Nesting Automated Design Modules In An Interconnected Framework

Jared Matthew Young
Brigham Young University - Provo

NESTING AUTOMATED DESIGN MODULES IN AN INTERCONNECTED FRAMEWORK

By

Jared M. Young

A thesis submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Mechanical Engineering

Brigham Young University

August 2005

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

Jared M. Young

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

____________________	____________________________________
Date	Jordan J. Cox, Committee Chair
____________________	____________________________________
Date	W. Jerry Bowman, Committee Member
____________________	____________________________________
Date	Carl D. Sorensen, Committee Member

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the dissertation of Jared M. Young in its final form and have found that (1) its format, citations, a bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

__________________	______________________________
Date	Jordan J. Cox Chair, Graduate Committee

Accepted for the Department

__________________	______________________________
Date	Matthew R. Jones Graduate Coordinator

Accepted for the College

__________________	______________________________
Date	Alan R. Parkinson, Dean, Ira A Fulton College of Engineering and Technology

ABSTRACT

NESTING PDGs IN AN INTERCONNECTED FRAMEWORK

Jared M. Young

Department of Mechanical Engineering

Master of Science

This thesis seeks to extend the PDG methodology by developing a generalized formal method for nesting PDGs in an interconnected system. A procedure for decomposing an individual PDG into reusable modules will be defined and a software architecture will be presented which takes advantage of these reusable modules.

This method breaks the PDG structure into discrete elements known as PDG objects, PDG modules and PDG services. Each of these elements forms a distinct unit of reuse and each can be seen as a “little” PDG.

Two different industrial implementations of this method are presented. These examples show that it is possible to share PDG services amongst multiple PDGs and provide a mechanism to create a PDG for a complicated system.

ACKNOWLEDGEMENTS

This work would not have been possible without the contributions of faculty, students, my loving family, and my Heavenly Father. I am extremely grateful for this help and the opportunity to study at this wonderful university.

Special thanks to Greg Roach for his willingness to listen and his mentorship.

I appreciate the patience of my loving wife Kim and her constant encouragement in the completion of this work. I couldn’t have accomplished anything without this help and the prayers of my children.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1 INTRODUCTION

This thesis seeks to develop a generalized formal method for nesting automated design modules in an interconnected system. A procedure for decomposing automated design modules into reusable elements will be defined and a software architecture will be presented which takes advantage of these reusable elements.

The proposed method defines a process for decomposing a product into reusable design modules. This is done by recursively dividing the product into smaller and smaller chunks and then regrouping these chunks into reusable structures. These reusable structures use ideas from object oriented design and are realized as web services. These techniques require an extensive knowledge of object oriented design, web programming and service oriented architectures.

In this era of “faster, cheaper, better”, companies are focusing on improving the product development process. The global market is becoming more and more competitive and customer requirements are becoming more and more specific. Production in many industries is changing from make-to-store and make-to-market to make-to-order. As consumers become more educated to market options, the demand for custom products increases. This increase in the demand for custom products is forcing market shifts towards mass customization [1].

The broad, visionary concept of mass customization was first coined by Davis [2] and promotes mass customization as the ability to provide individually designed products and services to every customer through high process agility, flexibility and integration [3]. Mass customization systems may thus reach customers as in the mass market economy but treat them individually as in the pre-industrial economies [2].

New flexible design, manufacturing and information technologies give companies the potential to deliver higher variety at lower cost. Experience has shown, however, that companies are limited not by the technology but by organizational barriers and the resistance to changing the way things are done. It appears that the largest hurdle lies not in advancing the technology, but in advancing the product development process itself. Companies are still working with the same basic philosophy for product development that they were using in the 1960's regardless of the fact that technologies and tools are so dramatically different [4]. These product development capabilities are the basis for successful competition. Successful product development, in our globally linked and increasingly competitive economy, requires fundamentally improved approaches to organizing the product development process. In order to develop products efficiently, companies must take advantage of their existing knowledge and work coming from previous engineering, software code, analyses and parts instead of "reinventing the wheel" each time.

Roach presents a new "object oriented" product development process called a Product Design Generator (PDG) [1]. The PDG forms a flexible, new methodology for product development where product variation is built into the product development process and is achieved through scalable and in some instances modular parametric models [5]. The PDG methodology provides a systematic approach to the creation of automated design modules. The method allows for the development of reusable design and production modules for specific product families and is able to take full advantage of the latest CAD/CAM/CAE tools and capabilities. The systematic approach used in the PDG will be extended to provide improved reuse and the ability to connect PDG modules together.

There are, however, some current limitations to the PDG methodology. For example, there are no current strategies that allow a PDG to be used as a component in another, higher level, PDG. The use of a PDG within a higher level system PDG is referred to as nesting the PDG. The ability to nest a turbine disk PDG, for example, inside a gas-turbine engine PDG would be invaluable. This would allow the system designer to design the turbine disk in the context of the entire engine system and allow the designer to leverage the knowledge captured in the entire system. Secondly, up to this point each PDG has been designed as an integral application. The implemented PDG can not easily be broken down into components and these components can not be executed independently without significant modification. Because nearly all PDGs share some of the same functions, these functions have been duplicated in each of the existing PDGs. If the individual PDGs were to be decomposed into reusable modules, these functions could be shared by many PDGs. This would eliminate duplication and reduce the time required to construct a new PDG. A modular approach would not only allow for reuse of PDG components but would allow these components to be used in different systems and contexts. Lastly, there is no standard software architecture or framework for rapid PDG development and deployment which could take advantage of these reusable modules.

A generalized formal method for nesting PDGs in an interconnected system will be developed. In order to develop this method, it will also be necessary to develop a procedure for decomposing an individual PDG into reusable modules. These methods must have the same attributes enumerated by Roach in [1] namely they must (1) be engineered for reuse, (2) be flexible and (3) optimize and integrate the latest tools. These methods will then be applied in the design of an Atmospheric Resistor and a turbofan engine PDG.

CHAPTER 2 LITERATURE REVIEW

As previously stated, the aim of this thesis is to extend the Product Design Generator methodology by devising a generalized formal method for nesting PDGs in an interconnected system. First, the idea of mass customization is presented. Second, a review of the PDG methodology is presented. Third, a review of some of the latest web-enabled product development tools are reviewed for comparison. Next, a review is made of the commercial integration tools for product development. Finally, a review of distributed agent-based systems is made because of the benefits of autonomous agents and their distributed architecture.

2.1 Mass Customization

Mass Customization (MC) can be defined either broadly or narrowly. The broad, visionary concept was first coined by Davis and promotes MC as the ability to provide individually designed products and services to every customer through high process agility, flexibility and integration. MC systems may thus reach customers as in the mass market economy but treat them individually as in the pre-industrial economies [2]. This is a shift for manufacturing industries from the Fordist paradigm of production, in which economies of scale are being replaced by economies of scope, competition through price by competition through innovation, standardization and hardware automation by more programmable and flexible technologies.

New flexible manufacturing and information technologies enable production systems to deliver higher variety at lower cost. There is also an increasing demand for product variety and customization, even segmented markets are now too broad as they no longer permit developing niche strategies.

The idea of customizing products on a large scale with prices comparable to standard products has been around for some time. But due to insufficient technological infrastructure, implementations of the process were largely unworkable until relatively recently. With the overwhelming rise and utilization of the internet as an e-commerce and information transportation mechanism, a viable infrastructure is now in place [7]. These new internet technologies are seen as the main enabler to mass customization as they allow for rapid gathering and dissemination of information. Information can be regarded as the most important factor for the implementation of MC. Mass customization is successful only when it can cover this need for information and communication both purposely and efficiently. A distinctive feature of new internet technologies is that they enable direct communications between customers and suppliers [25]. These customers may be within the organization itself or external to the organization.

DaSilveria et al. identify an opportunity for the design of an effectively decentralized, service based, architecture for MC systems, although they do not express it in exactly those terms [7].

