Externalizing Component Manners to Achieve Greater Maintainability through a Highly Re-configurable Architectural Style

Ali Arsanjani
IBM Global Services, USA
aarsanjani@acm.org

Hussein Zedan
Software Technology Research Laboratory, DeMontfort University, UK
hzedan@dmu.ac.uk

James Alpigini
Penn State Great Valley, USA
jia7@psu.edu

Abstract

The maintenance and evolution of distributed, heterogeneous software components; including both legacy and green-field subsystems is described through a highly re-configurable architectural style. It is shown how this architectural style is realized through identification, separation and externalization of a formal specification of the manners of the application domain and its components.

The approach is based on the notion of Enterprise Component (EC). An Enterprise Component is defined as an architectural pattern that is leveraged to provide a uniform mechanism for management of component boundaries between otherwise entropic systems consisting of multiple legacy systems coexisting with newer, object and component-based application programs.

EC’s are identified through a domain decomposition that includes mapping business architecture onto component-based software architecture. Extensions to current methodologies and architectural practices to support and realize such a style are presented.

1. Introduction and Motivation

This paper is based on work done at IBM Global Services 1998-2002 across multiple projects in various industry segments and domains. Over fifteen projects were studied in the fields of telecommunications, banking, insurance, mortgage, e-government, financial services, patents, etc. Among these projects, a set of common challenges and problems were identified. First was the difficulty in providing harmonious functionality that involved the integration and consolidation of multiple “stovepipe” legacy applications, often with newer, “green-field”, J2EE™-style applications that capitalize on web application server models. The two sub-problems in this category were: a) integrating legacy with legacy, b) integrating legacy with green-field.

A second problem was the immense functional redundancy that exists within a typical enterprise application portfolio. The sheer number and lack of harmony between applications contributes a complexity that introduces architectural entropy; disallowing the application of cost-effective, non-intrusive changes as software evolves to meet changing requirements.

This led to the third and perhaps more significant problem of managing the evolution of their heterogeneous distributed application portfolios. Although the applications may have contained parts that would provide the same basic functionality (perhaps in a slightly customized fashion for each product line), each of the applications was not appropriately decomposed (or modules) to expose those quanta of functionality that another application could reuse.

This had led to “functional duplication”: an overlap of functionality across the application portfolio. The functional units or identities that provided a cohesive set of behavior in the object-oriented sense were missing. This lack of componentization or poor decomposition into modules was only part of the problem. Even in systems that were supposedly object-oriented, the collaborations between components were often “hardwired” into the application disallowing a “rewiring” of the components to satisfy new business flows. But the change in business flows and component collaboration were key to maintaining robustness in response to business change. Flow adaptation or allowing the programmatic, dynamic re-configuration of a business flow in a business-to-business interaction was therefore extraordinarily difficult to achieve.

Other issues included: 4) Web enablement of legacy applications, assimilation of heterogeneous systems inherited through acquisition, 5) rapid introduction of products and services, 6) managing large quantities of business rules with the solution falling within a spectrum rather than one solution type and 7) the need to integrate legacy with newer development paradigms and 8) issues relating to programming-in-the-large or “megaprogramming [1].” These problems will be addressed through the creation of a highly re-configurable architectural style (HRA) using the Externalization process of Variation-oriented Design (VOD).
It is a common misconception that software maintenance begins when the completed product is delivered to the client and deployed into a production environment. While this is formally true, in fact decisions that affect the maintenance of the product are made from the earliest stages of design or of business modeling. One of the key design decisions is the application flow, workflow, and business flow or if they exist, collaboration between components. If there are no components, then the collaboration is merely the application control flow. Without an appropriate separation of concerns that decomposes an application into functional (legacy) or object-oriented (green-field) quanta of interaction, or software components, encapsulation of the design decision of the flow is not possible. Identification of flow precedes encapsulation (objects) or separation (object or procedural code).

A study of the abovementioned fifteen applications points to three root causes of many of the problems stated above; categorized in terms of identification, separation or encapsulation and externalization. These activities of variation-oriented design, aimed at creating more maintainable systems, often culminates in Externalization, which is the main theme of this paper.

