Stratified frameworks

Colin Atkinson and Thomas Kühne
AG Component Engineering
University of Kaiserslautern, Germany
{atkinson,kuehne}@informatik.uni-kl.de

Keywords: components, refinement, architecture, stratification

Received: August 16, 2001

Finding the optimal level of abstraction at which to document the architecture of a system has long been a problem in software engineering, particularly for large and complex systems. In this paper we argue that providing just a single abstraction level is inappropriate, and that instead, multiple architectural descriptions should be developed and documented, each capturing a specific aspect of a system's realization at a particular level of abstraction. Further, we argue that such a stratified architecture is especially valuable when used to organize a framework. After explaining the basic motivation for the work, and defining the basic principle of stratification, the paper illustrates the approach in conjunction with a small case study. The paper then discusses the methodological issues associated with the creation, application and maintenance of stratified frameworks.

1 Introduction

Although it is generally accepted that an optimal representation of software architecture includes multiple views, the structural view still invariably plays the central role. However, today's enterprise systems have reached such a level of complexity that a single "components + connectors" view of a system's structure as popularised by Garlan and Shaw [1] is no longer adequate. Not only has the scale and functionality of software increased, but with the advent of component technologies the boundary between the application and system level services (so called middleware) has significantly blurred [2]. As a result, it is no longer clear what level of abstraction provides the best overall structural description of complex software systems.

In the following we argue that for complex, industrial-scale software systems it is no longer appropriate to think in terms of just one structural view of a software system, but that instead it is better to provide multiple structural views (or strata), each elaborating upon a different aspect of the system's overall functionality. Since the strata in such a multi-level architecture each provide a complete description of the system's structure, they are not layers in the traditional layered architectural style. On the contrary, they each describe the entire structure of the system but at different levels of abstraction and with respect to different aspects of the system's overall functionality. Different stakeholders can therefore understand the system's structure at the level of abstraction which best matches their individual needs or tasks.

The advantages of such a multi-level view of system structure are twofold. First, since the relationships that connect individual strata reflect those that would result from a process of top-down, step-wise refinement, starting from the highest-level structural view, the approach can serve as the basis for an architecture development methodology. Instead of describing the whole structural architecture of a system in one fell swoop, system developers can focus on separate aspects of the system individually, and gradually work towards the detailed description [3]. A multi-level view of system structure therefore provides a powerful basis for separation of concerns and step-wise progress in the software development process. Moreover, with a reasonable degree of rigour in the structural description [4], the approach can provide a foundation for a generative approach to software development.

Second, by viewing the higher level strata as an end in themselves rather than a means to an end the concept of multiple views forms the basis for a powerful model of component-based enterprise frameworks. Regarding the upper-level structural views (or strata) as merely stepping stones in the creation of the real (most detailed) view makes them second class citizens, and makes them vulnerable to the neglect that befalls most intermediate "documentation" artefacts in software engineering. However, by viewing all strata as first class citizens of a framework, they become more stable software assets that are related more in space than in time. Multiple structural viewpoints within a component-based framework enhance the range of possible parameterisation points, and thus facilitate more flexible and straightforward instantiation.

In this paper we introduce and explain the concept of architecture stratification, and show how it can be used to increase the flexibility of component-based enterprise frameworks. We give a small example of what a stratified framework looks like, and then elaborate upon the processes by which such frameworks can be created, instantiated and evolved. Finally we discuss related principles and research areas in software engineering.

2 What Is Stratification?

In this section we explain the basic motivation for architecture stratification, and describe fundamental principles for its realization. We then consider the ramifications of stratification from the perspective of individual system components.

2.1 Which Architecture is *the* Architecture?

Consider a very simple client-server system in which a client component requests some form of service from a remote server component. At the highest level of abstraction the structure of this system could be captured using a class diagram of the kind in Figure 1.

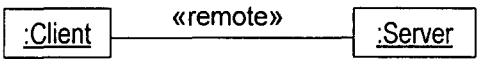

Figure 1: High-Level Client-Server Architecture

The stereotype, «remote», is used here to capture the high-level properties (or semantics) of the interaction between the client and server. This view of the system's structure conforms to the fundamental "components + connectors" concept of software architectures popularized by Garlan and Shaw [1]. However, it is not the only structural view that has this property. It is perfectly possible to provide alternative representations of the system's structure that conform just as well to "components + connectors" concept of architecture. Figure 2, for example, provides an alternative view of the structure of the system that explains how ORBs (Object Request Brokers) serve to mediate the interaction between the client and the server.

