Patterns for Implementing Grammar-oriented Object Design

Ali Arsanjani, Senior Consulting I/T Architect
IBM National eAD Center of Competency,
Center for Architectural Excellence, Maharishi University of Management

Introduction

In designing and implementing enterprise systems and product-line architectures using Grammar-oriented Object Design (GOOD), as in any other method of software architecture design, the Architect needs to make certain key design decisions.

This paper outlines the design decision points, the tradeoffs and solution spectrum that should be taken into consideration when designing software architecture infrastructure and tools to support Grammar-oriented Object Design.

Prologue

This pattern language has a broad context spanning a spectrum of possible situations. You can Pay As You Go and each context will have its corresponding specific problem associated with it:

1. You want to be able to build rapid prototypes but you want extensible ones that initially cover an end-to-end architecture to which functionality can be pluggably added.
2. You want to design a product line or component-based architecture for an enterprise that crosses business lines and product line boundaries.
3. You want to be able to build a Highly Re-configurable Architecture that does not require intrusive changes to accommodate new functionality in business flow.
4. You want the business to drive the technology and allow the business objectives to drive the decomposition of the application architecture into enterprise components, the collaborations of which are mediated, orchestrated through an externalized Configurable Flow.
5. You want to harmonize business processes across multiple organizational boundaries; sometimes across international ones. You need to concisely represent the business process flow in order to identify commonality and variability for business process consolidation, workload reduction, quality and possible re-engineering. But reams of voluminous documentation of business processes, although perhaps a start is not conductive to meta-modeling. A concise representation of the business flow is needed. And yet

6. You want a dynamic representation of the business process flow. A static set of diagrams is a start but you want to be able to trace through the execution of the business process; add components and apply what-if scenarios.
7. You want a Flow Language and Architectural Description Language that accommodates the needs of your particular industry.
8. You want to program at a high level in your own business language. This programming would define the high level structures and flow that the lower level technical architecture could implements. The business modelers and owner representatives, the subject matter experts define the enterprise components and their manners, the technical architects and developers build the system to that more formal yet executable, practical specification.

Corresponding to each of these situations, we have some specific problems that are often encountered, namely:

1. How do you create an extensible prototype without coupling the solution's View and Model to the Controller, yet allowing Pluggable expansion of the project?
2. How do you start your design of product line architectures?
3. How do you avoid making intrusive changes to a software architecture even when there is a lot of changes you anticipate down the road, as the project unfolds?
4. How can you ensure that the business is steering the project in terms of goals and traceable support for business processes?
5. How should you start to reconcile business process differences across product lines or across multiple organizational boundaries; sometimes across international ones?
6. How can you avoid static and "dead" documentation? Can we somehow provide dynamic, animated, live documentation that acts as a program (self documenting program documents the business processes)?
7. How can you define a generic flow or ADL for a given industry segment without coding in a third-generation language and yet have executable capabilities?

Thus, the application of Grammar-oriented Object Design (GOOD) and it's implementation (discussed as a pattern language in this paper) are applicable in a broad spectrum of situations outlined above. They form the context for the set of patterns outlined below.

Grammar-oriented Object Design (GOOD)

Context The context of this pattern runs along a spectrum. Depending on where you are in the life-cycle and what you want to accomplish, you can leverage Grammar-oriented Object Design to a degree that is applicable to your project. We outline several potential milestones along that spectrum below and include this as the context of the pattern.

9.

Problem You want to program at a high level in your own business language.

How can you define an executable formal, practical high level specification for the structure and function (flow) of your business while maintaining alignment with business goals?

Corresponding to the context above, here are problem statements that address each of the context elements outlined along the spectrum above. If the reader finds this too detailed at first reading, they encouraged to take a look at the more general encompassing problem statement above.

This programming would define the high level structures and flow that the lower level technical architecture could implements. The business modelers and owner representatives, the subject matter experts define the enterprise components and their manners, the technical architects and developers build the system to that more formal yet executable, practical specification.

Forces Current general –purpose programming languages allow you to solve general technology problems and are often very low level from the business modelers perspective and out of reach. There is a Huge Gap (See Bridge the Gap) between business models and software models that need to be bridged. So you need a domain- specific language to represent your business flow.