2.2 Product Design Generator Methodology

The goal of the product design generator methodology is to create a new detailed development process for mass customization. This new process must provide a more complete definition of the product to include process information and knowledge and alleviate the rework in the product development process caused by design iterations upstream. It must also allow for iterative design activities while providing single pass generation of the product artifacts. The process should be scalable, reusable, consistent and able to take advantage of the capabilities of new CAD/CAE/CAM and IT technologies.

The PDG developed by Roach provides a method for product development where product variation is built into the product development process. The PDG is a computer-based tool that is used to automatically create all of the design artifacts and supporting information necessary for the design of a product that is customized to meet the needs of a specific customer [1]. It is an automated process that transforms a set of customer specifications to product deliverables. The customer specifications are transformed to product deliverables through a set of intermediate transformations or mappings. These intermedi-

$$\begin{tikzcd}
& B & T \arrow[l, bend right] & \\
C \arrow[r, "\mathfrak{D}"'] & M \arrow[u, "\mathfrak{P}"] \arrow[r, "\mathfrak{G}"'] & A \arrow[u, "\mathfrak{I}"] \arrow[r, "\mathfrak{S}"'] \arrow[d, "\mathfrak{E}"'] & U \arrow[llu, bend right] \arrow[lld, bend left] \\
& K \arrow[u, "\mathfrak{R}"] & V \arrow[l, bend left] &
\end{tikzcd}$$

Figure 2.1 Schematic Representation of the PDG

ate transformations are made up of behavioral predictions, company rules and best practices, the generation of design artifacts, vaulting strategies, testing procedures and design artifact delivery procedures [5].

The transformation of customer requirements into detailed designs is a complex process and can be represented by the following equation:

$$F(\Phi) = \Omega \tag{2.1}$$

where Φ represents the design requirements, F is the design transformation, and Ω is the set of process outputs. Thus, in order to deal with the great complexity inherent in the design process, F, Φ, Ω are decomposed into subsets and mappings (see Figure 2.1). Φ is decomposed into two subsets, customer requirements, ***C***, and company conventions and rules, ***K***. Ω is decomposed into five subsets, product behavior metrics, ***B***, product artifacts, ***A***, product deliverables, ***U***, test and validation metrics, ***T***, vaulted artifacts, ***V***, and master parameter list, ***M***. The dependencies between the different sets are represented by a series of intermediate transformations or mappings. The design map, **D**, takes the set ***C*** and transforms it into a subset of ***M***. The rules map, **R**, transforms the set ***K*** to a subset of the set ***M***. The predictive map, **P**, represents a set of predictive models transforming a subset of the set ***M*** to the set ***B***. The artifact instantiation map, **G**, is a set of parametric design artifacts that transform a subset of the set ***M*** to the set ***A***. The test and validation map, *I*, is the collection of test procedures that transforms a subset of the set ***A*** to the set ***T***. The archiving map, **E**, represents the procedure for archiving and vaulting product artifacts which transforms a subset of the set ***A*** to the set ***V***. The product delivery and support map, **S**, transforms a subset of the set ***A*** to the set the set of final deliverables, ***U***. [1]

2.2.1 PDG Construction

The PDG is constructed by (1) selecting the product concept and embodiment, (2) developing the product generation schematic (PGS), and (3) constructing the reusable intermediate functions and integrating them into the automated PDG application [1].

The first step in creating a PDG is to identify the solution context. Many solutions exist, for example, to reduce the noise produced by venting a gas to the atmosphere. Silencers, mufflers, and atmospheric resistors all meet this criteria. The creation of a PDG, therefore, would begin by selecting one of these concepts. In this case, an atmospheric resistor may be chosen to solve this noise problem and an atmospheric resistor PDG created.

The second step in the construction of a PDG is the development of the product generation schematic (PGS). The PGS is a visual representation of the overall transformation function and serves as an aide in planning PDG construction. In the process of developing the PGS, the transformation function is defined, the membership of the various sets are enumerated, and the actual transformations between the sets are defined. The PGS is constructed in four steps. First, the best-practice process for designing the product is decomposed into the various sets. Next, parametric models are identified to serve as intermediate transformations between the sets. With these parametric models now identified, the parametric models are planned and designed. Next, the governing parameters for all of the intermediate transformations are gathered and reconciled into the master parameter list. The final step is to formalize the design process sequencing by creating a storyboard.

The last step in PDG construction is the construction of the reusable transformations and integrating them into an application. First, the intermediate transformations are implemented as executable models in the specific tools used to produce the various design artifacts. Second, the storyboard and executable models are integrated into an automated application. [5]

The PDG provides controlled variation of possible designs and allows for the rapid creation of custom designs. This significant reduction in design cycle time has been shown in at least two industrial applications. Because variant designs may be produced so rapidly, the cost of developing custom designs is significantly reduced. The PDG has been shown to be flexible enough to utilize the capabilities of the latest CAx tools and has proven to be repeatable and consistent [1]. It is also interesting to note that the development of a PDG often leads to a reorganization of the product development process itself. A more detailed account of the PDG methodology can be found in [1].

2.3 Web-Enabled Product Development

Web technology is playing an increasingly important role in product development and is rapidly changing the way engineering is done. Implementation of product design systems on the internet allows designers to gain worldwide access to valuable design information and design algorithms that they can easily incorporate into their day-to-day design activities. These web-based systems allow engineers to control complex engineering programs through a simple and familiar interface. Over the last several years, more and more efforts have been put into web-based product development applications as many industries have distributed their product development to geographically dispersed locations.

Siddique and others developed a web based template used in creating variants of a family of coffee makers [14]. This work is based on the Product Family Architecture and is limited to the automatic creation of CAD models using ProEngineer. He extended the work in [15] to a product family FEA module to analyze components of lawn trimmers and edgers. Ninan and Siddique also created a CAD/FEA template for web-based bicycle design using Active Server Pages (ASP) technology. This example uses Pro/E to generate the solid model and ANSYS as the FEA software. Flores et al [18] developed an automated web application for the creation of a sheave used in an elevator assembly. The architecture was similar to that of the coffee maker template but was complicated by adding an optimization loop to the application using the iSight optimization package. The main focus in this work is the development of a "CAD Services" specification which would integrate CAx applications using a component architecture. This specification would then be used to eliminate some of the limits found in the existing CAx application programming interface (API). Wong [13], developed an automated system for designing and optimizing industrial silencers. The silencer application uses Pro/Toolkit and an optimizer to produce variant silencer designs. One of the most complete of the examples found in the literature is a web-based support system for gear design [8]. This application allows the user to design different gear types and configure them into a simple gearbox assembly. The application then selects materials and bearing information from a database, runs a finite element model and specifies manufacturing process information. A prototype application architecture is proposed by Karne for the future development of manufacturing tools [11]. In this case, manufacturing objects are modeled with object data models and are stored in an object-oriented database management system. Tumkor [35], presents a web-based design catalog for shaft and bearings, which provides some information to the remote designer and aids in designing a shaft and in selecting the rolling element bearing simply and correctly. The user can download the CAD or FEM models to carry out further investigations. Wang et al. [39], presents a web-based tool for custom cell phone design where users can select a phone style and customize its appearance by interacting with a Java applet which manipulates a VRML model of the mobile phone. Mulberger et al [52] describe interactive web-based platform customization as an extension of product family design. In their work, a web-based framework is presented and applied to the design of new refiner plates used in pulp and paper processing. They then add optimization to the framework and create a web-based application for general aviation aircraft design. This framework, however, is not scalable nor is it extensible.

It is apparent in the literature that web technologies have successfully been applied to specific product development applications. While these may appear outwardly to be similar to a completed PDG implementation, it is clear from this survey that the above applications lack the formal framework of the PDG and thus the ability to be reused. They also lack an extensible framework. Each of these web based product development applications was designed as a stand alone application. They were not designed as scalable web applications and many of them can not even support more than one simultaneous user which quickly erases the advantages of a web based product design environment. They were not designed to be extended nor were they designed with the idea of reuse in a higher level system framework.

