Master's Thesis

Managing product variants in a component-based system

Jacob Gradén D01 & Anna Ståhl

Department of Computer Science
Faculty of Engineering LTH
Lund University, 2009

ISSN 1650-2884
LU-CS-EX: 2009-33

Managing product variants in a component-based system

Jacob Gradén, Anna Ståhl

Supervisor: Lars Bendix
Lund University

November 13, 2009

Abstract

Today's markets are fast-paced, with many different customers and requirements on products. More than ever before, it is necessary to be able to provide each customer with a *tailor-made product*, corresponding to just that customer's needs. At the same time, maintaining different products is costly and strains resources. By reusing code and producing tailored variants of the same basic product, the customer's requirements can be met while keeping costs under control.

Component-based systems are becoming a popular way of managing product variants and promoting code reuse. Component-based systems are based on stand-alone components which can be combined in various ways to produce different product variants – essentially using the same building blocks to construct different products.

However, the many different requirements and product variants introduce *complexity* which needs to be managed while retaining flexibility, so that creating product variants is facilitated. This means that not only components and products must be managed, but also information pertaining to them, such as technical relationships between components and business requirements on products.

This master thesis suggests a *support tool* to help in creating and managing the different components and products, and outlines the capabilities such a tool should have and the opportunities it would present.

Contents

List of Figures

List of Tables

1 Introduction

This section outlines why the thesis is interesting and the problems it tackles. The purpose is also presented, as well as the thesis structure.

The world is moving from standardized mass-produced software controlled by the producer, to tailor-made products with wide possibilities for end-users to select just the features they want – or even change and update their products after delivery. To perform well on such a market, companies must be able to supply many different but similar products, and to do so quickly.

Traditionally, large software systems have been written as one entity, which has grown over time. This is the monolithic approach. A more modern approach is that of the component-based system (CBS), where smaller parts of the system are developed separately and the system as a whole (the product) is constructed by combining the parts. Different variants of the product can be achieved by choosing from the available parts, which enables a high level of code reuse and cuts down on the time needed to create a new variant.

Managing CBSs is however non-trivial. Not only must they support the same functions as the monoliths do; they also introduce problems of their own.

1.1 Problem

At its core, the central problem is simply this: *How can the variants of a component-based system, and their constituent components, be managed efficiently?* This is however a large field, with many facets.

A component-based system can be very flexible. Components are meant to be mainly independent, which means they can be combined freely to create any number of variants. There may however be relationships between components – for example when one component needs another to function properly, or when two components are mutually exclusive.

The resulting combination of components (the *configuration*) may also impose restrictions on possible combinations. For example, components may belong to different layers of the system – such as operating systems and applications – and one restriction might be that at least one component from each system layer is required. Components may also have properties, and another restriction might be that all components in the configuration have the same value for a specific property.

There is a lot of complexity involved as well. The sheer number of components in a large CBS makes it difficult to manage manually, and when components are developed and exist in different versions over time, the amount of components becomes staggering. Add to that the possibility to create configurations, and variants thereof, by combining components in different ways, and it is clear that complexity will become a major hurdle.

1.2 Purpose

A great deal of flexibility can be achieved by working with a CBS, but it comes at the cost of complexity. The ultimate purpose of this study is to allow for that flexibility by helping to manage the complexity.

One way to manage such complexity is manual labor, which has obvious drawbacks – lack of speed, exactness etcetera. A tool-supported approach would be preferable, for example in the form of an application which helps with the task of combining the different variants of the system from its constituent parts. This thesis aims to present an analysis of the considerations that would apply to such a system. Benefits and drawbacks are discussed, and a possible application design is suggested.

1.3 Business case

Using CBSs allows for great flexibility in creating variants of a product, facilitates parallel development and can reduce compilation times. Using a support tool for managing CBSs keeps the time and costs down, simplifies communication between departments, allows for statistical analysis of components and product variants, and greatly reduces the risk of creating broken or nonsensical product variants.

1.4 Context

While the problems outlined above may occur in many different disciplines, this study has limited the scope to software development. This is a large field, used by a plethora of companies in all kinds of industries, and the results are applicable to any of them who work with variants of software systems.

1.5 Thesis structure

The Background introduces concepts used in the rest of the thesis and which the reader should have some familiarity with. The Analysis explains in more detail the problems outlined above, and presents possible solutions. These solutions are utilized in Design, together with supporting structures, to create a full picture of a system for managing product variants in a component-based system. Limitations, future work and a general discussion are found in Discussion, and the most important findings from Analysis, Design and Discussion are presented in Conclusions.

2 Background

This section presents concepts necessary for understanding the rest of the thesis. It can be safely skipped if all areas are familiar.

There are many challenges to overcome, and potential gains to benefit from, when introducing tailor-made products to customers. On a technical level, there are different ways of managing the customization process, and on an organizational level, there are questions regarding how to make the customization process quick and easy enough to be valuable. These areas are briefly presented here, together with a general background of neighboring subjects. First, though, it is important to understand why variation occurs in the first place.

2.1 Causes of variation

Variation is the concept that a defined piece of software may look and function in different ways depending on the situation. The archetypal example is that variation in a software system is introduced by the producing company in order to allow for both customization and code reuse at the same time. This is done by inserting *variation points* into the software – places where variation may be introduced [12]. An example of a variation point is the background color of an application; another may be which applications are part of an application suite.

Variation points need to be given specific values in order to be useful. When all variation points have been given values, a *variant* is created [12]. Variants differ from each other by having different values for the same variation points, or possibly even different variation points. If the same source code is used to create two different variants, they typically have the same variation points but different values for at least one of the points; if most of the source code is the same, but one of the variants has more source code than the other, there may be a difference not only in values but also in variation points. It is also possible for two variants of the same program to have no common source code at all, but this is inefficient.

Introducing variation provides the possibility to choose between several options and allows for the development of customized products. This can be necessary for different reasons: customer requirements, laws, technical reasons, etcetera. When companies are customers, they are usually interested in what is visible to the end-user – such as their brand-logo or the portfolio of applications included. Variation could also be caused by legal demands – for example a law that forbids inclusion of a specific language in products that will be sold in a certain country. Customer requirements and legal demands are usually introduced on a high abstraction level, whereas technical reasons are more specific – for example, a program can be built for several different devices with different screen sizes.

Variability is the technical equivalence of variation. While variation encompasses all kinds of changes, variability is sub-divided into specific forms. Customer requirements and legal demands are examples of *external variability*, as opposed to technical reasons, which cause *internal variability*. Internal variability may be very important to customers, but they are never aware of it. Screen size is a typical example of this: the development team realizes that the same basic program can be created in different variants and create the variation point *screen size*, which is never seen by the customer, but is used during development. External variability, by contrast, corresponds to variation points which the customer is aware of. Internal variability often appears when refining external variability, for reasons such as maintenance, scalability, portability and so on.

The variability pyramid in Figure 1 [12, p. 72] shows abstraction levels of the development process and illustrates the amount of variability at each abstraction level. External variability is most common on a high abstraction level, while internal variability becomes more common further down on a more detailed level.

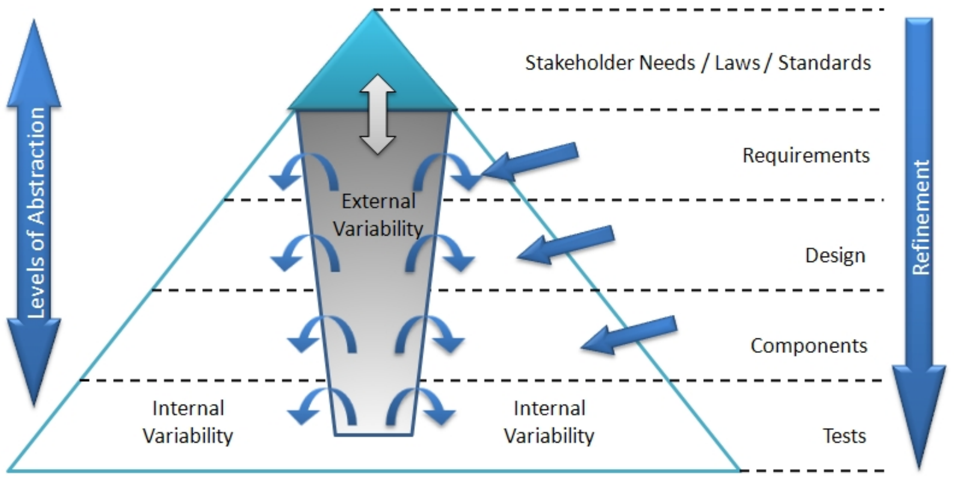


Figure 1: Variability pyramid

The variability described above is *variability in space*, which is when different variants are intended to coexist at the same time. There is also *variability in time*, which is a completely different matter. Variability in time implies that one variant is meant to replace another; typically, only one (the latest) is ever used. In common configuration management vocabulary, variability in space gives rise to *variants*, whereas variability in time produces *revisions* [10]. There are instances when older revisions are necessary, but as a general rule the latest revision is always the best – all else being equal.

In order to actually use the possibilities of variation – supplying variants of products to customers – the variation points must be given specific values, so that variants are created. This can be done in two different ways: *static configuration* or *dynamic configuration*.

2.2 Static vs. dynamic configuration

Product variants can be created either before or after compilation, using several different methods. Choosing a variant is a form of configuration, and doing so before compilation is called *static configuration*, whereas choosing after compilation is called *dynamic configuration*. Statically configured products cannot be changed, which is the reason for the name, whereas dynamically configured products can.

Static configuration is done by selecting source code for compilation. This can be done simply by compiling different files for different products or by using compiler features such as conditional compilation, where directives in the source code inform the compiler about which code to compile and which code to ignore.

This has the advantages that each variant becomes small, that it is very simple to create new product variants – simply change the source code and recompile – and that a great deal of flexibility can be achieved; but the drawbacks are that a lot of work is required to define each variant (not to mention making sure that all *other* variant still work) and that a full recompilation must be done for each variant. For small systems with only a few variants, this is not a serious problem, but for large systems with many variants, a lot of time and resources are required.

Dynamic configuration eases these pains. In a perfectly dynamic configuration, each line of source code needs to be compiled only once, no matter how many variants are created. The simplest way of achieving this is to move all decisions regarding variants into the executing code itself; the program may for instance read a settings file when it starts, and only then will the decision be made on which variant is actually in use. This can be accomplished for example by changing from conditional compilation (such as #ifdef) to ordinary control structures (if).

A drawback is that dynamically configured products potentially become very large. If the specific variant to be used is determined when the program is started, all variants must exist until then, and one of them must be chosen. If there are a lot of variants, this could cost dearly in space requirements. Depending on the specific technique used, dynamic configuration may also require a lot of manual labor, just like static configuration.

A completely dynamic system also has the problem that the total amount of choices may not always make sense in all combinations. Imagine for instance a program which can have the background-color green or blue, and which can also work either in graphical mode or text mode. It makes perfect sense to set a color when the program is running in graphical mode, but not so much when it is in text mode. If a naive method is used – simply looking at the settings one by one, disregarding their interdependencies – the program may even crash.

The very best of both worlds is of course to cut down on compilation times and manual labor, and still produce small programs in all the right variants and with no incompatible choices. One of the most promising ways for doing so is component-based systems, as opposed to the monolithic structures which are common to many software systems today.

2.3 Monolithic vs. component-based systems

There are two extremes on how to design software systems: All parts of the system can be fully interconnected with all others, or each part can be separated from the others as much as possible. The former is the *monolithic* approach, and the latter is that of a *component-based system* (CBS). Both have advantages and disadvantages.

Monolithic systems can require less design and architecture, because all parts are allowed to affect each other. It can also be easier to make changes and implement new functionality: instead of adding logging to one part, some other part can be modified if it is easier to implement logging there. Since there are no boundaries, it is also simple to add features which concern many different parts – simply add the necessary code to each one of them. However, these advantages turn to disadvantages as time passes.

Quick changes made to different parts of the system without a clear design scale badly. If the logging feature later needs to be modified, all code pertaining to it must first be found – and if it is spread out all over the system, this may not be easy. As many different features are added in many different places, the system risks deteriorating into a state where even simple modifications take a lot of time and require extensive regression testing to make sure that everything else still works. The basic problem with this is that a lot of work has to be performed which provides no utility – while modifying code does provide utility, finding it and testing the system afterwards does not.

Even in well-designed monolithic systems, there are still – by definition – connections between different parts. This means that modifying just one part becomes difficult, since it is likely that the changes will affect other parts as well, in ways which are sometimes subtle and hard to catch with tests. Impact analysis is also made very hard, since changes to one part may cause ripple effects in other, completely unexpected, areas of the source code. Additionally, when variation points cannot be introduced, parts of the system – sometimes the entire system – must be duplicated and changes made, in order to create variants. This introduces the problem of double maintenance [2], which means that all changes which are made to the common part of the two copies must be maintained for each copy.

On a final note on drawbacks with monoliths, there is the problem of size and compilation times. If completely static configuration is used, the resulting product may be kept just large enough for its purpose, but this requires compiling all source code pertaining to each variant, which can easily take hundreds of hours for a complex system with many variants. Also, even though each variant is not very large, the combined size of variants can become a problem since all of them must be stored if reproducibility is to be maintained[1]. If completely dynamic configuration is used, compilation only has to be done once, but the resulting binaries will contain all different variants and will therefore be quite large. Compromises are of course also possible, but there will always be a trade-off between size and compilation times.

CBSs are built around the idea that different parts of the software should be clearly separated into different components, in stark contrast to the monolithic idea. This requires careful design and architectural decisions and therefore has a longer start-up period and is generally harder than development on monolithic systems. It also requires continual work on design and architecture, and from time to time new parts must be created and old ones removed. Introducing new features may also require more work than would be required in a monolithic system, for example in the form of introducing a component which performs logging, and then calling that component from all other components in the system which require that feature.

The primary advantage, however, is that even though more time is spent on implementation seen feature-for-feature, the design is kept clear and it is easier to locate where changes must be made. Theoretically, it is also possible to perform regression testing on *just* the affected component or components once changes have been made, though in reality more code must be tested. Even so, testing can be made more efficient by focusing the test efforts on the components where faults can primarily be expected to reside, and then performing more cursory tests on the rest of the system. In other words, more time can be spent performing work which provides utility, and less time spent on performing work which simply has to be done, but provides little utility.

CBSs also have advantages in terms of size and compilation times, especially if dynamic configuration is used. In that case, each component only has to be compiled and stored once, which cuts down on both compilation times and storage required. The specific product is then created by pointing to the actual binary components which are part of it. If static configuration is used, there can still be savings because

[1]If *everything* is known about creating the binaries, this is not necessary, but that is very complicated. Exact source code must be stored, as well as exact versions of compilers and other programs used – and the programs must produce deterministic output, which is not always the case when optimization is used. Compiler flags must also be known, and possibly environment data such as the OS version it was compiled on – perhaps the entire computer must be stored.

not all components always need to be compiled in different variants. For instance, even if a dozen product variants are created, they may all have the same values for all variation points in the logging component. This means that that specific component only has to be compiled and stored once. This would be impossible in a monolithic system.

Even in a CBS, there are of course connections between components, as exemplified by the dependence on the logging component above. This is necessary and allowed, but the general rule is that if components have no immediate connections, it should be possible to combine them freely. This is very important because if that property is lost, the system starts moving towards a monolithic system where different parts can affect each other in undocumented ways. There are certainly situations where two nominally unconnected components cannot work together, but it should be avoided if possible and documented when the situation is unavoidable.

Whereas a monolithic system has only one level of variability (variation points in the source code), a CBS has two: Variation points inside each component, and the selection of components which make up the actual product. Product variants are known as *configurations* in CBSs, and the fact that a configuration can consist of different components can in essence be seen as an additional variation point where the available components can be seen as the values to assign to that variation point. This allows CBSs more freedom in creating product variants – the variation can occur either on the level of selecting components, quite simply by including different sets of components, or on the component level, as in monolithic systems.

Figure 2 and Figure 3 illustrate how components are created from the underlying source code and added to the repository. This thesis is concerned mainly with components which contain compiled binaries, but it is technically possible to create components which contain uncompiled source code.

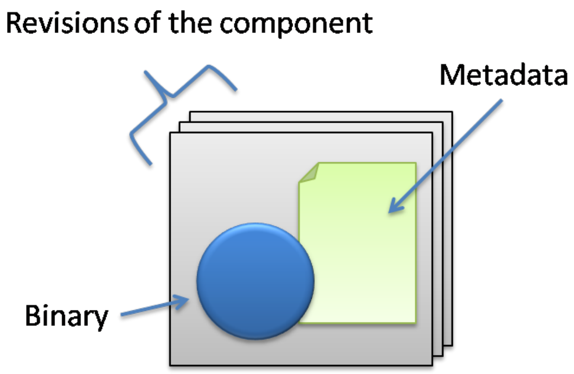


Figure 2: Component

The binary in a component is quite simply the compiled source code file or files. Technically, there could even be several binaries inside one component, for example if the main executable of a component requires binary libraries. The metadata contains properties and information regarding the component. Section section 3.2 explains components in more detail.