This process is summarized in the next section on variation-oriented design (VOD). Section 3 brings the discussion within the context of component-based development and integration and discusses how components have manners. Section 4 elaborates on the VOD process of Externalization and discusses the process of introducing change into a software architecture or component. The process of Externalization is seen to take place across a spectrum, with the target of externalization starting out to be data, then behavior and finally manners.

Once we arrive at the externalization of manners that drive an application or large-grained component, we enter the realm of grammar-oriented object design (GOOD), in section 5, which takes component-based development one step further into the realm of product line architectures. The application of GOOD produces a HRA Style. Section 6 then describes the anatomy of GOOD step by step, showing how a domain-specific language can be used to implement the manners of a component (or the entire application) and through externalization, can drive the application. The notion of externalization is formalized using domain-specific languages. The focus is on the externalization of highly changing items in the architecture and application. Strategies and patterns for both anticipating and managing the changes are demonstrated. This concept is closely related to the notions of stability and symmetry in software architecture [2] built upon VOD. VOD seeks to produce more stable and balanced software architectures through the process of Externalization. The paper is concluded in section 7, which shows the application of GOOD to build the components of a product-line architecture.

2. Variation-oriented Design: Adaptability Through Gradual Externalization

Thus, in order to build a more easily maintainable distributed architecture, a highly re-configurable software architecture (HRA) is needed [1,5,7,4,3]. Key to realizing this goal is the separation of changing from less-changing, more stable aspects of software architecture and the corresponding key abstractions within the domain of study. Parnas describes this in terms of ease of extension and contraction [4]. Variation-oriented design identifies commonality and focuses on building pluggable variations through identification, separation and externalization of variations. One of the highest volatility variations is the high-level application flow of a set of loosely coupled components. These can be wired together in multiple ways based on current business needs. There is no reason to hard-wire them if they tend to change. Externalizing these behavioral variations is described below.

The methodology and architecture support needed to build a highly re-configurable architectural style is discussed and the stages of identification, separation and externalization of self-descriptive configuration and adaptation information pertaining to an Enterprise Component (EC) are described. Domain decomposition and subsystem analysis create the EC’s that cut across legacy and green-field applications and factor common elements of business functionality into a highly re-configurable system of components.

**VOD Process Summary.** Identification refers not only to the identification of changing or varying elements in the problem domain, but also in partitioning the functionality of a domain. We refer to this as domain decomposition; decomposition of the domain into cohesive units of business functionality or services that align with business goals. The two types of decomposition are structural and dynamic: the software quanta and their collaboration or manners.

Separation or encapsulation, as per Parnas [5], pertains to hiding the design decisions that may tend to change within the software quantum. Although this is traditionally considered to be primarily the encapsulation of data, VOD [6] generalizes this notion to include the reification of the variation so that new componentized variations can be pluggably added according to the open-closed principle [7].

The second aspect of the encapsulation of design decisions pertains to the dynamic re-configuration of collaborations between components. Externalization signifies gaining the ability to dynamically re-configure a set of distributed software components that span legacy...
and green-field. Collaborations and manners -- the rules governing the behavior of components, tend to be hard-coded and locked deep within the application.

Once the aspects or axes of variation upon which many design decisions rest, have been identified, the dynamic collaboration between components and the rules governing that collaboration, are formally specified using a domain-specific grammar. The process of formally specifying the manners of such component collaborations in a domain-specific (business) language is called Grammar-oriented object design (GOOD) [8]. The result of this step can be represented either internally or externally to the software component (smaller scale) or application (larger scale). Automated adaptation of the flow of an application, which requires access to the current flow configuration of the component, will not be feasible if the manners are specified internally.

Externalization, on the other hand allows dynamic re-configuration and flow adaptation. Externalization involves taking the encapsulated or separated design decision about the dynamic aspect of application flow and makes it re-configurable by representing it as meta-data within the grammar of a domain-specific language. An Enterprise Component’s configurable profile can then execute that externalized specification of its manners. Alternatively, the mapping between the specification and realization of a component’s behavior can also be generated using approaches described by Shlaer and Mellor.