2.4 Commercial Integration Tools

Several commercial software applications have been developed to aid in the integration of engineering tools. FIPER, is a four year project sponsored by the National Institute of Standards and Technology designed to create an environment that allows an engineer to easily integrate various pieces of software to form a simulation environment. Kao et al. describe the use of FIPER for real-time business-to-business (B2B) collaboration through the internet. In this case FIPER is used for virtual collaboration of aircraft engine combustor design between GE Aircraft Engines - a jet engine manufacturer, and Parker Hannifin - a gas turbine fuel nozzle supplier. In this case the steps to design a functional nozzle are shared between GE and Parker. The combustor design at GE specified interfaces for the fuel nozzles, this information was then passed to Parker where the nozzles were designed and analyzed for vibration. When this analysis was completed, the necessary information was passed back to GE for aerodynamic analysis. The process was structured in such a way as to allow iteration between the two companies to occur. This case study shows the potential for real time collaboration across the internet [26]. Another similar commercial software package is ModelCenter by Phoenix Integration. The ModelCenter technology is very similar to that of FIPER. Similar case studies exist for the ModelCenter technology.

Each of these tools is geared toward the integration of simulation code and lack a formal strategy for their use. Although these tools provide the capability to create a component based analysis model, they provide no strategy as to how to decompose a system into reusable entities.

2.5 Object-Oriented Theory

Some of the first ideas for object-oriented programming and theory come from a biological analogy. It was postulated that the ideal computer would function like a living organism; each “cell” would behave in accord with others to accomplish an end goal but would also be able to function autonomously. The “cells” could also regroup themselves in order to attack another problem or handle another function. The human body is divided into trillions of cells, each performing a specialized task. Like objects in software produced with-object oriented programming, human cells do not know what goes on inside one another, but they can communicate, working together to perform complex tasks. A developer can focus on one simple module at a time, making sure it works properly, and move on to the next. Not only is building a system this way easier, but the system will be much more reliable. And when it does break down, it is simpler to fix, because problems are typically contained within individual modules, which can be repaired and replaced quickly [21].

2.5.1 Objects

An object represents either an abstract concept or a physical entity in the world, both real and perceived [22]. An object can be a person, place, thing, or attribute -- anything tangible or intangible that exists or is perceived to exist.

Abstraction allows for easier management of complex ideas. A description of a real-world object, situation or process can be simplified in an abstracted model to emphasize aspects that are important to a user of a model. Other details not needed for understanding are suppressed and displayed only in more detailed models. The idea of abstraction in object-oriented development models is to distill the essence of a problem to understand it better. In building systems of objects, an abstract model emphasizes the external view of an object, with the implementation details hidden inside the boundaries of the object [23].

Classification is the process of identifying sets of objects that belong together because they share a particular concept, such as a feature or function [22]. Object classes are generic representations of any object that is a member of that class. These groups are identified by a specific characteristic that makes logical sense.

Encapsulation follows directly from the idea of abstraction. Only the minimal details required in the understanding of an object should be used to represent that object in the abstract model. Encapsulation, also known as information hiding, provides a conceptual barrier around an object, preventing users of the object from viewing its internal details.

Generalizations/Specializations begin with concepts in the most general sense that become further and further refined. A generalization/specialization is commonly referred to as a "type-of" relationship. For example green is a type of color or mustang is a type of automobile. Using generalization, object types can be organized into hierarchies, which form increasingly general types [22]. More general object classes are sometimes referred to as parents, while the specialized classes are referred to as children. The general objects encompass the more specialized objects.

Aggregation relationships depict "part-of" hierarchies. Aggregates are assemblies composed of other objects, or components. For example, an engine, transmission, wheel and body are part of a car. The engine in turn is composed of fuel, cooling and ignition systems. An aggregation can be thought of as a complex object that is composed of other objects. Conceptually an aggregation is an extended object, viewed as a unit by some operations, but actually composed of multiple objects.

When objects have been encapsulated to insulate the outside world from the details of the object structures and behaviors, there needs to be a way to interact with these structures and behaviors. Messages provide this mechanism. The total set of messages that an object can respond to comprises the behavior of that object.

Polymorphism is the ability of two or more object classes to respond to the same message, but in different ways. The meaning of the commands that are passed between objects is packaged with the objects, so a client object does not need to be aware of which server object the message is being sent to. Polymorphism allows the similarities between different object classes to be exploited. Since it is possible to have different responses to the same message, the sender of the message can simply transmit it without regard to the class of the message receiver [23].

Inheritance allows a class to inherit features from a parent class. Multiple inheritance extends this concept to allow a class to have more than one parent class and to inherit features from all parents. A more complicated kind of generalization -- multiple inheritance -- does not restrict a class hierarchy to a tree structure. Generally the child class assumes all of the properties of the parent classes and adds new responsibilities of its own.

2.6 Agent-Based Systems

DaSilveira et al. [7] identify a void in the literature on how to implement the information management processes required for mass customization and identify the potential use of decentralized autonomous agents. Its implementation could potentially reduce the complexity and increase the flexibility of the PDG.

Agent based systems represent one of the most promising computing paradigms for the development of distributed, open and intelligent software systems. Agent technology has been widely applied in the engineering design fields since the early 1990s [53]. An extensive survey of Multi-Agent systems in intelligent design and manufacturing as well as conceptual design can be found in [54] and [55].

Agent-based architecture enables a truly cooperative coordinated computing style wherein members of the agent community work together to perform computation, retrieve information, and serve user interaction tasks. An autonomous agent (1) is not controlled or managed by any other software agents or human beings; (2) it can communicate and interact directly with any other agents in the system and also with other external systems; (3) it has minimally sufficient knowledge about other agents and its environment; and (4) it has its own goals and associated motivations [10]. In an agent-based system of more than one agent, each agent is usually modeled after a natural subdivision of the application for which the system is built. This natural division can be represented by function (e.g. Process Planning), or by physical entities of the system (e.g. CNC machine).

Baker [6] explores the use of autonomous agents for use in a manufacturing process planning and scheduling application. Here the agents are used as the tools for efficiently reconfiguring available production resources. In this case an agent architecture decomposes the system sufficiently to address both information modularity and the physical realities of manufacturing. Shen [10], uses agents to create a web-based collaborative design system. Xue [17] uses a similar architecture to create a distributed database for concurrent design where the tasks of collaboration are distributed among the agents. These agents are accessed at a specified internet node and are accessed by its address and port number.

Emerging standards from the Foundation for Intelligent Physical Agents (FIPA) are playing an increasingly important role in developing intelligent, distributed and collaborative applications. These standards are helping with the innate difficulties of inter operation between heterogeneous agent communities and rapid construction of multi-agent systems using platforms and toolkits that implement FIPA specifications [53].

2.7 Summary

The literature reviewed here provides insight into possible architectures for a nested PDG framework.

CHAPTER 3 BACKGROUND

This chapter provides the background necessary for the development of a system of interconnected PDG components. The foundation is laid by looking at the existing PDG applications. Secondly, we look at various enabling technologies such as distributed computing, service oriented architectures and web services. Next, we look at product modularity, design patterns and product architectures. Lastly, we look at the concept of fractals.

3.1 Foundation

As in any architectural endeavor, the system of PDGs must be built on a solid foundation of component PDGs. In order to build this foundation, a review of the implemented PDGs will be made to determine a logical approach for splitting the PDG into its constituent building blocks. It will then be possible to use and share these modules in multiple PDG applications. This, in turn, will form the basis for a generalized modular approach to the construction of a product design generator.