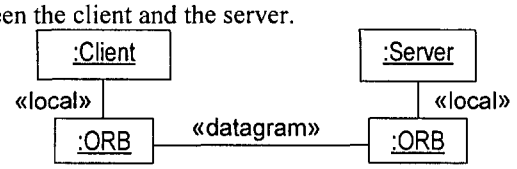

Figure 2: Low - level Client - Server Architecture

Figure 2 represents an equally valid structural description of the system. The only difference between the two views is the level of abstraction at which the structure is described. Figure 1 provides a high-level view of the system, involving a semantically rich interaction between the client and server, while Figure 2 gives a more detailed view that elaborates upon how the interaction between the client and server is realized.

Since both provide acceptable structural representation of the system consistent with the "components + connectors" model, the immediate question that arises is which is the correct, or the best, one? In other words which diagram represents the *real* architecture? The implicit answer in most contemporary methods is that the lowest, non-executable description of the system structure represents *the* architecture. Descriptions that are executable are typically regarded as an "implementation" or a "program" rather than an architecture, while higher-level descriptions are generally viewed as being incomplete or merely models.

Were one forced to select just *one* architecture to represent the structure of the system, the more detailed version of the two architectures would probably be the best choice. However, a better solution is obtained if more than one architectural description can be chosen. This is because higher-level views of the structure provide descriptions of the system that are of more value to certain stakeholders than the lowest-level view. For example, the user or customer is probably going to find Figure 1 a much more useful representation of the system than Figure 2. Therefore, higher-level views are valuable assets in their own right, and do not have to merely represent stepping stones along the road to the "real" (most detailed) architecture. In other words, rather than being just a means to and end, higher level structural views represent a valuable "end" in themselves.

The basic premise underlying the concept of stratification is that a complete representation of a system's architecture should contain all relevant abstraction levels. Moreover, the relationships between the different levels should be carefully and explicitly documented so that they represent a single coherent, whole, rather than a set of disjoint structural viewpoints.

2.2 Interaction Refinement

Most component and architecture description languages include the concept of "connectors" to capture the interaction protocols through which components, or architectural units, can be connected together. Ideally these connectors should be first class citizens of a description language, amenable to the same set of manipulations as components themselves. However, to date, no entirely satisfactory model for connectors has been found. Introducing the notion of abstraction levels, referred to as strata in the following, enables connector semantics to be understood in terms of lower level components.

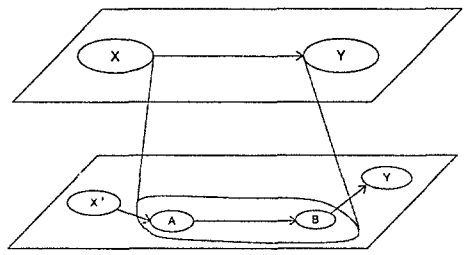

Figure 3: Interaction refinement

Figure 3 shows how a high level interaction between components X and Y is realised in terms of additional interactions and components at a lower level of abstraction. In other words, the interaction between X and Y is reified into components (A and B) and other interactions which reside one stratum lower in the hierarchy. Viewing connectors as collections of components at lower architectural strata, provides a clean model for their access and manipulation.

Note that the original component X in Figure 3 has to be adapted to X' in order to communicate with A rather than

with Y. This is an important aspect of stratification and is discussed further in section 2.3.

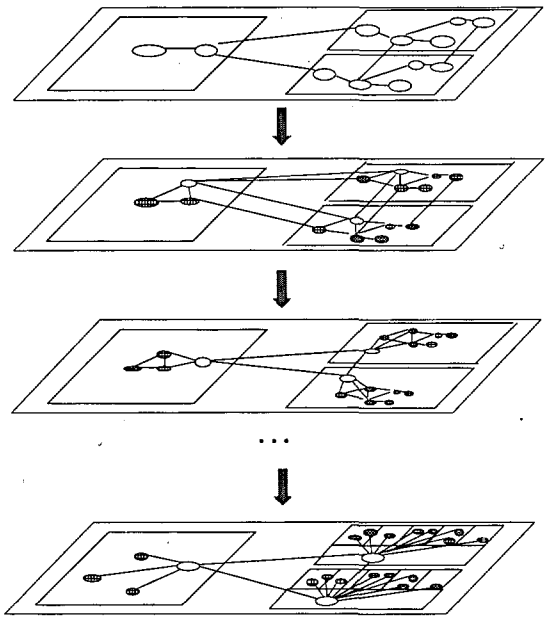

Figure 4: Structure of Stratified Architecture

In general, a complete stratified architecture is comprised of multiple strata as illustrated in Figure 4. Each stratum represents a refinement of the stratum above, and thus typically contains additional components and interactions that explain how interactions in the higher strata are realized. In Figure 4 the new components in a stratum are represented by white ovals, while those projected down from the stratum above are solid. The abstraction levels in a stratified architecture range from a high level, analysis-like description of the system down to the most detailed implementation-oriented view. The highest level will be most useful for understanding the overall solution strategy. This stratum is populated with components and connector types that are key to understanding how the system is organized. In the lowest stratum one will find the usual complex web of objects where it is hard to "see the forest for the tree". However, this is often the only appropriate level of abstraction to resolve realization details.