3. Components and Manners

Component-based Software Engineering (CBSE) implies application construction through the assembly of prefabricated, configurable, and independently evolving building blocks. Emerging software component models prescribe standards for the collaboration of independent components and are aimed at improved development productivity and greater resilience of software to changing requirements.

Current approaches to component-based software development seem to be inadequate for the creation of reusable and changeable software architectures. The apparent lack of design information in today's components is considered to be one of the most significant problems of software development based on components. This can be traced back to the lack of an abstract specification, which encapsulates each design decision type. Thus, changing the design decisions through maintenance tend to adversely impact the software architecture; increasingly its “architectural entropy” and rendering it more “brittle”: maintenance tends to degrade the structure of software, ultimately making maintenance more costly [9]. The more brittle an architecture is, the less it can accommodate changing requirements.

Many of the unsuccessful projects encountered in various industry sectors result from the inability to rapidly re-configure running code of production systems in the face of new requirements that literally emerge as the system is promoted into production. This inability is founded on a tacit assumption: that software components can be used “as-is.” Components are not to be used as software Legos™ [9].

Changing software at run-time can be prohibitively expensive, time-consuming and error-prone. In addition, every change that was not factored into the design infuses entropy into the software architecture. This makes the software more and more brittle over time, until new changes become prohibitively expensive, cause poor performance or are infeasible to implement. In this paper, the nature of changes in the process of building reusable component-based architectures for stand-alone applications and for product lines is examined.

Any object, or class, has identity, state, behavior [10] and in most cases, rules [29]. These statically defined classes can only be changed at compile time, thus making it difficult and time-consuming to make rapid alterations to their structure or function at run-time.

The process of adapting an object-oriented application to changes in requirements that share a common stable basis and yet differ in certain aspects has been called VOD. A traditional paradigm tends to concentrate on the common aspects that are not interesting from a perspective of software maintenance, which comprises up to 70% of the cost and effort in software development projects [49,50]. Instead, VOD focuses on identifying the changing aspects of a software system and separating them out from the less rapidly changing elements [3].

Manners. The notion of manners was introduced [3,8] to account for capturing a multi-dimensional concern [11] for large-grained software components; a concern that cuts across the four aspects of business events, context and state analysis, business rules, and business flow. Manners are the rules governing the behavior of a component within a given state and context.

A component subscribes to and responds to the triggering of business events. Upon invocation, a component instance does not “blindly” execute the method invoked. Rather than an unconditional execution of the request, the component analyzes its current business and technological Context and State. For example the component may assume the role of the invoker, as passed in by the session entity that has authenticated the role. If the State is a valid one and consistent within a given operational context, then the necessary business rules are (pre-) selected, analyzed and, once the rules are exercised, if the conditions hold true, actions are fired. These actions can be fine-grained methods or they may be a
business flow of a large-grained component that define the sequence and alternation of method calls between a group of collaborating classes or components that enact a business transaction. The identification and specification of these aspects of a Component have been described by Arsanjani in the Enterprise Component pattern [28].

4. Introducing Change: The Process of VOD

Change typically involves previous design decisions (based on requirements) and how they are altered or extended to take into account new or modified requirements or constraints. Properly designed software architecture provides/maintains stability in the face of change. The impact of change on architecture may be related to non-functional or functional aspects: making decisions on such concerns as resource balancing and allocation; constraint conflict resolution; as well as, business factors, skills and time-to-market concerns. The tenet proposed herein is that the basis for this stability and resilience in software architecture is based in the decision of “to which extent to externalize what.”

Thus the principles of VOD are employed to introduce placeholders into the software architecture where changes are anticipated as likely to occur. As shown in Figure 1, these changes transition progressively through the phases of identification (definition), separation (isolation), consolidation (encapsulation), informal externalization and managed externalization. Johnson, Roberts and Foote suggest isolation and reification of volatile code elements. Gamma et al. describe the foundation of design patterns to be based on variation-oriented design [21].