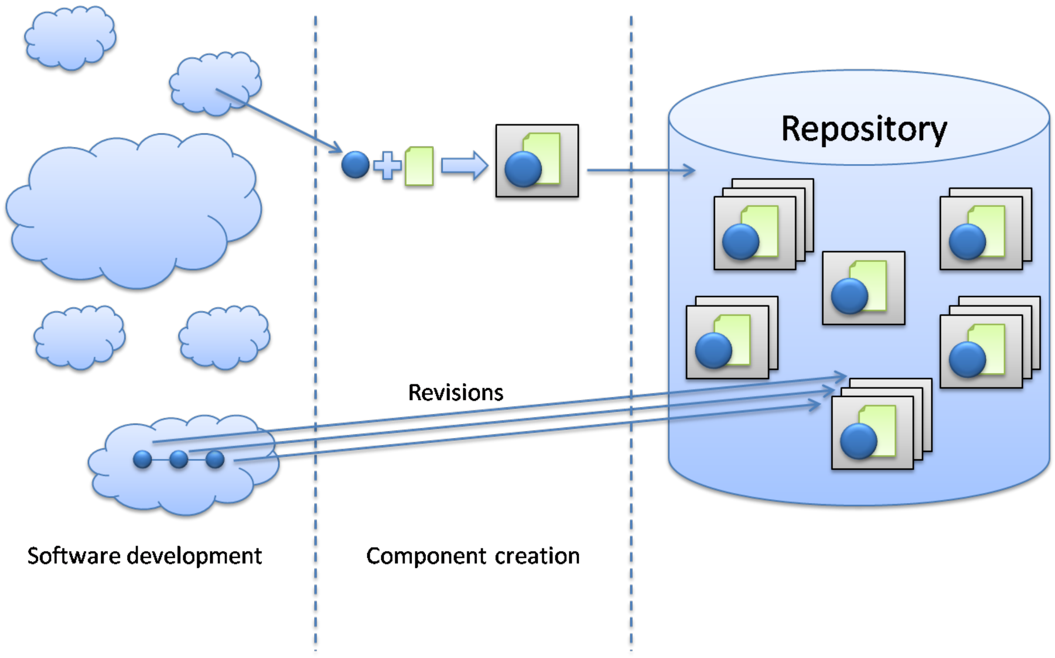


Figure 3: Component creation process

The remainder of this thesis works from the assumption that the a component-based system is used, primarily together with dynamic configuration; a CBS is a necessity, but the type of configuration is not strictly required to be dynamic.

3 Analysis

This section presents an analysis of the challenges and benefits related to component composition. Different alternatives for overcoming each challenge are explored together with their specific advantages and drawbacks, and the necessary requirements for each benefit are laid out.

As established in section 2.3, a component-based system (CBS) solves many of the problems inherent in a monolithic system. Component-based systems are however not without problems – the most obvious being that the price of flexibility is complexity, which must be handled. While this can be done manually, manual labor is more error-prone and slower than an automated system. On the other hand, automated systems only go so far – some things must still be performed manually, such as deciding what a configuration should look like and resolving any problems for which there are no preexisting automation rules.

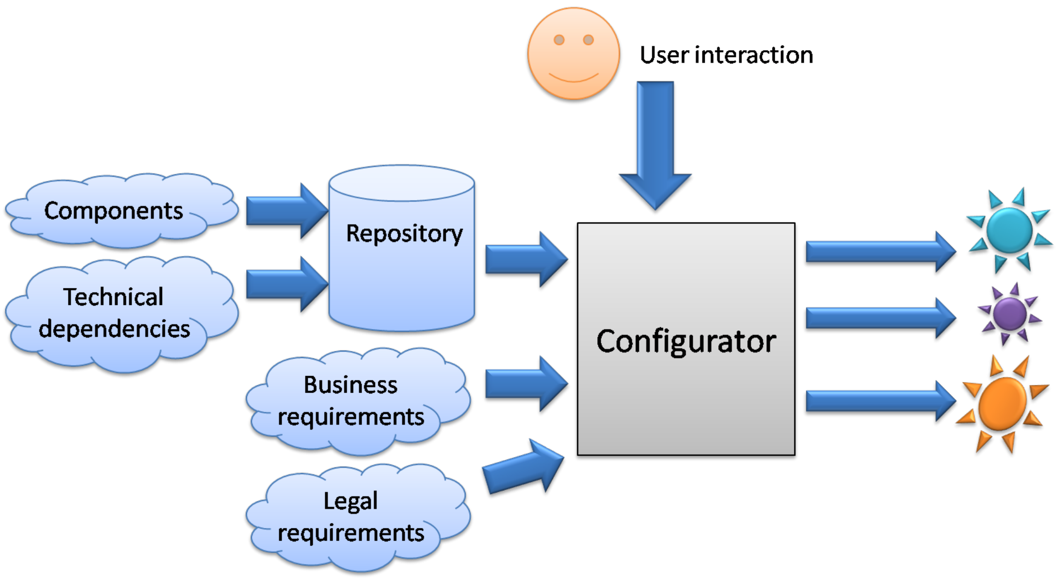


Figure 4: The composition process

A configuration consists of many different components, each of which has properties and may have relationships with other components, and in addition, the configuration itself may have properties. From a technical perspective, the central challenge with component-based systems is how to create configurations which are both complete and consistent. Completeness means that all components which should be present are in fact present, and consistency is the requirement that all components are able to function together. There are also the questions of how to store components, configurations and their properties, and how to ensure that testing and quality assurance is possible and efficient. These issues are analyzed in the following sections.

This section takes the challenges with CBSs as a starting point and analyzes what would be required of a semi-automatic system designed to aid in the process of composing complete and consistent configurations – a *configurator*, which would work along the lines of Figure 4. The first step is to identify possible use cases.

3.1 Use cases

There are several strategies for creating a configuration. Different users might use different strategies; one user might not care about which operating system (OS) is used since the focus is on choosing specific applications, while another user might start by choosing the latest OS version, since a new feature from that version is required. A component-based system opens up for both these approaches, and more.

Figure 5 shows how a configuration can be created from components of different types, and how those components can be connected. More details on this are presented in section 3.2 and the following sections.

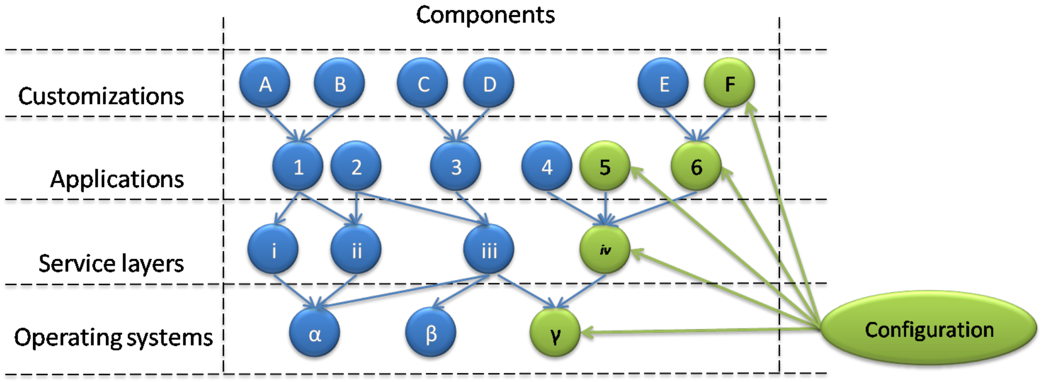


Figure 5: Anatomy of a configuration

The term *producing company* denotes the company which develops components and releases configurations. The term *user* denotes somebody who uses the configurator. This is typically an employee at the producing company where the configurator is used. *Customer* refers to the company or person who buys the products from the producing company. The *end-user* is the physical person who actually uses the product.

3.1.1 Day-to-day usage

In daily work, the configurator is used to assemble components into configurations and to work with those configurations. This can be done by a large number of roles, spanning from configuration managers, quality assurance personnel, testers and developers, to customer contacts and possibly even the customer or end-user directly[2].

Bottom-up configuration creation The user creates a configuration by selecting components from the most fundamental to the most specific. The OS is selected first, then the service layer. Next, applications are selected, and finally, customizations are added for a specific customer and market or markets.

The choice of OS limits the available service layers; the choice of OS and service layer together limit available applications; and the choice of OS, service layer and applications limit the available markets and customers.

Top-down configuration creation The user creates a configuration by selecting components from the most specific to the most fundamental. Customizations for customer and markets are selected first, then applications. Finally, a service layer and an OS are chosen which support the required applications.

The choice of markets and customer limits available applications; the choice of markets, customer and applications together limit available service layers; and the choice of markets, customer, applications and service layer limit possible selections of OS.

Mixed-mode configuration creation The user creates a configuration by selecting components and settings in any order. The OS may be selected first, to explore which applications are available for a new version of that specific OS; or the target may be a specific customer on a specific market, and applications, service layer and OS may be of secondary importance.

Selecting a component for inclusion may mean that other components must or must not be selected. A certain application may require a specific version of a service layer; or selecting a specific market may disqualify certain applications because they lack support for it. The choice of all earlier components determines the list of available components for the next choice.

Saving, loading and changing configurations Once a configuration has been created, it should be possible to save it, so that the creator or other persons can

[2]BMW's tool *Build Your Own* (`http://www.bmwusa.com/byo`) is an example of a configurator geared towards customers.

use it later on. It should naturally also be possible to load a saved configuration. Additionally, when a configuration is loaded, it may need to be changed – new versions of components may become available, or changes may be required for some other reasons.

Just like a component, a configuration may exist in several versions. Once a configuration is changed and saved, a new version must be created – that is, changing and saving a configuration must not replace the original, but rather create a new version of it.

Configuration verification The user creates a new configuration or loads an existing to verify that it contains all components which are needed (completeness), that all components can be used together (consistency), that each component has been verified in isolation and that the entire configuration as a whole has been tested. This is primarily useful after a problem has been resolved (see that use case, below), to verify that the configuration is in fact usable.

Configuration inspection The user creates a new configuration or loads an existing, and inspects its properties to gain information regarding it. Properties may include things as simple as which components are part of the configuration, or may be more advanced notions such as which licenses are used by any of the constituent component. Inspection can take the form of pre-defined standard queries or custom queries used on a case-by-case basis, and may answer questions such as *What happens if component X is added to configuration Y* or *Who is impacted by changes to component Z*.

3.1.2 Administration

While daily usage mainly focuses on fetching components from a list (the *repository*) and combining them, administrative usage focuses on maintaining the repository[3]. This is sometimes carried out by configuration managers – when users request changes to the repository – and sometimes by other users, directly, through the configurator.

Creating a new component A new component is required, which will have to be approved, implemented, and tested. The source code and binary data may or may not be present when the component is added to the configurator system; if they are not, the component must not be used until they have been added.

[3]Somebody must also set up process guidelines and directives for how the repository may be used, but this relates more to the organizational process than to the configurator itself.

Changing component status When the status of a component changes, for example as a result of testing, this should be reflected in the configurator. This may also occur when a component should no longer be used and is deprecated, or when components are changed for any other reasons.

Problem resolution The user creates a configuration, selecting components which are desired but currently mutually exclusive. This results in a configuration which contains all needed components but which will not work properly – complete but inconsistent. The configuration is saved and sent to development for analysis and implementation.

Development personnel open the configuration and perform the necessary tasks to make sure the components can in fact work together. The versions of components in the configuration will need to be updated. This results in a properly working configuration. The configuration is saved and the original creator is notified.

3.1.3 Advanced usage

Once the basics are provided for, a few more use cases can be envisioned.

Notifications For large systems, the configurator may have many different users. When one of them changes something – creates a new version of a component, for example – other users may wish to be notified – for example owners of configurations which include earlier versions of that same component. This is however case-specific, since owners of old configurations no longer in use probably have little interest in notifications regarding them.

Statistics Among the more intriguing possibilities with a CBS is the prospect of statistical analysis. The owner of a component may for example be interested in finding out if that component is used at all anymore, to determine if it should be deprecated or not. Quality assurance may be interested in identifying components which are commonly used together, to make sure that they are tested more carefully as a group. Marketing operatives could make use of the knowledge that customers who select one specific component normally select a certain other component as well, and suggest this other component to a new customer who requests the first one.

Quality aid When defects are located in a component and subsequently fixed, a user may wish to identify configurations which might also suffer from that defect – starting with a component and working towards affected configurations. When components need to interact with each other, support can also be provided in the form of indicating which other components, apart from the faulty, that may need scrutiny – if the defect affects the internal workings of a component upon which other components rely, it is possible that their behavior needs to be changed too, as a result of repairing the original defect.

Requirements fulfillment If complete traceability is available, such as being able to find requirements affecting a certain component, the user should be able to start not with components but with requirements. If a specific requirement is selected, components which implement that requirement should be indicated to the user. For example, the requirement "Bluetooth capability" could be selected, which would disqualify all operating systems without that functionality; or the requirement "customer-specific application Alpha-Acme" could be selected, which would be satisfied only by the actual component for the application Alpha-Acme (though possibly by more than one version of that component).

The advantage to this, compared with no link to requirements, is that the person responsible for creating configurations needs no information about which components implement specific requirements, since that is known by the system. By providing this layer of abstraction, the administrative burden of creating configurations which adhere to customer demands, or are in other ways related to requirements, is reduced.

3.2 Components

Component-based systems are built from individual, stand-alone components which interact using well-known interfaces. An analogy is that of meals in a restaurant: There are distinct components (ingredients) which need each other to work properly, and from the same set of components, many different configurations (dishes) can be built. There are also relationships between components (fish should be served with one wine and red meat with another; fish and red meat should not be served together, etcetera). In a software system, a typical component could be an application or the operating system (which is in itself commonly created from sub-components).

Components are developed over time, and changes cause components to come in new *revisions* or new *variants* (see section 2.1). Revisions and variants are collectively called *versions*. They create possibilities, but also complexity. In this thesis, it is implicitly assumed that components (and for that matter configurations) can always occur in different versions. For reasons of brevity, this will be described explicitly only when it is of specific interest – but the issue will be present whenever components or configurations are used.

Components have properties, can belong to different layers in a configuration, may need other components to work properly (relationships) and can be combined into suites of several components. The following sections analyze this in depth, starting with component properties.

3.2.1 Properties

Depending on the actual situation, components may have any number of properties. Among the more intuitive are its name, the date it was created, the date it was last modified and who is responsible for the component – and there can of course be more, depending on the specific situation. For the purposes of creating complete and consistent configurations, however, only those properties which affect the consistency are really interesting. Completeness is not affected by properties, but by relationships (section 3.2.3).

There are many properties which could affect consistency, but they can mostly be handled in the same basic way. Two examples are presented here: *customer* and *market*, which illustrate slightly different cases.

The terms *customization*, *customer* and *market* are used widely in this thesis. A customization is a component which provides settings, images or other data, and allows for customizing the appearance and behavior of another component. Customizations come in two basic variants: Customizations for a specific customer, and customizations for a specific market. These can also be combined, so it is possible to have one customization which provides the necessary information for one customer on, say, five markets at the same time.

A customization may be specific for one customer, or it may be customer-generic; and it may be specific for one or many markets, or it may be market-generic. Generic settings are defined by the producing company and can be seen as sensible defaults, suitable for all customers or markets. Customizing a component means assigning values to one or more variation points.

Customer When configurations are created for a specific customer, that customer may want to tweak the product – for example if the configuration is then re-sold to an end-user, to highlight the customer's brand. This can be handled either by introducing components which are created specifically for a customer, or by having generic components whose appearance or behavior can be modified using special customization components. This is illustrated in Figure 6, where the component to the left is created specifically for one customer, and the component to the right is generic and uses one of the three customization components to receive the right settings. It is obvious that the amount of code reuse can be high when customizations are used.

The first approach is costlier, but allows for more flexibility for the customer. The second allows for more code reuse and is therefore faster and cheaper, but provides fewer opportunities for the customer to customize the component, since no new variation points can be introduced into the component – only new values can be introduced. There is also a middle road, where components specific for a customer

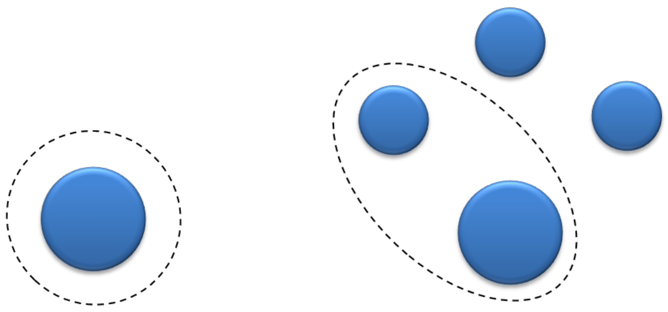


Figure 6: Customizing components

are created, but customization components are still used. This requires a large initial investment for each customer, but once that is done, code can be reused and smaller changes made through the customization component.

If a customer-specific component is created, which may be used only by that customer, it should obviously have the *customer* property set appropriately – the component Alice, for example, could have the customer property set to Acme. The generic component Bob, however, has an empty customer property, signaling that it is generic. If Acme want to customize Bob, this is done by adding both Bob and a customization component, say Charlie, to the configuration. Charlie contains customizations to Bob specific for Acme, and therefore has the customer property set to Acme, but Bob still has an empty customer property. Figure 7 illustrates this.

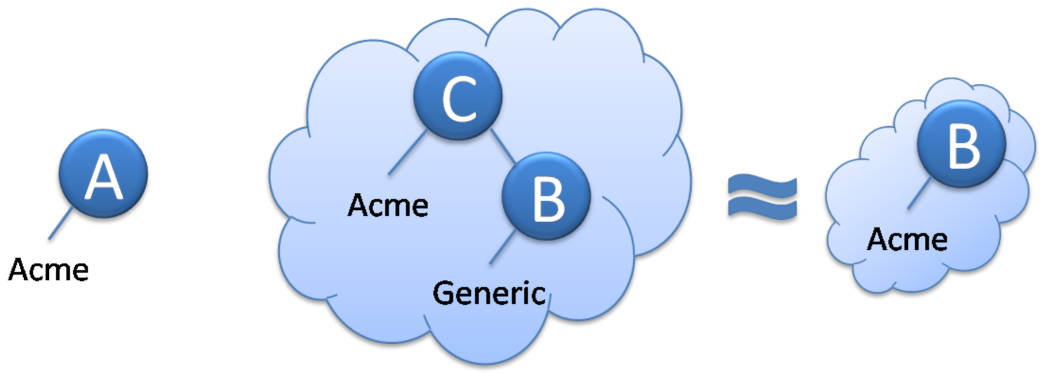


Figure 7: The customer property

A component can have only zero or one values for customer – it is either generic, or targeted for just one customer. This is in contrast to the *market* property, which can have zero, one or many values.

Market Products may be customized not only for a specific customer, but also for a specific market. A market can be a city, country, continent, or any other form of area, and can often be created from several sub-markets. Markets are created when certain areas require specific settings – for example a specific communication protocol or settings requested by the end-users. The main difference is that while it is hard to imagine a product made for two different customers at the same time, a product can very well be made for two different markets.

Sometimes, the market is defined entirely by the producing company; sometimes by the customer. Those markets need not be the same – *Generic: Europe* and *Acme: Europe* could differ in the included countries, for example. There can also be inheritance relationships: the Europe market may require high-level settings which are common for the entire EU area, while the France and Germany markets require lower-level settings which are specific for those areas.