The key to effective stratification is the selection of appropriate sets of interactions within one stratum whose collective refinement elaborates upon a well-defined aspect of the system's realization. For instance, if in a refinement step starting from a business logic stratum one elaborates upon how remote interactions are realized, this will give rise to a "communication stratum". The next level down could be devoted to dealing with persistence issues and so on. Depending on the application domain the same sequence of types of strata will be useful in structuring a system's architecture according to its emergent (often non-functional) properties.

It is important to realize that each stratum of a stratified architecture is complete in its own right with respect to the level of abstraction it addresses. In contrast to layers, which only encapsulate a certain subset of a system, each stratum describes the entire system, albeit with varying degrees of abstractness. Clearly, all levels of abstraction are useful in a certain context. Someone trying to understand the overall structure of the system is best served with a high level view. Another person, whose task it may be to change the way data is marshalled over a network, gains more from looking at the communication stratum. Each stratum, therefore, provides the appropriate level of abstraction for a specialist working on a particular system property.

The multi-level separation of concerns provided by a stratification approach is even more valuable when it is used to organize a framework, since it offers enhanced opportunities for parameterisation. A fundamental concept in framework technology is the idea of so called "hot spots", i.e., points of adaptability. With a conventional, "single abstraction level" approach, all hot spots reside at the same level, and it is not clear what their individual role is within the complexity of the detailed architecture. In contrast, a stratified framework distributes hot spots over the various strata according to their place in the abstraction hierarchy. Hence, it is much easier to understand the role that a point of variability plays in the overall framework organization and what are the implications of plugging in a certain new behaviour.

Not only are the hot spots easier to see and understand in a stratified framework, but they can also have a greater range of effects on the eventual system's functionality. In conventional object-oriented frameworks, the hot spots are typically hooks to replace one *object* with another. This means that although the application developer is free to adjust the objects in the architecture, the interaction mechanisms, in contrast, remain relatively fixed. However, interactions, which also have a critical bearing on the system's functionality, are equally as likely to change as objects if not more so[1]. This is especially true when a framework has to be evolved to fit an extended or slightly different application context. By expressing interactions as objects in a lower stratum, a stratified framework can offer more flexible parameterisation than traditional object-oriented frameworks. Interactions within the framework can be made parameterisable by providing points of variation in their realization stratum. This allows application engineers to influence interaction semantics, as well as object semantics, by providing the required adaptation objects tailored to the precise needs of the customer.

2.3 Component Metamorphosis

As well as introducing new kinds of components and interactions, a given architectural stratum (except the top level) also contains projections of the components in the stratum above. The precise nature of the projection depends on the nature of the interaction refinement

[1] In fact, the success of object-oriented architectures partly lies in the fact that objects are more stable architectural abstractions than functions (i.e. interactions)

defining the relationship between the strata. Sometimes a component at one level will appear in the lower level completely unchanged. Often, however, the implementation or even interface of a component will be changed, either due to a change of its interaction partners, or a change in the nature of the interactions. For example, Figure 3 shows how component X is changed to X' because of the refinement of its interaction with Y. In the lower stratum the new version X', only communicates directly with A rather than with Y.

Because of the obvious analogy with the biological development of insects, we refer to the set of changes applied to a given component as it is projected across the different strata as *metamorphosis*. When viewed from the perspective of an individual system component, stratification can be understood as the successive metamorphosis of a component to its most detailed form in the most detailed architecture.

Consider, for example, the elaboration of an aspect of a system, such as authorization, within a given stratum. Authorization cannot be handled by a single module alone, but is a cross-cutting concern, i.e., it cuts through component boundaries and its introduction causes changes that are scattered in a non-local manner throughout the system. In a stratified architecture, however, the strata above the aspect-elaborating stratum do not deal with the aspect in an explicit manner, but defer matters of authorization to the stratum that explicitly elaborates upon the realization of this aspect (i.e. the authorization stratum). When the authorization interactions from this stratum are refined in lower level strata, the spreading of the aspect-related details starts until the bottom level is reached (see section 3.1 for a sample application).