4.1 Changing Type and Attributes

In the context of business logic, a class is a conceptualized semantic entity in the business domain, having identity, state and behavior. In this paper, components are defined as deployable units of functionality that have contracts and manners. If the attributes of a class changes in type and number, due to requirements changes, as they typically do, especially within the beginning of the system life-cycle where more exploration and adaptation is necessary, the attributes are externalized and stored as Variable State [14] or Property Lists.

In order to facilitate on-demand alterations at run-time, adaptive object models (AOM) have been suggested in [12]. AOMs use meta-data to generically describe the data of a class as in a data dictionary using Property Lists and Type Objects. This approach focuses on representing an object’s state or attributes as meta-data as in Beck’s Variable State Pattern [13]. Identity is classified using Type Object, whereas its state or attributes are handle using Variable State. AOM’s have use meta-data to describe the state or attributes of an object but have not provided a means to externalize behavior nor rules. Grammar-oriented Object Design creates a highly re-configurable architectural style by externalizing the collaboration, business flow and business rules of a group of collaborating components as a domain-specific language.

Changing Component Manners. The external specification of a component is defined by its contracts [7]. Contracts pertain to pre-conditions and post-conditions on methods. A component’s service, translated internally to a method call (in an object-oriented scenario) requires a context. The context of a component’s method may include meta-data, as well as, the roles and access rights the component’s invoker is permitted to use. The EJB deployment descriptor is a good example of this notion. A component can dynamically change its behavior by altering the grammar that defines it manners. In this way, this paper adds to the literature of Active, Dynamic or Adaptive Object Models [6] limited by representing state or attributes as meta-data and proposes a new HRA architectural style, that represents collaborative behavior as a domain-specific business language describing the business flow of the system as a whole and the manners of components as a smaller part of the overall system.

5. Grammar-oriented Object Design: An Overview

Building Product Lines. Product line architectures [14,15,16] cover a family of applications often aligned with specific business lines in larger organizations; e.g., in a telecommunications domain, the business lines might be wireless, wireline, DSL, Cable, voice messaging, etc.

Traditionally, building product-line architectures based on component-based software engineering and
service-oriented architectures have been difficult tasks. Large organizations are finding it difficult to have redundant software products that service given products or business lines but differ by small sets of variations. Business application development has been no less challenging. Ever-changing requirements morph the intent, structure and function of the deployed application in new and often unanticipated ways. New or modified attributes, behaviors, business rules and business flow make changing such software architectures particularly difficult and error-prone within business-mandated shorter and shorter development cycles.

Variation-oriented design is used to address the configurability issue at a methodological level. This entails externalization and encapsulation of the more rapidly changing elements of a domain or architecture. At the same time, a “common-denominator” of more stable, more slowly changing aspects of software application architecture is retained. Software stability is thus retained.

Once identified through variation-oriented analysis and design, the variations in an application or family of applications, such as a Product Line, can be used to alter the configurable profile for new configurations of attributes, behavior, rules, or business flow. The configurable profile includes the domain-specific language, which the dependent software component can cache, interpret or execute based on the strategy employed. This allows the execution of its business flow as determined through grammar-oriented object design (GOOD).

Thus, based on the nature of the prospective changes, one may intend to add/modify attributes, behavior, business rules and business flows that were usually considered to be immutable aspects of the “model” in a standard model-view-control architecture. User interfaces tended to change, schemas tend to change and rules tend to be changing as new products and services are added or tempered for market response.

**Externalizing Domain Behavior.** Adaptive architectures and flexible software that adapt “just-in-time” are vital to meeting the multiple constraints often demanded by changing environmental and application requirements. Important constraints that need to be met include modulations in power and performance, variations in resource-availability (e.g., cost, space limitations, battery and memory capacity) and changes in application requirements (e.g., quality of service). For example, the explosive growth and expected pervasiveness of mobile, embedded and high-end systems that impose multiple constraints that are not completely known before execution or evolve constantly at run-time have created new challenges in both compiler and hardware technology. Instead of merely externalizing data or parameters, the very behavior of an application domain can be externalized. Once externalized and stored, the manipulation of behavior is akin to the manipulation of data; the same principles apply.