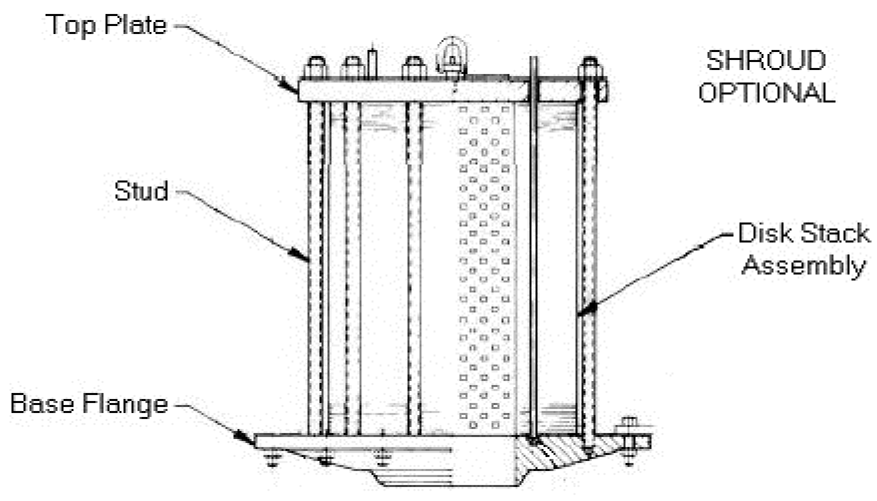


Figure 3.1 Anatomy of the Atmospheric Resistor

3.1.1 Atmospheric Resistor PDG

3.1.1.1 Introduction

The first PDG built using the formal PDG methodology was the atmospheric resistor PDG and was developed for a manufacturer of custom valves. An atmospheric resistor is a fixed orifice fluid device used where any gas or steam needs to be released to the atmosphere (Figure 3.1). Resistors are designed to prevent noise generation instead of simply muffling noise at the outlet. These products are typically used in power plants, oil and gas production facilities, pulp and paper mills, and other process plants where safety considerations prohibit prolonged exposure to high levels of noise. OSHA standards require noise levels in these plants to be kept below a threshold value for worker safety. Noise control is achieved by controlling the expansion rate of the fluid thereby controlling

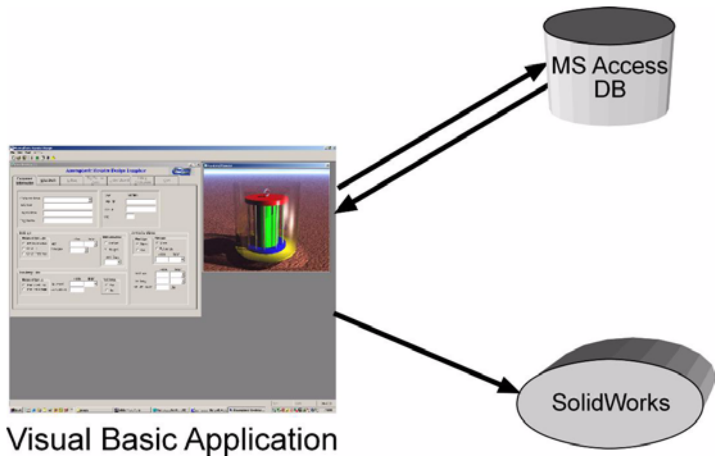


Figure 3.2 Atmospheric Resistor PDG Architecture

the fluid velocity. Because the process plants requiring resistors are so varied, each resistor is custom designed to the specific application.

3.1.1.2 Architecture

The atmospheric resistor was designed as a stand-alone visual basic application (Figure 3.2). This stand alone architecture is simple because it is self contained and easy to install. This simple architecture does, however, have some drawbacks. First, the application and all supporting software (e.g. MS Access, SolidWorks etc.) must be installed on the user's local machine. This means that each user must have a software license for each of these programs. Second, visual basic is relatively slow computationally when compared to other programming languages. Third, visual basic programs only run on the Windows platform, they are not portable to linux or any other unix variant. Forth, MS Access does not scale well to large databases with many simultaneous users. The greatest drawback from this architecture, however, is that the internal mappings may not be shared or reused without significant modifications and recompiling of the code itself.

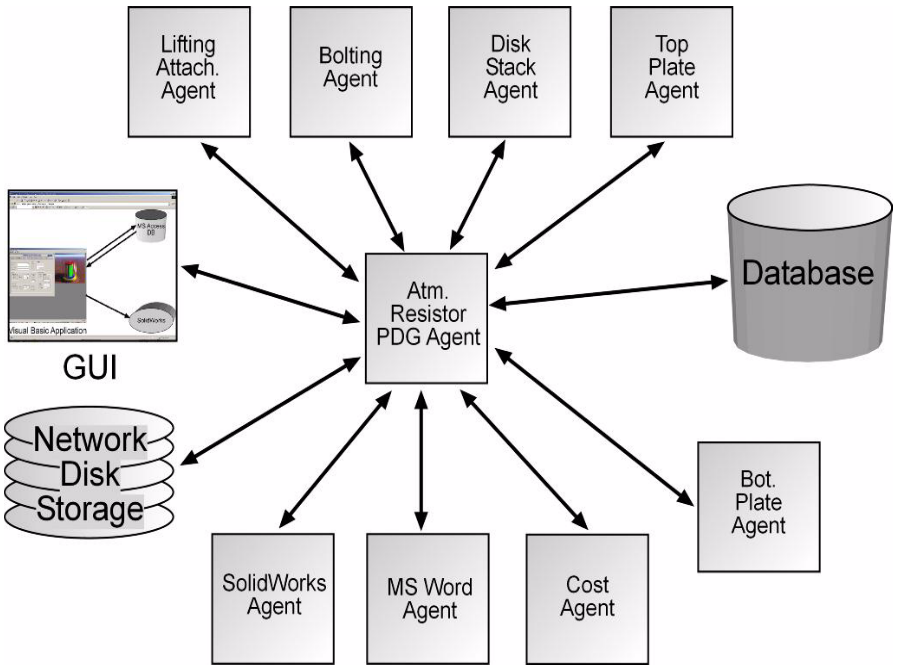


Figure 3.3 Agent-Based Architecture for the Atmospheric Resistor PDG

Figure 3.3 shows how the architecture would change for a service oriented atmospheric resistor PDG. Here the G mappings have been modularized into the SolidWorks service and the MS Word service. The P mappings have been divided into logical modules to give the most flexible and reusable architecture. The valve manufacturing company, for example, uses the same disk stack in a different control valve application, uses the same bolting rules across many different valves and uses the same rules for designing welded and flanged pipe connections as contained in the bottom plate service. If a PDG were to be constructed for a different control valve application, these elements would not be re-created and using these components becomes a simple connection to the relevant component. This greatly improves the maintainability of the overall system and provides a mechanism for reuse.

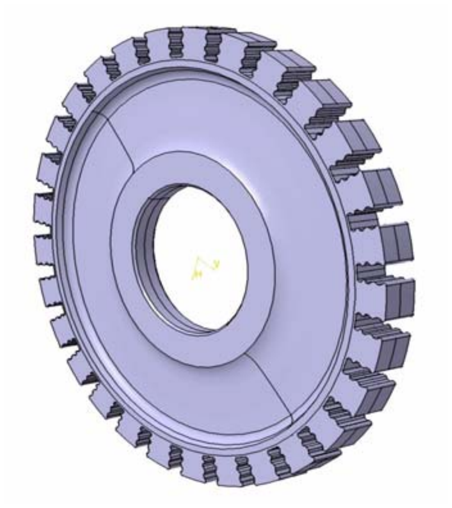


Figure 3.4 CAD Model of a Typical Turbine Disk

3.1.2 Turbine Disk PDG

3.1.2.1 Introduction

The turbine disk PDG was developed for a commercial aerospace company. The basic function of a turbine is to transform a portion of the kinetic energy and heat energy in the exhaust gases to mechanical work, thereby driving the compressor and other accessories [20]. The solution context for this particular PDG is that of an axial flow turbine. A typical axial flow turbine is made up of a number of rotating airfoils that are inserted into slots in an otherwise solid disk (Figure 3.4). The disk functions to maintain the circular motion of the airfoils and couples them to one of the rotating engine shafts. For simplicity, the airfoils were considered only as loads on the turbine disk.

