Accepted for publication in Software: Practice & Experience (Wiley)

Comparing Practices for Reuse in Integration-oriented Software Product Lines and Large Open Source Software Projects

Jilles van Gurp, Christian Prehofer, Nokia
[jilles.vangurp|christian.prehofer]@nokia.com
Jan Bosch, Intuit
jan @ janbosch.com

Abstract. This paper compares organization and practices for software reuse in integration-oriented software product lines and open source software projects. The main observation is that both approaches are successful regarding large variability and reuse, but differ widely in practices and organization. To capture practices in large open source projects, we describe an open compositional model, which reflects their more decentralized organization of software development. We capture key practices and organizational forms for this and validate these by comparing four case studies to this model. Two of these studies are based on published software product line case studies, for the other two we analyze the practices in two large and successful open source projects based on their published developer documentation. Our analysis highlights key differences between the practices in the two open source organizations and the more integrational practices used in the other two cases. Finally, we discuss which practices are successful in which environment and how current practices can move towards more open, widely scoped and distributed software development.

1 Introduction

This paper examines organization and practices for software reuse in integration-oriented software product lines and open source software projects. The main observation is that both of these approaches are successful regarding large projects with large variability and reuse, but differ widely in practices and organization.

Software product line development has emerged as a successful solution for reusing software artifacts between software products (e.g. [Jazayeri et al. 2000][Clements & Northrop 2001][Weiss & Lai 1999][Bosch 2000], [SEI 2006]). We refer to these approaches as integration-oriented software product line (SPL) methods, which are based on the notion of having a centrally maintained reusable asset base that forms the basis for derived products.

On the other hand, we see that there are successful but different practices for software reuse in large open source projects such as Eclipse and Debian. In a wider sense, we can view these as software product lines that can be easily adapted to different instances or products and where software reuse takes place successfully. These projects have a more open and distributed approach to software reuse, both on a technical and organizational level. For instance, Eclipse and Debian are extremely large open source projects that despite their size, scale of development and scope are widely used and highly successful. Their software forms the basis of several commercial software offerings in very different domains. Their development spans many organizations and they release new versions and features frequently while also maintaining a good level of quality and reuse.

In general, the goal of a software product line is to minimize the cost of developing and evolving software products that are part of a family of related products. A software product line captures commonalities between software products and provides reusable assets. The reusable assets of an integration-oriented software product line generally include a pre-integrated & tested, reusable component platform with components and variation points. By relying on a software

product line, product developers are able to focus on product specific issues rather than issues that are common to all products. They reuse the platform and some of the reusable assets while relying on the variation points to add the product specifics.

As an example, the Nokia Series 60 smart phone platform now measures many million lines of code (MLOC). Nokia, and several other companies are currently shipping dozens of different products per year that are based on this platform. It is the software platform with the largest market share in the smart phone market. Nokia has three product lines that each use their own customized version of the Series 60 platform for creating mobile phone products. These product lines corresponded to the (former) enterprise, mobile phones and multimedia business units that each have very different requirements. In addition to the three internal Series 60 related product lines, Series 60 is also licensed to external companies that extend the platform with additional functionality. In some cases, these licensees even base their own product lines on Series 60. For example, Panasonic's successful use of a Series 60 based software product line was presented at SPLC 2006 [Jarrad 2006].

A recent reorganization in Nokia has brought these three business units as well as the platform development business unit that develops the S60 platform together in one large new business unit called “Devices”. Additionally, Nokia announced on 24 June 2008, the acquisition of Symbian and the creation of the Symbian Foundation which aims to act as an open source organization representing the various Symbian and S60 dependent companies in similar fashion as the other open source organizations discussed in this article. These changes illustrate the transition from integration-oriented to compositional software development that is going on in the software industry currently.

The motivation for this paper comes from a general trend in large software systems to have distributed development and ownership. This trend presents challenges for integration-oriented software product lines, such as the S60 platform described above. However, many open source projects are already based on the assumption of distributed ownership and development, and practices in those projects are designed for this.

Consequently, the goal of this paper is to capture and compare the practices of both approaches and to understand when to apply which practices. For instance, there may be disadvantages to using an open, more distributed approach in small-scale product lines that should use an integration-oriented approach. If, for example, a small number of variations are needed and these are well planned, an open approach can lead too high cost of product derivation and an inefficient distributed organization. On the other hand, we see that, integration-oriented software product lines face several challenges in large-scale software development in an open environment and are evolving more towards open platforms. Our earlier work in [Prehofer et al. 2008] identifies several trends regarding the scalability of software product line platforms: the scope of successful platforms and their derived products is widening, development spans multiple organizations, and time to market for new features is increasing due to increased complexity. Additionally, [Krueger 2006] discusses similar limitations of using a top down, integration-oriented approach when scaling SPL development.

To facilitate the comparison, we present a model of the organization and practices in large open source software projects, which we call open compositional approach. Then we extract key practices where these approaches differ from integration-oriented software product lines. We validate these practices by analyzing four cases studies. Two of these studies are based on published integration-oriented software product line case studies. For the latter two we resort to analyzing the practices in two large and successful open source projects. The main goal of this analysis is to discuss how the cases relate to our model and key practices. We show in our analysis that the practices in the two open source cases strongly differ with the two SPL cases in many respects regarding our criteria.

In summary, the main contributions are as follows:

- We describe an organizational model for open compositional software development as used in large open source software projects.
- We identify the key, differentiating practices of successful large open compositional software development.
- We validate our analysis by a systematic analysis of four openly available case studies of software product line and open source software projects.
- We discuss the challenges of large software product lines in current open and distributed software environments and discuss the changes needed to implement a compositional approach in the organization.

Note that this paper is mostly about organizing software development and facilitating composition and least about technical composability of components. This focus differentiates it from component based software engineering practices [Brown & Wallnau 1999][Szyperski 1997] that were popularized in the past decade. The organizational aspects we discuss include: decision making on roadmaps and requirements, decision making by component teams and between component teams, testing practices, and versioning.

In the following, we first introduce background on software product lines and then present the model for open compositional practices in Section 2. In Section 3, we analyze the four software projects by contrasting their practices against this model. Our evaluation methodology, related work and further discussion are provided in Section 4 and we conclude our article in Section 5.

2 Comparing Integrational and Open Compositional Development Practices

In this section, we describe practices and organization for integration-oriented SPLs, followed by a model for open compositional software development as employed in several large open source projects. We discuss reuse in this setting and then we present a list of key practices which differentiate both approaches. The case studies in the next section are then validated against these.

2.1 Integration-oriented SPL Development

Integration-oriented software product line methods are based on a centrally maintained reusable asset base that forms the basis for any derived products. The reusable assets include a software product line architecture with a pre-integrated and reusable component platform with components and variation points. Ideally, the product derivation process consists of just configuring the integrated platform using its variation points [Bosch 2000]. However, in practice it involves extensive product specific development and sometimes modification of the platform assets.

Typically there are multiple relatively independent development cycles in companies that use software product lines: one for the software product line itself (often referred to as domain engineering), and one for each product creation. Deriving a product from a software product line typically means taking the latest version of the software product line and using that as a starting point for developing a product. This involves two activities. First components are selected and unneeded functionality is removed. Where possible, pre-implemented variants are selected for the variation points in the software product line. Secondly, additional variants are created for the remaining variation points. In some cases, evolution of both product line platform and software product may make it necessary to repeat these activities at later stages (e.g. when updating a product with the latest product line components for a new product release).

A SPL with many variation points and pre-implemented variants can potentially support a wide variety of products. However, a SPL with too much variation (i.e. unneeded variability) makes product development needlessly hard due to the associated complexity, and complicates maintainability of the platform. To prevent this, SPL development practices include activities

related to requirement scoping and variability modeling. Effectively, a SPL approach results in a top down, centralized planning of reuse. Requirements are gathered centrally. Based on this, a product line architecture and its components are designed, realized, tested and delivered as a pre-integrated platform that product developers can use to quickly assemble products. Subsequent development iterations in a SPL are generally focused on supporting new features identified to be desirable in next generation products.

A key advantage of this integration-oriented product line approach is that product developers benefit from the integration effort that was spent on creating the component platform - they can focus their attention on aspects that are specific to the product that is being created. Case studies such as the CCT study [Clements & Northrop 2001] discussed later on in this article have shown that substantial benefits are feasible with respect to e.g. reduced product creation cost and faster time to market.

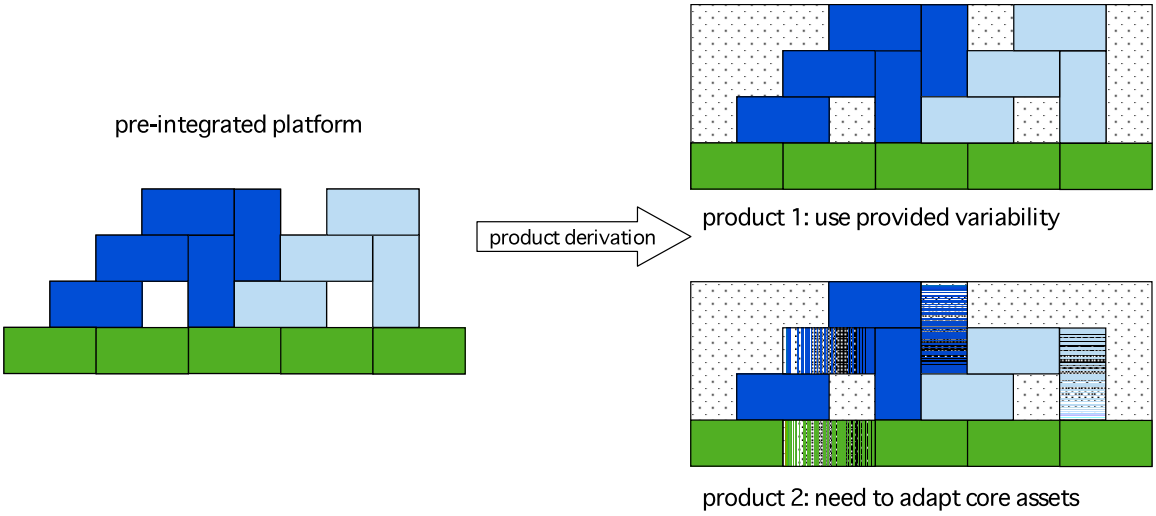

Figure 1. Product Derivation from an Integration-oriented software product line

In Figure 1, the concept of a SPL is illustrated. On the left, there is a pre-integrated platform. The white space represents those parts of the platform that need to be extended/completed during product derivation (using the provided variability). In the pre-integrated platform a few architecture components can be identified (represented in different colors), that each can consist of several subcomponents (represented here as rectangles). The layer at the bottom represents the component platform.

Components in a SPL may be libraries, frameworks, subsystems and any other large assets that large software systems can be decomposed in at a medium to coarse-grained level. (E.g., individual classes would be unsuitable because large systems typically have many thousands of those).

The two products on the right represent two examples of a product derivation. The dotted space represents the custom development done as part of the product derivation. As illustrated in product 2, derivation is not always ideal and changes to architecture components may be needed beyond the variability provided in the pre-integrated platform. To illustrate this, some of the components are partially transparent and one sub component has been replaced by a custom sub component. However, the end-product is still recognizable as a variation of the pre-integrated platform, however. It can easily be seen that such custom development can cause problems with respect to maintenance and testing if for example the platform evolves and changes need to be propagated to the product development.