GOOD solves this problem by externalizing a component’s and application’s manners. This can be represented as a business domain-specific language that can be adaptively modified at run-time.

### 5.1 A Highly Re-configurable Architectural Style Implements a Model-driven Architecture

This approach helps realize the notion of a model-driven architecture [18] by representing that model as an executable business specification. The model representation includes a representation of the manners of the domain as a grammar in a domain-specific language. In this way, the HRA implements a model-driven architecture and significantly reduces the risk associated with software maintenance by recasting a portion of the more complex problem of software evolution into the well-known context of grammars and domain-specific languages that define application and component configurations for a given business domain.

Externalized configuration includes anticipated axes of variation that have been separated out from the application and rendered non-intrusively (re-)configurable rather than requiring intrusive and often widely impacting code changes (customization) that is more error-prone; leading to longer cycles of software development, testing and deployment.

The process includes identifying requirements, abstracting them into a domain-specific language that can be externalized as meta-data in a grammar; thus allowing code-generation and run-time re-configuration of large-grained Enterprise Components. This helps reduce a portion of software maintenance to software configuration. This highly configurable architectural style is demonstrated to help build reusable components that form the cornerstone of component-based software architectures for families of applications.

### 6. The Anatomy of GOOD

This paper expands on the extension of current methods such as the Unified Process of Software Development based on the work in [8]. The extensions comprise a set of activities and artifacts. One such a artifact is subsystem analysis in which the business domain is partitioned into cohesive business-process level subsystems with appropriate “manners” allocated to each subsystem using a use-case grammar. The focus of this paper is process by which externalization of component manners can be accomplished to yield highly reconfigurable architectures.

Especially noteworthy is the representation of a business domain-specific language in the form of use-case
grammars included in each use-case description. The following Use-Case Grammar describing Online Order Entry Component’s Manners might appear as the last entry in a typical use-case description or as a separate artifact:

<table>
<thead>
<tr>
<th>Online Purchase = { [Identification], Presentation, Selection, Purchase, [Identification], Confirmation, Order Fulfillment}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identification = (Challenge User with Login, Verify UserID and Password, Present Valid Options)</td>
</tr>
<tr>
<td>Presentation = (Display Product Catalog, Navigate Product Catalog)</td>
</tr>
<tr>
<td>Selection = (Browse Product Catalog, Select Product Item, Shopping Cart Operation, Selection)</td>
</tr>
<tr>
<td>Shopping Cart Operation = { (Add Item to Shopping Cart</td>
</tr>
<tr>
<td>Checkout = { (Verify Billing and Shipping Address</td>
</tr>
<tr>
<td>Purchase = (Review Order, Review Terms of Agreement, Acknowledge Terms of Agreement, Submit Order</td>
</tr>
<tr>
<td>Confirmation = (Send confirmation number to user, Generate Order)</td>
</tr>
<tr>
<td>Order Fulfillment = { List Orders, Pick and Ship Orders}</td>
</tr>
</tbody>
</table>

The use-case grammar is a new artifact, which combines the notion of a structured use-case with those of subsystem partitioning and domain-specific languages [24]. The externalization of manners, in this instance as a use-case grammar can be done in an XML file called a Configurable Profile (CP) as described in the EC pattern above. An EC reads in its CP and uses its mediator’s workflow manager to parse the DSL’s grammar and based on the actions inserted in the DSL to invoke methods on the component’s business objects.

Once a domain analysis [17] is conducted and business language analysis [18] is completed, the key abstractions of the domain are partitioned in terms of interacting subsystems that may eventually be realized as a set of collaborating software enterprise components. Arsanjani [28] gives a pattern for the composition of such components. Manners are assigned to each subsystem based on the business rules that govern its behavior [19]. Subsequent variation-oriented analysis is conducted [11] to: a) separate changing from non-changing aspects and features; b) verify what changes have occurred; handle changing aspects using patterns [20] [21]; c) partition the domain into subsystems and define manners for each subsystem and their interactions; and, d) use the three layers of interface, abstraction and concrete realization in the aggregate inheritance pattern.