Inheritance can be modeled either internally in the components, in which case it is not visible in the configurator, or between components, in which case it is visible in the form of relationships. In the latter case, components having France as market will indicate that they also need an additional component which has Europe as market and supplies the common settings.

Inheritance is the main reason why a component can have several values for the market property, such as *Europe and France.* Another reason is end-user choice, where a typical value could be *France, or Germany,* which would allow end-users to choose the settings they prefer. However, if only France and Germany are selected, then the common settings for Europe will not be present. To rectify that, the complete value should be *Europe and France, or Europe and Germany.*

Another issue is that of overlapping markets. Including the *Scandinavia* market, for example, may or may not be the same as including the *Sweden*, *Denmark* and *Norway* markets, which may easily lead to confusion when configurations are created. If a number of markets do add up to a common market, there should be some mechanism to identify this and suggest to the user that *Sweden, Denmark and Norway* could be replaced by the *Scandinavia* market.

In order for this to be feasible, it must be possible to inspect settings for all concerned markets, and to know when settings must be exactly identical and when to compose them. The name of a region is an example of a setting which must be exactly identical; whereas a flag corresponding to that region could be composed, so that Sweden, Denmark and Norway supplies one icon each, and Scandinavia supplies all three of them.

It should be noted that a market is not the same as a language – a customization for a specific market may state that some languages should be included, and that other languages must not be included, but markets and languages are not equal.

Customers and markets As mentioned above, markets can be defined either by the producing company or by the customer. While not a strict hierarchy, this implies that the customer property is on a higher level than the market property – for a given customer (or no customer at all), there is a well-defined set of markets.

The opposite view – that markets are on a higher level than customers – could be taken, but since the generic market Europe is not necessarily the same as the Europe market for Acme, this could lead to subtle problems and great irritation among customers. It is also the customer who decide what should exist on a market, not the other way around. Figure 8 exemplifies the hierarchy of component properties with respect to customers and markets.

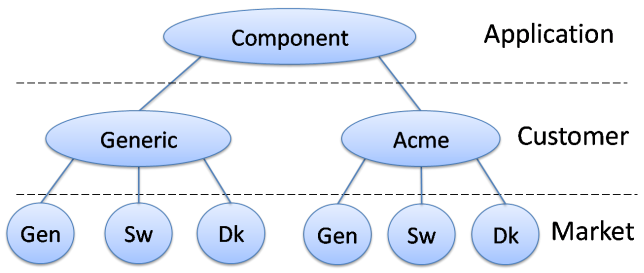


Figure 8: Customer-market hierarchy

It should be noted that even though the generic customer (no customer) is chosen for the component, it is still possible to choose a market for which to customize the component. One of these markets is the generic market (called Gen), which is equivalent to no market customization. If no customer and no market is chosen, the component should simply use its default values. If customer and/or market is set, the appropriate customizations from those should be used. It should also be noted that the generic market Denmark is not necessarily the same as Acme's Denmark – for instance, Greenland may or may not be supported in one of them.

A hierarchical structure does pose one major problem, namely what to do if there are conflicting settings on the customer and market levels. Should the customer settings be used, to ensure that a product is always consistent for a customer no matter which area it is targeted for; or should the market settings be used, to ensure that a product is specifically customized to the greatest possible degree?

The decision could be left to the user, in which case the configurator should provide support to identify the ramifications of each choice; or the situation could be identified as an error and combinations of customer and market which are in conflict could be prohibited from being used. The first alternative offers more direct user control but requires more knowledge from the user, while the second alternative is safer but risks causing delays when customization components need to be updated to work with other customizations.

3.2.2 Layers

Different components serve different purposes; some components interact with hardware while others provide GUI features and are of more interest to end-users. A common approach is to create an architecture where components belong to different layers, such as the one presented in Figure 9. The solid arrows denote an actual requirement – an application may, but does not have to, require that a certain service layer is present – whereas dashed arrows indicate that a customization is made for a component. The customization will work, although fulfill no purpose, even without the component it is made for.

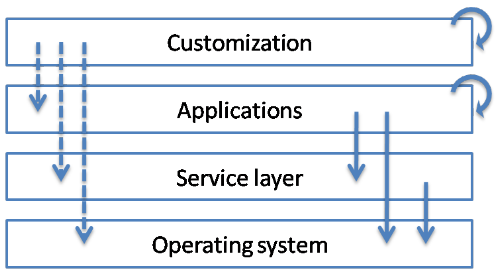


Figure 9: Layers

The operating system is at the lowest layer, mainly with the purpose of talking to hardware devices. The service layer provides a set of services, libraries and APIs and serves as a connector between applications and operating system. Applications are at the third layer, providing features for end-users. Each layer may have customizations[4], making them the top layer.

Layers have different characteristics, such as how many components can be used at the same time in a layer. There can typically only be one operating system in a configuration at a time, and the same goes for the service layer. There can however be many applications, and possibly even several customizations for each application – one customization for each market the application can be used on, for example.

While not a technical requirement, the most intuitively logical design is that components on one layer may be allowed to require that other components are present on the same or lower layers, but not on higher ones. It makes little sense for an operating system to demand that an application is present – if such a situation occurs, the application should either be part of the operating system, or there should be an external requirement from the configuration, which says that the application needs to be included.

[4]From a configurator perspective, there are no technical reasons why components on a certain layer may or may not have customizations, but from an architectural standpoint it might make little sense to allow customizations of the service layer, for example.

Other architectural concerns may also apply – it may for instance be desirable that components may have relationships only with components on the same level or the level directly below, to preserve maximum modularity. The examples in this thesis implicitly assume that relationships between components are from higher to lower layers, or inside the same layer; however, this is not a technical necessity.

3.2.3 Relationships

In their most general form, relationships are connections between two components which say something about how they affect each other. Relationships can have at least three different causes: technical, business-related or legal reasons. Different organizations with different needs may add more relationship types, or not need all of them.

For the purposes of this thesis, the term *relationship* will refer strictly to the *technical relationships* – for example that component Alice will be technically unable to work properly unless component Bob is present. *Business-related* and *legal* relationships are called *rules* and are considered in section 3.6. Technical relationships always relate to components or versions thereof, whereas rules can be more general and relate to components, properties of components, other rules or relationships, etcetera. It also makes sense to manage the technical relationships in close connection to the components they relate to, whereas rules are handled on a higher level of abstraction and are bound less tightly to components.

When the configurator reads relationships between components, they must be tied to specific versions of components, such as *Alice 1.4.3.7*, in order for the system to know exactly which items and versions thereof which are affected. To facilitate understanding for users and ease the administrative burden, it is however better to allow for the creation of more general relationships, such as *Alice* $\geq$ *1.4.0.0*, or simply *Alice*. It is then up to the configurator to translate this to all the specific versions of the component: *Alice* $\geq$ *1.4.0.0* will be translated to *Alice 1.4.0.0*, *Alice 1.4.0.1*, and so on. If only specific relationships, such as *Alice 1.4.3.7*, were allowed, new specific relationships would have to be added every time a new version of Alice or Bob were added, since older versions of Alice may work together with the new version of Bob, and vice versa.

This can be applied to both ends of a relationships, meaning that a relationship between *Alice* and *Bob* is just as valid as one between *Alice 1.4.3.7* and *Bob 4.1.1.2*, which is just as valid as one between *Alice* and *Bob* $>$ *4.1.1.2*, and so on. This provides a very high degree of flexibility with little added complexity for the user – the complexity is hidden inside the configurator, where the translation must be made from the general to all the specific versions.

In the remainder of this thesis, the type of relationship is specified only when needed – most properties of relationships apply regardless of whether *Alice*, *Alice* $\geq$ *1.4.0.0* or *Alice 1.4.3.7* is considered.

Relationships can be specified with one of two basic premises: Either two components are allowed to be combined unless there is a relationship between them stating the opposite; or two components may be combined only if it is explicitly permitted. While neither is fool-proof, the second approach has better odds of guarding against poor combinations. The cost is however that if most components can be freely combined, which is often the case, this leads to a great amount of relationships. Taking versions of components into consideration, the relationship web moves towards a combinatorial explosion, which is distinctly undesirable. The premise that all combinations are allowed also encourages separation of components and leads to a greater number of possible configurations.

There are a number of possible technical relationships between components. The following are the most common, and also the most important. Please note that there can also be connections between components in the form of rules, as specified in section 3.6; examples are *recommends* and *suggests*. The difference between them and the technical relationships here are that the technical relationships are strict – they must be observed – whereas the others are not. Also note that if the underlying principle is that everything is prohibited unless allowed, an additional relationship is necessary, namely *allows*. This thesis assumes that components may be combined unless explicitly forbidden, and therefore does not list *allows*.

Requires If component Alice requires component Bob, then Alice cannot work without Bob. Bob may however work with or without Alice – the relationship is one-way only. Alice should never be used without Bob. *Requires* is commonly found when Bob is a library component of some sort which Alice uses, or when Bob is some other kind of supporting component.

Conflicts If Alice cannot work together with Bob, they are in conflict. This relationship is two-way: If Alice conflicts with Bob, then Bob also conflicts with Alice. Alice and Bob should never be used together. Components most commonly conflict with each other when they perform the same function or use the same resource – operating systems are the prime example, or two applications which must use a single sound system at the same time for the same reason, but in different ways.

Replaces If Alice provides everything that Bob does, then Alice may replace Bob. This is a somewhat complex relationship, since it means that Alice and Bob are on some level compatible, and may indicate that either one could be used – if Charlie requires Bob, and Alice replaces Bob, then Charlie would probably be satisfied with

Alice instead. The inverse is not true, however – the relationship is one-way – so if Charlie requires Alice, and Alice replaces Bob, Charlie would probably *not* be able to settle for Bob.

Replaces is commonly found when a component is deprecated and should no longer be used, but a new component is provided which can be used instead of the old one. This is somewhat similar to the concept of revisions, where a new revision of a component generally replaces and older. In a sense, the replacing component can be thought of as a new revision of the replaced one – except that for some reason, an entirely new component was developed, rather than just creating a new revision.

It should be noted that even though Alice replaces Bob, Alice and Bob may still work together. To specify that only Alice should be used, *replaces* and *conflicts* could be used together. Typically, though, Alice should be used instead of Bob.

Breaks As a special case of *conflicts*, Alice may break Bob. This means that Alice and Bob may be used together, but Bob will not be able to function. This is akin to a one-way version of *conflicts*. Typically, however, Alice should not be used together with Bob either.

Pre-requires If Alice pre-requires Bob, then Bob must be present and working *before* the addition of Alice can start. If all components are combined at the same time, then *pre-requires* and *requires* are equivalent.

Pre-requires is typically used to solve the problem of dynamic changes to a configuration – that is, when a configuration has already been created and later needs to be modified. It is for example useful if Bob is the scripting language in which Alice is written – in that case, running Alice is utterly impossible without Bob. To ensure that Alice can in fact be installed, Bob must first be installed. If an error occurs in installing Bob, then the installation of Alice should not even be attempted.

Some relationships can be found automatically, while others need to be added manually. *Requires* and *pre-requires* are among the easier to find, since they are often explicit, for example in the form of include statements and file extensions, respectively. It is however not trivial, since knowledge of the source language is required, and because there may be many mechanisms in the same language for expressing that Alice needs Bob. Indeed, it is not always possible: A component may require another component for implicit reasons which are never stated in the source code.

Most of the excluding relationships must be found manually, for example through testing. *Conflicts* and *breaks* are prime examples of this. Code reviews may be useful for catching this, or simply trying to test, only to have the tests fail. *Replaces* must typically also be found manually, but this relationship is often a desired effect of redesign, and is thus known even before development on the new component starts. Either way, when such relationships are found it must be easy to update the affected components.

Relationships are a good example of why it is necessary to make information explicitly available. The knowledge about relationships between components is typically found among developers, but needs to be used for example by account managers or other users from the marketing side in order to create configurations.

If the relationships are not specified explicitly, marketing must communicate directly with development when they need to know them, which takes a lot of time and creates unnecessary work for both parties. If on the other hand the relationships are made explicit, development only need to perform the work of documenting them once, and they can then be used by any user of the configurator. It can also be a great help to developers themselves during technical tasks, such as impact analysis, to see directly which other parts of the system are affected by a planned change.

3.2.4 Feature dependencies

The relationships above are direct, in that a component explicitly has a relationship with another specific component. There are also indirect relationships, namely feature dependencies. These are of the type that Alice requires the feature Friend. Bob and Charlie both supply the feature Friend, and Alice will then be able to use either Bob or Charlie to function properly.

Two important characteristics of feature dependencies are that they allow Alice to choose one of several supplying components, and that a direct relationship is transformed into an indirect one. This means that Alice's dependency on Friend needs not be updated if a new supplier is added; all that has to be done is to specify that the new component, say Dave, provides Friend, and Alice will be able to rely on Bob, Charlie or Dave.

Additionally, there is no need for Alice to have relationships with specific versions of the other components – so long as they supply the Friend feature, they are usable. For the same reason, components can be freely updated with new revisions or variants, or deprecated, and no relationships need to be updated. This means that feature dependencies are very useful in limiting complexity.

To complicate the situation, features can be thought of as interfaces[5], and interfaces can change. Two obvious ways of managing this are to use new names (some Friends may become BestFriends), or to use versions on features (Friend 1.0 may become Friend 2.0). To avoid unnecessary clutter, versioning is the better approach.

[5]Actually, a component which implements an interface can be regarded as providing the feature of that interface. In other words, feature dependencies can be used directly when a component requires a specific interface, provided by another component, in order to work.

Typically, only one component should be selected to fulfill a feature dependency, since the components may otherwise interfere with each other. This can vary, however, and it may be possible to select two, more, or all components, depending on the system architecture. A feature may for example be a web browser; several browsers can be installed, allowing the user to choose between them.

3.2.5 Component suites

Sometimes, components are strongly connected and commonly used together as entire component suites. This is a powerful way of simplifying administration: instead of having to select several different components, one single suite can be selected. The *Internet* suite, for example, could be made up of the components *Web browser*, *Youtube player* and *MSN client*.

A clever way of providing component suites (also called *virtual packages*) is to create a new component – the suite itself – which has no code and provides no functionality. Instead, the suite has relationships with its constituent components. Selecting the suite means the configurator will automatically select all required components referred to by the suite.

This is not as straightforward as it might seem. Component suites are in essence configurations of their own, which means that they may themselves exist in different versions, and that they have all the power and challenges from that world. The components in a suite may for example be specified using feature relationships, in which case the actual constituent components to be used must be decided upon, possibly by interacting with the user; or there may be components in the suite which require other components, outside the suite. This brings complexity, which can either be restricted or handled.

Restricting complexity makes for an easier but less powerful system – component suites can for instance be required to use specific relationships rather than feature dependencies[6], which would mean that no further user interaction is necessary once a suite has been selected for inclusion. Handling complexity means, in practice, applying the same logic on a suite as on the configuration itself. Since suites can also be components in other suites, the mechanism to handle them must be general.

A final problem is that of supporting the user. If a set of components are selected which can be replaced by a component suite, the composition system should be able to detect that and ask the user whether to keep the components as separate entities or to replace them with the suite. This allows for a clearer understanding of what actually goes into a configuration.

[6]Instead of saying that *Alice* is part of a suite, a specific relationship would point to *Alice 1.4.4.3* or similar; and instead of saying that a component providing *Friend* is part of a suite, one of the components known to provide *Friend* would be required.

3.2.6 Complex relationships

Relationships do not necessarily have to be between just two components. Alice may for example require *Bob or Charlie*, where either one of the two is enough. This is called disjunction, and is very similar to feature dependencies. Alice may also require *Bob and Charlie*, where both are needed. This is conjunction, which corresponds to component suites. Relationship management systems are normally based primarily on conjunction, meaning that if a component has several relationships, all of them must be fulfilled, not just one.

There is no reason why relationships cannot be even more powerful, stating for example that exactly two components out of three possible must be installed. This is however beyond the scope of this thesis, and the gain appears small compared to the cost of implementing it. It should also be noted that all such requirements can be transformed into the longer but easier form *Alice and Bob, or Alice and Charlie, or Bob and Charlie*, and so on.

Relationship trees arise when components have many levels of relationships, such as in Figure 10, and can be quite complex. Introducing support for arbitrarily large trees is therefore non-trivial, and if it is not commonly used, costs may very well outweigh benefits – most trees can be modeled with feature dependencies and component suites, at the cost of introducing more such than may be necessary or indeed appropriate. Basic support for conjunction and disjunction is necessary, however, for example to be able to define suites and feature dependencies in the first place.

One compromise between the need for flexibility and the cost of complexity is to allow just conjunction for relationships, and then use feature dependencies for disjunction, as in Figure 11. This can be expected to work well because in many cases where disjunction is used, the relationship is actually a form of feature dependency, and there is no need to use component suites in relationships since this can be expressed with ordinary conjunction. This also reflects the nature of both feature dependencies, which refer to the technical back-end, and component suites, which refer more to the customer-centric front-end.

3.2.7 Problems

There are several problems which can arise when dealing with technical relationships, and which must be handled. They are presented here. Similar problems have been discussed in other settings, notably by Syrjänen [13].

Dependencies and anti-dependencies On a basic level, relationships must be fulfilled. If Alice requires Bob, and Alice is selected but Bob is not, then the selection must be expanded to include Bob. Each time a selection is changed, a new sweep

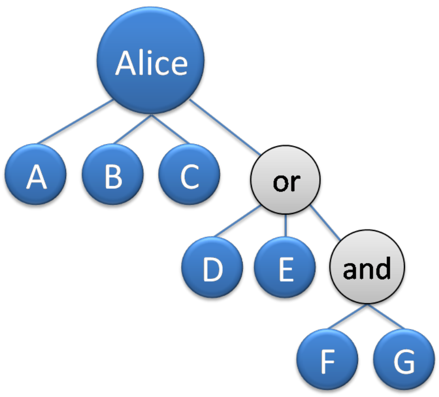


Figure 10: A relationship tree

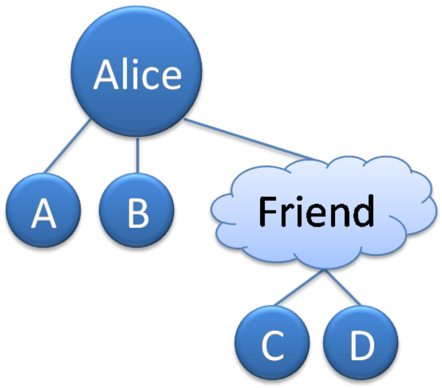


Figure 11: A simplified relationship tree

must be made to see if additional components are required – Bob may for example require Charlie, and so on. This is the problem of dependencies.