Integration-oriented SPLs can be characterized as process centric and centralized and generally have a lot of overhead in the form of coordinating between development teams, requirements engineering, and central organization. This overhead is justified by the reduced cost

for creating new products. However, with the increasing size of software product lines, the cost associated with these inefficiencies grows proportionally to the development effort, which over time, may expose companies employing this approach to disruption from competitors with more efficient development practices.

2.2 Open Compositional Development in Open Source Software

Several large open source projects have established decentralized practices for software development. In this and the following sub section, we introduce a model of these practices and organizations. We will then show in Section 3 how this reflects the practices in open source projects. This model for open compositional development is based on the following observations that we see in large software projects, including open source software projects:

- **Distributed software ownership.** It is increasingly rare, or even impossible, for organizations to develop all of their software in-house. Consequently, ownership of software is fragmented among many stakeholders including licensees, business units, development teams, copyright owners, third party software providers, etc. Note that we use the term ownership loosely here in the sense that the owner takes care of software development & maintenance and key decisions affecting the software, but not in the legal sense of owning copyrights, intellectual property or licenses. The 2nd product derivation in Figure 1 illustrates a potential outcome of conflicts with respect to ownership: product teams start making modifications to product line assets. The difference between the compositional and integrational development styles here is mainly that the compositional approach takes this decentralization of responsibility as a key organizational principle whereas in the integrational model there is much more emphasis on centralized and top down driven decision making.
- **Different goals of owners.** The owners of parts of a large software system typically have different missions and goals. They are pursuing their own interests such as: creating market leading products, developing the best open source framework, implementing a standard/specification, representing the interests of a broad developer community, optimizing development cost for a particular range of products, etc.
- **Organizations control a decreasing part of their software.** Software systems are getting larger and larger (measured in millions of lines of code, MLOC). Assuming that software developer productivity is more or less constant (some cost prediction models such as COCOMO [Boehm 1981] assume this), it requires increasingly larger teams to develop such systems. Consequently, large software products require a large part or major pieces of the software to be developed externally and then integrated.

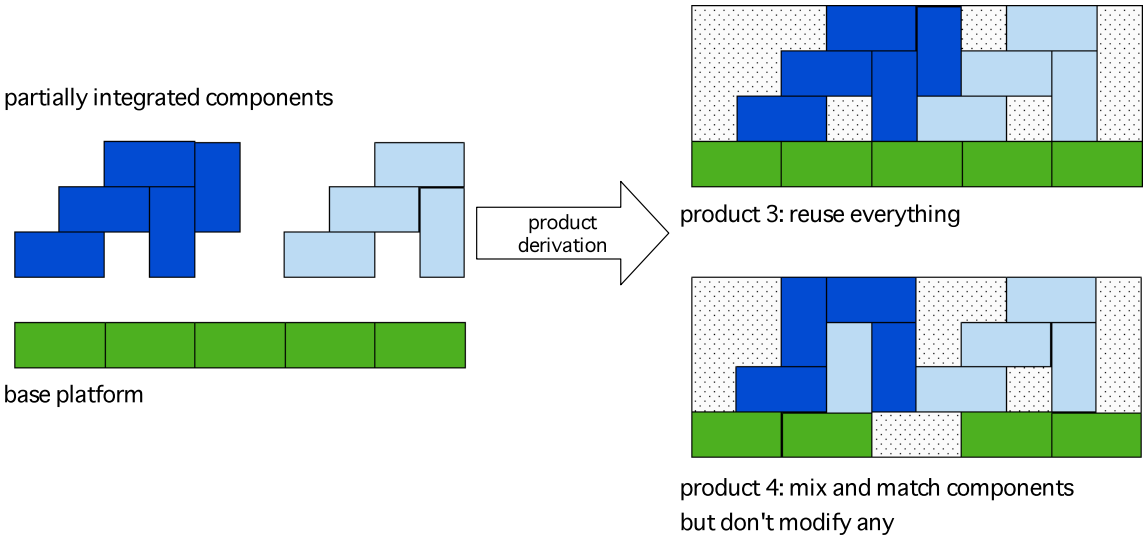

Figure 2. Composition oriented product derivation.

The open compositional model is illustrated in Figure 2. In this approach, product developers select partially integrated components consisting of sub components and integrate these with their own components. However, there is no pre-integrated platform and product developers are free to select from the available (sub) components. If possible, the partial integrations are used as a whole, as in product 3. These partial integrations represent the partial architecture that component developers design their components for. When necessary individual sub components can be mixed and matched as well if needed thus adapting the component architecture to product needs.

However, the individual sub-components are either used as is or not at all. Additionally, they are configured and extended using their variation points rather than modified and bended as in product 2 in Figure 1. Product 4 illustrates this by reusing only one of the components and rearranging the sub-components of the other one substantially.

Clearly, product 4 is something that would not fit well in the integrated platform of Figure 1, and would require significant changes to this integrated platform. This conflict is illustrated in product 2 with the partially transparent blocks. In the compositional approach there is no pre-integrated platform and the product developers simply take any components they can use, preferably in combinations that are known to work and fit with the architecture (e.g. the partial integration on the right) but if necessary in new combinations as well (such as the parts on the left in product 4). Any sub components that cannot be used are completely replaced with custom versions. Even the base platform and architecture itself may be modified if the product requirements make this necessary (illustrated by the custom component in the platform layer).

To ensure composability, the components share a common base platform. This base platform provides only support for key variability and composition mechanisms that are shared by all components and products. The base platform can be thought of as a micro kernel like in [Buschmann et al. 1996], i.e. a minimal product line architecture that is relatively small compared to the components and products that use it. In many product line architectures, including the two discussed in Section 3, such a base platform can be identified as part of the overall product line architecture, which also includes components and interfaces.

A key difference in the compositional model is that this base platform is separated from components and interfaces and that there is no pre-integrated platform. The components are provided by small teams of component developers that operate independently from each other and from product development teams.

The notion of independence is what makes the approach open. The goal is to create an open environment where small groups of people are responsible for nothing but the software they build

themselves. Responsibility of these people involves more than just building the software and also includes making important decisions related to *how* to build, *what* to build and *when* to build as well as how to interact and integrate with dependent and depending products and components. The rationale behind this is that competence and knowledge required for making such decisions lies primarily inside the team of people actually working on the software component. Effectively this decentralizes ownership of the components.

Of course, building products such as product 2 where parts of the pre-integrated platform are taken and modified is still possible in the compositional approach. However, it should be noted that in this approach doing so amounts to taking ownership of the sub components involved.

2.3 Organizing for Open Compositional Development

In this section, we discuss organizational practices of the compositional approach, which clearly need to reflect the ownership and dependencies of software development. An integration-oriented SPL is based on strong ownership and central control of an asset base that is aligned with product strategy in an organization. The main purpose of an open compositional approach is to reduce central control in order to create openness and flexibility. However, this is not sufficient by itself, since many important advantages associated with integration-oriented software product lines are missing such as:

- Cost reductions by exploiting commonalities between products [Bosch 2000].
- Strategic planning of reuse and product roadmaps [Clements & Northrop 2001]
- Improved efficiency, time to market, productivity and quality [Clements & Northrop 2001]

However, as discussed earlier in the paper, we believe these benefits will be lost in the future as software development evolves as we have discussed.

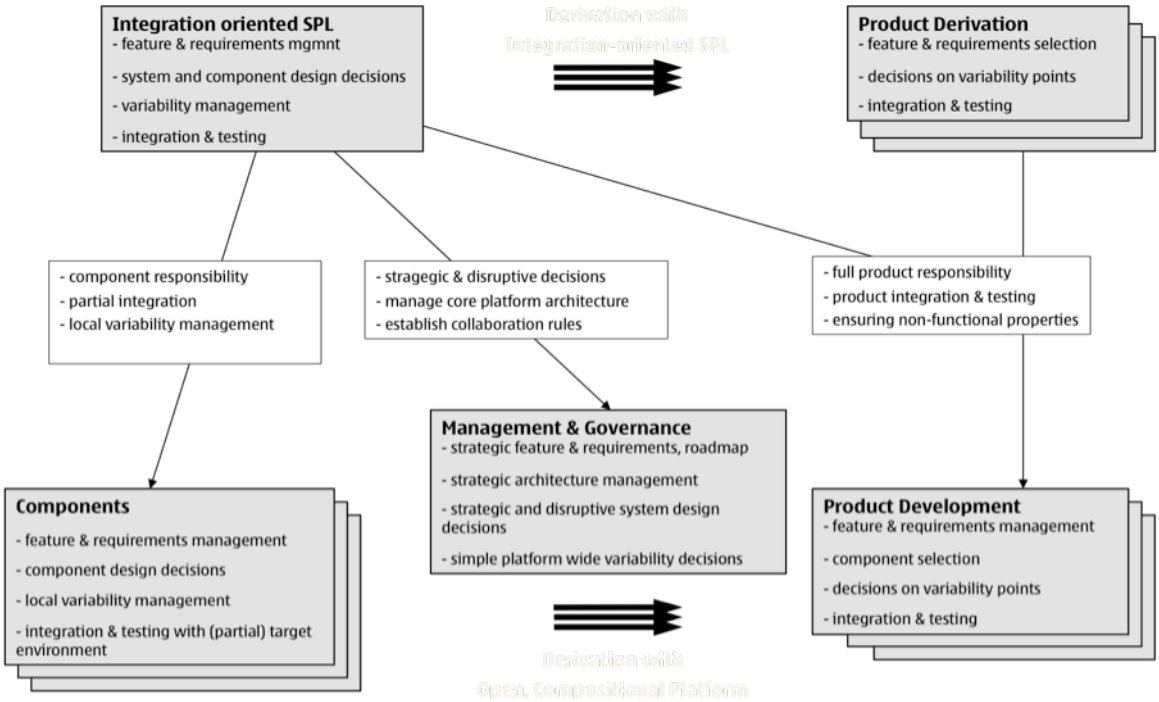

Figure 3. Comparing of responsibility in integration to compositional development

The key, guiding principle is to place responsibility where decisions can be taken best in a distributed, open development. The open compositional model can be seen as distributing

responsibility from the more centralized integration-oriented SPL approach to the following new organizational entities, as illustrated in Figure 3. Figure 3 compares product derivation in both approaches, where the thin lines illustrate corresponding responsibility (shown in light boxes) as follows:

- **Components.** There are several such organizational entities, one for each component in the platform. These organizational entities may be internal or external. A key element of our approach is that in both cases they should be treated as autonomous entities rather than elements of a hierarchical organization operating according to a strategy defined at the top of this hierarchy. Component teams have a high degree of autonomy. They are responsible for all development phases of their component. Responsibility includes planning development work, taking decisions, roadmaps, quality assurance etc.
- **Architecture:** Although there is no centralized authority for controlling the architecture, architecture is central in the coordination between teams. The architecture needs to evolve in order for the overall ecosystem or family of systems to support new functionality. The evolution primarily occurs through interaction between connected teams. But for cross cutting functionality, engineers that are trusted by their peers may develop initiatives to drive architectural changes.
- **Product Development.** Developing software products in the open compositional approach involves taking components and infrastructure, initiating development for product specifics, and doing integration work. This work is done by separate product teams that are responsible for the product(s) they develop. Similar to component teams, product teams may be internal or external and feature a large degree of autonomy. This includes the option to use the platform partially or not at all, if this is the best way to build their product.
- **Management & Governance.** While component and product teams are autonomous, a certain degree of orchestration is required to ensure that components can interoperate, fit together, and to ensure the right components are built. However, due to the autonomous status of these teams and the potentially heterogeneous nature (e.g. several competing companies, open source projects, etc) of the consortium that these teams form, this central governance takes a substantially different and more limited form than would be the case in a integration-oriented platform approach.