There have been numerous examples of highly successful implementations of software based on domain-specific languages (DSL) [22,23,24]. A business language is an industry or business domain-specific language that characterizes the key manners or rules governing the behavior in the domain’s partitioned set of subsystems. Once such language that has been successfully employed is grammar-oriented object design (GOOD). The primary process steps of the methodology extensions in GOOD are: a) Identifying the business language for a given business domain, b) Partitioning the domain into subsystems based on subsystem analysis, c) Identifying variations within the subsystem manners and applying necessary patterns [21,22] through variation-oriented analysis and design and finally, d) Writing use-case grammars that define the manners for the subsystem and the context in which it will interact with other component interfaces, once deployed and executing the control flow in a component framework through pluggable micro-workflows that implement the manners.

### 6.2 Identify, Separate, Encapsulate and Externalize

The process of externalization of manners starts with the encapsulation of data or behavior, which are the variant aspects of the software system. These are often in a method within a class. Subsequently, the frequent need to alter the internal data name types or behaviors within methods suggests a separation of the class from the code fragment; first into a method of its own and later, as the need and frequency of change increases, to an often subordinate class of its own. Fowler [23] describes this as refactoring a method into an object. Reification and meta-modeling [25] exponents suggest the encapsulation and separation of the repeatedly altering mechanisms into a properties file. Arsanjani [26] suggests the externalization of business partner contracts in an XML-based markup language, which can be dynamically reflected and adapted, based on trading partner negotiations.

Parnas suggests that design decisions should be hidden from the external view of the component consumer. The internal view of the component builder and the internal/external view of the component configurator (formerly mis-represented as “assembler”) can be described by an abstract specification provided by its manners. Encapsulation may not be the solution to dynamically re-configurable systems; internal separation of the variations grouped under a common abstract variation point may often be necessary. Many patterns exhibit this behavior [21, 23].

### 6.3 Externalization: Transforming Behavior into Data

Behavior is specified as interfaces and realized as code within a class method. Using meta-modeling, this code is abstracted into a meta-language which defines the valid interactions and business flow between collaborating classes. Thus, the behavior has been converted into data
by taking it to a higher level of abstraction and representing it in a grammar of a business domain-specific language, that can be driven by a simpler parser than the one used for the program code as a whole.

This approach extends the state of the art from mere representation of meta-data for variable state attributes to separating an application in multiple dimensions:

**Dimension 1: Language**
- General-purpose programming language (e.g., Java, C#) for coding the fine-grained business rules and logic.
- Special-purpose, domain-specific language for coding the larger-grained application flows between and within large enterprise components.

**Dimension 2: Structure, Function and Intention**
- Partitioning and decomposing the structure into business process level functionality using domain decomposition resulting in subsystems
- Using subsystem analysis to decompose components into their constituents as describes by Enterprise Component pattern
- Partitioning the dynamic aspect of the application flow logic and separating and externalizing it from the rest of the component’s state and behavior.
- Partitioning based on business level goals and intentions

**Dimension 3: Stages in Component design, realization and usage**
- Design model: using manners to define an abstract specification for component and system behavior; gray-box
- Realization model: internal functioning and building of individual components; white box representation
- Usage or Assembly model: how components are wired together; contracts and interfaces; black box representation.

Each of these n-dimensions of separation of concerns are essential to creating a maintainable software architecture.

The definition of a component is a critical undertaking. Defining the right component boundaries is critical to initiating the re-configurable architectural style through component identification, conceptual separation, physical reification or encapsulation and externalization of a semi-formal specification for the flow logic within the application and within components to be captured in an externalized, configurable component profile. Thus, the system and their components would be “soft-wired” and allowed to evolve independent of their static and dynamic configurations.

