;
2. A brief discussion of the entity's functionality within a subsystem;
3. A proposal that the entity could be supplied to future subsystems as a product via a product-line;
4. An inter-project request for a meeting to discuss the shared abstract interface of the product.

Proposals like this, carried out during a current project, require a budget (and time) set aside for product-line work. The Maintenance teams on all projects would book time to the same cost code for this work, and members of different teams should report to the chief architect or product-line co-ordinator as they make proposals for the product-line. It is essential that no one project takes the entire burden of setting up the product-line and the specification of the abstract interfaces. Indeed allowing one project to do so in the absence of other projects' representatives is likely to result in an inflexible interface due to the likely bias towards that project.

4.2 Negotiating the Abstract Interfaces

Once a meeting has been agreed for the proposed product and product-line, development teams with relevant domain experience should attend the meeting to discuss the degree of variability that the products could have. Even if these teams are working on mature subsystems rather than new implementations, they are likely to have valuable experience relating to previous changes in their requirements. They may further

have suggestions on the weaknesses of current designs or implementations that have caused maintenance difficulties and these considerations should be brought to the meeting to help define an abstract interface that is agreeable to everyone. A checklist for such a meeting should ensure;

- the entity can form a part in future subsystems
- the abstract interface is acceptable to all stakeholders
- the abstract interface of the product is unrestrictive in terms of the expected concrete implementations
- the abstract interface can be traced to generic requirements for all projects

Depending on the degree of commonality that can be agreed between the project teams, the abstract interface can be very simple (maximum implementation flexibility) or well-defined (restrictive implementation, but providing better levels of reuse). These needs conflict, and some projects may be unhappy with the prospect of having to use a tightly defined abstract interface. In this case, and perhaps while the product-line is relatively immature, it may be safest to keep the interface specification to the bare minimum in order to ease the negotiations. As the product-line and the process of negotiation matures between the teams, more commonality is likely to be found and agreed on, and this can be added to the existing interfaces.

5 Introducing the Technique on an Existing Project

One of the main advantages of the technique given in this paper is the possibility of applying the pattern to a single entity at a time. This permits a low-risk migration towards a product-line strategy and allows the onus of developing the product-line to be shared between projects. Maintaining a series of mini product-lines is easier if the processes for doing so are built up gradually, with people learning from their mistakes and making improvements on the process.

Starting with a single entity also allows metrics to be brought in earlier, so that reuse can be monitored and the benefits assessed. However, it should be noted that serious consideration of the metrics gathered before the product-line matures (i.e. after several products have become available) might provide misleading data. The principal benefit of getting metrics in place early is that a process can be established and consideration given to what exactly should be measured. After these are in place, data can be gathered with greater confidence once the next project uses the shared interfaces and architecture. The steps can be summarised as follows:

- A product-line co-ordinator is appointed — this could be the chief architect, for example;
- The development team selects a single entity as a candidate product for future subsystems;
- The team goes through the specification process outlined in the previous section;
- The product-line co-ordinator works with the team to set up meetings with developers from other projects so that the process of negotiating the abstract interface can begin;
- After the abstract interface is agreed, it should be published and all projects informed;
- The reasoning behind the proposed abstract interface must be clearly documented;
- Generic requirements should be established that reference the product-line structure so that current and future projects can trace the reuse intention specified by the abstract interface;
- Work on the concrete implementation of the product for the current project can then begin, this work is costed and carried out by the project team that made the initial proposal.

During this process, it is likely that the current project team will have the greatest say on the abstract interface specification, as their project has the foremost need of being completed on time. It is the responsibility of the product-line co-ordinator and other teams to ensure that the proposals of the current project team do not impose an interface that is likely be restrictive in terms of their experience. They should bring their own projects' requirements into the negotiations as if they were currently relevant, or at least use their knowledge of requirement change requests to indicate the most probable variation that could occur from one implementation to another. There are many aspects to product-lines which remain undetermined, but by trying out the process on just one or two entities, the risk of major costs being spent on a process that later fails to give real benefits is kept as low as possible.

6 Conclusions

This paper has shown how a product-line can be developed in a C++ OO environment. The technique provides low risk and low set up costs. The basic principle of *encapsulate and abstract points of variation* is simple, and providing a product-line co-ordinator has been appointed, the following stage of abstract interface negotiation is straightforward. The process has the following advantages:

- a low risk, minimal impact migration towards a product-line strategy
- cross-project negotiations over the shared interfaces are easy to identify, plan and budget
- transmission of expertise across projects as the product-line process matures
- incremental definition of the architecture allows early metrics for pilot projects
- start up costs, design and development for new products are reduced

It is important that developers realise the entities they design could be products in a product-line of use to other development teams. They should raise queries over potential products with the product-line co-ordinator or chief architect, who would then take on responsibility for managing the specification and negotiation of the interfaces, while the developer is able to continue their localised work on the current project. This seems to be a way of developing product-lines within an organisation without undue disturbance to ongoing projects. Project start up times should also improve, as much of the infrastructure to an entity should have been queried and put in place as part of its abstract interface specification.

6.1 Issues

As the process and product-line matures, the abstract interface specification to a product becomes larger and less flexible. Variant products struggle to fit within the shared architecture. At such a point, the product-line co-ordinator needs to review whether the variation is such that a new product-line may be warranted. By keeping the product-lines on a small scale, the risk of moving outside a mature product-line architecture is minimised. If it *is* unavoidable, the cost of doing it is lower.