Anti-dependencies are the opposite problem: If Alice conflicts with Bob, and both Alice and Bob are selected, then one of them must be removed. This may potentially leave orphan components in the selection: If Bob requires Charlie, and Alice is kept while Bob is removed, then maybe Charlie is no longer necessary and should be removed – or maybe Charlie should be selected, even though Bob is now removed. This is the problem of anti-dependencies.

Cyclic relationships Suppose that Alice requires Bob, Bob requires Charlie, and Charlie requires Alice. Unless care is taken, this will cause an infinite loop when the component relationships are mapped. The loop must also be handled if components are added to an existing configuration, because no two components can be installed at exactly the same time – the loop must be broken up into a linear process with a start and an end, as shown in Figure 12.

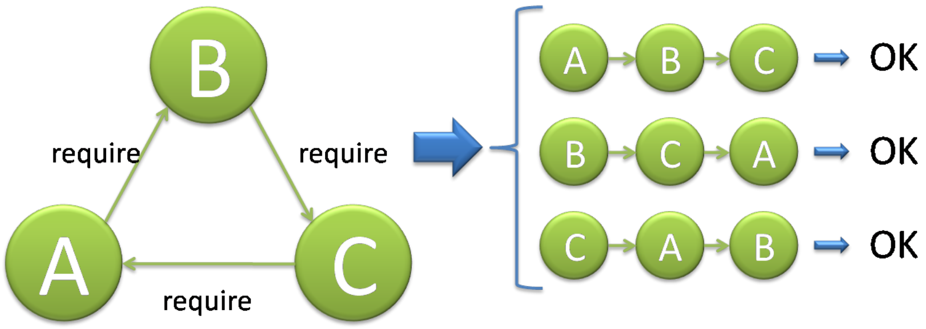


Figure 12: Cyclic relationships

There is also the problem of unselectable components, due to cyclic anti-dependencies. If Alice requires Bob, Bob requires Charlie, and Charlie conflicts with Alice, then Charlie, or Bob and Charlie, can be selected, but Alice cannot: Because of the anti-dependency, Alice is completely unselectable. Figure 13 illustrates this.

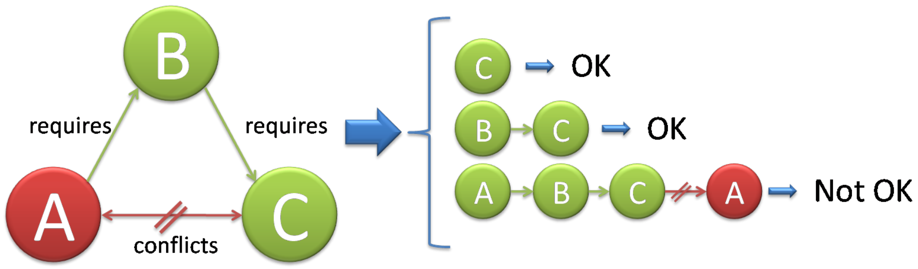


Figure 13: Cyclic antidependencies

Conflicts in practice It should be noted that even though Alice and Bob do not theoretically cause conflicts, they may in practice behave in ways which cause problems. If both Alice and Bob need to create a specific file, say, /users, then whichever component creates it first will have its changes overwritten by the other. While the components may still work together, one of them may not behave as expected, and the problems caused may be very hard to identify.

Missing relationships Related to the conflicts in practice are missing or unknown relationships, which can occur simply as a result of human error or because the relationships are hard to detect. Some of these can be automatically detected, as outlined above, but most can be caught only with the help of testing and debugging. When missing relationships are found, the next problem is what to do with them – should the component be updated with new information, and, if so, what should happen with configurations containing that component, where the new relationship is no longer fulfilled? This is discussed in section 3.4.3.

3.3 Configurations

In component-based systems, configurations are the top-level objects – they are products. A configuration is in essence just a collection of components, and by varying the components, different product variants can be created. However, since components have versions and relationships, collecting them poses challenges. Configurations themselves can also exist in different version – namely when the collection is changed as components are added or removed – and have properties of their own, in addition to properties which come with the components.

Configurations do not need to consist of many components, although they normally do. An empty configuration (containing zero components) is useless, but a configuration with only one component may be useful: This is a way of transforming just that component into a product. The same goes for small configurations, containing only a few components (*partial configurations*), and for configurations which represent entire systems with all the necessary components (*full configurations*).

It is worth noting that component suites are equivalent to configurations. The only difference is that suites are used internally, when configurations are created, whereas configurations are used externally, when configurations have been created and are being distributed as products – there is no technical difference between them, and they could be handled in the same way. Figure 14 illustrates different cases of configuration, where the full system (right) contains another configuration (suite).

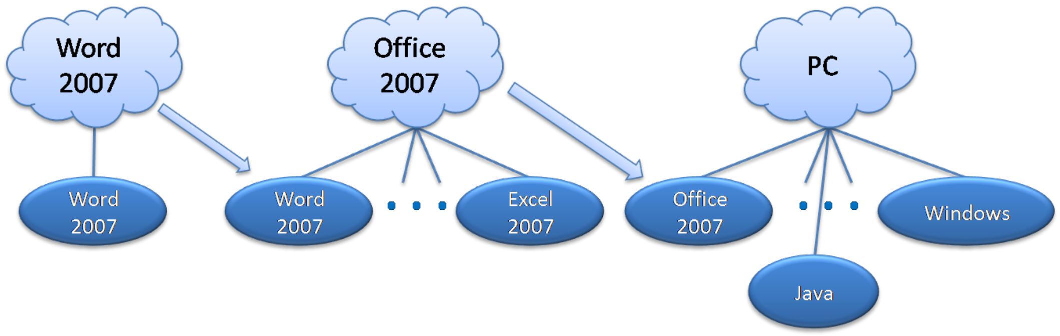


Figure 14: Examples of configurations

3.3.1 Properties

A configuration has two classes of properties: Properties which are specific to the configuration itself (*static properties*), and properties which depend on the the components it contains (*dynamic properties*).

Static properties are set directly and manually by a user, and should generally never be changed – if they need to be changed, a new revision of the configuration should be created. There are however exceptions, which are further discussed in section 3.4.3. Static properties for a configuration could be for example its name, the date it was created and who is responsible for it.

Dynamic properties on the other hand are calculated from the configuration's components and their static properties according to preexisting rules. Typical examples of dynamic properties are *completeness*, *consistency*, *customer*, *market* and *test status*. The first four are analyzed below, and test status is dealt with in section 3.5.

Completeness A configuration is complete if it contains all needed components. There are two reasons a component may be needed, corresponding to two different kinds of completeness: *layer completeness* and *relationship completeness*.

Layer completeness is the requirement that components from certain layers must be present – typically, the operating system is required to be able to use components from any other layer. Required layers differ between configurations: partial configuration may have no required layers at all, whereas full configurations do. Certain layers simply must be present for the system to work. The reason is that if there is no operating system, all the applications of the world can be part of the configuration and it still will not be usable – which is why layer completeness is important.

Relationship completeness is the requirement that all components are present which are required by other components – the interesting relationship is *requires* (and, if it is used, *pre-requires*). If Alice is selected, and Alice requires Bob and Charlie, then a configuration containing only Alice would not be relationship complete; nor would a configuration containing Alice and Bob, or Alice and Charlie. Only if all three are included will relationship completeness be achieved. If a configuration is layer complete but not relationship complete, then most parts of it may work as expected, but some parts will not. It is obviously unacceptable that parts of a system simply do not work – which is why relationship completeness is important.

This type of completeness, *technical completeness*, is based on layers and relationships, is strictly technical, and its applicability may differ between configurations – especially for layer completeness. However, technical completeness does not fully solve the problem of checking if configurations are usable – a configuration consisting of an operating system and a service layer would be technically complete, but hardly usable, since it would contain no applications. In order to ascertain that a configuration is usable, it must not only be technically complete, but also abide by all rules (see section 3.6) which are applicable to it. One such rule could be that applications must be present in full configurations.

Consistency Consistency is similar to completeness in that it is a sort of sanity check for configurations. It is based solely on the relationships between the components in a configuration, and a configuration is said to be consistent if all anti-dependencies of all components are respected – that is, no components in the configuration are in conflict with each other. If a configuration is inconsistent, some components will be unable to function – applications may for example not start – which is why consistency is important.

Conflicts must be respected in order for a configuration to be consistent. Depending on the software system being built, components which have relationships of types *breaks* or *replaces* may or may not be allowed to be part of the same configuration.

This could be specified using rules (see section 3.6).

Just like with completeness, *technical consistency* does not cover all problems. A configuration can be technically consistent and still make no sense from a business or legal perspective. Technical consistency is however in some way the most basic level of consistency: if a configuration is inconsistent from a business perspective, it may still be technically consistent; but a technically inconsistent configuration is inherently broken and can therefore not be consistent from a business perspective.

Customer The customer property of a configuration is directly determined by the properties of its constituent components. Each component can either have no customer set, or exactly one. In a sensible configuration, each component will either be generic, or have the same customer as the others – meaning that the configuration as a whole will either be generic or have one customer. If mistakes are made, however, two components may be selected which have different customers. This will cause the configuration to receive conflicting values – in effect, a configuration will be created which is intended for two customers at the same time. This is hardly sensible, and should be disallowed – with the one exception that a configuration may at the same time contain some components which are generic and some components which are targeted for one specific customer. The generic customer can simply be disregarded and considered as being compatible with an actual customer.

Another aspect is that if a component is chosen which belongs to a certain customer, then it should be possible to check whether other components which are generic could be interchanged for compatible components which belong to the same customer. An example is displayed in Figure 15. In performing this check, however, care must be taken to ensure that versions are compatible and that no other errors are introduced. Ultimately, the choice of which component to use should be left to the user, but automated support is a plus.

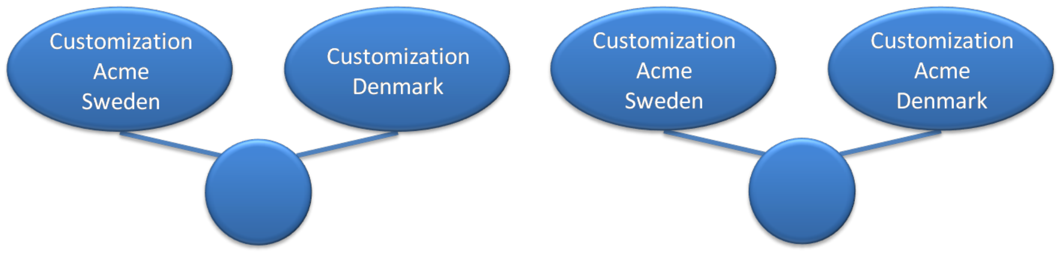

Customization Acme Sweden
Customization Denmark
Customization Acme Sweden
Customization Acme Denmark

Figure 15: Suggesting components

In a sense, these checks can be regarded as a form of consistency: *customer consistency*.

Market While a configuration can be intended for only one customer (or be generic), it can be targeted for several markets. This means that it is not necessarily a problem if different components in the same configuration are targeted for different markets. It may however still be preferable – if all components support both Denmark and Sweden and the entire system is set to Denmark by default, but Alice supports just Sweden, the end-user may be rather confused when settings and languages differ between Alice and all other parts. This requires the same type of automated support as above, to allow for *market consistency*.

A further complication regarding the market is that it is in a sense both a static and dynamic property. The actual value is determined by the components' values and is therefore dynamic, but the desired value needs to be stored as a static property, since not all components may be able to support it. Figure 16 illustrates this: the leftmost illustration depicts the configuration having the market of Sweden, and since all components also have the market property set to Sweden, the configuration's actual market is Sweden.

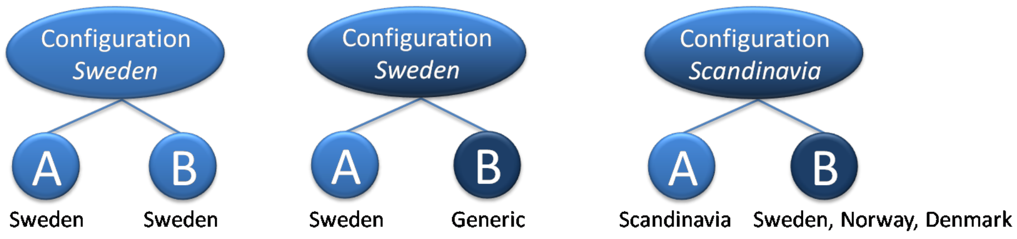


Figure 16: Desired and actual market

The illustration in the middle shows the same situation, except that one component is generic, so the configuration's desired and actual markets are not the same. This may be acceptable, or it may be necessary to ensure that all components can have Sweden as a market. In the third case, however, the situation is more complex. The desired market is Scandinavia, but one component does not support that. It does however support the markets Sweden, Denmark and Norway. Is this acceptable, or is a new component necessary, which does support Scandinavia? Only the user can answer that question.

A similar challenge is that of market suites – Figure 17. The markets Sweden, Denmark and Norway are three distinct markets, and the market Scandinavia is yet another. They may share many settings, but they are not identical, and Scandinavia cannot be created by merging the three others. However, for ease of use, it may be desirable to create a suite of the markets Sweden, Denmark and Norway – the ScandinaviaSuite. That suite is however not equivalent to the Scandinavia market: If ScandinaviaSuite is selected, what will actually happen is that the markets Sweden, Denmark and Norway are selected, and the software system will be able to use any one of them – possibly allowing the end-user to choose between them – but if the actual market Scandinavia is selected, there will only be one market available to the system, albeit one that is appropriate for use in either of the countries Sweden, Denmark and Norway.

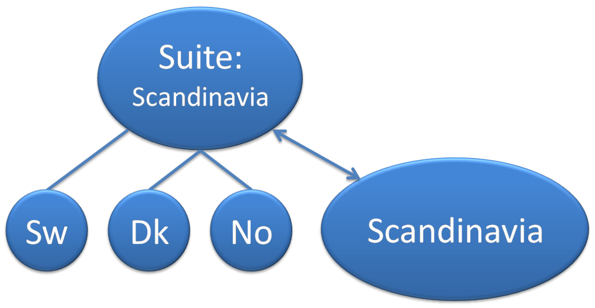


Figure 17: Market suites

This also brings up the same question as for component suites: What if the user selects the markets Sweden, Denmark and Norway? The composition system should be able to identify that this corresponds to an existing suite and ask the user if the suite should be selected instead. Maybe the user even intended to choose the *market* Scandinavia, and the system should be able to catch this, and possibly ask the user to confirm whether the suite or the market should be used. This does require a manually entered connection between ScandinaviaSuite and Scandinavia, but is otherwise uncomplicated.

3.3.2 Problems

Configurations are a powerful way of managing even complex product variants, but they do have their own problems. These are however primarily related to dynamic problems which occur when changes to an already running configuration are necessary. These issues are not the primary focus of this thesis, but they can affect the design of the composition system and should therefore be considered before that design is decided upon.

Identification There are many situations where configurations need to be identified, outlined in Use cases, such as when components must be added or removed, when a configuration needs to be verified, etcetera. Configurations are not static entities, mainly because they have dynamic properties. The selection of components in a configuration is however static, in the sense that it can be changed only by a user's direct actions. This means that although configurations' properties may change, configurations themselves can be identified by the components, and versions thereof, that comprise them. Advantages to this approach include that two formally different configurations which are in practice the same would be recognized as being the same; but also means that all components must be inspected to know if two configurations are the same.

Also, configurations – like components – exist in several revisions; each time a component is added, removed or updated to a newer revision in the configuration it results in a new revision of the configuration. Unless configurations are specifically saved, earlier versions will be lost, which may or may not be a problem. If earlier versions ever need to be retrieved, it certainly is, and configurations should be handled just like components: Stored in some kind of repository and never really changed – instead, new revisions of configurations should be created.

One specific case where identification is important is when a configuration needs to be updated on the device, under control of the end-user. This is a dynamic problem which is greatly affected by the design of the composition system: If the entire list of components and versions is transmitted to the update server it will mean a lot of data, which is costly in terms of time and bandwidth.

If, on the other hand, configurations are handled like components with revisions and unique identification numbers, only that identifier needs to be transmitted. Then, when the configuration on the device has been updated, the new set of components – known by both the device and the update server – can be inspected and the configuration identification can possibly be changed, so as to keep the list of local differences as small as possible.

Optimality In several cases, more than one version of a component may be used, for example if Alice requires Bob version 2.0 or greater. Bob may exist in a hundred different revisions greater than 2.0, but only one of them must be chosen. A common tiebreaker is to pick the latest component, but there are other issues to consider first. Do all versions of Bob have the same relationships, and are they all satisfied? Have all Bobs been tested and accepted, separately and together with all other components in the configuration? Are there more open bugs in any of them than in others? Are they all roughly the same size or are some larger than others, thus costing more in time and transfer fees than others? All else being equal, choosing the latest version is still a good tiebreaker, but "all else" must first be verified as being equal.

In some cases, there may be not only several versions of a component, but more than one component as well, for example if Alice depends on a feature and there are many components available to satisfy the requirement. Not only is the set of candidates greater than for just one component, but components may differ more from each other than versions of the same component do. It may be desirable to allow for expressions to decide which component to select, where several properties can be used in calculating the best suited component, and even generate a ranking between possible components. Such expressions would be able to handle even complex evaluation criteria in a simple way. An example could be $componentSize \cdot 10 + componentAge \cdot 8 + requiredComponents \cdot knownBugs$, where lower values would be better.

Efficiency Apart from the problems above, there is the problem of being able to create a configuration easily and quickly. It may not be wise, for instance, to recalculate the entire relationship graph every time the selection changes, for reasons of speed and response time. Rather, the graph should be calculated only once, for the entire system, and then used. This problem is exacerbated if dynamic updates of configurations must be possible on the devices running them, since those devices may not be as powerful as the servers used by the producing company.