A key difference with integration-oriented SPL approaches is integration testing. A product line generally consists of a pre-integrated and tested set of components that product developers can customize to their needs. The approach here is to perform full integration testing only during product development. Additionally, the components used in a product have already been tested by their component teams. Finally, if those components are used in other products, they have been exposed to the integration testing in those products as well. Since components depend on other components, component teams need to select (preferably tested) versions of those components in order to test their own component. This leads to a situation where components have an associated recommended set of components for which a partial integration test has been performed by the component team. If these sets of compatible components are managed and evolving properly, this saves considerable effort compared to integration-oriented approaches.

Making product specific modifications or even using untested development versions of a component is allowed. However, the responsibility to do the required testing then transfers to the product team. Consequently, in order to maximize benefit from the component testing effort that component teams already did, components are best reused without modifications and with their recommended set of dependencies. If changes are needed to a component, the best way is to negotiate with the component team to make these changes in a next release of that component. The product team can either wait for this release or choose to use a development build. Only as a last resort, the decision to make the modifications as part of the product development should be made.

This practice is very different from that in many software product line organizations where it is common to make modifications to, or influence the development of, product line assets during product derivation in order to address urgent product issues. A key benefit of the compositional approach is mainly that it no longer hides the cost of such product specific modifications. As argued earlier, extensive product-specific modifications affect time to market and integration-testing cost as such modifications make products more expensive.

Comparing the organization, the entities operate autonomously but not in isolation of each other since there are dependencies between them. These dependencies are primarily related to use-relations and not to ownership or customer relations. For example, a product team depends on a (sub) set of components and a component team makes use of some components provided by other teams. The presence of such dependencies creates a shared interest on managing, coordinating and deciding certain things centrally. For example, integration of components becomes a lot easier, if component teams agree to use the same component middleware. Similarly, product teams can operate a lot more effectively if components are released on a predictable, joint schedule. Recently, Mark Shuttleworth who leads the company behind Ubuntu made a case for doing so across Linux vendors [Shuttleworth 2008].

Assuming that component teams strive to have their components used in as many products as possible, it makes sense for related component teams to synchronize their roadmaps and technology infrastructure. The management and governance part of the approach exists to facilitate this type of cooperation between component teams and product teams. It includes the following:

- Deciding on strategic directions and decisions that go beyond the scope of individual components is a main responsibility. This includes high level architecture and roadmap as well as key platform wide design decisions related to e.g. cross cutting concerns such as security. Such decisions are often disruptive and may affect large parts of the platform and products that use it.
- Platform wide rules and guidelines that are basic or easy to agree, and that most components depend on is the second main task. This includes harmonization of development practices regarding SW tools, release and bug management as well as testing practices. Other examples are system wide requirements and features like storage of user preference or recovery information that are used by many components. In addition, variation points, which have system wide effects, should be governed to be consistent in naming and usage.

This split of responsibility leaves the key development decisions for the component developers and can create an eco-system for collaboration and healthy competition between different organizations or organizational units.

2.4 Variability Management & Planned Software Reuse

A key topic in traditional SPL research is variability management. SPLs have been characterized as "planned reuse" [Clements & Northrop 2001][Bosch 2000]. The general idea is that SPLs allow organizations to capture commonalities in the platform and identify where their products differ from each other. The product line architecture then captures this knowledge in the form of components and variation points, i.e. specific points in the software where the functionality of the platform can be made to vary. This variation is then put to use during product derivation when product developers bind specific variants to the variation points (e.g. by configuring, providing plugin components, interface implementations, etc). Component platforms such as Koala [Ommering 2004] and the OSGi framework, which are used in respectively the Philips and Eclipse case studies we discuss in the next section, provide technical support for this.

A key difference to the integrational SPL approaches is that there no longer is centralized planning of variability because component teams operate autonomously. Given the documented success of SPL approaches in establishing measurable reuse and reductions in amounts of product specific code (e.g. in the case study we discuss in Section 3.1), this might be perceived as a disadvantage of the approach presented in this article approach.

However, compositional development as we outline it here is not completely decentralized and still has the notion of a base platform. As discussed earlier, the base platform ensures composability of components. This composability is achieved by supporting specific variability mechanisms in the base platform architecture that components may use to provide variation points. Since all components employ the same mechanisms, it is easier to compose them. Additionally, even though there is no central planning of reuse, there still is local planning of reuse since component teams can plan and optimize their designs individually taking into account the variability needs of users of their component.

As discussed in Section 3, all cases we present in this article feature a micro kernel like base platform providing support for variability mechanisms. Thus, we claim that planned reuse benefits realized in SPL approaches may be preserved and are not necessarily lost when adopting our approach. The amount and scale of reuse in the two predominantly compositional cases we discuss in Section 3 provides some evidence for this claim.

2.5 Key Practices for Open Compositional Development

A compositional approach affects the way components and products are developed. Building on the above, we have extracted a list of essential practices to characterize the compositional style of development and to distinguish it from the integration-oriented style of development. In Section 4, we validate these practices by comparing four case studies against them. In practice, there may be a mix of integrational and compositional that reflects the needs of the stakeholders involved, as the case studies in Section 3 illustrate.

The list of practices is structured by the different aspects of development.

1. Requirements & Roadmapping

   a. Component teams own their requirements. Component owners decide which requirements/features to implement, how to do that, with what priority and when. Given their technical and domain knowledge, component owners are the most qualified individuals to decide on component requirements. As discussed, the trends outlined make it increasingly harder to do centralized decision making and planning. Therefore, micro-managing component requirements centrally does not make sense in the open compositional approach.

   b. Component roadmaps based on long term needs. The long term anticipated needs rather than the short term needs of products drive component roadmaps. Because component developers are independent from product developers, component roadmaps are less influenced by short term product needs. Instead, they are based on the component developer's perception of what will be important in the long term. While this may create short term problems for some product developers, it prevents those problems from affecting technical integrity of components, from disrupting component development, and thus from propagating to other users of the same component.

   c. Products & components have shared roadmap A central roadmap is used to coordinate component and product development. It lists only strategically important requirements for future products. Details are left to the individual component teams. Items on the list represent the consensus between the component & product teams on what to build next as well as the agreed strategy for addressing key product requirements and a schedule for doing so.

   d. System-wide quality responsibility at product development. System wide, non-functional requirements (e.g. performance, security, throughput, real time behavior, etc) are ensured during product development, integration and testing. Products have product specific

configurations of components and therefore any system wide properties are unique to that configuration. Of course, many of these quality requirements have a crosscutting nature and require selecting and using particular components that support them as well. This is part of the Management and Governance responsibility, which includes ensuring that these components function properly in the context of different planned products. However, ultimately quality needs to be validated as part of product integration testing. (see practice 4a).

2. Architecture & variability

   a. Base platform. A base platform exists that embeds overall architecture guidelines and principles with a strong focus on interoperability and composability of components. This platform may resemble e.g. a micro kernel or object broker architecture [Buschmann et al. 1996]. These components do not form a complete, integrated platform but instead provide infrastructure common to all products and components.

   b. Base platform managed as separate components. The base platform consists of a small set of components that most other components will depend on. These components encapsulate the architectural principles that guide composition. The importance of the base platform may also imply a more central governance for these components.

   c. System wide variability managed through base platform architecture. The base platform architecture also serves as a means to configure central variability needs (e.g. preferences, component registries, etc.) since it facilitates the variability realization techniques that components use [Svahnberg et al. 2005].

   d. Components provide variation points locally. Instead of identifying variability from centrally maintained feature models, component developers try to anticipate extensibility and variability needs of their users. Additionally, they make use of mechanisms supported by the base architecture to offer variation points in the components.

3. Integration testing of components

   a. Test recommended dependencies. Component releases are tested against recommended, representative versions of dependencies to other components. For these dependencies, interoperability is verified by the component development team.

   b. No full component platform integration testing. Unit tests and/or test applications are used as a substitute for full platform integration testing when testing components. These tests and applications are designed to maximize coverage of component features and test the interaction with relevant dependencies.

4. Product Integration testing & bug fixing

   a. Full integration test is part of product development. The combination of components used in a product is unique and therefore only tested during product integration testing.

   b. Product testing does not result in immediate, major component maintenance. Component issues found during product integration testing are unlikely to result in major short term action (e.g. extensive feature work) by component teams unless doing so is in their interest (e.g. addressing critical security or stability bugs). Product needs may of course trigger the addition of items to the long term component roadmap. Additionally, components that are used in a lot of products are likely to get a lot of feedback from product testing. Consequently, on the long term, component quality benefits from product integration testing (Also see 5c).

5. Component Versioning & dependencies

   a. Decentralized component releases. Components are versioned independently from each other and are released according to individual roadmaps. However, as noted in Section 2.3, there is a benefit in synchronizing the release moment for related components. Often this takes the form of a platform or product release where all the component teams work to get the best possible version of their component delivered in time and actively test against other components. However, it is possible to have intermediate releases for some components or to adapt to other product/platform roadmaps on a per component basis.

b. Documented recommended dependencies. Component documentation includes a description of dependencies on other components as well as which specific versions of components are recommended/have been tested.

c. Prefer stable components. Product developers select stable component versions at the beginning of product development and are unlikely to use unreleased, development versions. Assuming component developers have an interest in their latest work being adopted in products, this provides some incentive to have short release cycles (which 5a enables).

d. Low risk dependency fulfillment. Product developers will also favor fulfilling component dependencies with versions recommended in the component documentation to minimize their own testing effort. When external partners are involved certified correct behavior of specific combinations of components and support contracts may also play a role. Additional factors in the decision process regarding component dependencies may be non technical issues such as reputation of the people working on it, legal issues (e.g. licensing or IPR), etc.

6. Organization

a. Separate product development. Product teams are separated from component teams in the organization and from each other.

b. Minimal platform coordination. Component and product teams operate independently from each other and may even be part of separate organizational entities (e.g. sub contractor or open source project). This means that coordination between teams is limited to direct dependencies.

c. Strong component ownership. Component teams have strong ownership in the sense that the same people are involved with the component development over long periods of time and decision power lies primarily inside the team.

d. No imposed use of components. Similarly, product teams have a large degree of freedom in selecting & developing components that match their requirements and product architecture. Of course, the selection of any component will trigger dependencies on the base architecture components and thus limit the freedom to compatible & recommended components (see 5d). This freedom is further constrained by testing cost and cost of taking ownership in case of deviations from recommended components or introduction of product specific components.

7. Process

a. Locally customized processes. Component & product teams have internal processes that are optimized for their environment. There may be central guidelines on how to organize, however the details of implementing such guidelines are not decided centrally. This allows component teams to function even if they are in different organizations, countries, continents, etc.

b. Limited team dependencies. Dependencies of component teams to each other and to other product teams are limited. Component teams are in charge of initiating and coordinating their internal processes. Needs of other teams of course play a role in decisions related to that but in principle no central coordination takes place.