**7. GOOD in Action**

**Building Enterprise Components for Product-Line Architecture.** A recurrent solution that was applied across many of the projects mentioned above was the Enterprise Component (EC) compound pattern. This design structure provided a uniform design mechanism and terminology that could be used across all development teams within minimal instruction. In addition, EC was used as a checklist to validate that all relevant aspects of a component-based design had been addressed. In this manner, multiple teams could work with the same basic design template and factor in the placeholders necessary for an evolving software component’s design needs.

Frequently, enterprises with multiple business lines (such as retail, wholesale, institutional or credit card, traveler’s checks, financial services, insurance, etc.) would find the need to fulfill many product lines through a configurable component-based approach that would address the needs of their families of applications. It was soon realized that the emphasis on the design of components (the parts) should be shifted to the emphasis on the configuration of components within a business context (the whole). This allowed the identification, conceptual separation, physical reification or encapsulation and externalization of a semi-formal specification for the flow logic within the application and within components to be captured in an externalized, configurable component profile. Thus, the system and their components would be “soft-wired” and allowed to evolve independent of their static and dynamic configurations.

**Figure 2: An Enterprise Component for an HRA**

Externalization is a design-time or refactoring–time activity whereby variations are encapsulated and a description of the axes of variation of a component are externalized into a Configurable profile (CP). The participants in this pattern are described in [27] and are summarized below:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Participant</th>
<th>Responsibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service Interface</td>
<td></td>
<td>All use-cases supported by this large-grained component will be provided as services on its interface. This will keep it in line with the business needs and architecture (terms and thinking) as well as provide a services based architecture without degenerating into an unmaintainable set of fine-grained, unintelligent services implemented as a plethora of distributed objects; difficult to maintain or anticipate side-effects.</td>
</tr>
<tr>
<td>Mediator (Manager)</td>
<td></td>
<td>The role of coordinator or controller of collaborations between medium- to small-grained components within the</td>
</tr>
</tbody>
</table>
subsystem or enterprise component boundaries. Serves as a central focal point of business flow and collaboration that can be modified without impacting each individual sub-component’s behavior: thus providing greater maintainability and extensibility in an agile (run-time, configurable) fashion. Disassociates the coupling between medium-grained components so they can be reused outside of the current enterprise component context; e.g., in other by other enterprise components implementing business subsystems.

Handles the use-cases provided by the component services interface and ensures (“trusted component”) that the required component manners are adhered to (manners being the super set of how a component should behave within certain circumstances; consisting of: business rules, events triggering actions, context (e.g., what role determines what rules apply what behavior can be executed, meta-data for run-time configurability).

**Workflow**

The Manager or mediator will have a way of implementing its workflow or business flow or collaboration within the confines of the component and with other collaborating components through the interfaces it requires. Note every component provides its service interfaces and yet requires other interfaces from its peer enterprise components (outside of itself) in order to enact a business process transaction. Thus the workflow is a macro-rule that can be implemented by a micro-workflow by loading a configurable profile (an externalized XML script that determines its valid set of actions (manners) using the language of the business domain; a business domain specific language.)

**Business Rules**

Every component such as Account Manager will have business rules unique to it, such as account rules, fraud protection rules, etc., which will be a combination of hard-coded, configurable, parameterized and edit rules.

**Factory**

This component is responsible for creating all dependent components within the enterprise component base don the enterprise component’s configurable profile. This profile defines the elements the enterprise component needs to carry out a business process...

**Business Components/Business Objects**

The medium or fine-grained objects and components that will be needed to implement the larger component’s functionality. These business objects / components may be reused across product lines and are context independent. They are governed by the overall business flow of the workflow and are constrained by their own Business Object Rules.

**Rule Objects**

These reified business rules constrain individual fine-grained or medium-grained business objects and components. They implement the rule object pattern [27]

**External Broker**

Responsible for interfacing with external systems through messaging, JDBC (database connectivity) or other protocols that may require specialized adaptors to be built or managed for the particular component.

**Message Driven**

The Enterprise Component is a message-aware large-grained component and can generate and receive asynchronous messages from a message bus.