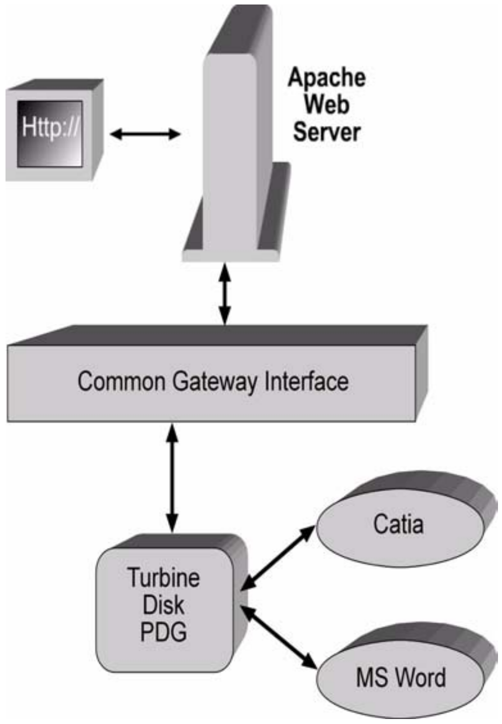


Figure 3.5 Turbine Disk PDG Architecture

The turbine section of an aircraft engine is located immediately after the combustor section and absorbs most of the energy created in the combustion process. Consequently, the turbine is the most highly stressed component in the engine [20]. The stresses on the turbine disk come from the extremely high temperatures of the combustion gases, the enormous inertial loads due to rotation at tens of thousands of rpm, and thermal cycling during the mission. The problem becomes even more difficult when space requirements, weight and cost are considered. The difficult objective is to design a disk to withstand the stresses, fit within a specified spatial envelope, weigh as little as possible, and meet the specified life requirements. These conflicting objectives make turbine disk design an inherently iterative process, sometimes taking months to execute.

3.1.2.2 Architecture

The architecture of the existing turbine disk PDG is slightly more complicated than the resistor PDG because it is web-based (Figure 3.5). A web-based application does, however, provide some significant advantages for product development. First, the application need only be installed on a single machine. This allows a company to run the application on a higher-powered computer while users of the application access it from smaller and cheaper computers. A server-based architecture also lessens the burden of upgrades and configuration management because any changes need only be made once on the server. Second, the user does not need the CAD/CAE software installed locally on his/her computer. The models can be accessed using free VRML viewers. Third, the web is easily accessible from virtually anywhere in the world. This permits a company to share tools between all of their globally dispersed locations. Notwithstanding these advantages, the turbine disk is subject to the same constraints regarding the reusability and sharing of models and mappings within the PDG. That is to say, the mappings can not be reused without modifying and recompiling the code.

3.1.3 The Compliant Constant-Force Spring PDG

3.1.3.1 Introduction

A constant-force compression spring is a device that exerts a constant or near constant force for a given displacement. These types of springs have application in robot end-effectors, electrical contacts, grinding operations, etc. The solution context for this particular PDG is that of a compliant constant-force mechanism. A compliant mechanism is a

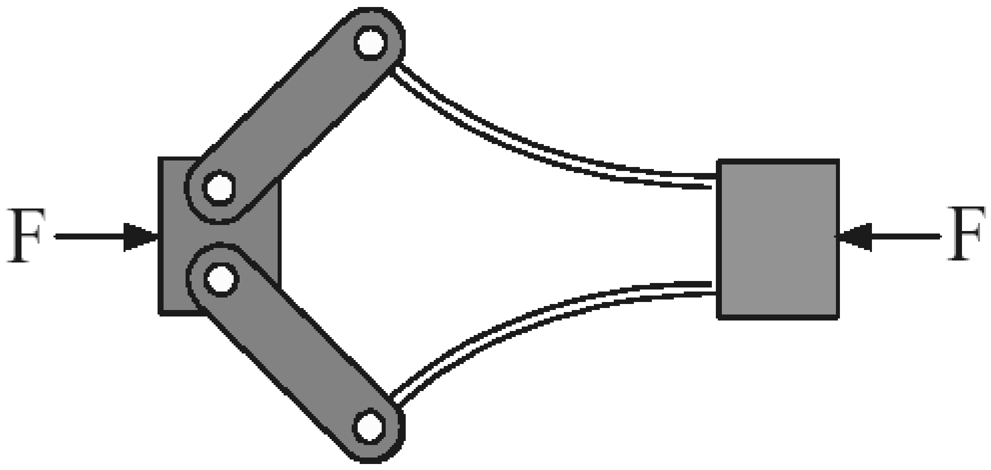


Figure 3.6 Constant-Force Compression Spring

mechanism that gains some or all of its force and motion from the deflection of flexible segments [1]. The configuration chosen for the PDG is shown in Figure 3.6.

3.1.3.2 Architecture

The constant force spring PDG was implemented as a visual basic application. The storyboard was implemented in Visual Basic and the workflow was tightly integrated into the user interface itself. The underlying behavioral models are also tightly coupled to the PDG implementation as are the artifact models in Excel and Word. This architecture makes it difficult to insert process steps into the design process and impedes the reuse of individual behavioral and artifact models and mappings.

3.1.4 Class Projects - Turbofan Engine Components

The same aerospace company that sponsored the axial turbine disk PDG also sponsored the projects for a graduate level course. The projects were divided into four compo-

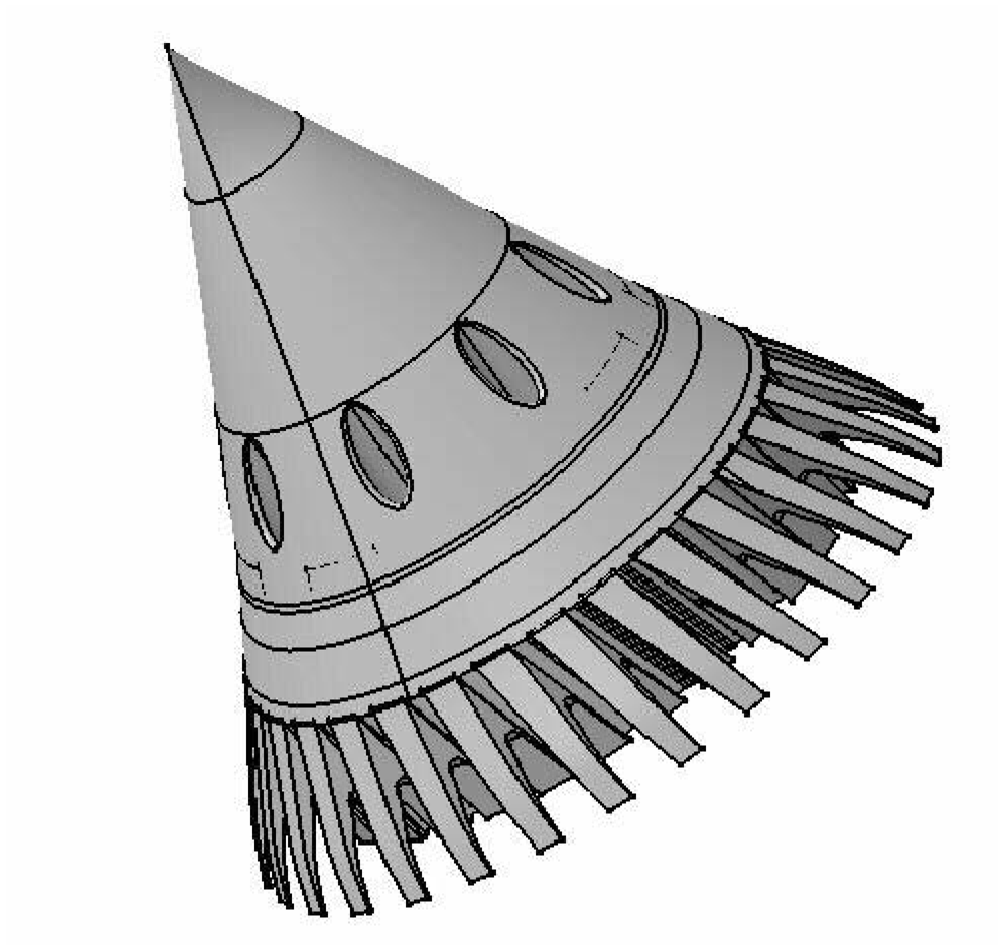


Figure 3.7 CAD Model of Fan Disk and Cone

nent groups; the fan disk and cone, a fan blade, a fan stator, and a turbine nozzle for an industrial turbofan engine.