3 Evaluation of Four Projects

We have evaluated the software practices in four successful software organizations based on the approaches and criteria in the previous section. The goal of the evaluation is to show that the key practices in some major open source projects are different from those in software product line projects. In other words, we show that our criteria are essential practices where these projects differ. We analyze four cases below based on publically available material and have carefully selected these cases studies. We do not aim at empirical or quantitative validation of "correct" practices, but rather argue that the case studies are prime examples of their approach.

Two of the evaluations are based on published case studies on software product lines. The first evaluation discusses the Control Channel Toolkit (CCT) case study from [Clements &

Northrop 2001]. However, we base ourselves mostly on the more detailed technical report in [Clements et al. 2001] which outlines in quite large detail the organization and architecture of the platform. The second analysis is based on the practices at Philips Consumer Electronics that Van Ommering discusses in his article on product populations [Ommering 2002] and related articles [Ommering & Bosch 2002] as well as his PhD thesis [Ommering 2004]. Arguably, Philips is less integration-oriented than the former since Van Ommering also discusses some of the practices we include in the open compositional development.

The other two studies are based on publicly available material about the development practices in two large open source organizations. We will see that these organizations apply most of the compositional practices we listed above. The Eclipse Foundation and Debian, represent extremely large open source communities whose software forms the basis of many commercial software products in different domains. Both are organized as a not for profit foundation and their development spans many organizations across the software industry and other open source organizations (e.g. the Apache Foundation and the Mozilla Foundation). Additionally, these two organizations are similar to many other open source projects with similar scale (e.g. Apache Geronimo, Sun's Glassfish, Mozilla Firefox). We have selected these two open source projects as their practices are well documented and openly available.

The aim of the evaluation is to compare these organizations to the practices identified above. We also want to see how practices correlate to the size and scope of the development in these organizations. Therefore, for each of the cases we discuss:

- The size and scope of development and organization. We estimate the number of people working on the software platform and the size of the software measured in lines of code.
- How they organize their development, e.g. responsibility of the different organizational units with respect to the SW development process.
- The base architecture used in the platform.
- Whether or not they implement the practices listed in Section 2.5. This is done by scoring the practices on a scale from --/-/±/+/++ where -- means leaning strongly towards integration-oriented approach and ++ means leaning strongly towards compositional approach. We use ± whenever we lack the material to decide either way. The scores are explained below and then summarized in Table 2 in Section 3.5. The judgments are based on the referenced material. As we evaluate very large projects we aim to capture the predominant practices, knowing that there may be exceptions.

In the remainder of this section, we first discuss the four cases in Section 3.1 to 3.4. A summary, including Table 2 with an overview of the evaluation scores, and analysis is provided in Section 3.5.

3.1 Control Channel Toolkit (CCT)

The Control Channel Toolkit (CCT) is a toolkit for developing spacecraft command and control software developed by Raytheon for the US National Reconnaissance Office (NRO). CCT users are government contractors that use the CCT to build software. The CCT is described in [Clements et al. 2001] as neither a complete software system nor a complete software product line. Instead, it is intended as an asset base with reusable components, architecture, testing procedures, development practices, etc.

Platform size and scope. The CCT was designed by a core team of architects and then implemented mostly in-house by a group of approximately 20-50 developers. According to the case study, "Typical command and control systems exceed 500,000 lines of code". Additionally for the first product based on the platform they claim that "Total SLOC developed by Spacecraft C2 is 76% less than planned", which suggests that the remaining 24% comprises reused code of assets from the CCT platform. Based on this information we estimate that this platform is in the order of 200-300 KLOC. This discussion of course excludes any products built with this platform

since this study was published as well as features and components added to the platform since then. An overview of examples on the CCT website [CCT 2008] indicates that the platform has evolved substantially since the introduction.

Organization. The CCT platform development is taken care of by Raytheon. The organizational structure outlined in [Clements et al. 2001], includes separate teams for software architecture, component development, testing and maintenance. This indicates a top down integrational approach with centralized decision making. CCT products are not developed by Raytheon but by separate companies. It seems that the CCT platform was realized using a waterfall model style development where first requirements were specified, then an architecture team designed the platform, and finally a larger team implemented the platform and its components.

Base architecture. The CCT architecture and variability mechanisms are based on CORBA. In addition to common CORBA related services and variability facilities, the CCT base architecture provides 6 "standard" variability mechanisms. A summary table in [Clements et al. 2001] (Table 2) gives us the information that the CCT platform has 100 components on top of this base architecture. Furthermore, licensees can add components by wrapping legacy components as CORBA components or by implementing specific CORBA interfaces.

Practices. Even though CCT has 100 components, no separate component teams are mentioned and instead it is mentioned that all components were developed by "component engineering". So there are likely no component specific roadmaps or autonomously operating teams (1a & b, 5a, 5b, 6 b & c, 7a & b). Additionally, while the components share a single CORBA based platform architecture, it is managed, tested and released as a whole and not separately (2a & b).

Raytheon does not develop products itself. Instead, the platform is licensed to third parties who build products (6a). The roadmap for the platform is based on the needs that emerge from product development and customer feedback (1c, 4b).

The case study mentions that there are 110 variation points, which is very close to the number of components. This suggests that this results from the central architecture rather than that this concerns extensibility added by component developers (2d). Additionally, product developers always use the full platform (thus relying on the platform engineers to make any choices related to which components to use) and use the provided variation points to differentiate rather than hand picking components (2c, 5c & d, 6 d).

Four levels of testing are identified in the report: unit testing, requirements verification, platform integration testing, and quality testing (1d). This suggests that components are tested together as part of platform integration testing (3a & b, 4a).

3.2 Philips Consumer Electronics

Philips Consumer Electronics is a division within Philips that manufactures TVs and other consumer electronics. The examples used in [Ommering 2002] are based on Van Ommering's eperience as a software architect in this division.

Platform size & scope. While he does not cite specific size of the software discussed in the article, Van Ommering does specify that "it takes 100 engineers two years to build the software for a high end television" and that "between 100 and 200 software developers are currently involved, distributed over 10 sites" [Ommering 2002]. Additionally it is indicated that so far (i.e. as of 2002) only a few products have been produced with the platform. Based on this, only some very rough estimate can be made of the size of the platform discussed in that article. We conservatively estimate it to be around 2 million lines of code.

Organization. The organization structure is discussed in quite good detail in [Ommering 2002]. Philips has product teams spread over four business units that create software products. They reuse software created by capability teams that are funded by product teams and that act as

an internal software house. The capability teams support the product teams but are physically located in a different place than the product teams. While Product teams strive to create software for their product, capability teams strive to create a unifying platform and balance the long term platform needs against the short term product needs. The terminology in the article (capability teams) suggests that Philips has multiple component teams with some degree of ownership and expertise. However, platform design and management appears to be a highly centralized still.

Base Architecture. Philips has created its own component model consisting of a Microsoft COM like architecture and an Architecture Description Language called KOALA. KOALA has several innovative features that are discussed in [Ommering 2002]. Highlights are that it has the notion of provided and requires interfaces as well as the notion of Glue models that allow modularization of product specific code as opposed to e.g. modifying components using e.g. C macros. These features provide variability mechanisms that may be used during product development to configure the platform with the correct components.

Practices. The vision behind KOALA reflects part of the vision behind the open compositional approach discussed in this article. Consequently, several of the practices that we list are applied by Philips. For example, road mapping is explicitly mentioned in the article as a tool used to plan "which components are developed when and by whom, and how they are used in products" (1a). However, the roadmaps are driven by current product requirements rather than a long term strategy (1b & c, 4b & 5c).

KOALA is intended as a compositional platform [Ommering & Bosch 2002]. This means that system variability is mainly achieved by using the KOALA ADL to define product configurations. (2c). Aside from the KOALA base architecture, there is no complete central product line architecture (2a). It is not clear exactly how KOALA is managed as part of the platform (2b).

Philips has a four step testing procedure that includes testing components against stable versions of dependencies (3a). However, they also do extensive platform integration tests in a phase called pre-integration (3b & 4a) to test both functionality and quality (1d).

While components are developed more or less independently (5a), products are based on a pre-integrated platform, which means that product developers merely use what is provided in the platform (6d) rather than what is recommended in the component documentation (5b & d).

Van Ommering classifies his organization as having separated product development (at a different location) from component development (6a). So called capability teams contribute a "single software product" to product teams and are funded by the product teams rather than by a central component organization (6b, c). Also, they work independent from each other (7b) however tools (i.e. KOALA), processes and practices are governed centrally (7a).

3.3 Eclipse Platform

The Eclipse foundation oversees the development of the Eclipse platform and a wide range of sub projects that depend on this platform. A wide range of software companies (e.g. IBM, Oracle, BEA, JBoss) have based their development tools on the software developed by the Eclipse foundation and are also active supporters of this foundation. Considering the large number of Eclipse based projects and products, the Eclipse platform can be seen as a highly successful platform for developing developer tools.

Platform size & scope. The open source network Ohloh [Ohloh Eclipse 2009], which provides information on a large number of open source projects based on their version management systems, claims that the Eclipse Platform project is (16-08-2009) 7,504,106 LOC and deduces using the COCOMO cost estimation model [Boehm 1981] that the effort to develop that amount of software amounts to 2277 person years. According to [Eclipse 2009], the 2009 Galileo release of Eclipse, which includes 34 Eclipse projects (including the Eclipse Platform), consists of over 24 million lines of code and contributions from over 380 developers.

Of course, spending that many person years with just a few hundred people [Eclipse 2009] takes more than a decade and the Eclipse project is only six years old at this point. However, it should be noted that it is common for open source committers to review and commit code on behalf of others without the right to do so (i.e. their personal code production is much lower than the amount of code they commit). This phenomenon is also analyzed in [Mockus et al. 2002]. Consequently, the number of developers working on Eclipse is probably much larger than the number of people with the right to commit changes on the repository. Additionally substantial amounts of code are regularly contributed by various companies involved in the Eclipse Foundation. Development for such code is not accounted for in these statistics either.

Organization. The Eclipse Foundation hosts top level projects that each develop a major Eclipse product or component. One of these is the Eclipse platform. Each of the top level projects can have several sub projects. At the foundation level, there is some shared technology infrastructure and decision structures in place. Key goals of management at this level is resolving any conflicts, ensuring IPR is protected properly (e.g. copyright of individual contributions, patents, planning synchronizing releases of projects, setting out global strategy, etc. The Eclipse platform project hosts both core Java development related sub projects as well as shared infrastructure projects such as the Equinox OSGi framework that is used in every Eclipse project. Eclipse sub projects further break down into components and modules.

Components generally have a single owner that works together with other developers. Within a sub project, component owners work together in determining roadmaps, initiating new component development, resolving bugs and issues in the bug databases, etc. Component owners have far reaching decision power on anything regarding their component. Generally, write access to a component in the Eclipse source repository is limited to component owners and a small number of trusted developers. Consequently, component owners control all changes on their component. The Eclipse organization is often described as a meritocracy, meaning that decision power is based on demonstrated skills and achievements. In order to affect development of components one needs to either provide acceptable change sets to component owners or convince them to make the changes.