The need for changes to components as they are projected into lower strata is simply a testimony to the fact that components typically cannot be used "out of the box" and that a pure black-box "plug & play" composition strategy rarely works in practice. To make component composition really work, components have to offer some open implementation facilities [5] and connectors have to be grey-box connectors [6].

3 Example stratified Framework

To illustrate how the principles of stratification explained in the previous section would be applied in practice, in this section we walk through a small example. The subject of the example is a simple banking system which stores and accesses accounts on behalf of customers. Key requirements for this facility are that the bank and the accounts are potentially remote, access to accounts must only be granted to authorized users, and interactions between the bank and accounts must be secure (i.e. non-interceptable).

In the discussion below we focus on only a small part of the systems potential functionality, but the ideas explained can be scaled up easily to the other parts of the systems.

3.1 Structure of a Stratified Framework

Naturally the top-level stratum is the simplest, since it describes the structure of the system using the semantically richest connectors. The class diagram in Figure 5 shows that the class `Bank` interacts with the class `Account` by a means of an interaction labelled with the stereotype `«remoteSafe»`. This conveys the fact that the interaction is a semantically rich connector and "wraps up" the requirements for remoteness, authorization and security identified above.

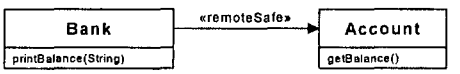

Figure 5: Application Stratum

The next stratum elaborates upon the realization of the authorization aspects of the system's functionality, as illustrated in Figure 6.

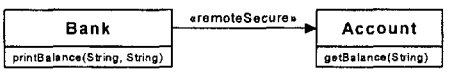

Figure 6: Authorization Stratum

In principle, this is done by defining the additional components and interactions that are involved in the authorization process, and by defining how the components and interactions from the Application stratum are changed. In this case, no new components are required but the interaction between the `Bank` and the `Account` components needs to carry the information needed to perform the authorization. In particular, methods `getBalance()` and `printBalance()` now are elaborated to feature one more parameter. A so called PIN (personal identification number) has to be provided when requesting a service from the bank and is checked by the account before access is granted. Note, how the annotation of the association has changed from `remoteSafe` to `remoteSecure`, as the safety aspect of the interaction is addressed. It remains to specify that the interaction in addition is remote and secure.

The next stratum elaborates upon the distribution aspect of the system. As illustrated in Figure 7, this uses the broker pattern as defined by Buschmann et al. [7] to realize distribution.

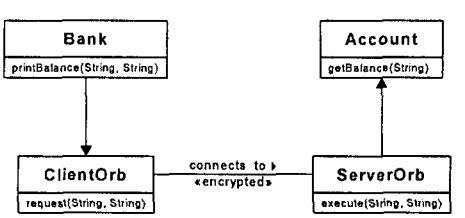

Figure 7: Distribution Stratum

In this particular refinement, two new component types and one new interaction type are introduced. Note that this implies a change to the required interface of the `Bank` since it must now interact with its `ClientOrb` rather than with the `Account` directly. The `Account` component type, on the other hand, is totally unaffected by this refinement.

The final stratum in this example elaborates upon the realization of encryption. As illustrated in Figure 8, this is achieved by the introduction of an additional `Encryption` component type that is used by the `ClientOrb` and `ServerOrb` components to respectively encode and decode message before they cross the network.

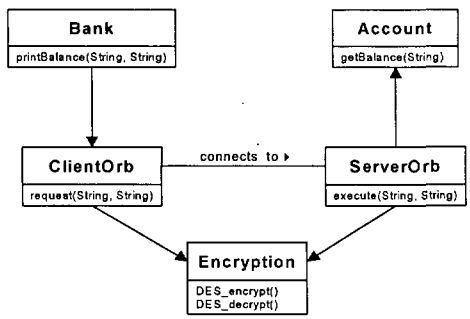

Figure 8: Encryption Stratum

Further strata would be defined to elaborate upon successively lower level details of the system's realization until a level is reached that can be translated directly into an executable form. In general, strata are defined to focus on the realization of each of the key aspects of the system realization. Since the aspects affecting a system tend to be domain specific so are the strata.

3.2 Refinement transformations

Documenting architecture at multiple levels of abstractions as shown in the previous section is a useful activity in its own right. However, it is unrealistic to expect that all the strata can be made mutually consistent manually.