3.4 Repository

The repository is the central location where everything needed for the CBS is stored: components, configurations, relationships between components, etcetera. Table 1 lists typical *configuration items* which need to be stored in the repository.

Table 1: Configuration items

Item name	**Notes**
Component	Exists in different revisions and/or variants, each of which may have different properties.
Configuration	Exists in different revisions and/or variants, each of which may have different properties.
Relationship	Connects a revision of a component or group of revisions of a component, to another revision or group of revisions.
Feature dependency	Each component may supply zero to many features. Relationships are used to indicate that a component depends on a feature.
Component suite	Functions like a partial configuration. Exists in different revisions and/or variants, each of which may have different properties.

The exact format may vary – the repository could for instance either store information regarding components, such as their versions and properties, in one part, and the relationships in another part; or the relationships may be documented together with properties; or both properties and relationships may actually be part of the component; and so on. The components themselves may be stored in the repository, or the repository may only hold a reference to the actual location in an external system – there are many variants of the same basic premise. As long as the capabilities of the repository remain the same, the differences pertain mainly to optimization and other implementation-specific concerns. There may even be more than one repository, but this thesis assumes that only one repository is used, unless specifically noted.

The focus in this thesis is not on the technical aspects of the repository. The important characteristics of the repository are described on a general level, and the implementation details are left for future work – efficiency considerations, cost-benefit compromises, etcetera. It is however important to notice that these issues must be handled before detailed plans for a configurator are laid down. Milligan [11] presents more details on why these aspects are important.

One important thing to note about the repository, which is very relevant for the configurator, is whether components are stored as source code or as binaries. From a composition perspective, it is much easier to manage binaries. Source code must be compiled, which means that every such component *pre-requires* the compiler, and that the compiler version, all flags etcetera must be stored for future reference. This is not necessary for binaries, and the focus can therefore be exclusively on composition.

It should be noted that whether source code or binary components are added to the repository, this is not a repository where active source code development will take place. Rather, all day-to-day development will take place in a dedicated repository, and once a component reaches a certain stage it will be added to the configurator repository. Revisions in the development repository do not have to match those in the configurator repository; and the development repository may have many branches, variants or even entire components which are never added to the configurator repository.

Capabilities Regardless of the form the repository takes, there is one basic capability it must have, namely support for proper versioning of items. There are also a great many capabilities which users may wish that it had, such as allowing for variants and concurrent work[7] [6]. The distinction between capabilities of the repository itself, and of tools working with the repository – such as the configurator – is somewhat blurry, since tools can often simulate many aspects even if the repository does not support them directly.

[7] *Component* development is not carried out on this repository, but creating *configurations* may still require parallel and concurrent work by several users.

Without versioning capability, development on many different components over time cannot be accommodated, and the component-based aspect collapses. This is very similar to just about all existing version control systems such as CVS[8], Subversion[9], ClearCase[10] etcetera. Depending on how complex configurations become and how many persons work on them, there might even be an interest in providing features like branches or tags.

In addition to versioning, the repository could provide support for answering questions such as *Who added component Alice*, *What was changed between revisions X and Y*, *What does the entire history of changes look like for configuration Z*, etcetera. These capabilities are not strictly required, because the same questions can be answered by performing manual work, but would be valuable to improve efficiency.

Concurrency There are three main possibilities for handling the concurrency problems which occur when different persons work with the same components or configurations at the same time[11]: Serialize the work using locks, allow and support parallel work using merges, or ignore the problem and place the checks outside the system. The latter is the easiest: users are left to handle the issue themselves, and the process is defined so that only one user or organizational unit is allowed to make changes to the repository, though many others may still be allowed to read from it. As long as there is only one user at a time making changes to a specific item in the repository, there will be no concurrency problems[12]. There may however be drawbacks in terms of efficiency, since people cannot collaborate on the same item at the same time.

Serialization is the second simplest approach. The checks are then internalized into the repository, so the users do not have to ask each other whether or not it is safe to change an item – whenever a user starts working on an item, that item is locked and no other user may modify it. The efficiency problem is still present, but the manual checks will become automatic so there is no longer any risk that users will make mistakes and accidentally modify the same item at the same time, and users will be spared at least some manual work.

Parallelization is the most advanced approach, and is present in all modern versioning systems. Several users can modify the same item at the same time, and the first one to save changes will quite simply be allowed to save. The second user will

[8]http://www.nongnu.org/cvs/

[9]http://subversion.tigris.org/

[10]http://www.ibm.com/software/awdtools/clearcase

[11]This applies when making changes to configurations, or to the properties of components – not when the actual components are modified, since this is done in the software development repository.

[12]If two or more users do make changes to the same item at the same time, the last one to save the item will overwrite all earlier changes, which will then be permanently lost.

however not be allowed to save changes, and will instead be notified that the item has already been changed by the first user. The second user must then look at all changes and decide what the end result should look like, and only then can the item be saved – and so on for all the other users attempting to save.

Distributed development For large companies, there is also the problem of multiple sites. When development on components and configurations needs to be geographically distributed, it may also become necessary to provide for multiple repositories. This is a problem that even powerful versioning systems have not fully addressed, although completely distributed version control systems such as git[13] may indicate a solution. Other possible solutions include a master-slave-setup, where all changes must be made directly to the master and the slave is used only to speed up reading, or repository synchronization.

3.4.1 Evolution

When a repository is created, it is normally small and it is possible – even easy – to get a complete overview. Over time, however, as new components are added, and revisions and variants complicate the picture, the repository becomes larger and more complex. The overview is lost and achieving a full understanding of even a single component may require considerable effort.

Mechanisms must be in place from the very beginning to help counter this problem and keep the repository usable. A typical example of this is a method to trace all configurations where a given component is used, to quickly see if it is used at all, and whether or not those configurations are still active. Sorting and filtering capabilities are also a bonus, to facilitate administrative tasks. Whether this is technically carried out by the repository itself or the configurator is less interesting.

Clean-up When more and more components and versions thereof are added, together with relationships, large webs of component interdependencies[14] risk being created. A simple solution to this is to regularly remove old components which should no longer be available to ordinary users. Since it is still desirable to be able to recreate old configurations, the components should not actually be deleted from the repository, but instead marked as not being available when new configurations are created[15]. This will keep the amount of components available for daily use limited, thereby helping the user to retain an overview.

[13] `http://git-scm.com/`

[14] Also known as *The dreaded Italian spaghetti-symptom.*

[15] Although some roles, such as configuration managers and support personnel, may need to be able to create new configurations with such components, or update existing configurations.

A more demanding clean-up is also possible, in the form of architectural decisions. It should be possible to identify newly developed or updated components which rely on old components, or components which have relationships with many others, and consider whether their relationship webs can be pruned. While this costs in terms of development effort, the benefit is a smaller and healthier system which can be combined more freely. At some point, entire component webs may even be marked as deprecated, and replaced with newer solutions. Crnkovic and Larsson [5] present some interesting thoughts about this, namely that development on a component is most active in the beginning and end of its useful lifespan, eventually reaching a point where it is better to start on a new design than continue working on the old.

Commit checks Whereas clean-up is performed on the contents already in a repository, commit checks are run before any changes are allowed to be made to the repository. The basic idea is to apply the same checks that are run when a configuration is created, and make sure everything looks good. Some problems can be caught, but not all – if they could, there would be no need for checks when creating configurations.

A typical example of a problem which can be caught before commits is that of cyclic anti-dependencies, as introduced in section 3.2.7 and Figure 13. If components Alice, Bob and Charlie are added to the repository at the same time, applying the relevant checks will reveal that while Bob and Charlie can later be used, Alice in essence conflicts with herself and can never be used. This should probably not be allowed – it makes little sense to create components which cannot be used, and the user may be confused when trying to select such a component. It may however be of interest during planning stages, to use the configurator as a tool in deciding on the system architecture, for instance if other components have dependencies with Alice.

An entirely different case is that of unsatisfied dependencies. Assume that the repository is empty, and that two components should be added: Alice and Bob, where Alice requires Bob. Alice has been completed, but Bob is still in development and cannot yet be added to the repository. Should Alice be allowed in, knowing that it is not currently usable but that it will be in the future? In the end, this can only be answered on a per-case basis, but the problem is obvious: An unwary configurator user may be surprised when trying to add Alice to any configuration, since Alice cannot be selected until Bob has been added to the repository. A possible compromise would be to allow the addition, but not display Alice in the configurator until Bob has also been added to the repository, but then there is little use in adding Alice at all. Ultimately, this is not a technical problem at all, but a management problem caused by the development planning process.

3.4.2 Adding items

Components need to be added to the repository in order to be available for inclusion in configurations. The point at which they are added can however be discussed – should they be added as empty placeholders as soon as a decision has been made to create them; or when the first versions, which may lack several features, are ready; or should they be kept out of the repository until they are completely finished?

All alternatives have advantages and drawbacks. The later a component is added, the more users can depend on it being finished, but this also means that it may take a long time before a component shows up and can be used, lengthening turnaround time. Adding components early means times will be shorter, but it will probably be necessary to create more versions as bugs are found and corrected, and new features become available. If an early version of a component is part of a configuration, the configuration will also need to be updated to instead use a newer component version.

There are also several possible compromises between alternatives. The most natural is to add components as soon as possible, but to restrict usage to those users who actually have a reason to use the components – and to warn them when a component is not safe to use. Before coding has started, for example, maybe only developers and configuration managers should be able to select the component, and everybody else should see nothing at all of it, or should perhaps see that it exists but that it is unfinished and therefore can not be added to configurations. Once early versions with some features have been finished, more users could then be able to use the component, culminating in a final released component which has been fully tested and approved, and which can be added to a configuration by any user.

A similar model is that of separated stages, where the component is added to one repository when it is planned, then moved to another repository when code is added, moved to yet another when it is stable enough for testing, and moved to the final repository when it has been tested and approved – akin to the idea of staged integration lines, described by Appleton et al [1]. This requires more repositories, to which different users may or may not have access, but otherwise works like restricted usage. The main advantage is that it is very clear what stage a component is at, but the drawback is that it is harder to follow the development of a single component since it is spread over different repositories.

3.4.3 Modifying metadata

Components and configurations are stored in the repository together with their properties and relationships. If configuration management rules are followed strictly, a specific version of a component must never be changed once it has been added to the repository – a new version must be created instead. While this is a sensible strategy for the component itself – its source code and behavior should never be allowed to change unless a new version is created – it is not necessarily the best approach for the properties and relationships, collectively known as the component's *metadata*[16].

Consider a configuration which contains the components Alice and Bob. Neither Alice nor Bob has any relationships, and both of them have been tested for bugs and approved – everything is just dandy, and the configuration is saved. However, an attentive developer realizes that Alice actually requires Charlie too, and that the configuration will not work. Even if a new version of Alice is added to the repository with the "new" relationship, the configuration will still indicate that everything is fine, since it contains the old version of Alice.

If, on the other hand, the existing version of Alice is corrected and the relationship is added, the configuration will correctly indicate that Charlie is needed, but the system will have lost the possibility to exactly recreate old configurations[17] – if the configuration is now loaded, it will indicate a problem and Charlie must be added before anything can be done. The incorrectly "possible" combination of only Alice and Bob is gone.

A similar problem is that of Bob's markets, and while the problem with Alice was one of missing relationships, this is a problem with properties. Bob's *market* property has been incorrectly set to Norway, but in reality, Bob was designed for Denmark – a simple human error has been made somewhere along the road. The question is the same: Should a new version of Bob be added to the repository, or should the existing one be changed?

If a new version is added, which has the correct value *Norway* for the market property, there is a risk that users will select the older version of Bob when creating new configurations, instead of the newer version, simply because one of them appears to work with Denmark and the other does not. Alternatively, the old version could be replaced by a new one with the correct market, and the old version marked as being deprecated.

There is also the problem of when to add items to the repository: If no metadata must ever be changed, then it is impossible to add untested items to the repository, test them afterwards, and then set the test status property (see section 3.5). Instead, items can be added to the repository only when they have been tested and accepted. This may be a benefit for many users, but it means that the test department has little use for the repository, since all items there have been tested and accepted. While this can be solved by having several different repositories (one for untested items, one for accepted and so on) it increases complexity in the system.

[16]A component's metadata is actually any kind of information regarding the component, but properties and relationships are the most important kinds.

[17]To be precise, the exact *configuration* can be restored, but its exact *metadata* cannot.

Over time, when components become old and are eventually replaced, it may also be necessary to remove them from active use, as discussed above. How should this be performed? It is very counter-intuitive to add new versions of components and mark them as deprecated, while allowing the older versions to reside in the repository without being marked as deprecated.

The very best solution is to manage versions not only of components, but also versions of their metadata[18]. This means that it is always possible to recreate all previous configurations, components, properties and relationships – but at the cost of a large system and much complexity.

A compromise may be to allow only certain metadata, notably relationships and statuses, to vary inside the repository without creating a new revision, while others are frozen. Such metadata could be the test status and a few similar status properties, and there would probably have to be restrictions on what kind of user was allowed to change properties; in which circumstances it would be allowed; and how notifications should be sent once updates are made.

3.5 Status modes

Both components and configurations can have several different statuses modes, and these status modes (statuses) can have different values over the course of the item's life cycle. Both the status modes themselves, and the values available for each status mode, must be decided upon by the producing company – large, complex products may need many modes and many possible values for each mode, while relatively small products may need only a few. The status modes can be thought of as different status dimensions, which may depend on each other in non-trivial ways.

One status which should be common to most development situations is *test status*. This will be used to explain the basic workings of status modes; first for components and then for configurations. The basic premise is the same for other status modes, some of which are also presented below.

3.5.1 Component test status

When development on a component has resulted in a new version, that version generally needs to be tested to ensure high quality; even if tests are also part of the development process. While this can be performed in a separate system, there are advantages to doing so inside the world of the configurator; the most obvious being that those users who need access to new versions quickly can start using them immediately and that work on configurations and work on testing can run in parallel, while untested versions can still be kept hidden from users who should not use them.

[18]Akin to sixth normal form as defined by Date et al [7].

The test status of a single component is rather simple: A component can have only one of a limited number of values for its test status. Those values could typically be *Untestable*, *Untested*, *Tested – rejected* (or just *rejected*) and *Tested – accepted* (or just *accepted*). Different situations will probably call for different levels of detail. A new component starts with a test status of untestable or untested, depending on when it is added to the repository – if it is added without any code, or a skeleton code which performs no work, it may start as untestable; if it is added when it is nearer to completion, it will more likely start as untested. It will then be tested and subsequently have its test status updated to rejected or accepted; or it may never be tested, for some reason. New versions of a component should always start as untested, and then be updated to rejected or accepted – or, if something has gone terribly wrong, maybe even untestable.

There is a simple order to these values. *Untestable* indicates a broken or useless component, and is the lowest possible value a component can have. *Rejected* indicates that the component works well enough to be tested, but that it does not perform as expected. *Untested* is in some aspect better than rejected, since there is still the possibility that the component will work, even though it provides less information than *rejected* and even though the component may move from *untested* to *rejected* or even *untestable*. The only desirable test status is of course *accepted*. The order between values may differ for different companies but the concept of ordering should still be available, and when more (or fewer) values are used, the same kind of ordering should be available for them too. This is relatively simple for components, but configurations introduce entirely new problems.

3.5.2 Configuration test status

While a component has just one single test status at a particular point in time, a configuration consists of many different components which potentially have different values for test status. The fundamental question for the test status of a configuration is how to make sense of all these underlying values and be able to tell whether a configuration will work or not.

A case where this is very simple is presented in Figure 18, leftmost. Even if two of the components have been tested and accepted, the third has been rejected, and the configuration as a whole cannot work if one of its components is broken – it essentially receives a test status of rejected. The third component should be updated to a new version which can be tested and accepted, and the configuration updated to use that component instead.

This is an example of the principle of the least common denominator – if the first two components would instead be *untested*, the configuration as a whole would still be *rejected*. The principle of the least common denominator is at the core of

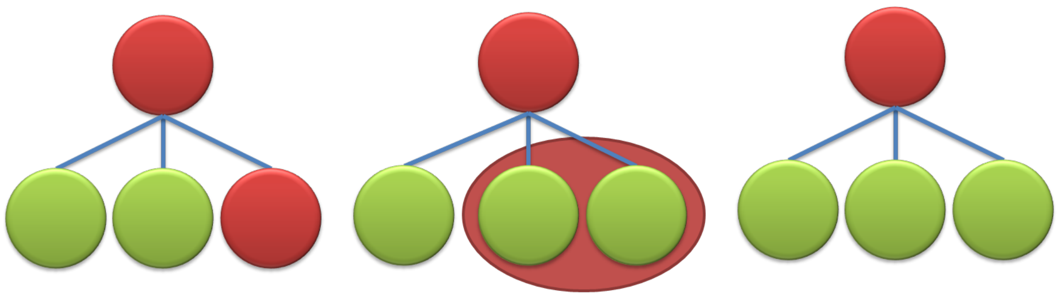


Figure 18: Configuration status

calculating the status of a configuration from its components: The configuration can never get a better rating than that of the lowest of its constituent components, since it makes little sense to say that a product works even if some of its parts are broken. It can however receive a *lower* rating than that of its components, since even parts which work independently may break when they have to work together.

The middle illustration in Figure 18 exemplifies this. Each component has been tested and approved in isolation, but components may affect each other and therefore need to be tested together, too. In this case, although all components work individually, two of them disrupt each other and do not work together – so the configuration, again, receives an effective test status of rejected. The only way to make absolutely certain that components work together is to verify the entire configuration as a whole, as depicted in the rightmost illustration. In so doing, not only should each component be tested individually, but also in combination with all other components, and the results from this system test should be stored so that other configurations can make use of that information – otherwise, the same work may need to be performed several times, which is of course unnecessary.

In the end, the only way to be *entirely* sure that a product is of desirable quality is to test the entire configuration. However, this takes time and ties up resources. In some cases, an entire system test is not required to make an educated guess about a configuration's quality. During development of a new product, this kind of analysis can be used to direct development and testing towards the areas where the probability of faults is highest, and a full system test can then be performed when the product is nearing completion.