You need to document the flow, but reams of paper documentation are difficult to keep in synch with reality, are difficult to convey to other parties and very difficult to check. You need a concise representation.

Business process models are essentially static. You need a dynamic representation that simulates the business in order to identify and communicate problem areas .

Solution Define the structure of the business in terms of its constituent functional areas and subsystems (that typically encapsulate business process level granularity). Define the business flow through a business domain-specific language that provides a formal yet executable specification for the manners of the components within the system.

Resulting Context You can adaptively define new manners by adding new production rules to your semi-formal executable specification. You now need a business compiler to drive the business flow and structure your system.

Implementation Considerations You need the tools to do this by defining a grammar that describes a business domain-specific language. You need a Business Compiler to execute this skeleton of your business, your business blueprint, to which you can additively, non-intrusively add and rapidly re-configure software components that have the potential to be used across the business lines in your enterprise.

Known Uses Systems and projects developed using GOOD since 1989: computer-based training, accounting, back office customer care and sales lead, GUI manipulation, telecommunication switching and provisioning, healthcare, taxation, banking, insurance, mortgage systems.

Business Architecture

Context Businesses do not seem to spend resources focusing on creating precise business models or business architectures [2]. Thus, business requirements carry this imprecision or equivocation forward. They may provide details for a few specific business rules that are know.. They embark upon I/T projects that are aimed at supporting the business and providing business value, without a clear mapping between a business architecture (mostly non-existent) and a software architecture. The software architecture is usually pieced together through the interpretation of vague business requirements documents by technical staff such as architects and developers. This usually leaves a huge gap between the business expectations and their technical interpretation.

The business goals are not tied to the services and features that are required of the software architecture and the applications running on it.

Problem Business models alone are insufficient to help drive software architectures that are critical to supporting them.

Forces Not tying technological decisions to business goals and objectives creates a mismatch and gap between business and I/T within an organization.

Bridge the Gap

Solution Therefore, let the business analysts build a business architecture before modeling the software architecture, helping to bridge the gap between business expectations and software design of applications. They can use some precision and more rigor inspired by the software or business engineering community to create an architectural blueprint of their business, so they can understand, fine-tune and enhance it in the face of rapid changes in the market, legislation, customer trends and i/t support infrastructure.

In order to create a model of the business architecture before modeling the software architecture you may need to define and design the cross-business line enterprise wide product line architecture, with the components and services needed to fulfill business requirements.

Resulting Context You have a Goal Model, Conceptual Model, Process Model and Rule Model [2] to design and fine tune your business. These models define your business architecture and serve as a blueprint to understand the current business functioning, provide context for enhancing current processes and serving as a frame of reference for re-engineering the business.

Implementation Considerations The Process Model will be implemented using a Business Compiler. The other three models will be implemented in the component-based code that will be written as actions that the Business Compiler will execute. For example, the Conceptual Model will outline the components and their properties and methods; the rule model will outline the set of Rule Objects needed to implement the business process model. The Goal model will ensure that all services required by the business are defined on the component interfaces and implemented.

Known Uses Arsanjani Extensions, Eriksson-Penker Extensions to UML, RUP Profile for Business Modeling.

Bridge the Gap

Context The gap between business and I/T comes from different cultures, tools, modeling techniques, ownership models, driven by different (often diametrically opposed) agendas and objectives.

Problem There is often a huge gap between business and information technology (I/T) causing conflict and contention, detracting from the value I/T can add to the business; disallowing the business to be properly supported by I/T aggravating business problems, creating customer satisfaction problems, market response time lags, failure to meet business objectives in a timely fashion.

Forces Businesses have huge differences in strategy, tactics, scope and objectives with I/T. Their success factors are different and often conflict.

Should the business drive I/T or should I/T drive the business (e.g., in cases where the “software is the business”, on the WWW)?

Solution Defining the mapping from Business Architecture to software architecture is key. Therefore, create a Business Architecture and seamlessly map that onto a Component-based Configurable Software Architecture.