3.1.4.1 Fan Disk and Cone PDG

Five students were asked to develop a PDG for the fan disk and cone of the engine. The fan disk is the axial disk that supports the fan blades. The nose cone of the engine is attached to the disk. This represents a simple assembly of two components. Also, the fan

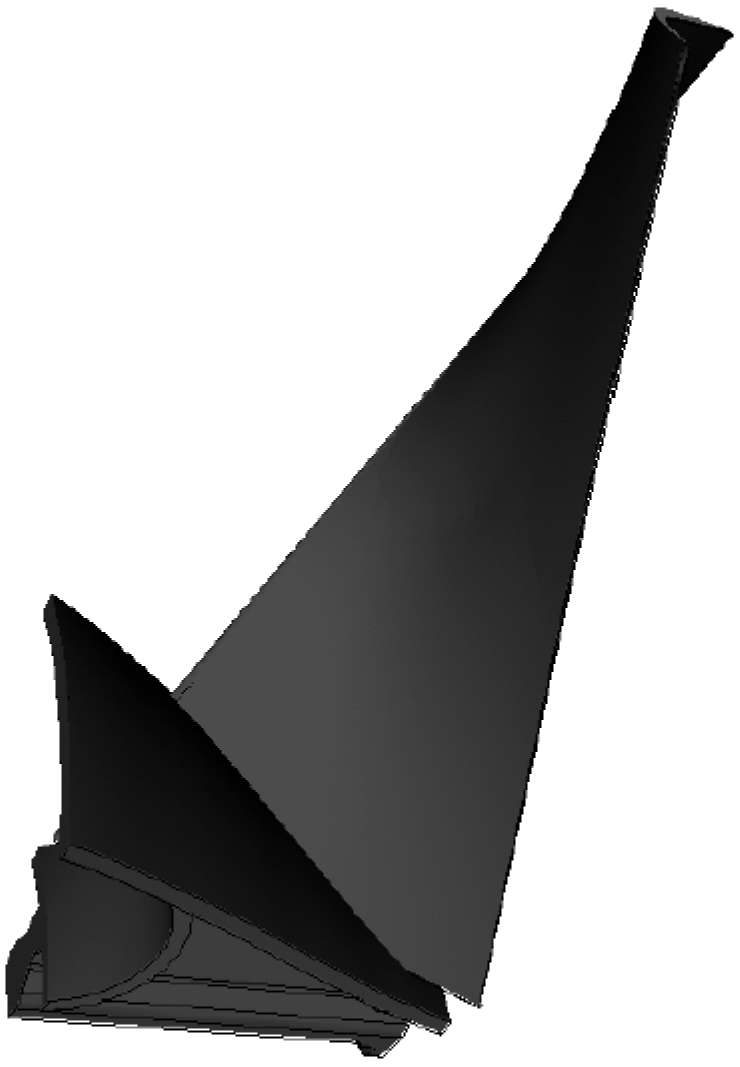


Figure 3.8 CAD Model of Fan Blade

disk and cone team had to collaborate with the fan blade team to ensure that the parametric relationships between the disk and the fan blades would always be maintained [1].

3.1.4.2 Fan Blade PDG

In this project, five students were asked to develop a PDG for the engine fan blade that is attached to the fan disk. The fan blade is composed of four sections: the airfoil, the platform, the attachment, and the shank. The fan blade required a parametric definition of the airfoil cross section in order to design the blade because of the cross section changes that were required. This same parametric definition was also used by the fan stator project team and the turbine nozzle team. Thus, a parametric, generic blade cross section model was developed for all three blade project teams to use [1].

3.1.4.3 Fan Stator PDG

In this project, four students developed a PDG for the engine fan stator. The fan stator is located directly behind the fan in the engine. The stator is a fixed ring of blades that act as diffusers to decrease the air velocity and increase the pressure. The fan can be considered to be the first compression stage of the engine. The fan stator assembly is composed of a number of stator sections which contain one to several vanes. The fan stator required the parametric vane cross section model in order to design the vanes in the stator.

3.1.4.4 Class Project Architecture

Because each of these projects was implemented in a class setting, the implementation architecture was kept simple. User interfaces were implemented using MSExcel, MSAccess, and Visual Basic and their functionality was relatively basic as compared to the PDGs discussed previously. These projects do, however, illustrate the need to reuse models in more than one PDG and show the necessity for a framework that provides communication between the various PDGs. Without this framework it will be difficult to create a maintainable system level PDG derived from the component PDGs. The fan disk and fan blade teams, for example, coordinated the model elements pertaining to the attachment

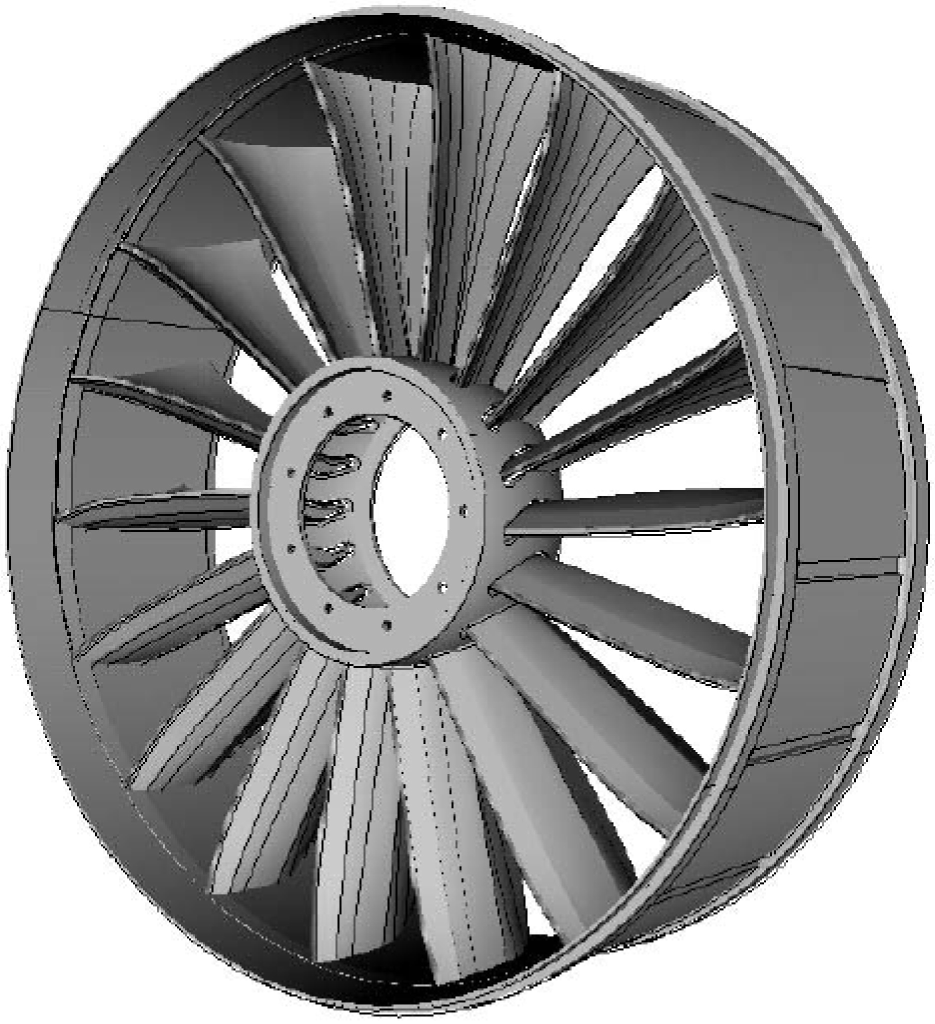


Figure 3.9 CAD Model of Fan Stator Assembly

features on the blade and the disk. In doing so, work was duplicated as each team had to have knowledge of attachment design practices in order to calculate the attachment geometry. Any iteration between the two PDGs required a manual export of the relevant parameters. In this case, it would have made much more sense to share a common model that could be used to design the attachment regions of the disk and blade. Furthermore, this approach would provide improved maintainability of the system because changes and updates could be made to a single component within the system. Otherwise, changes made to one model would cause the second model to be "out of synch" with the first, creating a fragile, hard to maintain software environment. The same could be said about the parametric airfoil model that was used by both the fan blade and fan stator teams. In this case, each team used the same parametric model for the airfoil definition but a separate copy of that model was used by each of the groups. This practice acts in contrary to the idea of reuse that is so important to the PDG methodology. Benefits of scale, and speed of PDG creation can not be realized without the reuse of PDG components and a structure to allow components to be assembled on an ad hoc basis into a higher level system.