A typical example could be when an existing configuration is used as a basis for a new one, and the new configuration uses newer versions of components than the old one did. If the old configuration had had all its components accepted as working together, and the new component versions have been accepted individually, chances are that there will be little problems integrating them into the configuration, and testing resources can be directed towards other areas where they are better needed.

This kind of reasoning is especially applicable when components have no relationships between them, which should generally be the default case in a well-designed CBS. If a configuration consists *entirely* of components which have no relationships at all and where each component has been individually tested and accepted, it should be possible to say that the configuration as a whole will work and therefore should receive a test status of accepted. To be on the safe side, however, it may be more intelligent to assign a test status such as *tentatively accepted* to the configuration, and then make entirely sure by actually testing it.

Configurations may also have the opposite situation: They may be *accepted* even if some components are *rejected* individually. While this may not be desirable (or even allowed) for business reasons, it is entirely possible to handle technically by separating the test status into one dynamic part and one static part. The dynamic part would then be based on the analysis described above, and depend on the test statuses and relationships of components; whereas the static test status of a configuration would quite simply be set by a user. If the two differ, the static test status could be given priority, with the interpretation that "The user knows there are problems, but the configuration as a whole is good enough" – or the lowest value of the dynamic and static properties could be chosen, with the interpretation that "Everything must be tested and accepted before the product is delivered to customer".

Since component suites are in essence configurations, they can be handled very much the same way when they are selected for inclusion into configurations. Feature dependencies are another matter, however, since a feature dependency is satisfied by selecting just one component, whereas component suites require that all its components are added. Because of this, the logic is inverted for feature dependencies: It is enough that one of its components has been approved, for the entire feature to be considered approved. The problem of testing components together with each other still applies, but if just one of the components supplying the feature has been tested with those other components, that is enough.

Finding the test status of a component is simple. Calculating the test status of a feature dependency, component suite or configuration is considerably harder, but the payoff is shorter turnabout times and the possibility to direct testing efforts to where they are most needed. Complicating the picture even more is the fact that there are also other status modes.

3.5.3 Other statuses

Test status is probably the most fundamental status mode, and is intuitively understandable to most persons who work with software development. There are however several other statuses too, which can be of great help in working with a component-based system. Four of them are exemplified here to give an idea of what statuses may look like and be used for, and the problems which come with having several different status dimensions. More status dimensions can be required, depending on the producing company.

Deprecated Primarily applicable to components, feature dependencies and component suites, but sometimes also to configurations, the status *deprecated* typically takes one of only two values: yes or no. When an item has been deprecated it should no longer be available for use in new configurations, and not even visible for most users. Marking an item as deprecated indicates that it has reached the end of its lifespan and should not be used. Sometimes, an item marked as deprecated is also on the receiving end of a *replaces* relationship, so that old configurations containing the item can be updated to use its successor, if any.

In addition which component to use instead, it may also be wise to note when and why a component was deprecated, and who did it. The owners of configurations using the item should also be notified of the change, as well as owners of components with dependencies on the item, so that appropriate actions can be taken.

Update needed There are many reasons why an item may need to be updated. The latest revision of it may for example have been tested and rejected, necessitating a new revision, or it may depend on another component which was recently updated or it may simply be decided that a new feature is needed. To signal this, the status *Update needed* may be set to yes. Again, as for *deprecated*, it is prudent to also record when and why it was done, and by whom. Once a new version of the item has been added to the repository, *update needed* may be safely restored to no on the original item, since the version system indicates that the later item is available.

The main reason for including the status update needed is that it enables automating a lot of work when creating configurations. If a revision of a component is selected for which *update needed* is set to yes, the configurator can immediately warn the user of this – or may not even allow selecting that revision to begin with. If such a status is not available, the user must manually check to see if there are circumstances which require a new revision, for each component added to the configuration; or the user adding new revisions of a component must notify all other users whose configurations include that component. The same applies when other components have relationships with a component which needs to be updated.

Production stage Primarily applicable to configurations, *production stage* indicates how far in the process of creating a product for a customer the configuration has come. Possible values include *planning*, *design*, *development*, *pre-fabrication*, *fabrication*, *deployment*, *delivered*, etcetera – again, depending on the producing company and the specific situation. The usefulness of this status is that some actions may look different depending on what production stage a configuration has reached.

Notifications are an example of this. Typically, if a configuration is still in the planning stages and a new revision of a selected component becomes available, the owner of the configuration may want the configuration to be silently and automatically updated so that the new component revision is used instead of the old one. For a configuration in pre-fabrication, it may be desired that new components cause notifications to the configuration owner, who can then manually decide on the best action to take; while nothing at all should be done if a configuration has already been delivered – not even delivering a notification. Notifications also allow for a smart way to propagate information such as *update needed*, above, to owners of components or configurations.

Support level Like production stage, the *support level* applies primarily to configurations. It denotes the level of support the producing company is willing to extend to a certain configuration, and may take values such as *unsupported*, *supported*, *guaranteed* etcetera. An unsupported configuration may be one which should work, but for which there will be little or no support offered if it does not, perhaps created on the direct request of the customer; a supported configuration could be one which has been created after discussions between the producing company and a customer and for which there will be general support; and a guaranteed configuration could carry promises such as "on-site support within 24 hours" or similar.

Unlike production stage, support level is of great interest for persons outside the producing company – customers may even be more interested in having support than in having many features. It is also of interest internally, perhaps especially when bugs are reported because bugs afflicting components which are part of guaranteed configurations should be given higher priority than bugs which only affect unsupported configurations.

3.5.4 Combining statuses

Many times, it is less interesting to consider the many different status values that a component may have, and more interesting to reduce the many status dimensions and their values to one single combined status. This applies even more to configurations, which can have even more statuses since they are made up of components, and those components and their relationships may put the configurations in any number of situations. In essence, this reduction corresponds to combining all relevant status values in order to answer the question "Is this good enough for my purposes?".

Different users in different positions may have different interests. If a certain configuration is in active use by an important customer and that customer has filed a bug report, it matters little whether components in that configuration have been marked as needing update, or even being deprecated – what is important is to work with the corresponding versions and fix the bug. On the other hand, when new configurations should be created it is important that all components are of the latest versions and work well with each other.

There are basically two ways of combining different statuses to one single value: Either set up a hierarchy of statuses, or create a more general rule where boolean expressions, weighting and similar operations are possible. The first is simpler but much more limited than the second, and not all hierarchical combinations may make sense. Also, the list containing the hierarchical mapping between values for all statuses and the combined value will grow exponentially in the number of statuses and their values, which will quickly make it hard to maintain. Some kind of rule, allowing for expressions to calculate the combined status, would be preferable.

Figure 19 and Table 2 illustrate the two approaches for a small example. Note that even though just two statuses are used, the table becomes large for both combined statuses, whereas the expression remains compact.

Overall status provides an example of how statuses can be combined to provide information about a configuration at a glance, whereas *Test action* illustrates a combined status specifically designed to supply information about whether testing needs to be performed – a veritable plethora of other combined statuses can be imagined. Whether expressions or tables are used, automation will help the user greatly – in one case to evaluate the expressions, and in the other to find the correct value in a very large table.

```
if(testStatus == Untested && productionStage < Deployment &&
                             productionStage > Planning) {
    testAction = Test;
} else {
    testAction = None;
}
```

Figure 19: Calculating combined status using expressions

Table 2: Reading combined status from hierarchical mappings

Test status	**Production stage**	**Overall status**	**Test action**
Accepted	Delivered	Delivered	None
Accepted	Deployment	Deployment	None
Accepted	Fabrication	Fabrication	None
Accepted	Pre-fabrication	Pre-fabrication	None
Accepted	Development	Development	None
Accepted	Design	*Impossible*	None
Accepted	Planning	*Impossible*	None
Untested	Delivered	Warning	None
Untested	Deployment	Warning	None
Untested	Fabrication	Warning	Test
Untested	Pre-fabrication	Warning	Test
Untested	Development	Untested	Test
Untested	Design	Untested	Test
Untested	Planning	*Impossible*	None
Rejected	Delivered	Error	None
Rejected	Deployment	Error	None
Rejected	Fabrication	Error	None
Rejected	Pre-fabrication	Warning	None
Rejected	Development	Rejected	None
Rejected	Design	Rejected	None
Rejected	Planning	*Impossible*	None
Untestable	Delivered	Error	None
Untestable	Deployment	Error	None
Untestable	Fabrication	Error	None
Untestable	Pre-fabrication	Warning	None
Untestable	Development	Broken	None
Untestable	Design	Design	None
Untestable	Planning	Planning	None

3.6 Rules

Rules are a generalization of relationships. Relationships exist between two components; rules can apply to many items at the same time. Relationships can only exist between components[19]; rules can apply to components, component properties, configurations, relationships, other rules, and more.

Relationships are explicitly technical in nature and must be observed if the configuration is to work; rules are non-technical and configurations may work even if rules are violated. However, the repercussions of shipping a product which violates rules may be even worse than delivering one which violates relationships. Finally, relationships and rules have different origins and are added or removed by different roles, for different purposes. Relationships are added or removed when components are modified, whereas rules can be changed independently of items in the repository.

Rules have the potential of removing a lot of complexity from the user, but that complexity still exists and must be managed somewhere else. One aspect of this is that somebody has to add, remove and generally maintain the rules, which is not necessarily effortless. Another is that there must actually be a system in place to process the rules. That system is however beyond the scope of this thesis – here, rules are simply assumed to work in some magical way. The paragraphs below outline what rules should look like and what they can be used for.

Types of rules There are primarily two types of rules: rules from within the producing company, and rules from outside it. *Business rules* can come both from within and outside, but are used here to exemplify rules from within the company. *Legal rules* typically come from outside and exemplify that type of rules. Different companies may need others.

Examples of business rules could be *All configurations produced during 2010 must include component Alice*, to promote that component, or *Components which have the customer property set to Acme and the market property set to Europe are recommended to support the Chinese language*, because Acme want general support for Chinese in Europe. The first example is an absolute requirement, whereas the second is a suggestion to the user. There can also be negative rules: *Components with the market property set to Sweden must not be included in configurations with the market property set to Denmark.*

Legal rules are more definitive. They are of the type *The Danish language must never be present in configurations with the market property set to Sweden*, because of language laws, or the opposite: *Configurations with the market property set to Germany must include the German, French and Turkish languages*, again because of language laws.

[19]Component suites and feature dependencies are ultimately converted to components.

Completeness and consistency Introducing business rules and legal rules also introduces two new types of completeness and consistency (see section 3.3.1), namely *business completeness* and *business consistency*, and *legal completeness* and *legal consistency*. A configuration is *business complete* if it contains all components indicated by business rules, and *legally complete* if it contains all components indicated by legal rules. In the same way, configurations are *business consistent* if all negative business rules are adhered to (no components or properties in a configuration violate business rules), and *legally consistent* if all negative legal rules are adhered to. This only applies when rules are definitive, not for suggestions and similar rules.

This means a configuration can, at the same time, be technically consistent, business inconsistent and legally consistent; and the same applies to completeness. In order for a configuration to be *complete*, it must be complete with respect to all three kinds; if it is incomplete with respect to one of them, it is said to be *incomplete*. The same applies to consistency.

Adding rules Just as commit checks are meant to catch components which can never be used for technical reasons (section 3.4.1), components which violate rules can be caught before they are allowed into the repository. The reverse must however also be considered, namely that when new rules are introduced, components which already exist in the repository may no longer be usable – for instance, a component with market Germany and support for just the German language would no longer be usable if the language rules above were introduced, since it would then need to support the French and Turkish languages too.

This kind of checks need to be made whenever a new rule is added. One option is that the rule should not be allowed to be added at all, which would of course be negative since the rule can be assumed to fill an important purpose. An alternative is components in the repository which are no longer selectable should be marked with *update needed*, *deprecated* or some other similar status. This has the drawbacks that those components can no longer be used – which might be exactly the intended effect.

A third alternative is that the rule could simply be added with no change to existing components and configuration, which would make the rule take effect only on items which are added from that point forward. This should perhaps even be possible for the user to select when rules are added. Finally, since rules can be active during only certain periods – in other words, relate to dates as well as to components or other items – some problems can be avoided using this mechanism.

The basic premise for managing the complexity of CBSs is to hide it from the user and make the configurator perform as much work as possible automatically. In order to succeed with this, some information must be made explicit – for example relationships, properties, and rules. Introducing rules means that the configurator becomes more capable, but also more complex. In essence, flexibility and ease-of-use for the user comes at the cost of complexity inside the configurator, where it must be managed by whoever is responsible for developing and maintaining it.

Implicit rules A subtle point about rules is that even if explicit rules, as outlined above, are not introduced, there will still be *implicit rules*. These are the fundamental assumptions and restrictions which the configurator place on the composition of configurations, such as *Relationships must be observed for a configuration to be complete and consistent* or *There must be no more than one operating system*. If introducing explicit rules is too much work and not worth the effort, then the most important of those rules could instead be hard-coded into the configurator in the form of implicit rules.

Configurator behavior Rules can also be used to define the very behavior of the configurator itself. This is on another level than the layers specifying permitted compositions, and could for example specify that *Test personnel are not allowed to create new configurations*, or *Configuration managers are allowed to break rules X, Y and Z* (for instance, create inconsistent configurations to see the repercussions). This kind of rules could be specified in the same way as ordinary rules and checked using the same system, but would need to be applicable not only to components and other items inside the system, but also to users and capabilities of the configurator.

In a sense, this is reminiscent of allowing for script languages inside a program. Several applications have done so already, Microsoft Office being an example, with exceptional results. The important gain is that users are given increased flexibility within the confined and therefore safe limits of an existing system, while the complexity handled by configurator rules is hidden during everyday use.

Configurator rules could also be used to handle updates, discussed in section 3.5.3. A typical example of this could be when a new component version is made available and the user is working on a configuration which uses an older version of the component. There might be a rule which, depending on several different variables, automatically updates the configuration, informs the user, or does nothing. This would remove the need for users to manually check whether items have been updated, allowing them to focus on decisions which actually must be made manually.

Priorities Rules could also be given priorities, to facilitate understanding for the user. Rules with the highest priority are run first, and could for example be the implicit rules which absolutely *must* be adhered to. The rest of the rules can then be run in descending order, passing definitive business and legal rules, and only when all those rules have passed will the suggesting rules be run. This would mean that the user would be shown only one problem at a time, and the most important problems first. There is little point in adding suggested components if a configurations is technically inconsistent, for example. Still, the user could be given the choice on whether to display one problem at a time, or all at once. Priorities could also be used as a simple ways of providing different users with different possibilities. Some users may be allowed to break rules below a certain priority, whereas other users must adhere to all rules.

4 Design

This section builds on the results in Analysis to present a comprehensive outline of what a configurator should be able to provide in terms of features and user support. A possible program sketch is also provided, to provide a concrete example of what it could be like.

At the very beginning of the analysis, an overview of the configuration process was given. This overview can now be made a bit more detailed, and is presented in Figure 20. The concepts of components and their properties, statuses and relationships, as well as business and legal rules, have been explained. The repository has been introduced, and the problems and possibilities of configurations are known. All of this can now be combined into a complete overview, to provide a design proposal for a configurator.

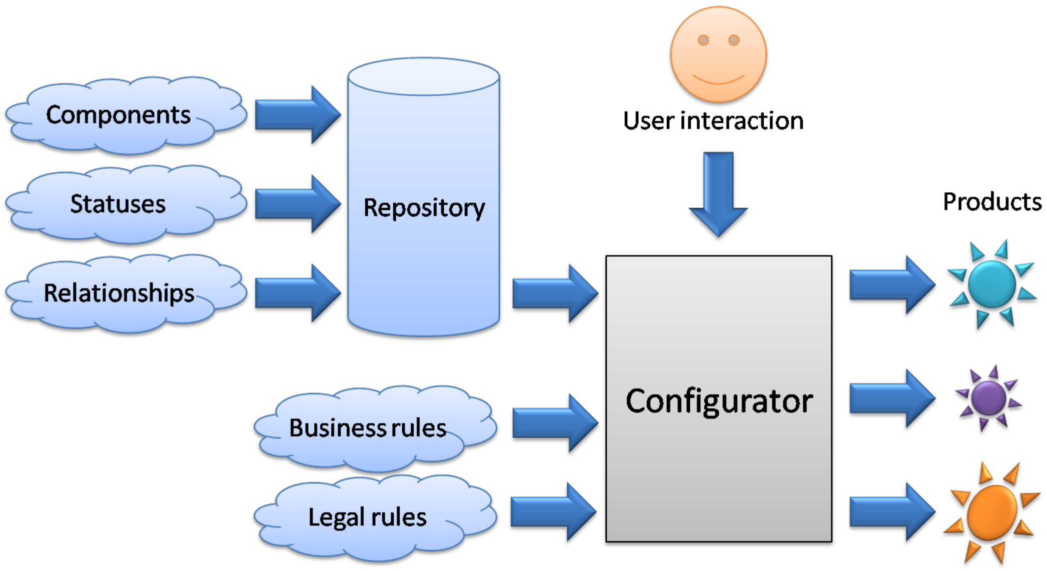


Figure 20: The composition process, revisited

It is obvious that any configurator must support version management of components and configurations, and their properties, to be able to handle the repository and the difficulties inherent in a system where both components and configurations may exist in revisions and variants. In fact, a configurator can be seen as a form of version management system. Dart [6] elaborates on issues which any version management system should be able to handle. It must also support at least the most basic use cases (see section 3.1).

There are also a great many details to take under consideration, such as whether customers should be considered to be on a higher level than markets, or the other way around (section 3.2.1). Considering the options, advantages and drawbacks as presented in the analysis, the following proposals seem to be the most natural and efficient decisions for the important aspects of the *first iteration* of the configurator.

Customers are considered to be on a higher level than markets – that is, in order to select a customization for a market, the customization for a customer must already have been selected. If a setting is defined both on the customer level and the market level, the customer takes precedence, to retain consistent settings per customer.

Layers are used, as defined above: operating systems, service layers, applications and customizations. When a relationship is created between components, it must either stay within one layer of the configuration, or must be from one layer to the layer immediately below, in order to promote separation of components on different layers. Customizations are the exception: they are allowed for operating systems and service layers as well as for applications. Complex relationships are not sufficiently cost-efficient and therefore disallowed. All relationships for a component must be fulfilled (conjunction), and feature dependencies are used to implement disjunction.