Solution Detail Model the business; create a Business Architecture by using methodology extensions that support component-based development. Map the business seamlessly onto software elements:

- Conduct Focused Domain Modeling to identify Functional Areas, Business Processes
- Identify business processes in the domain, map these to subsystems
- High level use-case map to business processes
- Refine them further into lower levels use cases

- Put the groups of cohesively related use-cases on a façade for an Enterprise Component
- Subsystems map to Enterprise Components
- The Workflow or Business Flow defines the collaboration of larger grained Enterprise Components

Resulting Context
You have software architecture that is driven and directed by business objectives, guaranteed to dovetail with and support the business.

Implementation Considerations
A software architect needs to work with the business modeler/architect(s) to produce this mapping. This is a joint business-I/T effort not a mandate coming from management to I/T nor is it a bottom-up I/T driven design decision that is bubbled up to become a business level constraint.

Business Compiler

Context
You have a Business Architecture in place and want to allow the business modelers to define and represent their own business process flows, and yet you need a degree of precision to create prototypes, production systems and software in general that will support the business being modeled.

Modelers have a strong prior knowledge or their business domain; a subject matter expertise that is invaluable to the successful execution of a software development project that is designed to support the business.

Problem
How can we create an intuitive and yet powerful executable representation of the manners contained within a business domain? Should we use a third-generation language (like Java or SmallTalk) to model a business? How can we empower business modelers in a non-dangerous way to express their subject matter expertise?

Forces
Business requirements are too vague; software needs more precision.
Business people cannot write software; software engineers very often have little expertise in the business but know the systems [presumably] well.

Programming languages are ideal for I/T constructs; but are too low-level for business modeling and flow definition. On the other hand, domain-specific languages are too restrictive to be used for low level coding.

Solution
Implement a portion of the the Business Architecture using a Business Compiler. Allow the business modelers to create their own domain-specific language that will describe the business flow and drive their business. Use a Business Domain Specific Language to represent the business process flow within the business architecture.

Solution Detail
Represent the processing steps of a business component or the business flow for the entire business (two levels of detail) as a domain-specific language. This is called Grammar-oriented Object Design (GOOD)[]. The business is modeled using object-oriented and component-based notions and yet the flow is governed by a grammar that is written as a domain-specific language by the business modelers with the help of the Chief Architect.

A set of Pluggable Actions can be added at run-time on-demand to accommodate new or changed functionality in various B2C, B2B and P2P kinds of scenarios.

Resulting Context
You can now prototype your architecture end-to-end by flow. You can then Pluggable add software functionality using traditional methods or components to provide support for an increasing number of use-cases based on a release plan for an iterative, incremental development approach.

You now have a highly configurable controller that can share views and models.

Implementation
The Chief Architect helps and facilitates the Business Modelers to enunciate their

Considerations	Business domain-specific language and represents that as a Configurable Workflow /Profile that will drive Enterprise Components to participate in business processes.

GOOD Scanner

Context	You clearly need a scanner to accept and tokenize input. In this case, input is not a regular input stream but may be the interpretation of messages, events as input tokens. You can embed quite a lot of knowledge about the application domain in the scanner and thus ease the burden of the parser.
Problem	How complex should the scanner be?
Forces	But the more logic you put in the scanner, the more complex and inflexible it gets.
Solution	You have a spectrum of solutions: 1. The scanner does basic lexical analysis and returns simple token and token types 2. The scanner knows more about the application domain and groups a set of tokens (e.g., fields) into a context and passes this context to the parser as a grouped token type for faster/easier processing on behalf of the parser.
Resulting Context	You have a scanner that is working in concert with your Parser. You can identify types of terminal symbols that occur in the domain including strings, events, selected objects.
Implementation Considerations	You can use a scanner generator or hand code a scanner for your domain-specific language.
Known Uses	DSL's often have their own scanner [][][].