Today it is rare to find examples of software development approaches in which just the code and a single, very high level architectural description are kept mutually consistent, so it is not feasible to expect this to be achievable when there are even more system representations. Therefore, a stratification tool is needed that can organize and help apply -

1. an annotation language and
2. a set of refinement transformations,

both of which are typically domain dependent. In the example used above we exploited the fact that one could use annotations such as `encrypted` or `remoteSecure` and that there are relations such as `remoteSafe = authorized + remoteSecure`. The task of the annotation language is thus to provide labels that are used to refer to certain interaction semantics and relations between them.

Each atomic label of the annotation language refers to an individual refinement transformation. Such a transformation takes one component scenario involving a labelled interaction and creates a new scenario in the stratum below. The resulting scenario has less rich interactions and possibly additional components implementing the interaction semantics of the initial interaction (see Figure 3). A tool that fully supports the stratification approach therefore needs to –

- manage the annotation language.
- support the definition of refinement translations.
- apply the refinement translations, thus keeping the strata in sync with each other.
- facilitate the reengineering of detailed scenarios to high-level interactions by applying refinement translations in the reverse direction.

In particular, for the purpose of discovering and documenting the architecture within already existing software, the last point is useful in creating high-level views from representations with too much detail. Naturally, in creating a stratified architecture one will *use* and *define* both the annotation language and refinement translations in an interleaved fashion. Both sthese assets, although typically domain dependent, are reusable for other systems in a similar domain.

4 Developing, Using and Evolving Stratified Frameworks

The previous sections have respectively defined and exemplified the structure of a stratified architecture. In doing so they have essentially focused on the *product* of stratification, but have said little about how stratified architectures are created, used and evolved, that is, the *processes* related to stratification. In fact, just as a modeling language such as the UML can be understood and used without an associated process, the concept of architecture stratification is useful as an organizational principle on its own. It is, therefore, useable with a variety of methodological approaches. In this section we explain the most important methodological issues associated with stratified frameworks, and suggest some solutions.

A framework oriented approach to software development raises three main questions –

- how should the framework initially be developed ?
- what are the necessary actions to instantiate the framework so that applications meet the needs of a specific customer?
- how should the framework be evolved over time in response to change requests?

4.1 Developing Stratified Frameworks

In addition to the usual framework development activities the following five main activities have to be

incorporated into a process for creating and using stratified frameworks-

1. Identify candidate strata
2. Order candidate strata
3. Identify hot-spots and allocate them to strata
4. Elaborate strata
5. Partial Implementation

4.1.1 Identify Strata

The first challenge in the development of a stratified framework is the identification of the appropriate strata. This question must be tackled from both a top down and bottom up perspective. From the top down perspective, the key aspects of the system as identified in the requirements specification are extracted and analysed. Each system aspect that is likely to involve numerous objects is an indicator of a possible stratum. But it is also possible that an aspect does not introduce objects at all but only changes method signatures and implementations (e.g. the "authorization" aspect in the example in section 3.1). From the bottom up perspective, the key reusable components within standard or predefined architectural solutions (e.g. CORBA) are evaluated and considered for the problem in hand. By carefully balancing the needs of the application against the potential solutions an optimal set of strata is gradually distilled. In general, a strata-inducing aspect could be anything that is not essential to a very high level analysis model of the system. A good indication for proper aspects is the one would expect them to appear in similar systems within the system's domain.

4.1.2 Order the Strata

The next step is to order the strata according to their dependencies and levels of abstraction. A key tenet of the stratification approach is that strata can be arranged in a strict order, such that a stratum on one level depends on (is refined to) only the stratum immediately below. Often, however, analysis of the initial candidate list of strata reveals mutual dependencies, or strata that are dependent on two or more other strata. Such situations must be rectified either by removing strata from the list, or by splitting strata into more specialized strata. At the end of this process, a list of strata must be identified in which each stratum only depends on (is refined to) the stratum immediately below.

4.1.3 Identify and Allocate Hotspots

The next step is to analyse the intended future uses of the system with a view to the identification of the key points of variation, or "hot spots". This activity is akin to the scoping and variability analysis activities found in product line approaches. Once candidate hot spots have been identified, a first pass is made at allocating hot spots to the identified strata. This is driven by a consideration of the aspects of the system that are affected by particular hot spots. One straightforward technique for accomplishing this is to consider one hot spot at a time and proceed down the strata from top to bottom until the hot spot's purpose is given meaning by the corresponding stratum. Strata above this stratum suppress the level of detail that the hot spot is addressing and strata below are likely to contain abstractions which support the realization of hot spot components. A concrete definition of the hot spot cannot be completed until the architecture of the stratum has been elaborated, but an initial allocation of hot-spots to strata can be made based on the purpose of each hot spot.