Base Platform. The Eclipse base platform includes an implementation of the OSGi (Open Services Gateway initiative) framework (Eclipse Equinox) that provides a standardized component framework on top of Java. This framework has been extended with several Eclipse specific concepts (e.g. Eclipse plugins, extensions and features) that are used to modularize the highly complicated functionality in the Eclipse community. All Eclipse Foundation products as well as external provided Eclipse based software consists of the Eclipse OSGi framework and a collection of OSGi bundles implementing the various components. Reuse is achieved by reusing bundles between projects. Bundles have a version, provided and required interfaces and are packaged. Additionally the OSGi standard includes standard services that are implemented by the Eclipse Equinox project.

Practices. The organization of Eclipse development is outlined in the Eclipse charter [Eclipse 2006] and the development process [Eclipse 2003] other than these broad guidelines, there is no centrally defined process (7a). Eclipse consists of several top level projects (including the Eclipse Base Platform) that each consist of a number of projects (e.g. JDT) that are each operating more or less independently (7b). As discussed, each sub project in turn is broken into components. Commit privileges on source repositories are managed at the component level. Requirements and design decisions concerning a component are decided by people with commit rights on the components (1a, 6c). A roadmap in the form of themes and priorities is defined centrally based on requirements elicited from all members, supporters and users of Eclipse (1c). The architecture council breaks work down into components, which effectively defines the project architectures (1b, 6b).

Equinox can be seen as a microkernel architecture (2a & b) with mechanisms for handling variability. Most variability in the other projects is realized through this architecture (2c) by using and configuring bundles (2d).

Officially, the Eclipse foundation only provides "frameworks and exemplary, extensible tools" [Eclipse 2003]. However, in practice, these exemplary tools are fully integrated products that undergo rigorous integration and quality testing (1d, 3b, 4a) and are used by millions of developers. Generally, these products correspond to top-level projects in the organization: each top level project is a separate product (6a). Technically, there is no difference between product-components and platform-components other than the project they are developed in. In practice, components in a different project are developed by different groups of people (i.e. the projects are operating as autonomous entities) meaning that they are on separate release schedules and that it is not easy to quickly fix something in a component from a different project (4b) whereas this is somewhat easier with components inside the same project.

Product configurations are defined as a list of component versions. The configuration is updated when the component teams indicate a new version of their component is ready (5a, b, c & d). The output of various top level projects and third party components can be combined on top of the microkernel architecture to form new products or create new product configurations (7d). Generally, such configurations are tested with stable versions of any dependencies. Component teams will generally reuse product configurations for testing except maybe for direct dependencies that they fulfill with a different version (3a).

3.4 Debian

Debian Linux is an open source Linux distribution that packages and tests open source software. The Debian project continually customizes, tests, and integrates open source software packages into the distribution. Every few years a stable snapshot of Debian is released as Debian Linux and for this stable set of packages extended support (e.g. security fixes) and stability is provided. Several companies develop their own Linux distributions based on Debian Linux. This includes mainstream distributions such as Ubuntu and also many embedded Linux variants such as, for example, Nokia's Maemo Linux platform.

Platform size and scope. The 3.1 release (2005) of the Debian Linux distribution consists of 230 MLOC [Amor-Iglesias et al. 2005] and over 8600 different packages. It more than doubled in size since the previous release (3.0, 2002), which was measured to be 105 MLOC [Wheeler 2002]. Wheeler also points out the vast amount of resources required to produce such amounts of software according to cost models such as COCOMO, which represent billions of dollars and thousands of person years.

We lack similar data for the recent Debian releases. However, Ohloh provides a number of 45 MLOC [Ohloh Debian 2009]. We believe that this figure merely indicates Debian specific code rather than total amount of source code of the packages they include. A reason for this is that most source code for Debian packages does not reside on Debian controlled version repositories but is imported from third party version repositories. Additionally we know from the release notes of the various releases that 4.0 included 18200 software packages and the recent 5.0 release 23200 packages [Debian.org 2009]. This suggests that the growth trend progressed since the 2005 release, which leads us to estimate current code size close to 500 MLOC. The release notes mention that the source code needed to build the Debian 5.0 release is shipped on “4 source DVDs or 28 source CDs”.

Organization. Debian, a sub project of the Software in the Public Interest (SPI) foundation, is organized as a small organization with elected leadership and many people fulfilling roles ranging from support and documentation to account management and public relations. Generally, these people are sponsored by companies with an interest in Debian development. The SPI foundation itself has few people on its payroll.

Debian has a social contract which outlines mission and goals for Debian. Furthermore, the processes and guidelines that govern the organization are outlined in the Debian Developer's Guide [Barth et al. 2006]. However, almost all the development is done in third party organizations that have their own identity, infrastructure and development practices. These organizations contribute source packages to Debian for testing.

The Debian development process is very simple: new source packages or new releases of packages are contributed to the unstable distribution of untested packages. People in the large Debian test user community that consists of many thousands of users with an interest in Debian Linux development, update their local installation from this repository regularly and tests software packages they are interested in. Packages are moved from unstable to the testing distribution when no major issues have been reported for several weeks. The testing distribution has a much larger set of users (including many end users). Additionally, many commercial Debian derived distributions (e.g. Ubuntu) integrate and adapt packages from this distribution. Finally, the testing distribution is declared stable once every few years. This relatively rare event is generally preceded by months of extensive and rigorous testing and a temporary stop of the regular process of unstable packages migrating to testing.

For many open source projects, Debian is an important vehicle to deliver their software to end users. Consequently, there is extensive collaboration in the open source community to ensure that the software is good enough for inclusion. This includes conforming to guidelines regarding packaging of the software and ensuring interoperability with other Debian packages in the testing distribution. It is common for large open source projects such as e.g. KDE or Gnome to deliver alpha milestone releases to the Debian unstable distribution or to provide packages that can be installed onto a Debian testing or unstable installation since the Debian user community is key to getting their software tested.

For some packages, substantial amounts of adjustments are done by Debian developers. For example, the Mozilla Firefox browser is patched with several Debian specific customizations to ensure interoperability. Such customizations are part of Debian's own software development and are not part of regular Mozilla Firefox development.

Base Platform. Debian has a base distribution that includes the Debian package manager and a minimal Linux system that implements standard UNIX architecture for e.g. launching services, directory layout, an installer and a tool chain for building and testing the distribution, etc. Most of the Debian specific development is aimed at evolving this implementation and at integrating existing software in it. Additionally many sub groups of packages may be identified that each have their own additional architecture (e.g. KDE and Gnome architectures for desktop software components and applications). However, we don't consider these to be part of the Debian base architecture.

Practices. Obviously, Debian does not develop all this software itself (6a). It merely integrates releases of the packages and produces software to facilitate integration, installation and testing (7a & b). Quality and stability are the main goals of this integration effort (1d). There is no central requirements gathering or even a roadmap. The integrated open source packages are developed externally according to package specific roadmaps (1a, c & b).

Other than the package management infrastructure and the Linux kernel, which are managed as ordinary packages in Debian (2b), there is little central architecture or coordination (2a, 6b). Of course individual packages share architecture between themselves (e.g. the KDE platform and various KDE applications). However, this is outside the scope of Debian. Any Debian specific variability is managed using the Debian package management and configuration tools (2c). Generally, package descriptions and scripts wrap around individual components to expose any internal variability (not designed for Debian specifically) (2d).

Generally, new component versions are uploaded to unstable whenever they are ready (5a) and then migrate to testing after integration problems have been addressed. Components in testing and stable have been verified to work well with each other (3a, 5b & c). However, it is possible to

upgrade or downgrade components individually by users (e.g. use a unstable component) however generally people prefer the recommended stable or tested packages (5d).

There is no central, full platform integration testing (3b). Instead, thousands of users install various combinations of packages (6d) using the installation tools from the central repository. Any problems that result from this testing are reported centrally and addressed by package managers. Package managers are not necessarily involved with the packages under their supervision. They merely adapt the product from external projects that have complete control over their software development (6c). In addition to the official Debian distribution, there are several commercial Debian derived distributions that each layer additional infrastructure, integration and testing (4a) on top of the base Debian distribution. While the Debian developers and developers of these companies are in close contact with the package developers, there are cases where there are no formal ties between the two. For example, it is quite common that integration issues are addressed with Debian specific patches to components (4b).

3.5 Summary

In the following, we summarize the above findings in succinct table format to compare the key practices which we have discussed in Section 2.5. Then, we relate these results to the size and openness of the projects. First, Table 1 gives an overview of the project size and organization. Table 2, presents an overview of how we scored the four organizations (referred by number + abbreviated title). The columns have been sorted from left to right by platform size.

Table 1 Overview of evaluated organizations

	CCT	**Philips**	**Eclipse**	**Debian**
Organization Size (people)	50	200	380+	8000+
Code Size (estimate)	200 KLOC	2 MLOC	24 MLOC	500 MLOC
Central Platform Architecture team	Yes	Yes	Partially	No
Autonomous Component Teams	No	Partially	Yes	Yes
Has Base Architecture	Yes	Yes	Yes	Yes
Design decision making	Centralized	Architecture and product level	At component and product level.	Decentralized
Number of Stakeholders	Raytheon for the platform + licensees	Only Philips	All members of Eclipse foundation + unknown amount of eclipse based projects and companies + users	Unknown/many

Table 2. Overview of evaluation results

Practices		**CCT**	**Philips**	**Eclipse**	**Debian**
1 - a	individual comp. requirements	--	++	+	++

b	long term comp. level roadmap	--	-	+	++
c	shared prod. and comp. roadmap	--	-	++	--
d	System quality tested only during prod. dev.	--	--	-	++
2 - a	base platform	-	+	++	++
b	base platf. = set of components	--	±	++	++
c	system variability through base arch.	-	+	+	+
d	comp provide local variability.	--	--	++	++
3 - a	comp. tested with recommended deps.	--	++	+	-
b	no full platform integration test	--	-	-	++
4 - a	integration test only in prod.	--	-	+	+
b	no comp. maintenance result from prod integration test	-	--	+	+
5 - a	decentralized comp. releases	--	+	+	++
b	documented recommended deps.	--	--	+	++
c	products use stable components	--	-	+	++
d	low risk dep. fulfillment	--	--	+	++
6 - a	separate product dev. org.	+	++	--	++
b	minimal platform coord.	--	+	+	++
c	strong comp. ownership	--	+	++	++
d	no imposed use of comp.	--	-	+	++
7 - a	locally cust. processes	-	-	++	++
b	limited team deps.	--	+	++	++
Compositionality percentage		8%	44%	75%	89%

We relate the size of the projects to the compositionality percentage in Figure 4. The vertical axis ranking is based on the *compositionality percentage* in the last row of Table 2, which has been calculated using by counting -- as 0%, - as 25%, +- as 50%, + as 75% and ++ as 100% and averaging over all practices. The other two axes represent the size of the organization in number of people and the size of the platform measured in millions of lines of code (MLOC). The compositionality percentage is a basic metric to capture the alignment with the above criteria.