Parser Design

Context	You can design a parser that is LL(1), LALR or just plain event driven.
Problem	What kind of parser should you use or design to accommodate a business domain-specific language?
Forces	The decisions on which approach to take reply more heavily on the application domain: a servlets-based application, EJB-based, message-based or merely back-end business processing scenarios are primarily considered.
Solution	Design a parser based on a combination of the target business and the target technical infrastructure.
Resulting Context	You now have a generic parser that can be used in all similar types of applications within the same domain, and possibly within similar domains; i.e., a parser servicing a family of applications and potentially multiple domains if the domains exhibit enough fundamental similarity. All that needs ot be changed is the grammar for each domain and you can still use the same parser to drive the enterprise business components for that domain.
Implementation Considerations	Each new domain will need a new grammar. Each new domain may potentially need a new scanner with greater or lesser knowledge about the domain of discussion. The actions that the parser will invoke may indeed start as intended to solely be confined to the domain. But the classes or modules that provide business logic (model) or controller or view capabilities may be resued across multiple applications because the way they were developed was fundamentally decoupled from the controlling element in the domain.
Known Uses	

Let the Parser Do the Driving: GOOD Parser Implementation

Context You do not want to write the processing of the business logic in an application domain. That is inside each of the components. You want to be able to orchestrate the flow between large grained components in a pluggably changeable fashion; by declaratively representing the manners of components in the domain.

Problem You want to separate the execution of the business flow, model and process from the business logic that occurs within each step of the business processes. You want to define overall flow, not intricate detailed logic.

Forces You can code, but you then have to maintain the code and this may not give you the features you need. So you do not want to embark on an all out development effort; rather, you want a fast, simple, end-to-end skeleton of the process. You then want to plug in actions in the form of components providing services. You want to declar the flow, not code it. You want to code the actions, not declare them.

Solution Use a parser to drive the flow, use actions in the grammar to handle calls or message sends to objects or components that provide the services you need. You can either implement the services yourself or buy them or reuse them.

Resulting Context You now can Pluggable add actions where you want ; but have the system working end-to-end. When you want to make changes to the flow, you change the grammar often by adding production rules. You add functionality by writing components and plugging them into the grammar using actions.

Implementation Considerations

Known Uses Many systems written using GOOD. Other domain-specific languages.

Pluggable Action

Context You need to be able to invoke actions that will provide Views, and persist Models in your software architecture.

Problem How do you design extensible software functionality on an ongoing basis?

Forces You have a spectrum of solutions: at the one end your parser “knows” about the code you will call when it encounters each action. This binds the parser to the implementation.

You can implement the actions as a set of Commands [], resulting in a large number of small-grained objects, each which handles a small portion of the business logic and functionality you want added or functioning within your system.

Should you implement the actions using reflection; on the other hand this increased flexibility may add overhead to your solution.

Solution Allow the run-time and compile-time adaptation of components to be added to provide new or modified functionality. Invoke the action from the Parser.

Solution Detail Parser Knows Actions. Import the class package of your service implementation, instantiate and invoke the methods statically at compile time.
Parser Sends Commands. Use an Abstract Factory in the Parser to create appropriate commands that are invoked when the Parser encounters an action.
Parser Reflects Actions. Specify a String representing the package and class name of the component that is the recipient of the message. Code the method as a String. Use reflection to actuate the invocation at run-time.

Resulting Context You have a system that can execute actions in a Pluggable and modular fashion allowing the dynamic adaptation of target action implementations at runtime to accommodate rapidly changing business requirements or negocoations in the context of business-to-business interactions over, for example, trading partner agreements tpML or ebXML.

Implementation Considerations	Using Web Services, specifically, the SOAP to invoke the Actions possibly through a middleware that uses message-oriented middle would be a robust and scalable solution approach.
Known Uses	Enterprise Application Integration approaches, Web Services, Externalization and Adaptation.

References

[1] Arsanjani, A., Alpigini., J., “Using Grammar-oriented Object Design to Seamlessly Map Business Models to Software Architectures”, Proceedings of the IASTED 2001 conference on Modeling and Simulation, Pittsburgh, PA, 2001.

[2] Penker, Eriksson, Business Modeling with UML: Business Patterns at Work, Addison-Wesley 2000.

[3] Arsanjani, A., Grammar-oriented Object Design: Creating adaptive collaborations and dynamic configurations with self-describing components and services, *Proceedings of TOOLS 2001,* IEEE Computer Society Press.

[4] Arsanjani, A., Zedan, H., Externalizing Manners to Achieve a Highly re-Configurable Architectural Style, Proceedings of the International Conference on Software Maintenance, IEEE Press 2002.