3.2 Enabling Technologies

Technologies such as distributed computing, service oriented architectures, web services, design patterns, J2EE, and Microsoft dot Net have recently provided the necessary capabilities to successfully create a system of interconnected PDGs and PDG components. These technologies along with emerging standards for inter connectivity provide industry accepted methods for connecting disparate systems.

3.2.1 Distributed Computing

CORBA, DCOM, and Java/RMI are distributed object technologies, while web services, are based on a messaging architecture, in which methods are invoked and data passed via messages. Traditional object-based approaches tend to be tightly coupled. In a tightly coupled system, the changes in the interface definition of one component require the rest of the system to be updated. Service or message based architectures tend to be loosely coupled meaning changes in one component can be tolerated as long as the underlying data and component definitions don't change.

3.2.2 Service Oriented Architecture

Service Oriented Architecture (SOA) is an architectural style for building software applications that use services made available over a network such as the web. It promotes loose coupling between software components so that they can be reused. Applications in SOA are built based on services. A service is an implementation of a well-defined business functionality, and such services can then be consumed by clients in different applications or business processes.

SOA allows for the reuse of existing assets where new services can be created from an existing infrastructure of systems and components. In other words, it enables businesses to leverage existing investments by allowing them to reuse existing applications or pieces of applications, and promises inter operability between heterogeneous applications and technologies. SOA provides a level of flexibility that wasn't possible before in the sense that:

- Services are software components with well-defined interfaces that are implementation-independent. An important aspect of SOA is the separation of the service interface (the what) from its implementation (the how). Such services are consumed by clients that are not concerned with how these services will execute their requests.

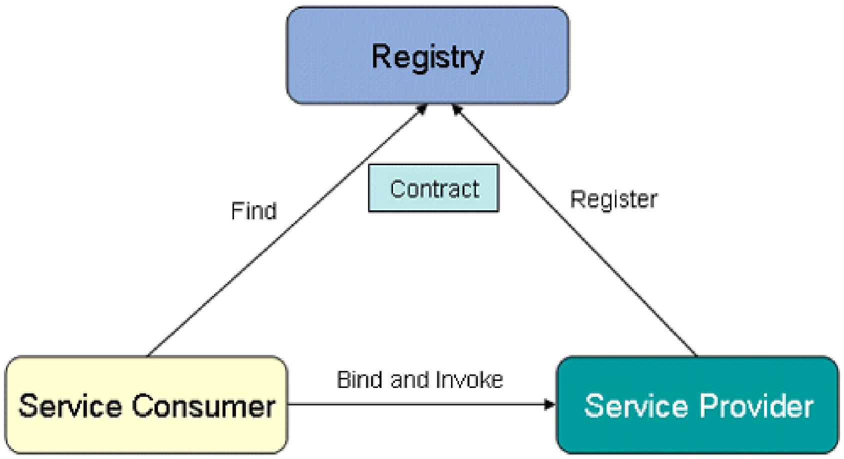


Figure 3.10 SOA's Find-Bind-Execute Paradigm

- Services are self-contained (perform predetermined task), modular and loosely coupled (for independence)
- Services can be dynamically discovered
- Composite services can be built from aggregates of other services
- Services stress inter operability
- Services have a network-addressable interface
- Services have course-grained interfaces
- Services are location transparent

SOA uses the *find-bind-execute* paradigm as shown in Figure 3.10. In this paradigm, service providers register their service in a public registry. This registry is used by consumers to find services that match certain criteria. If the registry has such a service, it provides the consumer with a contract and an endpoint address for that service.

SOA promotes application assembly because services can be reused by numerous consumers. For example, in order to create a service that calculates the attachment geometry, we build and deploy only one instance of such a service; then we can consume this service from any number of applications including the fan disk PDG and the fan blade PDG. The main theme behind SOA is to find the appropriate modularity and achieve loose coupling between modules.

The other key advantage of SOA is that it lets you automate business-process management. Business processes may consume and orchestrate these services to achieve the desired functionality. Thus, new business processes can be constructed by using existing services. For example, the design of fan disk can be represented by a business process that can asynchronously interact with the requisite services.

Like any distributed application, service-oriented applications are multi-tier applications and have presentation, business logic, and persistence layers. Figure 3.11 provides a typical architecture for a service-oriented application.

Another promise of SOA is that you can build a new application from existing services. An additional benefit that SOA brings is the standardization of business process modeling, often referred to as *service orchestration or service choreography.* The nesting of services can be arbitrarily complex and at the topmost level the entire business process ultimately can be viewed as a service. You can build a web-service-based layer of abstraction over legacy systems and subsequently leverage them to assemble business processes.

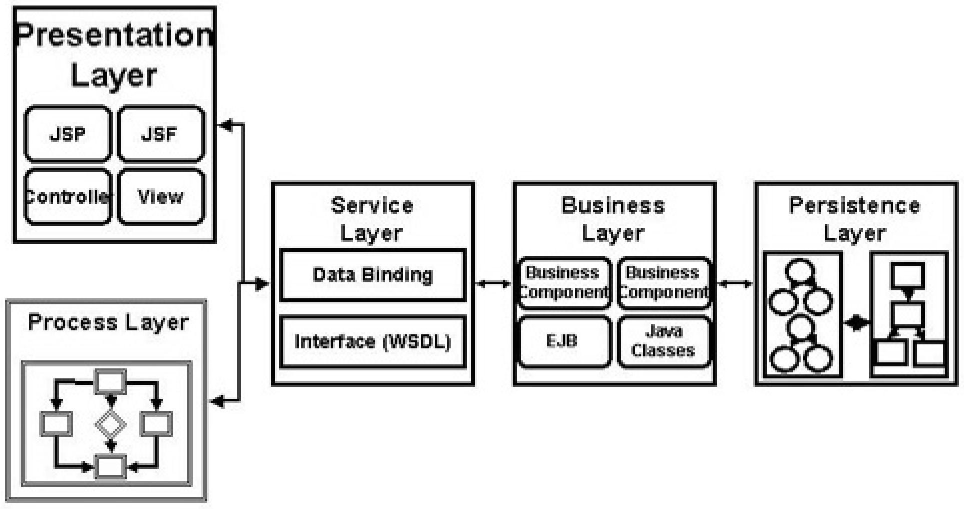


Figure 3.11 Different Layers of Service Oriented Applications

In addition, SOA platform vendors are providing tools and servers to design and execute these business processes. This effort has been standardized by an OASIS standard named Business Process Execution Language (BPEL); most platform vendors are adhering to this standard. BPEL is essentially an XML based programming language used to define a business processes.

3.2.3 Web Services

Although the original intent of the Internet was to send and receive pure textual information, the Internet gave birth to a new computation paradigm driven by “the network is the computer” and the “anywhere, anytime computation” metaphor. Central to this idea is the emerging technology of web services. The past couple of years have seen the beginning of a remarkable transformation in the business landscape. Connectivity, collaboration, and communication are all being revolutionized by the Internet and web services.

Web service technology is an important emerging paradigm for distributed computing that differs from existing approaches such as COM, CORBA, and Java RMI in its focus on simple internet-based standards such as XML, to address heterogeneous distributed computing. The web service model includes three parties in the general sense:

1. Service Provider: hosts the computational service or resource. This service can be of any kind ranging from a simple random number generator to a complex sequence of computations or optimizations.

2. Service Requester: the counterpart that invokes the service from the provider. Thanks to well defined standards for communication and description of the service, it is possible to universally connect service requesters to providers in a peer-to-peer fashion.

3. Service Directory: or broker. This is a service itself where the provider can publish services and the requester can search and discover services suiting the purpose.