4.1.4 Strata Elaboration

The most challenging activity is the elaboration of the individual strata. Apart from the top-stratum, the process is basically the same for each. The process of creating the top-level stratum is a little different, since it has much in common with the typical process of deriving an initial high-level architectural in a traditional object-oriented development method. The main difference is that care must be taken to keep the top stratum as abstract as possible, since the architectural features derived from specific aspects of the system should be elaborated in lower-level strata. This differs from traditional development methods where all aspects of the system are usually considered in the (single) architectural description.

The main difference between normal architecture development, and the development of the top-level stratum in a stratified framework is that the latter is allowed to make assumptions about services elaborated by lower-level strata. For example, in a stratified framework containing a stratum that realizes remote communication (i.e. in the style of an ORB), a remote interaction mechanism can be assumed as a basic architectural primitive (i.e. connector) by any stratum above it (including of course the top level). This is not so in regular architecture development, wherever functional requirement must be considered within the single overall architecture.

Once a first version of the top-level stratum has been defined, the next step is identifying which of the interactions at the top level are affected by the aspect handled by the stratum below. These interactions are then analysed in turn to identify appropriate refinements based on the facilities handled by the underlying strata. This can be done using the transformations identified above. The set of candidate refinement steps are then analysed and a common solution is distilled. A complete architecture for the underlying stratum is then determined, taking into account relevant architectural patterns or lower level design patterns depending on the current level of abstraction. Note that the realization of a particular stratum as well as the definition of the refinement steps mapping into the stratum lends itself to enactment by someone who is an expert in the area the stratum addresses. For instance, experts in distribution, fault-tolerance, persistence, encryption, etc. will work on their respective strata, potentially in parallel to some extent.

The previous activity is repeated until each stratum in the framework is elaborated and candidate architectures have

been selected. Finally, once the detailed architectures are available, the hot spots can be revisited and accommodated in the framework using one of the standard object-oriented techniques, as explained below.
One of the things that must be considered at each stratum is the concrete representation of the various hotspots allocated to that level. Various techniques can be used to capture hot-spots (for instance, see [8]), but for our purposes here it is sufficient to distinguish between so called "white-box" and "black-box" specialization [9]. A mature framework, which has been instantiated a number of times and has already undergone all major refactorings, will offer mostly if not exclusively black-box specialization. Here the hot spot has a well defined interface and is parameterised by plugging in a prefabricated component (e.g. by using pluggable adaptors or the Function Object pattern [10]). If a prefabricated component (packaged as part of the framework) is used one of the known framework behaviours is selected. If a new component has to be created than the framework will still behave within the range of possibilities opened up by the black-box hot spot. Analysing were that hot spot is located, i.e., which stratum it populates and which higher-level interactions depend on it gives a good picture about the scope of the variation introduced. Here the term traceability is given meaning, since it is possible to travel the refinement steps up and down to trace where the variation is being propagated. This advantage of the stratification approach becomes even more important when the other form of parameterisation, i.e., white-box specialization is used. In this case, a framework class is subclassed to override one or more methods. This form of framework adaptation is common when the framework is still evolving to its final form and it is not yet clear how the black-box hot spots should be modelled and where they should be located. Not only does this specialization variant require more intimate knowledge of the framework but it is also potentially much more dangerous, as erroneous overriding of behaviour could have disastrous effects on the integrity of the framework behaviour. Therefore, if white-box specialization has to be used it is doubly important and rewarding to travel up and down the strata, following the refinement transformations, to check the scope of the changes made.

4.1.5 Partial Implementation

The lowest-level stratum of a stratified framework represents the realization of the system containing all of the details introduced in the strata above. This is the stratum which is used to derive the implementation of the system. Depending on the level of detail present, i.e., on the amount of low level design aspects introduced by one or more strata above, the derivation of an implementation ranges from a standard "design-to-implementation" activity to a simple "one-to-one" mapping into a programming language.

4.2 Using the framework

Using a stratified framework to create a concrete application involves three main activities:

1. Variant Resolution
2. Hot-spot Instantiation
3. Implementation Completion

4.2.1 Variant Resolution

Variant resolution is essentially the same for all framework-based approaches to software development. The basic goal is to determine precisely what set of features the desired variant of the system requires in terms of the hot spots made available by the framework. Those features that correspond to default options, if these are available, can be immediately dealt with and should be documented with interaction annotations so that they can be validated in case the default option changes in the future.