**Configurable Profile**

Externalize the configurable aspects of the component into a profile. Example: externalize the business flow using a grammar in the profile (in XML) to be read in by the Workflow participant.

Thus, an enterprise component’s manners provide the opportunity for short turnaround re-configuration by targeting changes only to the externalized flow. The CP can then be changed (re-configured), by the Service Provider, Consumer or Broker at compile-time, startup (initialization) or runtime to provide dynamic customization of streamlined services.

**Figure 3: Structure of Enterprise Component with Externalized Manners in a Configurable Profile**

An example implementation of an Account Manager component might look like this:

**Figure 4: Account Manager uses Enterprise Component Pattern**

Its workflow can externalize its component’s manners in a Configurable Profile containing the following grammar:
Software components, especially large-grained ones are costlier to produce but more effective and valuable to an organization to own and apply. The total cost of ownership of the component is decreased through the ability to make changes to its functionality by reading in its description and re-configuring it, rather than making intrusive customizations through manipulation of source code at compile-time or code generation time.

It is useful to distinguish between compile-time and code generation-time because there are tools and approaches that use a transformational approach (automated code generation) rather than a refinement approach where the design is successively (manually) refined and ultimately coded [9,27]. It is also convenient to categorize roles and processes within DRS: There are three key roles of component-based systems within a highly re-configurable architectural style: Service Provider, Broker and Consumer.

There are three key processes for Highly Re-configurable Systems (HRS): configuration, customization and adaptation. Configuration and adaptation are differentiated by the fact that the former is a manual process done while the program may be in production and thus has to be reloaded. The latter signifies an automated process of dynamically reflecting upon a component’s manners and altering its behavior at run-time. Customization implies the intrusive changing of hard-coded functionality, requiring longer cycle times for bringing the evolved software back into production.

Agility can be described as a key trait of these systems. The Web Services and Component-based worlds or software engineering will find a common meeting ground within the context of Dynamic or Highly Re-configurable Architectures (HRA).

One of the key drivers of importance is to introduce the business goals into the forefront of the technology-dominated process by enabling the description and definition of a valuable software configurations or assembly of components and services they provide within a re-configurable architectural style.

Thus the configuration of business components is driven by a combination of the business driven goals that guide the selection and configuration of business components on a common infrastructure of technical components that satisfy non-functional requirements.

Thus, externalization is an extension of the notion of separation of concerns by not only partitioning the concerns but also enabling their physical separation to enable dynamic reconfiguration of static and dynamic aspects of a component based architecture driving an application.

We externalize variations to be able to re-configure them faster. To externalize something that can easily modify its axes of variation must be identified, encapsulated and partitioned. Placeholders are created for variation points and default implementations or mechanisms are plugged in to provide an initial level of acceptable functionality.

8. Conclusion

This paper addresses the problem of software maintenance of component-based systems by identifying, encapsulating and externalizing the axes of variation around design decisions. This includes the process of identifying requirements for the manners of a system and its components, formalizing and abstracting them into a domain-specific language’s grammar. Subsequently, externalizing this grammar as meta-data within the component’s configurable profile; allowing code-generation and run-time re-configuration of large-grained Enterprise Components. This process is called Grammar-Oriented Object Design. The resulting grammar is externalized, often into a configurable profile that is used within large-grained enterprise business components to interpret or execute their behavior. Variations are captured as pluggable, reconfigurations of an underlying common domain-specific language that can be used to generate, configure or reconfigure software systems. In this manner, the behavior, rules or business flow of a grammar-oriented architecture can be rapidly reconfigured to meet new requirements at run-time or at compile-time. This helps reduce a significant portion of software maintenance to software configuration. Thus, to change the application, you change its externalized flow grammar.

This approach enables a highly re-configurable architectural style to help build and maintain reusable components that are responsive and resilient to changing requirements; forming the cornerstone of component-based software architectures for product lines and families of applications by allowing non-intrusive, dynamic adaptation and variation of application flow.

9. References