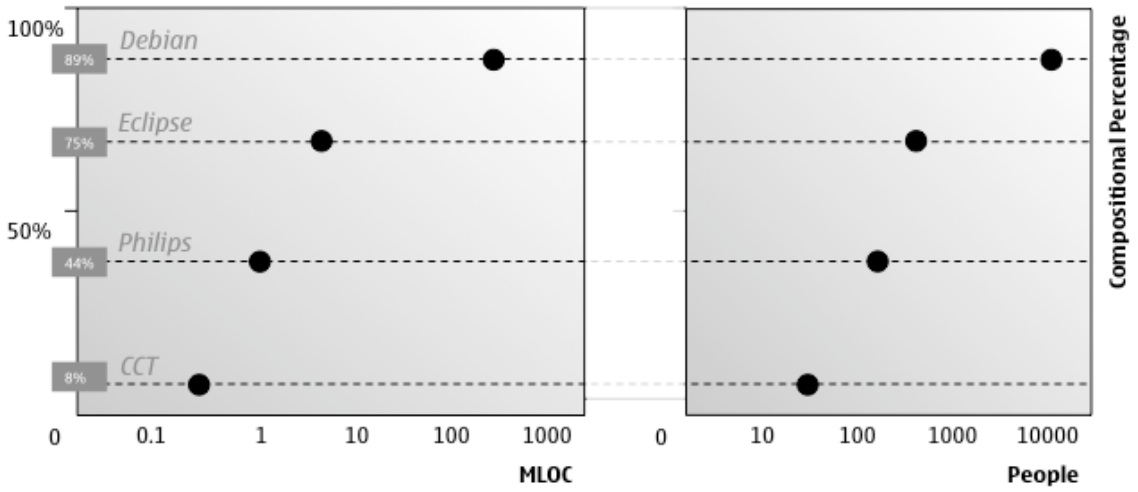

Figure 4 The cases ranked by estimated size and compositionality

CCT is clearly on the integrational side of the spectrum. Clearly, Philips has many compositional elements in its development approach since compositional development (as opposed to de-compositional development) is the topic in [Ommering 2002] and [Ommering & Bosch 2002]. However, there is still a great deal of centralized design and requirements related decisions in this case. Eclipse is an open source project but one with heavy involvement of a number of large software vendors. This shows in its centralized planning, requirements and design practices. Finally, Debian has only a minimum of central control to keep the project functioning.

Our evaluation has shown that the four cases we considered differ substantially in size of the software being developed, the number of people involved with the development, and degree to which they implement practices as suggested here. Indeed, we use an exponential scale for the software size in MLOC axis. To the best of our knowledge, most software product line case studies (see related work section for an overview) fit in the bottom left corner of Figure 4.

The main goal with this evaluation is to show that our criteria are suitable to characterize and differentiate the processes in the above projects. The data seems to conclude that growing the scope and scale of software development leads to a more compositional oriented style of development. While we conjecture this, we can only conclude that there are no known case studies of the integrational SPL style that are comparable to the Eclipse and Debian cases studies in terms of size. An explanation for this is that the more people and organizations are involved, the less practical it becomes to do things in a top down integrational fashion. As argued in the introduction, the trend we observe in our own and other organizations is that software development scope and scale is increasing. Consequently, successful software organizations currently applying SPL approaches may eventually be confronted with the problems we identified in the introduction.

4 Discussion & Related Work

Based on the above comparisons, we address the implications for integration-oriented SPLs and open compositional approaches. Furthermore, we discuss other related work.

4.1 Evaluation Method and Scope

Empirical evaluation of large software projects is important to justify development approaches. However, it also has number of inherent problems and potential limitations. We discuss these in the following for our evaluation.

- **Representativeness of the cases.** We have chosen to rely on well-documented projects from the literature and have deliberately chosen not to include case studies we have been involved in directly. We think that the cases are representative for SPLs and large open source projects. Each of the software systems can be argued to be very successful as well, i.e. the goal here is not to favor any of their practices but merely to compare them. The two selected open source organizations provide extensive documentation of their practices on their mutual websites. Also, much of the decision making related to these projects actually takes place in the open on e.g. public mailing lists. Unfortunately, this is not the case for most commercial software development. The limited availability of extensive, commercial case studies restricts validation in this area. Very few published SPL case studies exist with a sufficient/comparable level of detail to perform even the current evaluation. Furthermore, many commercial software organizations are very reluctant to provide such data for publication (for understandable reasons).
- **External studies.** We did not conduct the cited case studies ourselves and instead perform our evaluation based on published material from others. This somewhat limits the amount of useful criteria we can extract from each case and also makes it hard to compare the cases on some issues. Based on the material we've been able to extract, we think that there is sufficient material to support the validity of our conclusions.
- **Qualitative analysis.** We do not quantify the results (aside from using percentages to summarize the evaluation results). The reason is that this is a mostly qualitative analysis of existing, mostly qualitative studies without much quantitative data. Additionally the discussed organizations are very different in nature.
- **Compositional percentage metric.** The only quantitative metric provided in this article is the compositional percentage we provide. This might be mis-interpreted as a metric for organizations. Our main purpose of this metric is to be able to rank the organizations from integrational to compositional and capture trends. However, the criteria have to be considered for individual cases, as there may be different or context specific reasons for certain practices.
- **Validation of the open compositional model and criteria.** Our evaluation approach does not claim that the open compositional practices are complete or the single way of doing compositional development. As argued in the introduction, the approach is based on our evaluation of how open source projects function. The success of these projects is widely known and shows that the methods they use are effective.
- **Transition from integration-oriented to compositional SPL.** We also do not evaluate the transition between different practices. This is out of scope and there are very few documented cases where companies who have switched from traditional SPL approach to a compositional approach. To the best of our knowledge, the Philips case discussed in this section is one of the few that have made a partial transition.

4.2 Integration-oriented Software Product Lines and Scalability

In the following, we discuss the issues with current trends in SPL software development in the light of the above evaluation. Successful product line based organizations are confronted with a typical problem: their success causes them to grow market size, domain scope and software size. Consequently, over time, they absorb increasingly larger software components from outside the organization (e.g. through acquisitions or mergers [Ommering 2002], use of open source components, off the shelf components, or other third party provided software).

As argued in [Clements & Northrop 2001], software product lines were originally deemed optimal for small and medium software systems. Indeed the vast majority of case studies such as presented by [Clements & Northrop 2003], ourselves [Bosch 2000], and others (e.g. [Weiss & Lai 1999], [Jazayeri et al. 2000] and [Bass et al 2003]), mostly concern studies of software systems ranging in size from a few thousand lines of code to, a few hundred thousand lines of code. While there are of course some exceptions to this, very few SPL case studies discuss systems that are larger than that.

The observations in Section 2.2 lead to several key issues in the development:

- **Increase of (integration) testing cost.** As argued in [Muccini & Van der Hoek 2003], testing in the context of software product lines is inherently more complicated than testing standalone products. Especially when doing integration testing, changes in product line assets at the product level make it necessary to redo platform integration tests completed earlier. The sometimes considerable product-specific modifications made to the product line architecture and components that product developers make in order to differentiate, make it necessary to perform substantial amounts of additional integration testing. This leads to duplicated development and integration testing effort.
- **System wide design decisions and compromises limit product development.** As the scope of the platform and target products increase, the platform has to support more and more potentially conflicting requirements. Consequently, it becomes very difficult for an organization and their key architects to manage decision making and to ensure that the right technical decisions are taken. Difficult design decisions in the product line platform often lead to compromises that, while they address some product creation issues, also severely limit some later product developments where an alternative decision would have been a better match with product requirements**.** In an earlier case study on design erosion we argued this type of architectural brittleness to be inevitable for large software systems [Gurp et al. 2005].
- **Substitutability of non differentiating functionality.** Software features have a tendency to commoditize over time. Initially, when added to a product or platform, they have differentiating power. Over time, competitors implement similar features and they become less differentiating. As the platform is evolving and widening in scope, an obvious way to cut costs is to replace common or no longer differentiating functionality with "off the shelf" components (including open source) or to outsource the development and maintenance of such components. Essentially this requires that the product line architecture supports this with suitable interfaces, variability and development processes. Otherwise, the replacement of non-differentiating functionality with external components may lead to costly re-architecting and development.
- **Long time to market for differentiating platform features.** Time to market for new features in the platform (as part of a derived product) is increasing because product development generally starts from a stable version of the base platform. New, major releases of the base platform are slowed down by the enormous integration testing effort. Usually, platform releases occur less frequently than the derivation of new products. Consequently, time to market for new features in the base platform can be several years for large software platforms.

In summary, increasing scope and size of software platforms and organizations and the resulting reduced innovation speed are causing problems related to inflating testing cost, flexibility, replacing non differentiating software components, are putting pressure on time to market. Our above evaluation, suggests that incorporating compositional practices in their development may help them to address these issues since despite their scale and scope, the open source world seems to be really effective in producing good quality software.

4.3 Implications of the open compositional approach

An open compositional approach addresses issues and concerns we identify in the introduction and addresses the issues we have identified with integration based SPL development in the last sub-section. These are addressed as follows:

- **Increase of (integration) testing cost.** A key part of our approach is replacing platform integration testing with partial component integrations that individual component teams test. Component teams test their component against tested versions of components they depend on. This results in partial integration of component versions that are known to work well together and thus have been partially tested for integration beforehand. Full integration testing is only done as part of product development. This approach reduces platform integration testing effort and recognizes the well known fact (see e.g. [Muccini & Van der Hoek 2003]) that in a widely scoped platform, any combination of features and component in a product is likely unique for that product. Therefore, the most appropriate place to do integration testing is during product development. Consequently, the total amount of integration testing effort is reduced since it is now only done once instead of twice for platform and for product. As argued, product developers need to do a full integration test regardless of any platform testing due to the system wide effect that product specifics may have. This might be perceived as a disadvantage of the open compositional approach. However, it should be noted that the more diverse products are created with a particular component, the better it is tested. Therefore, in practice, testing effort becomes a shared responsibility rather than a centralized responsibility. Consequently, the more the platform components are used, the better they are tested. This argument is also frequently used in favor of open source software [Raymond 1999]. A potential risk here is that when several products identify the same issues, the cost rises due to the repeated testing effort. On the other hand such highly visible issues tend to be prioritized highly by component developers and of course potential workarounds for the issue as well as any knowledge about it may be shared and documented.
- **System wide design decisions & compromises limit product development.** By reducing dependencies between component developers and product developers and central management, each can operate more agile and benefit from locally available knowledge that is lacking centrally. This also limits the amount of decisions that is taken centrally and improves the quality of local solutions. A potential issue here is that less central coordination may lead to higher integration cost at the product level. However, one might also argue that it merely makes the associated cost more explicit at the product level.
- **Substitutability of non-differentiating functionality.** Removing most of the central decision making and giving component and product teams more independence means that they are free to use dependencies that fit in best with their team goals. While this does not directly improve the technical feasibility of replacing dependencies with alternative implementations, it does improve the likelihood that such a decision is taken if that makes sense, e.g. from cost perspective. Being autonomous means that teams take such decisions based on their local context of existing dependencies to components, depending components (if any), technical roadmap and requirements, budget, etc. These locally optimal solutions may differ from what is globally optimal. However, determining what is globally optimal is increasingly hard to determine in light of trends and assumptions that underlie the open compositional approach. Consequently, providing this level of freedom is important for product teams that need to support requirements not yet implemented in reusable components and thus aligns well with their mission to differentiate in the market.

- **Long time to market for differentiating platform features.** Product developers benefit from the above and can respond more agile to new requirements. When faced with unsupported requirements, they don't have to wait for a next platform release but can do any of the following: convince one of the component teams to address the requirement, use an external component, modify an existing component, or initiate product specific development to realize the requirement (e.g. as a new component). This approach also takes into account cost: while product teams can choose to "clone and own" an existing component, the cost associated with taking ownership this way is generally so high that this is not desirable. Some of the open source projects we discussed, for example, achieve high degrees of reuse precisely for this reason: it is legal to take the source code and make modifications but rarely worth the trouble of taking ownership. Secondly, the generally time consuming platform integration testing is removed from the path to market and product development teams are able to more closely integrate with component teams directly as well as synchronize roadmaps with them if needed.