4.2.2 Hot-Spot Instantiation

The next step is to actually resolve the non-default choices by providing concrete representations of the chosen features for each of the affected hot-spots. The advantage of stratification is that the hot-spots are clearly separated and defined with respect to the aspect of the system that they affect. The instantiation of hot-spots in a stratified framework is therefore much easier than in other non-stratified frameworks. First, it is clearer where to look for an appropriate hot spot because of their distribution to strata. Second, as detailed above, the refinement transformations represent paths to the places within the framework affected by the hot-spot allocation.
Since hot-spots are defined at different strata, the effects of the allocation of a hot spot must be carried down through the strata to the lowest stratum. This involves the reapplication of the refinement transformations where appropriate. As indicated beforehand, this is where stratification can benefit most from appropriate tool support.

4.2.3 Implementation Completion

Once the realizations of all non-default hot-spot resolutions have been mapped down to the bottom stratum, any modified or completed architectural elements must be translated to an executable form using the same techniques specified previously.

4.3 Evolving the framework

The final major activity involving in building and applying stratified frameworks is their evolution. No software system, or family of systems, is constant, so some systematic procedure is required for handling the inevitable requests for change. In the context of a stratified framework this involves three main activities:

1. Change analysis
2. Change Localization
3. Realization

4.3.1 Change analysis

Most requests for changes are received from users and customers of specific version of the system. Thus, before changing the framework, it is first necessary to determine whether the request can actually be handled by a reinstantiation of the framework with a modified set of parameters. If this is so, the existing framework is capable of handling the request and no framework level modification is necessary.

4.3.2 Change Localization

For those changes that are deemed appropriate for the framework, it is then necessary to establish two things. First, is the change "almost" covered by an existing hot-spot? If so, the hot-spot realization must be generalized to handle the change. If not, then it is necessary to repeat the "Identify hot-spots and allocate them to strata" activity in framework development, and adjust the framework accordingly to handle the requested change.

4.3.3 Realization

In both cases it is necessary to bring the whole framework, including the associated implementation, into a consistent state by reapplying all relevant refinement transformations. This ensures that all changes are reflected in lower strata and the executable code. Again, tracing out the affect of changes within higher strata and following them along the refinement transformations to all critical locations within the framework gives an extra means of validating the new framework version in addition to regression tests of the framework itself and its known instantiations.

5 Related Work

The concept of stratified frameworks is related to several other ideas which are currently under investigation in the software engineering community. In this section we briefly describe the most important of these.

5.1 Aspect-oriented programming

Aspect-oriented programming has received a great deal of attention recently as a way of separating out, and partially automating, the treatment of orthogonal aspects of a systems overall functionality [11]. The stratification approach shares the same basic philosophy of aspect oriented programming since it is built on the basic idea of separating concerns for orthogonal issues [12]. As explained above each stratum basically focuses on the realization of a specific aspect of as system.

Most aspect oriented approaches only deal with the code level of a system whereas stratification is an architectural approach as well. The main difference between stratification and the primary "aspect-oriented programming" approaches, however, is the strategy adopted for the description and manipulation of these aspects. In aspect oriented programming, the emphasis is on the semi-automatic integration of distinct aspects using special tools typically known as weavers. This is fine for a number of aspects that can be handled well by weavers, but in principle this approach suffers from a hard superimposition problem when independent aspects affect the same software abstraction. It is, for instance, a non trivial problem to decide the order of the modifications to be made in such a case. Many of the "concerns" which need to be addressed in large-scale industrial software development are too complex to be handled in this way. Sometimes *all* the details of a realization need to be available in order to find a bug or achieve the desired behaviour. In this case a pure version of the system plus a number of aspects and an implicit weaving strategy is insufficient. The stratification approach essential offers an alternative way of separating aspects using a more manual software engineering based approach (based on standard software development activities). The order of aspect introduction is fixed. Each aspect is defined on a complete system description (with regard to the required level of abstraction) and not on a pure, i.e., incomplete, system and possibly more aspects. While stratification successfully resolves superimposition conflicts with a linear and dependent aspect introduction, the downside of this is that it is not possible to view a system's architecture with an aspect B but without an aspect A if A is introduced earlier (higher up the hierarchy) than B. With an optimal ordering of the strata, however, it is unlikely that such a need will occur.

5.2 Reflective Architectures

Another community of researchers, which has pursued the idea of capturing distinct aspects of a system's behaviour within separate "architectures" is the reflective programming community. The basic premise of this community is that the description of a system should comprise two "architectures", one representing the standard application software, and the other "meta" architecture describing certain aspects of the support software that are amenable to change, sometimes even during execution of the software. Since the meta architecture allows aspects of the system to be modified (at run-time), which were previously considered fixed primitives of software construction in traditional approaches, the whole approach is known as a reflective architecture. Since is separates out the description of different aspects of the systems behaviour into separate "architectures" this approach has much in common with stratification. In fact, we view stratification as a generalization of the reflective architecture approach - a generalization which supports an arbitrary number of "architectures" rather than just two. Interestingly, in the case of a reflective programming language, such as CLOS, the reflective "stratum" must be viewed as being below the programming language stratum, as it elaborates and defines the meaning of programming language mechanisms (called interactions in systems). By the same token a meta stratum will be found *below* strata which depend on it, although the general interpretation of "meta" being higher and above something.