The configurator must be able to handle general relationships, in the form of *Alice* or *Alice* $\geq$ *1.4.0.0*, in addition to specific ones such as *Alice 1.4.3.7*. The basic assumption of the relationship system is that all components may be freely combined, unless explicitly forbidden. Feature dependencies are fulfilled by including exactly one component providing the feature, and component suites are fulfilled by including all components referenced by it. Component suites are also somewhat restricted in functionality – they should not be treated as general configurations. Rather, they must refer to specific versions of components, and if a new component version is to be used, the suite must itself be updated to a new revision. Suites must not use feature dependencies to indicate constituent components; only specific relationships.

Each time the user selects or deselects a component, the configurator should update the display of all other components, to indicate in real-time which components are now available for selection. No user, except for administrators, should be allowed to select components which conflict with each other – this would be useful for some users, but may come at a later stage.

If a selected component requires additional components, those components should be indicated to the user by the configurator, but not automatically selected. If they are selected and the first component is later deselected, the user must keep track of this manually. The exception to this is component suites, where all constituent components are automatically selected when the suite is selected, and deselected when the suite is deselected. Components which cannot be selected are not hidden from view, but marked as unavailable.

Configurations are allowed to be saved even if they are empty, in order to refer to them inside the configurator. Incomplete or inconsistent configurations can also be saved, but are not allowed to receive a *production stage* status greater than *development*. This applies to technical completeness and technical consistency, as well as to business and legal completeness and consistency when such rules exist.

Technical inconsistency is considered to occur if two components in the same configuration conflict with each other, or if one component breaks another. It is however allowed to have two components in the same configuration where one replaces the other. All components in a configuration must have the exact same values for market and customer, with the exception that an empty value can be combined with anything, otherwise the configuration is considered technically inconsistent.

The configurator must perform commit checks before components are allowed to be added to the repository. Components are only allowed to be added if they can later be selected for use in a configuration – components which suffer from cyclic anti-dependencies are not allowed, for example. A repository administrator should also be appointed, charged with the task of keeping the repository clean and warning when components risk becoming too entangled for the component-based system to work as intended. No items in the repository may be changed in any way unless a new version is created – with one very important exception: Components and configurations may be given new values for status properties such as test status, update needed etcetera, without necessitating new versions.

The configurator should display notifications whenever changes to one item affect other items – typically, when a new version of a component becomes available, all configurations which contain older versions of it should be notified. The exact form of notification may vary – a message inside the configurator, or an email or similar sent outside of it.

4.1 Application sketch

While certainly not a complete proposal, the sketches below give an indication of what a configurator could look like. Many features are not detailed in the figures, but are no less important because of that. There are several different ways of working with a configurator, and three of them are shown here, together with a sketch of how administration could be performed. Figure 21 illustrates the first approach.

This approach supports any of the three basic ways of creating configurations; top-down, bottom-up or mixed-mode. Each time a component is selected all conflicting components are marked as unavailable and cannot be selected; and any components which must be selected are also indicated. This provides direct feedback and helps the user to understand why a component cannot be chosen, allowing for easily making the choice between two or more components which conflict with each other. Ultimately,

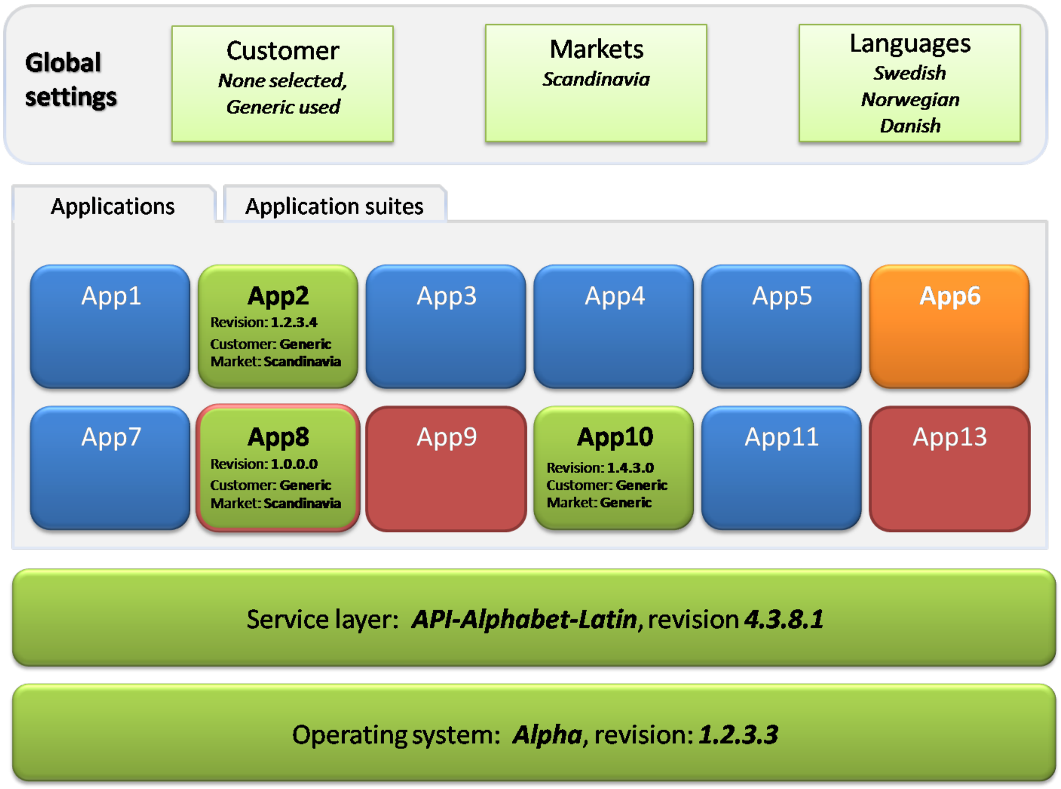


Figure 21: General design proposal

this is a means to ensure that configurations are complete and consistent.

It is also possible, at any time, to choose global settings. In the example, the market has been set to Scandinavia. The configurator should then mark components as unavailable if they cannot supply that market, and for the components which do support it, it should be automatically selected. Application 9 does not support Scandinavia, and has therefore been marked unavailable. Applications 2 and 8 do support Scandinavia, and the market property has been accordingly set. Application 10 does in fact *not* support Scandinavia, but the user has enough privileges to include components without regard to this rule, and has done so – the market has been set to Generic.

The test status, or perhaps combined status, as chosen by the user, should also be displayed so that it is directly visible to the user. The exact method for this is less important than the fact that it is actually displayed. Here, a colored line has been added to indicate when components are untested, rejected or untestable, as in Figure 22.

In addition to components, suites can also be added to the configurator, as indi-

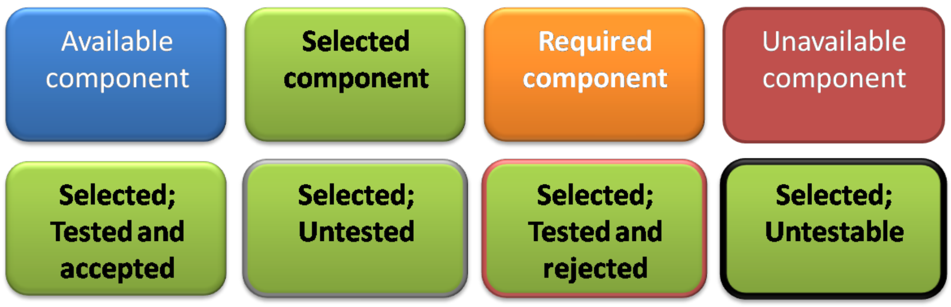


Figure 22: Component availability legend

cated in Figure 21 and illustrated in Figure 23. The rules for calculating suite status apply, so if a selected component in a suite is rejected, for example, the entire suite should be marked as rejected.

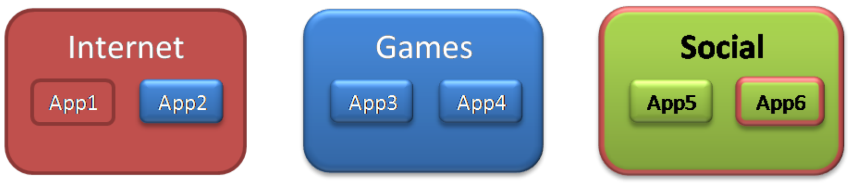


Figure 23: Handling suites

The method for calculating how component availability should be marked is based directly on the relationships as described in section 3.2.3, and is summarily illustrated in Figure 24. The leftmost situation illustrates how the five components are connected. In the middle, component B has been chosen, which causes C to be required and E to be unavailable. Finally, components A, B and C have been chosen, and D and E are left unavailable.

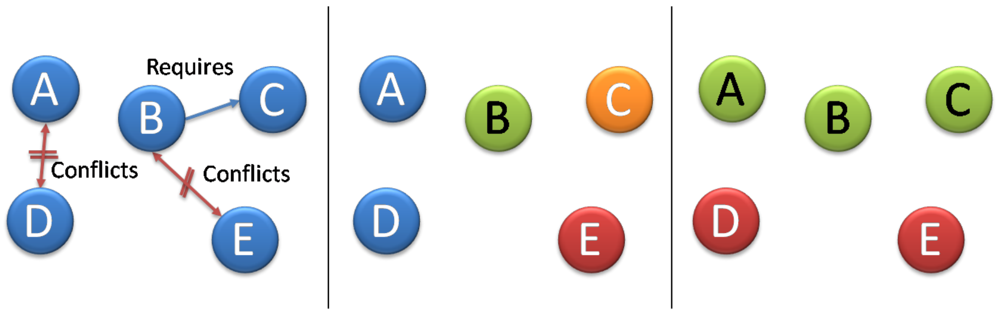


Figure 24: Component selection

Another approach is suggested in figures 25 and 26, namely that of a top-down-approach where global options must be set before applications etcetera can be selected. The same visual conventions are used here as above. It should be noted that when a component is selected, such as application 2 in Figure 26, a number of choices regarding it must also be made, for instance the revision, and customer and market. The latter must be specified even if the generic options are chosen. For revisions, the same rules as for components apply: revisions which can not be chosen or must be chosen should be indicated, and the test status (or combined status) should also be indicated directly to the user.

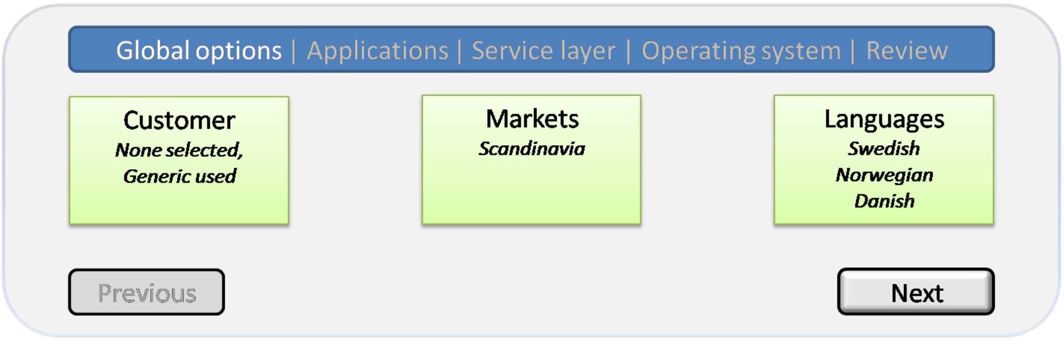

Global options | Applications | Service layer | Operating system | Review
Customer
None selected, Generic used
Markets
Scandinavia
Languages
Swedish
Norwegian
Danish
Previous
Next

Figure 25: Top-down approach, first step

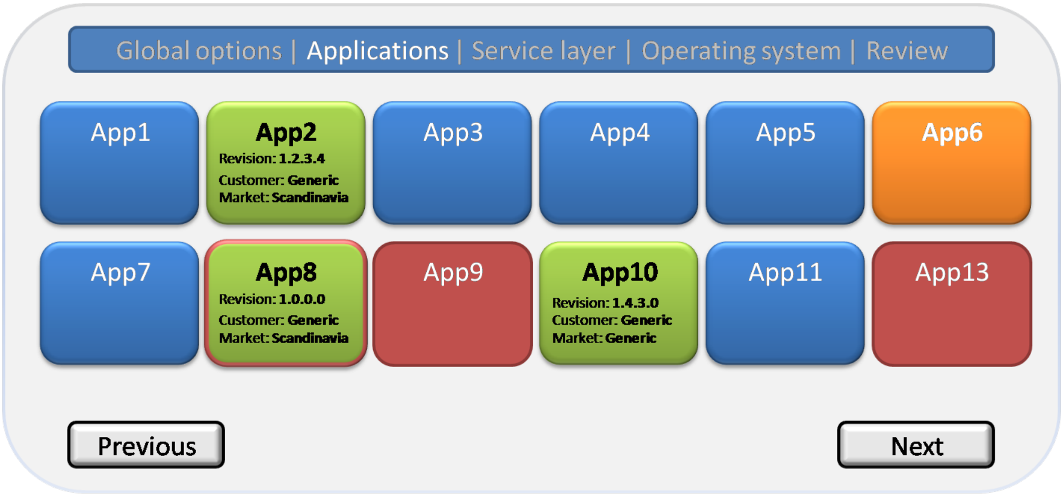
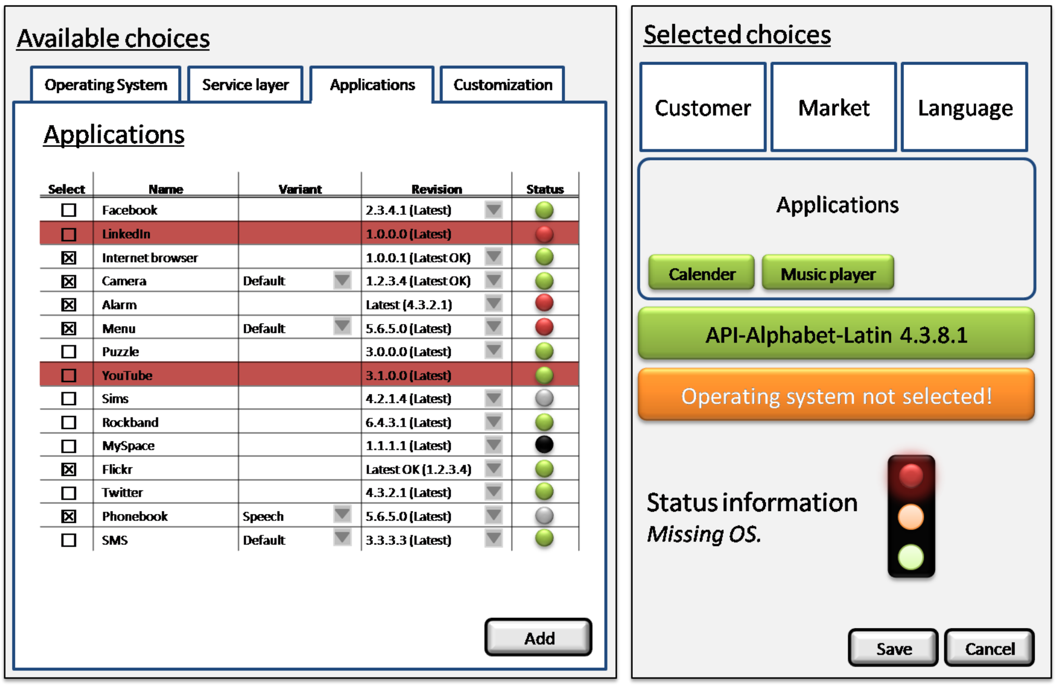


Figure 26: Top-down approach, second step

An important note regarding revisions is that it should not only be possible for the user to select a specific revision, such as 1.4.3.0, but also a few conceptual revisions – for example *latest*, *latest accepted* and similar notions. If this is done, the configuration should always be updated (automatically or after asking the user, depending on rules and settings in the configurator) when new revisions are made available. This corresponds to *partially bound configurations* [8].

The approaches above focus on the configuration as a whole, not on the details of individual components. These details are however the central interest for a different approach, illustrated in Figure 27. Components are selected from one layer at a time and added to the configuration which is displayed to the right. Components, variants, revisions, status etcetera can be seen at a glance and be added to the configuration once the user is satisfied. Global customization is still available on the configuration side.