Thus, the open compositional approach allows software development to scale along the trends we outline in the introduction and addresses some of the bottlenecks in the integrational SPL approach that currently prevent this. While open compositional development addresses issues related to these trends, it also raises a number of new issues. These concerns are resulting from our analysis of open source project, where such practices are applied. When applying the practices freely, some of the following issues may occur:

- **Potential wild growth of components.** This important issue also surfaced in traditional COTS component approaches where the decision to design for reuse sometimes resulted in developers wasting resources on improving software that did not necessarily need improving (e.g. by making it more extensible, reusable, flexible, etc.). Indeed one of the strengths of SPL approaches is preventing this by planning reuse using e.g. commonality and variability analysis based on requirements. As argued in this article, such central planning is not practical in the context of trends that motivate our approach. However it is important to preserve a link between development and business strategy, at least for components and products developed for one organization. This is why we emphasize the need for some central governance in Section 2.3. Additionally, in [Gurp 2006] we reflected on ways to adapt some variability related practices in SPL development to compositional approaches as proposed here. An important conclusion of that article was that there is no inherent need for centralization for variability management approaches, even if in current SPL practice these are highly centralized.
- **Cross cutting functionality.** Some features cross-cut components. This is particularly true for features related to quality attributes. For example, security related functionality has a tendency to do so. Consequently, there's a need for at least some consensus on how to realize such functionality across component and product teams. In the open compositional approach, this could be done as part of e.g. a base platform. We observe that indeed the two open source cases we discussed generally resolve such issues using central coordination. For example, in Eclipse, security concerns are (partially) addressed in its OSGI implementation (Eclipse Equinox) that can be seen as the Eclipse base platform. Additionally we observe that there are now many technical solutions for modularizing cross cutting functionality (e.g. Aspect Oriented Programming [Kiczales et al. 1997]). Consequently, given such technology, some cross cutting functionality can be managed autonomously, as we propose in this article.
- **Not all components are equal.** Inevitably, our approach results in a large number of products and components that all depend on each other. Assuming these dependency relations are mostly not circular, that means components at the bottom of this directed graph of dependencies are quite important because most other components and products depend on them directly or indirectly. This suggests that e.g. cost and the problem of wild

growth that was discussed earlier could be controlled and managed using such dependencies. Even though the approach we outline currently does not explicitly do so, this may be observed in e.g. the Eclipse foundation where some projects receive much more attention than others. Particularly, the components developed as part of the Eclipse Base Platform tend to be important to all eclipse sub projects.

- **Compositional vs. COTS Components.** Compositional development might be wrongly interpreted as a move back to the (Commercial Of The Shelf) COTS component approaches popular in the past decade. In those days, it was suggested that companies would either buy components developed by other components or use in-house developed components from a reusable component base. COTS approaches mostly pre-date both SPL approaches and open source practices that have emerged in recent years. The newer SPL approaches have in common that, unlike COTS they are not just about realizing the technical composition of components but cover the whole spectrum of software development and provide practices related to requirements engineering, architecture & design, processes & organization, testing, etc.

4.4 Related work

Product line case studies. In the past ten years, several books on software product lines have been published. In [Clements & Northrop 2001] several case studies are presented (we reuse the CCT case in this paper). Other works provide similar success stories [Weiss & Lai 1999] and [Jazayeri et al. 2000].

The selection of cases studies was mainly based on the availability of sufficient details on the case study. Several other case studies confirm our findings. For example, the CelciusTech case study in [Bass et al 2003] concerns a software platform in a small Swedish SME delivering products of up to 700 KLOC that seems similar to e.g. the CCT study. However, this case study is based on material that is at least 10 years old now (according to reports cited in [Bass et al 2003]). Another military software centric case study is provided in [Batory et al. 2000]. However, this study focuses on technical rather than organizational details of the approach, which makes it less suitable for our evaluation.

Additionally, we have been involved in several case studies on software product line development as well as software development in large software companies in the past. This includes our discussion of variability practices in the Mozilla platform, studied in [Svahnberg et al. 2005] and a discussion of variability practices in Axis Communications which is presented in [Svahnberg et al. 2005] and [Bosch 2000]. The Mozilla foundation, though smaller, shares a lot of its organizational principles with Eclipse. Axis on the other hand is comparable to the CCT case in size and scope of the product line.

Compositional development. This article builds on our earlier work [Prehofer et al. 2008] where we discuss compositionality in relation to large industrial software platforms.

In [Ommering 2002], the author outlines a component technology and development methodology for constructing software products from a platform of software components. In other words, it proposes a compositional style of development (as opposed to a de-compositional or integration-oriented approach). We used the article as the basis for one of our case studies and the evaluation indeed identifies many compositional elements in his approach. However, it still retains several integration-oriented elements from the open source cases we evaluated (see Section 3.2). While successful, we expect that his approach will need to adopt more compositional elements in the future allow Philips to scale development even further. The vision is further outlined in [Ommering & Bosch 2002] where the vision of moving to a more compositional approach is further detailed with a technical discussion of solutions related to compositionality and variability. In our view, such solutions are highly suitable for application in base platforms such as discussed in our article and found in all four cases we discuss.

In [Krueger 2006], the author identifies similar threats to traditional top down, integrational SPL development ("the first generation of software product line methods" in his words) as we do. Part of his suggested solution is to modularize currently monolithic feature model based approaches and to use so-called compositional profiles, which is similar to the notion of partial integration we discussed in Section 2.

Both Van Ommering and Krueger write from the perspective of a company that maintains its own platform and derives its own software products from it. In this article we broaden the scope to situations where many stakeholders and component owners are involved.

Variability management. In previous works, we published articles on feature modeling and variability realization techniques [Svahnberg et al. 2005]. A recent article discusses variability in the context of compositional development [Gurp 2007]. Feature modeling is a useful technology when centrally managing features for a software product line and similar techniques are also used in the FAST and PASTA methods discussed in [Weiss & Lai 1999], FODA [Kang et al. 1990], FORM [Kang et al. 2004] and ConIPF [Hotz et al. 2006]. Additionally, as argued in this article, adopting compositional product line development involves decentralization of requirements, architecture and development activities. Consequently, component variability is introduced locally at the component level rather than centrally at the product line architecture level. As discussed in Section 2.4 and in [Krueger 2006], this makes central management of feature models mostly redundant. Adapting these feature modeling centric approaches to compositional development will present some new research challenges. However, these are beyond the scope of this article. Feature models of course can still be useful for product configuration and derivation in a similar fashion as KOALA is used [Ommering 2002] or for guiding product composition as proposed in [Hotz et al. 2006]. KOALA is of course part of the software architecture of the case study we discuss in this article.

Agile Development methods. Feature and requirements driven development can also be done locally (as we propose) and is key to agile development [Cockburn 2002][Beck 1999] using localized processes, as we discuss in Section 2. Also increasingly associated with agile development is the notion of test driven development [Beck 2002][Nilsson 2006]. Test driven development practices are common in both Mozilla and Eclipse. Both projects make use of automated and very extensive test suites. Adopting such practices in component development methodologies can, partially compensate for the disappearance of full platform integration (a consequence of adopting compositional development). In addition, any automated tests may be reused during product integration testing to verify that product development does not break reused components unintentionally.

The practices associated with scrum [Schwaber 1995] overlap with the compositional practices listed in this paper. In scrum, self-organizing teams are formed that prioritize issues in a backlog and address these in the order of priority in short iterations (sprints). Requirements, architecture, design, testing, etc. are all part of the work a scrum team does. Scrum is widely used in our organizations and the first author is a certified scrum master.

Components. While the compositional approach can be combined with Commercial Off The Shelf (COTS) components and Component Based Software Engineering (CBSE) practices such as outlined in e.g. [Brown & Wallnau 1999] or [Szyperski 1997], the focus is here more on organizational aspects. COTS and CBSE practices are important for creating large, distributed software systems [Boehm & Sullivan 2000]. However, they have failed to create a market for reusable COTS components based on these techniques [Lang 2001].

Aside from niche markets such as the market for visual basic GUI components (based on the COM standard), these techniques are generally used as a platform and not to create reusable components for third parties [Wallnau et al. 2002]. What differentiates the open compositional approach, as well as the SPL approach (to which we contrast it) or open source development (by which the approach is inspired), is the focus on organization and development practices rather than just the technical composability of components.

In [Meyer & Seliger 1998], the authors note that it is possible to do concurrent platform development with small teams, very much like we do. Additionally they note that many applications exist now that act as a platform for third party plugins and that it is those plugins that provide differentiating features rather than the core application platform. However the approach they outline in that article for building such platforms is still top down and based on a business analysis and central architecture design.

Open source development. Others have advocated the adoption of elements of open source development in commercial development. E.g. [Mockus et al. 2002] did an extensive case study on the development practices in the Apache and Mozilla projects. Additionally, decentralized decision making is a key element in the Cathedral and the Bazaar [Raymond 1999]. Tim O'Reilly notes that extensibility and substitutability is an important factor in the success of open source projects since it enables the use of the software in contexts not foreseen by the developers of the software [O'Reilly 1999]. Finally, in [West 2003], West discusses the transition of three major software companies from a proprietary closed platform to a (partially) open source platform. West focuses more on economics and motivations of openness than on the practices related to this. In this article the focus is not on openness but rather on the practices used in open source communities and their applicability to application in large scale, commercial software development. Case studies such as the Philips case study we discuss clearly show the benefit of using such practices in a proprietary platform as well and we illustrate in this article that there is a clear benefit in further adopting practices from the open source community.

The aim of the open compositional approach is similar in that it intends to maximize flexibility to mix and match components for product developers (rather than being overly constrained by a product line architecture). Contributions (including our own [Gurp 2006]) to, and discussion at the first Open Source & Product Line workshop at SPLC 2006 and follow up workshops and a Dagstuhl session on this topic [Bermejo et al. 2008], also contributed to the compositional approach we introduce in this article. Finally, the issues that we elaborate on and address in this article were first outlined in [Bosch 2006].

5 Conclusions

The motivation for writing this article comes from our observation that current integration-oriented SPL development approaches are experiencing increasing inefficiency in response to increasing system scale, increasing use of global development teams, and increasing inter-organizational development. The resulting productivity for, especially large, systems is very low, which reduces the competitive position of companies maintaining it and depending on it. Additionally, it exposes it to disruption by more nimble competitors.

On the other hand, large open source projects are successful in reuse and variability. To capture this, we have presented a model for an open compositional software development approach that describes software architecture, organizational model and development practices. Instead of a highly centralized software process, these approaches use minimal centralized mechanisms and instead rely heavily on self-organization and architecture-centric coordination. Some elements for central management & governance may exist for the purpose of defining strategy, roadmaps and for guiding the development of the other two entities, components and product development. However, there is a healthy skepticism towards any mechanism that reduces the freedom of individual developers and teams.