Like all patterns that rely on inheritance as part of their reuse, requiring a change to the abstract interfaces can pose technical and managerial dilemmas. It is difficult and inconvenient to change an interface which as been used by several concrete implementations. The amount of code to change may be considerable if there are many layers of inheritance. Secondly, each implementation that used the original interface will need to be re-tested. If changes to an interface affect a lot of code, it may simply be easier to extend the interface rather than alter it. Extension can take several forms; parameter extension and interface extension are both permissable, though not in all circumstances. Parameter extension is possible where overloading is permitted. This allows additional parameters to be added and the compiler distinguishes the function as overloading a previous one defined higher up the hierarchy. Interface extensions can also take the form where another product is added to the abstract specification and existing implementations would then need to implement it. Unfortunately this last solution may not be allowed where no redundant code is permitted in the application, as such extensions may result in ‘dummy’ code being written to fool the compiler that an existing implementation matches the interface specification.

Given the difficulties of changing abstract interfaces where product-lines have been built up over time, it is essential that interfaces are proposed which are as flexible as possible so that they minimise the risk of requiring alteration. If cross-project negotiations are carried out which utilise the collective experience of those projects, there is every chance that the abstract interfaces will be flexible enough to cope with future variation of the entity. If the cross-project negotiations are weak, or unbalanced in favour of a particular project, then there is a danger of an interface being too specific to one project to be of use to other projects in later years.

6.2 Further Work

Clearly product-lines fit comfortably with the application of standard design patterns like the AbstractFactory pattern. However, much of that ease comes from the use of OO features like inheritance, and the ability

to not only specify abstract interfaces, but to do so in a way which forces their implementation if a sub-class inherits their interface. This bond between the abstract interface specification and the derived concrete implementation is of great value when ensuring standard solutions are built across the company.

Given the benefits of the approach outlined here, the next question to tackle is whether we can find a similar means of architectural specification for modelling tools such as Matlab Simulink and Stateflow. The difficulty here is to apply the same OO principles of abstraction and encapsulation to a toolset which is procedural both in its modelling tradition and in its model-generated code output. There are a number of possibilities that can be investigated in the immediate term. One is the possibility of treating outer Simulink blocks as 'wrappers' which depend on inner blocks to complete their functionality. Such wrapper blocks could be specified as abstract interfaces by using data input / output sources. Another alternative is to specify configurable sub-blocks as the abstract interfaces. The options provided by the configuration tie up the data sources automatically.

A common means of specifying variation points as products that form part of a shared architecture will add value to any company that uses a mixture of procedural and OO code. The method given in this paper achieves this for handwritten C++ code. The challenge now is to move the idea of abstract interface negotiation into the wider arena of model-generated, procedural code.

6.3 Acknowledgements

We gratefully acknowledge funding of the UTC work by Rolls-Royce PLC and the Department of Trade and Industry.

References

[1] J. Bosch. Product-line Architectures in Industry: A Case Study. In *Proceedings of the 21st International Conference on Software Engineering*, pages 544–554, May 1999.

[2] J. Bosch. *Design and Use of Software Architectures*. Addison-Wesley, 2000.

[3] L. Bratthall, R. van der Geest, H. Hofmann, E. Jellum, Z. Korendo, R. Martinez, M. Orkisz, C. Zeidler, and J. Andersson. Intergrating Hundreds of Products through One Architecture — The Industrial IT Architecture. In *Proceedings of the 24th International Conference on Software Engineering*, volume 24. ACM, New York 10036, may 2002.

[4] F. Buschmann, R. Meunier, H. Rohnert, et al. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley and Sons, 1996.

[5] K. Clegg. Design Guide Lines for C++. Technical Report YUTC/TN/2002.18, University Technology Centre in Systems and Software Engineering, University of York, May 2002.

[6] B. S. Doerr and D. C. Sharp. Freeing Product Line Architectures from Execution Dependencie. 1999. Presented at the Software Technology Conference.

[7] E. Gamma, R. Helm, R. Johnson, et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[8] L. M. Northrop. A Framework for Software Product Line Practice — Version 3.0. Software Engineering Institute, 2001. `http://www.sei.cmu.edu/plp/framework.html`.

[9] D. E. Perry. A Product Line Architecture for a Network Product. In *Proceedings of the Third International Workshop on Software Architecture for Product Families*, pages 41–54, Mar. 2000.

[10] D. C. Sharp. Reducing Avionics Software Cost Through Component Based Product Line Development. 1998. Presented at the Software Technology Conference.

[11] Z. Stephenson and J. McDermid. Tracing Features with Decision Models. In *Proceedings of the Second International Workshop on Software Product Lines: Economics, Architectures and Implications, Toronto, Canada*, 2001.

[12] Z. R. Stephenson. Outline Family Analysis and Specification Process. Technical Report YUTC/TR/01.02, University Technology Centre in Systems and Software Engineering, University of York, Dec. 2001.

[13] Z. R. Stephenson. Family-based Software Specification Guidelines. Technical Report YUTC/TN/2002.08, University Technology Centre in Systems and Software Engineering, University of York, Apr. 2002.

[14] B. Stroustrup. *The C++ Programming Language — 3rd ed.* Addison-Wesley, 1997.

[15] M. Svahnberg, J. van Gurp, and J. Bosch. On the Notion of Variability in Software Product Lines, 2000. `http://www.cs.rug.nl/~bosch/papers/SPLVariability.pdf`.