6 Conclusion

As the complexity of software systems continues to grow, and the boundary between applications and systems level software continues to blur, it will become increasingly difficult to visualize the structure of software systems using only one structural model. If all the "components and connectors" for all aspects of the system's functionality are crammed into a single structural model, it will become increasingly impossible to "see the forest for the tree." In this paper we have described a solution to this problem referred to as architecture stratification.

The basic premise of the approach is that several distinct models of a system's structure should be developed, each yielding a different level of abstraction, and focussing on the elaboration of different aspect of the system's functionality. As such, the approach can be viewed as extending the tenets of aspect-oriented programming to higher-level development concerns, but in a more software engineering oriented style.

The idea of creating a series of models of a system's structure, related by rigorous interaction refinement transformations, has value as a systematic technique for realizing the full, all encompassing structural model ready for implementation. When used in this way, the higher-level strata essentially represent stepping-stones, or milestones, along the route to towards the development of the "real" or ultimate all-encompassing architecture. The higher-level strata thus play a secondary role, and are likely to be neglected over time.

Greater leverage can be gained from the architectural stratification approach, however, when it is used to describe the structure of a component-based framework. When used for the purpose, the higher-level strata have the same weight as the lowest level stratum, and play a valuable role in describing the framework structure from the perspective of a particular stakeholder. The main value of the distinct strata is to provide a clean way of capturing functional (i.e. interaction-based) variation points in terms of components (albeit in a lower stratum), and to provide a clearer model of the effects of parameterised "hot-spots".

Another major benefit of the separation of concerns afforded by stratification is that is clarifies the role of proven architectural patterns in the structuring of a software system. The majority of architectural and design patterns that have been published to date are intended to be used together, but when a single, all-encompassing architectural model is used to describe the structure of a system, the application and interrelationship of specific patterns is all but lost. Stratification helps by providing distinct abstraction levels that focus on the deployment of only one or a few patterns, and thus the role and location of specific patterns, as well as their relationship to other patterns at different strata, can easily be discerned.

As well as explaining the basic principles of stratification, and illustrating their application in the context of a small example, this paper provided an outline of the primary development activities associated with the approach. The paper also discussed the relationship of the stratification concept with other leading architectural research initiatives. As a generalization of the aspect-oriented programming and reflective architecture approaches to software development, the principle of architecture stratification represents the next step along the road towards greater separation of concerns in the engineering of quality of software systems.

7 References

[1] M. Shaw, and D. Garlan (1996) Software Architecture: Perspectives on an Emerging Discipline, Prentice-Hall

[2] C. Szyperski (1998) Component Software, Addison-Wesley

[3] Nicholas Wirth (1971) "Program Development by Stepwise Refinement." Communications of the ACM, vol. 14, no. 4, pp. 221—227.

[4] M. Broy (1997) Towards a Mathematical Concept of a Component and Its Use, TUM Report I9746.

[5] G. Kiczales (1996) Beyond the Black Box: Open Implementations, IEEE Software, vol. 13. no. 1, pages 8—11.

[6] U. Assmann and A. Ludwig (1999) Introducing Connections into Classes with Static Metaprogramming, 3rd Int. Conf. on Coordination, LNCS 1594.

[7] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stal (1996) Pattern-Oriented Software Architecture – A System of Patterns, John Wiley & Sons.

[8] W. Pree (1994) Meta Patterns – A Means for Capturing the Essentials of Reusable Object-Oriented Design, ECOOP '94, pages 139—149.

[9] R. E. Johnson and B. Foote (1988) Designing Reusable Classes, Journal of Object-Oriented Programming, vol. 1, no. 2, pages 22—35.

[10] T. Kühne (1997) The Function Object Pattern, C++ Report, vol. 9, no. 9, pages 32—42.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier and J. Irwin (1997) Aspect-Oriented Programming, *In proceedings of the European Conference on Object-Oriented Programming*, Finland. Springer-Verlag LNCS 1241.

[12] C. Atkinson and T. Kühne (2000) Separation of Concerns through Stratified Architectures, International Workshop on Aspects & Dimensions of Concerns, ECOOP 2000, Cannes, France.