The main contribution of this paper is a list of practices that are characteristic of the compositional approach. Based on these practices, we have compared four software projects, covering both integration-oriented SPLs and open source projects. While all of the considered cases studies are successful in their scope, the question is to understand which practices are suitable in what context. Our evaluation shows that large-scale open software development can be performed successfully using practices that widely differ from SPL practices. The suggested trend

is that many software organizations find themselves part of an increasingly large ecosystem that develops the software they productize and consequently that a more compositional style of development is more appropriate. Although such an evaluation is inherently limited by the availability of details, we have used the data available in typical open source software processes. In this way, we consider such an evaluation as an important step towards an empirical evaluation method of software processes.

In the future, we intend to study additional cases to validate and evolve our evaluation framework. In addition, we aim to develop clear recommendations for architects, engineers and technical managers looking to determine the best way to organize software development in their organizations. Finally, we aim to evaluate the role of architecture in the composition-oriented approaches in more detail as it provides a very powerful and efficient coordination mechanism.

5.1 Acknowledgements

In the process of writing this article, we received valuable feedback from colleagues inside and outside Nokia in discussions and workshops. We would particularly like to thank Mikko Raatikainen from SoberIT, and Juha Savolainen from Nokia Research Center as well as the participants of the Dagstuhl session on this topic [Bermejo et al. 2008].

6 References

[Amor-Iglesias et al. 2005] J. Amor-Iglesias, J. M. González-Barahona, G. Robles-Martínez, and I. Herráiz-Tabernero, Measuring Libre Software Using Debian 3.1 (Sarge) as A Case Study: Preliminary Results, "Upgrade: the European journal for the Informatics Professionals", vol 6(3), June 2005.

[Barth et al. 2006] A. Barth, A. di Carlo, R. Hertzog, C. Schwarz, "Debian Developer's Guide", http://www.us.debian.org/doc/developers-reference/, 2006.

[Bass et al 2003] L. Bass, P. Clements, R. Kazman ,Software Architecture in Practice, Addison-Wesley , 2003.

[Batory et al. 2000] D. S. Batory, C. Johnson, B. MacDonald, D. von Heeder, Achieving Extensibility Through Product-Lines and Domain-Specific Languages: A Case Study, in Proceedings of International Conference on Software Reuse 2000, pp. 117-136, 2000.

[Beck 1999] K. Beck, Extreme Programming Explained, Addison Wesley, 1999.

[Beck 2002] K. Beck, "Test Driven Development", Addison Wesley, 2002.

[Bermejo et al. 2008] J. Bermejo, B. Lundell, F. van der Linden, Combining the Advantages of Product Lines and Open Source, Dagstuhl Seminar 08142, 2008.

[Boehm 1981] B. Boehm, Software engineering economics. Englewood Cliffs, NJ, Prentice-Hall, 1981.

[Boehm & Sullivan 2000] B. W. Boehm, K. J. Sullivan, Software economics: a roadmap, ACM Proceedings of the conference on The future of Software engineering, pp 319-343, 2000.

[Bosch 2000] Jan Bosch, “Design & Use of Software Architectures - Adopting and Evolving a Product Line Approach“, Addison-Wesley, 2000.

[Bosch 2006] Jan Bosch, Expanding the Scope of Software Product Families: Problems and Alternative Approaches, Proceedings of the 2nd International Conference on the Quality of Software Architectures (QoSA 2006), June 2006.

[Brown & Wallnau 1999] A.W. Brown, K.C. Wallnau, "The Current State of CBSE", In Proceedings of Asia Pacific Software Engineering Conference, Workshop on Software Architecture and Components, IEEE Computer Society, 1999.

[Buschmann et al. 1996] F. Buschmann, C. Jäkel, R. Meunier, H. Rohnert, M. Stahl, “Pattern-Oriented Software Architecture - A System of Patterns“, John Wiley & Sons, 1996.

[CCT 2008] Command and Control Technologies Corporation, http://www.cctcorp.com/spotlight.html, 2008.

[Clements & Northrop 2001] P. Clements, L. Northrop, Software Product Lines Practices and Patterns, Reading; MA, Addison-Wesley, 2001.
[Clements et al. 2001] P. Clements, S. Cohen, P, Donohoe, L. Northrop, Control Channel Toolkit: A Software Product Line Case Study, technical report CMU/SEI-2001-TR-030, Software Engineering Institute, Pittsburgh; PA, 2001.
[Clements & Northrop 2003] P. Clements, L. Northrop, Salion, Inc.: A Software Product Line Case Study, Technical report CMU/SEI-2002-TR-038 ESC-TR-2002-038, Software Engineering Institute, Carnegy Mellon, 2003.
[Cockburn 2002] A. Cockburn, Agile Software Development, Addison-Wesley 2001.
[Debian.org 2009] Debian Releases, http://www.debian.org/releases/.
[Eclipse 2003] The Eclipse Foundation, Development Process - Revision 1.0, http://www.eclipse.org/org/documents/Eclipse Development Process 2003_11_09 FINAL.pdf, November 7, 2003.
[Eclipse 2006] The Eclipse Foundation, Eclipse Platform Overview, http://www.eclipse.org/eclipse/eclipse-charter.php, 2006.
[Eclipse 2009] Eclipse Galileo Release Now Available, http://www.eclipse.org/org/press-release/20090624_galileo.php, June 2009.
[Gurp et al. 2005] J. van Gurp, S. Brinkkemper, J. Bosch, Design Preservation over Subsequent Releases of a Software Product - A Case Study of Baan ERP, Journal of Software Maintenance and Evolution: Research and Practice, 17(4), pp. 277-306, 2005.
[Gurp 2006] Jilles van Gurp, OSS Product Family Engineering, First International Workshop on Open Source Software and Product Lines at SPLC 2006, Conference PDF, p367-376 http://splc.net/prev-conferences/splc-2006.pdf..
[Gurp 2007] Jilles van Gurp, Variability Management and Compositional SPL Development, in proceedings of Software and Services Variability Management Workshop - Concepts, Models and Tools, HUT-SoberIT-A3, pp. 81-86, Helsinki University of Technology, 2007.
[Hotz et al. 2006] L. Hotz, T. Krebs, K. Wolter, J. Nijhuis, S. Deelstra, M. Sinnema, J. MacGregor, Configuration in Industrial Product Families - The ConIPF Methodology, IOS Press, ISBN 1-58603-641-6, July 2006.
[Jarrad 2006] Salah Jarrad, Product Line Adoption:. A Vice President's View. & Lessons learned, invited presentation at the 2006 Software Product Line Conference, Conference PDF, p498-525 http://splc.net/prev-conferences/splc-2006.pdf.
[Jazayeri et al. 2000] M. Jazayeri, A. Ran, F. Van Der Linden, “Software Architecture for Product Families: Principles and Practice”, Addison-Wesley, 2000.
[Kang et al. 1990] [Kang et al. 1990] K. C. Kang, S. G. Cohen, J. A. Hess, W.E. Novak, A.S. Peterson, “Feature Oriented Domain Analysis (FODA) Feasibility Study“, Technical report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.
[Kang et al. 2004] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, M. Huh, FORM: A feature-oriented reuse method with domain-specific reference architectures, Annals of Software Engineering, 5(0), pp. 143-168, 1998.
[Kiczales et al. 1997] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J. M. Loingtier, J. Irwin, Aspect-Oriented Programming, Proceedings European Conference on Object-Oriented Programming (ECOOP) 1997, pp. 220--242, 1997.
[Krueger 2006] C. W. Krueger, New Methods in Software Product Line Development, in proceedings of the 10th International Software Product Line Conference (SPLC 2006), 2006.
[Lang 2001] B. Lang, Overcoming the Challenges of COTS, news @ SEI, 4(2) http://www.sei.cmu.edu/news-at-sei/features/2001/2q01/feature-5-2q01.htm, 2001.
[Meyer & Seliger 1998] M. H. Meyer, R Seliger, Product Platforms in Software Development, Sloan Management Review, 40(1), pp. 61-74, 1998.

[Mockus et al. 2002] A. Mockus, R. T. Fielding, J. D. Herbsleb, Two Case Studies of Open Source Software Development: Apache and Mozilla, ACM Transactions on Software Engineering and Methodology, 11(3) pp. 309-346, 2002.
[Muccini & Van der Hoek 2003] H. Muccini, A. van der Hoek, Towards Testing Product Line Architectures, Elsevier Electronic Notes in Theoretical Computer Science, 82 (6), pp.1-11, Sep 2003.
[Nilsson 2006] J. Nilsson, "Applying Domain-Driven Design and Patterns: With Examples in C# and .NET", ISBN 0321268202, Addison-Wesley Professional, 2006.
[Oeschger & Boswell 2000] I. Oeschger, D. Boswell, Getting your work into Mozilla, http://www.oreillynet.com/pub/a/mozilla/2000/09/29/keys.html
[Ohloh Eclipse 2009] Ohloh, Eclipse Platform project, http://www.ohloh.net/projects/eclipse, 2009.
[Ohloh Debian 2009] Ohloh, Debian Gnu/Linux, http://www.ohloh.net/projects/debian, 2009.
[Ommering 2002] R. van Ommering, Building product populations with software components, proceedings of Proceedings of the 24rd International Conference on Software Engineering (ICSE 2002), pp. 255-265, 2002.
[Ommering 2004] R. Van Ommering, Building Product Populations with Software Components, Ph. D thesis, University of Groningen, 2004.
[Ommering & Bosch 2002] R. van Ommering, J. Bosch, Widening the Scope of Software Product Lines-From Variation to Composition, Proceedings of the Second International Conference on Software Product Lines, Springer, pp. 328-347, 2002.
[Prehofer et al. 2008] C. Prehofer, J. van Gurp, J. Bosch, Compositionality in Software Platforms, in A. De Lucia, F. Ferrucci, G. Tortora, M. Tucci eds., Emerging Methods, Technologies and Process Management in Software Engineering, Wiley, 2008.
[Raymond 1999] E. S. Raymond, The cathedral and the bazaar. http://catb.org/~esr/writings/cathedral-bazaar/, 1999.
[O'Reilly 1999] T. O'Reilly, Lessons from Open Source, Communications of the ACM, 42(4) pp 33-37, 1999.
[Schwaber 1995] K. Schwaber, Scrum development process, OOPSLA'95 Workshop on Business Object Design and Implementation, http://jeffsutherland.com/oopsla/schwapub.pdf, 1995.
[Shuttleworth 2008] M. Shuttleworth, The art of release, http://www.markshuttleworth.com/archives/146, May 2008.
[SEI 2006] Software Engineering Institute, Product Line Hall of Fame, http://www.sei.cmu.edu/productlines/plp_hof.html, 2006.
[Svahnberg et al. 2005] M. Svahnberg, J. van Gurp, J. Bosch, "A taxonomy of variability realization techniques, Software Practice & Experience", 35(8), pp. 1-50, 2005.
[Szyperski 1997] C. Szyperski, "Component Software - Beyond Object Oriented Programming", Addison-Wesley, 1997.
[Wallnau et al. 2002] K. Wallnau, S. A. Hissam, R. C. Seacord, “Building Systems from Commercial Components“, Addison-Wesley, 2002.
[Weiss & Lai 1999] C. T. R. Lai, D. M. Weiss, “Software Product-Line Engineering: A Family Based Software Development Process”, Addison-Wesley, 1999.
[West 2003] J. West, How open is open enough? Melding proprietary and open source platform strategies, Elsevier journal of Research Policy 32(7), pp. 1259-1285, 2003.
[Wheeler 2002] David Wheeler, More Counting Source Lines of Code (SLOC), http://www.dwheeler.com/sloc/, 2002 (last verfied 05-12-2006).