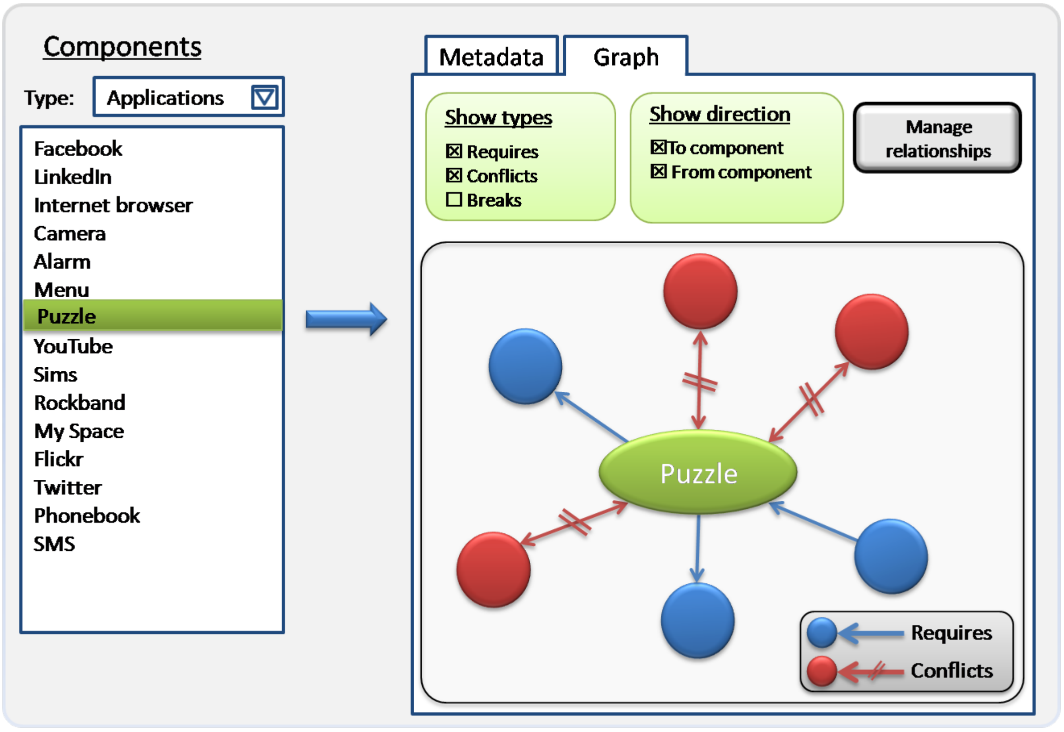


Select	Name	Variant	Revision	Status
☐	Facebook		2.3.4.1 (Latest)	●
☐	LinkedIn		1.0.0.0 (Latest)	●
☒	Internet browser		1.0.0.1 (Latest OK)	●
☒	Camera	Default	1.2.3.4 (Latest OK)	●
☒	Alarm		Latest (4.3.2.1)	●
☒	Menu	Default	5.6.5.0 (Latest)	●
☐	Puzzle		3.0.0.0 (Latest)	●
☐	YouTube		3.1.0.0 (Latest)	●
☐	Sims		4.2.1.4 (Latest)	●
☐	Rockband		6.4.3.1 (Latest)	●
☐	MySpace		1.1.1.1 (Latest)	●
☒	Flickr		Latest OK (1.2.3.4)	●
☐	Twitter		4.3.2.1 (Latest)	●
☒	Phonebook	Speech	5.6.5.0 (Latest)	●
☐	SMS	Default	3.3.3.3 (Latest)	●

Figure 27: Contents review proposal

The question of administration should also be dealt with when the configurator is designed. Whether it is a separate tool or part of the entire configurator, the considerations noted in the beginning of this section still applies. Several features should be available, including the relationship graph illustrated in Figure 28, the ability to edit metadata, statistics, etcetera.

As a final note on design, it is worth returning to the fact that not all users require the same things from the configurator. Regardless of the actual design, user profiles and views can be utilized to achieve a dynamic program suitable for many different users and roles.



Type: Applications

Show types
☒ Requires
☒ Conflicts
☐ Breaks

Show direction
☒To component
☒ From component

Figure 28: Administration

User profiles Different users may need to perform very different activities, such as creating configurations or administering components. Some actions should only be available to certain users; and sometimes the configurator should display slightly different information to different users. This can be easily accomplished with user profiles. Some options and actions can be hidden for certain profiles, and some users may even be allowed to switch between user profiles, typically from a more privileged profile to a less privileged one, in order to see the effect changes will have on other users.

Defining the actual user profiles to use is up to each producing company, but some general types of profiles can be suggested. *Configuration managers* is the most obvious profile, and should be allowed to do almost anything, including creating configurations which cannot work, performing administration, cleaning up the repository, creating rules as defined from business and legal requirements, and taking similar far-reaching actions. Configuration managers should typically also be allowed to switch to other profiles, to track down problems and see the effects of administrative tasks.

Developers could be allowed to create components and new versions thereof directly, since they have knowledge of properties and relationships; or they could be asked to submit changes to configuration managers, if tighter control is necessary.

Testers, likewise, could be allowed to update certain properties, notably test status. Testers would probably also need to work with configurations, in order to see what needs to be tested. *Account managers*, finally, typically only create configurations and do not edit components in any way – which could even be disallowed for that profile. More fine-grained profile restrictions can also be imagined, for instance giving certain persons access to act as testers for only some components, while remaining ordinary unprivileged users for all others.

Views No interface can be everything to everyone at the same time. A common and easy way around this is to split the interface into different views, where each view roughly shows just what is needed to perform a certain task. Many different persons can be expected to use the configurator, and to do so in many different ways, and views are a good solution to the problem of how to support them without cluttering the interface. A simple and elegant solution, albeit one which does require some effort, is to provide the different sketches above in the form of different views inside the same configurator.

5 Discussion

This section presents a discussion regarding the results – the context in which they are applicable, related studies, possible future extensions, etcetera.

There are a few things worth noting regarding the results. The study underlying this thesis was carried out mainly on one large company (more than 1 000 employees), with additional input from persons working in smaller companies. All companies are active in the software engineering industry. The results can therefore be expected to be applicable to most companies in that industry, but depending on the size and complexity of the products to be managed, some of the findings and suggestions may be unnecessary.

The configurator described in Design is geared mainly towards managing large products built from components which vary in several different dimensions, and where a very large number of products (on the order of hundreds or thousands) can be created from the same underlying components. For less complex systems, many aspects of the configurator are almost certainly overkill, for instance the more complicated parts of the status system.

A great deal of time and resources are required to introduce a component-based system into an organization. First of all, the product must be designed using components. This may be easy if the target is a new product which has not yet been designed, but altering existing source code to be modular in nature may not be cost-effective. Second, a suitable configurator must be created, or found and possibly extended. Third, the implicit properties, relationships and rules of the software must be found and made explicit.

The benefit is however very real. Quoting an existing vendor of configurators [9]:

> Companies that build configurable, multi-option, and customizable products are finding that a configurator can provide a competitive edge by reducing lead times, automating quotation documentation, increasing workforce efficiency, eliminating errors and rework, and increasing customer satisfaction.

The actual benefits realized depend on how much information is available. Deriving statistics, for example, is possible using almost any component system, so long as component and configuration data can be machine-processed. Other benefits, particularly the ability to work directly with requirements and integration between software and hardware components, require additional information, such as traceability. Some possibilities fall in the middle, such as impact analysis, which is possible when components and configurations can be read, but is greatly augmented if traceability is present.

5.1 Traceability

Traceability has been discussed in many configuration management articles and books [3, 12]. Traceability provides the ability to connect different items, for example components with their binaries, binaries with their underlying source code; source code with the underlying design, designs with their requirements, and so on. Figure 29 displays an example of such a hierarchy.

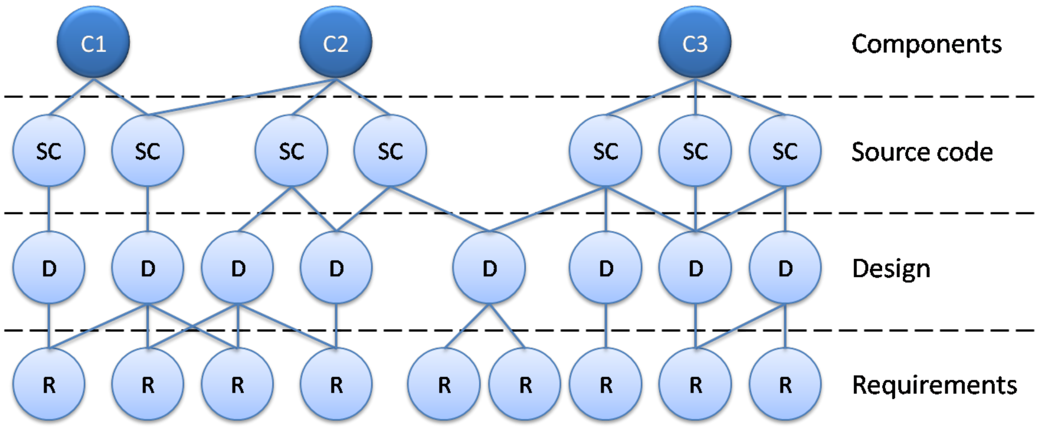


Figure 29: Vertical traceability

On a slightly more detailed level, most items are made up not only of sub-items but also of metadata, as displayed in Figure 30. There can also be traceability connections in more dimensions – test cases could be displayed. With traceability between a specific source code file and a certain test case, test cases could also be traced back to see which requirements were covered by tests. More advanced queries can also be answered, for example finding all requirements which are part of a configuration currently in use and where there is at most one test case. Similar queries could be answered if there is also traceability to and from hardware components.

Also, in many settings, the configurator is merely one piece of the puzzle. The user may need to select appropriate hardware for the software combined in the configurator (or vice versa), or there may already be a system for managing settings. In these cases, the configurator would have to allow for external references – meaning that not only must there be completeness and consistency between components inside a configuration, but also between components and hardware, and between components and settings, in all directions. In essence, this would mean merging software configuration management (SCM) and product data management (PDM) – and possibly other kinds of information management – into one single system, which Crnkovic et al talk about [4].

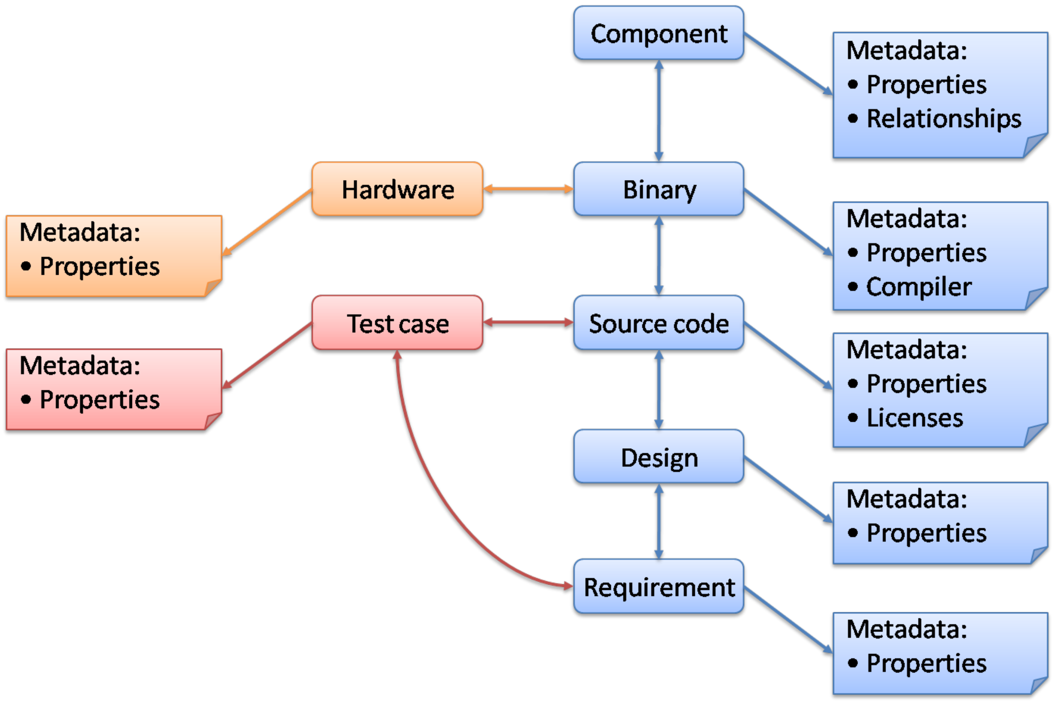


Figure 30: Traceability tree

5.2 Package managers

The initiated reader has probably noted that throughout the thesis, several issues already have a solution. Specifically, the problem of maintaining components and their relations is solved by programs such as the Debian package management system dpkg[20], the RPM package manager RPM[21], and similar. There are also front-ends, such as Apt[22], which help with some tasks related to composing configurations, for instance notifying the user of required or conflicting components. In and of themselves, however, these programs are not quite sufficient.

For one thing, these package managers serve a slightly different purpose than the configurator described in this thesis – they are focused on solving the dynamic composition problem of adding, updating and removing components on an existing system. They could be very useful on the finished product, to support changes to configurations in the field, but the configurator is required to solve the static composition problem correctly – the problem of composing a configuration *before* it is installed on a device.

[20] http://www.debian.org/doc/FAQ/ch-pkg_basics
[21] http://rpm.org/
[22] http://wiki.debian.org/Apt

There are certainly similarities between the two situations but there are also important differences, such as for whom the programs are intended (end-users versus producers), the available data (status information should probably not be visible to end-users) and capabilities of the programs (while some users are allowed to create incomplete or inconsistent configurations during static composition, this should never be allowed during dynamic composition).

5.3 Versioning systems

The configurator suggested in this thesis shares many traits with an ordinary versioning system (VS), the most basic of which is that both components and configurations are stored in different versions. In a sense, using this approach would mean having a hierarchy of such systems – requirements are managed in one type of VS, design documents can be stored in another VS, source code is managed in its own VS, the generated binaries may be stored in another, the components are stored in the configurator which is yet another VS, and the configurations may even be used in a further VS, higher up, akin to matryoshka dolls.

An alternative approach is to move towards one single system which is capable enough to handle all these aspects, and to handle them well – in essence incorporating all systems above into one. This would transform the hierarchical relations between them, into a linear one. Since this would be one integrated system, it would be possible to directly apply not only source code-versioning during development, but to directly, during coding, identify relationships which are needed at the configurator level, and so on. The drawback is obviously that such a system would become somewhat on the larger side.

5.4 Future work

There is still much to be done in this area, as evidenced by the fact that there are relatively few academic articles regarding configuration composition, and very few which focus on the problem of combining binary components; the work which has been done is mainly concentrated on combining components in the form of source code, and tends to drift towards architecture and system design.

One obvious starting point for future work is to evaluate alternatives for configurators in practice. Debian and similar system is one venue, which have the advantages of being free and well-documented, making them both cheap and extensible. Commercial programs are another venue, having the advantage that they can be expected to be more capable. Developing a configurator entirely in-house is also an option – a much more expensive option, but also one which provides complete flexibility and enables integration into existing systems.

The rules, as discussed in section 3.6, are another important area where more work is required: Finding the best strategy for achieving simplicity and capability at the same time and working out the details of the kinds of rules which are needed, how the underlying technical system for managing them might work, etcetera. A note on the underlying system is that rules can probably be separated into two parts: the *facts* regarding how items are connected, and the *logic* which dictates how facts can be combined to state whether a certain action is allowed or not. There will probably also be cases where rules can be combined or simply removed, and a way of detecting such cases automatically would be useful.

Also of interest, although tending towards compilers and text analysis, is the question regarding how to find relationships automatically, and for which relationships this is even possible. Automated smoke testing might also be a possible approach to identify both dependencies and anti-dependencies. Status data could possibly also be identified automatically, and details on how different statuses and relationships can be correlated to supply combined statuses and support to the user could prove very valuable.

Using traceability to create, or at least suggest, relationships or rules directly from requirements, design or source code would also be an interesting topic. Some rules can probably also be inferred from existing configurations.

6 Conclusions

This section presents the most important findings from Analysis, Design and Discussion in a clear and concise manner. For more details, please refer to those sections.

Using component-based systems allows for great flexibility, but at the cost of complexity. That complexity needs to be handled, and a semi-automated tool for helping to do so would be very valuable. Such a system could mitigate the risk of manual errors, provide statistics of component usage, and enable the tracking of license costs, among other things.

Components Components generally need to be *independent* of each other, and when they are not, the relationships need to be *explicitly documented.* Components have many *properties*, notably *market*, *customer* and *test status.* The test status may for instance take the values untested or approved, but many more can be envisioned.

Relationships Components may have *many different types* of relationships; among them requires, conflicts and replaces. Components may also have indirect relationships in the form of *feature dependencies*, which allow a loose coupling between service-providing and service-using components, or *component suites*, which is a method to group components into suites. Relationships are purely *technical.*

Rules Rules correspond to a higher *level of abstraction* than relationships and can be used to define requirements between components, component relationships, component properties, etcetera. They allow for managing more advanced associations between these items, thereby lessening the manual labor, and can be limited in time or space. Rules can be *explicit* or *implicit*, and different *types* of rules may be necessary, such as *business rules* or *legal rules.*

Configurations A configuration is created by specifying which *components* and their *versions* that are needed. Configurations have properties of their own, such as *production stage*, but also *receive property values* from their constituent components – an untested component, for example, means the entire configuration can receive a test status no higher than untested. Configurations may *vary greatly in size*, from single components sold as separate applications, to complete software systems.

Repository The repository stores all *configuration items* and their *metadata*, such as relationships, test status, etcetera. Changes and deletions are generally not allowed, only additions; but for some metadata, other considerations apply. To keep the repository clean, *commit checks* and controls for modifying items are needed.

References

[1] Brad Appleton, Stephen P. Berczuk, Ralph Cabrera, and Robert Orenstein. Streamed lines: Branching patterns for parallel software development. In *Proc. Third Conference on Pattern Languages of Programming and Computing, PloP, Monticello/IL*, number WUCS-98-25 in Technical Report, Washington Univ., Dept. of Computer Science, 1998.

[2] Wayne A. Babich. *Software configuration management: coordination for team productivity*, chapter 1, 2, 3, 5. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[3] Shawn A. Bohner and Robert S. Arnold. *Software Change Impact Analysis*, chapter An Introduction to Software Change Impact Analysis, pages 1–26. Wiley, 1996.

[4] Ivica Crnkovic, Ulf Asklund, and Annita Persson-Dahlqvist. *Implementing and Integrating Product Data Management and Software Configuration Management.* Artech House Publishers, 2003.

[5] Ivica Crnkovic and Magnus Larsson. A case study: Demands on component-based development. In *Proceedings of 22nd International Conference of Software Engineering*, pages 23–31. ACM, 2000.

[6] Susan Dart. Concepts in configuration management systems. In *Proceedings of the 3rd international workshop on Software configuration management*, pages 1–18, New York, NY, USA, 1991. ACM.

[7] C J Date, Hugh Darwen, and Nikos A Lorentzos. *Temporal data and the relational model : a detailed investigation into the application of interval and relation theory to the problem of temporal database management.* Morgan Kaufmann Publishers, 2003.

[8] Peter H. Feiler. Configuration management models in commercial environments. Technical report, Software Engineering Institute, 1991.

[9] Configure One Inc. Four types of configurators, 2008. URL: http://www.configureone.com/pdf/ConfiguratorWhitePaper.pdf.

[10] Axel Mahler. Variants: Keeping things together and telling them apart. In Walther F. Tichy, editor, *Configuration Management*, pages 73–97. John Wiley & Sons, Inc., 1994.

[11] Tom Milligan. Better software configuration management means better business: The seven keys to improving business value. Technical report, IBM (Rational), 2003.

[12] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software Product Line Engineering*. Springer, 2005.

[13] Tommi Syrjänen. A rule-based formal model for software configuration. *Report Series of Digital Systems Laboratory*, A55, 1999.