Web services allow applications residing on the Internet and intranets to work together in an integrated fashion. They make it possible to integrate systems that would otherwise require extensive development efforts and proprietary solutions. Web services provide a simple and streamlined mechanism for applications to communicate over the Internet/intranet without human intervention, and without the need to know the environment at each end point. This virtualization of service endpoints provides a flexible architecture that can be assembled in an ad hoc fashion at runtime. Web services standards such as XML, Simple Object Access Protocol (SOAP) and Web Services Description Language (WSDL) enable a new level of plug-and-play functionality in software. These technologies allow web services to be platform independent.

3.2.4 Web Service Basics

The most significant aspect of web services is that every software and hardware company in the world has positioned itself around these technologies for inter operability. No single technological advancement will have as great an impact on the way systems are developed as Web Services.

Web services use Internet technology for system interoperability. The advantage that web services have over previous interoperability attempts, such as CORBA, is that they build on the existing infrastructure of the Internet and are supported by virtually every technology vendor in existence. As a result of the ubiquitousness of the technologies they use, Web Services are platform-independent.

The computer software industry is moving towards a service-based model of implementation and delivery of software functionality for a wide range of applications. This architecture is more broadly known as a Service Oriented Architecture (SOA). While over-hyped marketing may dilute the precise definition of a web service, in general they possess such attributes as reusability, loose coupling, discrete functionality, programatic access, and internet accessibility.

There are several reasons for this trend towards “software as a service” for certain commercial and in-house applications, including:

1. The systems are easier to manage and the IT support can be centralized.
2. With standard service descriptions it is possible to assemble “best in class” solutions
3. Enforces corporate standards by defining data and the services that operate on that data
4. Legacy code can be incorporated into the information processing workflow
5. Services and clients are platform agnostic, and the links between them are loosely coupled

3.2.5 Existing and Emerging Standards

3.2.5.1 Extensible Markup Language

The eXtensible Markup Language (XML) is a standards-based data structure format for representing information; XML is “human readable” and computer platform independent.

3.2.5.2 Simple Object Access Protocol

Simple Object Access Protocol (SOAP) is an XML syntax for exchanging messages between applications or services on the internet; it supports communicating with, launching, and querying web services. Since it is based on XML, SOAP is also platform independent.

3.2.5.3 Universal Description, Discovery, and Integration

Universal Description, Discovery, and Integration (UDDI) is an XML protocol that enables automated storage and lookup of web services. It allows a service to describe its functions and supports the discovery of other services that perform a desired function.

3.2.5.4 Web Services Description Language

Web Services Description Language (WSDL) is an XML protocol that describes what a web service can do, where it resides, and how to invoke it.

3.2.6 Benefits of Web Services

3.2.6.1 Reusability

The promise of reusable legacy applications, databases, objects, and components have been largely unrealized. Web services can play a significant role in improving software reusability within organizations. The likelihood of reusability depends on several factors that Web services improve on: interoperability, modularity, central registry, and reduced compile time dependencies. Web services are built on open, interoperable, and ubiquitous standards, which maximizes their potential for reuse. Developers have two options to create functionality. They can either develop the functionality as part of the application that needs it or as a separate service. A function developed as a separate component and used as a service is more likely to outlive the original application.

3.2.6.2 Location Transparency

A service environment achieves location transparency, because the location is stored in a registry. A client finds and binds to a service and does not care where the service is located. Therefore, an organization has the flexibility to move services to different machines or to move a service to an external provider. The way a service is implemented is irrelevant. Therefore, if it becomes necessary to move a service from a J2EE platform to a .Net platform, no changes to the clients should be necessary.

3.2.6.3 Composition

Developers assemble applications from a preexisting catalog of reusable services. Services do not depend on the applications into which they are composed. Because services are independent, developers will logically reuse these services in many applications. The interface design process promotes the design of interfaces that are modular and independent from the application for which they are designed. Developers by nature want to reuse software unless it is more difficult to reuse than to build from scratch. One of the greatest impediments to reuse is determining the software available for reuse. The registry in a service-oriented architecture provides this single place to store service descriptions.

3.2.6.4 Scalability and Availability

A system is scalable if the overhead required to add more computing power is less than the benefit the additional computing power will provide. Because service clients know only about the service interface and not its implementation, changing the implementation to be more scalable and available requires little overhead.

3.2.7 Modularity of Services

3.2.7.1 Modular Decomposability

Modular decomposability of a service refers to the breaking of an application into many smaller modules. Each module is responsible for a single, distinct function within an application. This is sometimes referred to as “top-down design”, in which the bigger problems are iteratively decomposed into smaller problems. The main goal for service design is to identify the smallest unit of software that can be reused in different contexts.

3.2.7.2 Modular Composability

The modular composability of a service refers to the production of software services that may be freely combined as a whole with other services to produce new systems. Service designers should create services sufficiently independent to reuse in entirely different applications from the ones for which they were originally intended. This is sometimes referred to as bottom-up design. Sometimes, the composability and decomposability approaches to service design can create two different designs. The bottom-up approach is more focused on the application functions. The top-down design tends to be more focused on the business problem. It is important to use both methods to find the right interface for the service. The typical design begins as a decomposition exercise. When the designers get to a point at which they have exhausted the top-down design, performing a bottom-up analysis should validate the design. The bottom up design begins by defining the significant scenarios that the modules need to support. The significant scenarios will cover the important functional aspects of the modular design.

3.2.7.3 Modular Understandability

The modular understandability of a service is the ability of a person to understand the function of the service without having any knowledge of other services. The modular understandability of a service can also be limited if the service supports more than one distinct business concept.

3.2.7.4 Modular Continuity

The modular continuity of a service refers to the impact of a change in one service requiring a change in other services or in the consumers of the service. An interface that does not sufficiently hide the implementation details of the service creates a domino effect when changes are needed.

3.2.7.5 Modular Protection

The modular protection of the service is sufficient if an abnormal condition in the service does not cascade to other services or consumers. Faults must not cascade from the service to other services or consumers.

3.2.7.6 Direct Mapping

A service should map to a distinct problem domain function. The designer should create boundaries around service interfaces that map to a distinct area of the problem domain. This is important so the designer creates a self-contained and independent module. It is easy to pollute service interfaces with functions that: logically belong in another existing service, belong in a new service, span multiple services and require a new composite service, are really internal knowledge that should not be exposed through an interface. To directly map a services's interfaces to a distinct business concept in the problem domain, the service designer needs a good understanding of the problem domain.

3.2.7.7 Information Hiding

The service should never expose its internal data structures. Even the smallest amount of internal information known outside the service will cause unnecessary dependencies between the service and its consumers.

3.2.7.8 Loose Coupling

Coupling refers to the number of dependencies between modules. There are two types of coupling: loose and tight. Loosely coupled modules have a few well known dependencies. Tightly coupled modules have many unknown dependencies. Service oriented architectures promote loose coupling between service consumers and service providers and the idea of a few well known dependencies. A system's degree of coupling directly affects its modifiability. The more tightly coupled a system is, the more a change in a service will require changes in the service consumers. Coupling is increased when service consumers require a large amount of information about the service provider to use the service. SOA accomplishes loose coupling through the use of contracts and bindings. Since a service may be both a consumer and a provider of some services, the dependency on only the contract enforces the notion of loose coupling in service-oriented architecture.

3.2.7.9 Network-Addressable Interface

The role of the network is central to the concept of SOA. A service must have a network-addressable interface. A consumer on a network must be able to invoke a service across the network. The network allows services to be reused by any consumer at any time. The ability for an application to assemble a set of reusable services on different machines is possible only if the services support a network interface. Because of this requirement, service interface design is focused to a large extent on performance. Performance can degrade when objects are distributed across a network because of the chatter that occurs between fine-grained objects.

3.2.7.10 Coarse-Grained Interfaces

The appropriate level of granularity for a service and its methods is relatively coarse. A service generally supports a single distinct business concept or process. A service may still be implemented as a set of fine-grained objects, but the objects themselves are not accessible over a network connection. A service implemented as objects has one or more coarse-grained objects that act as distributed facades. These objects are accessible over the network and provide access to the internal object state from external consumers of the service. However, objects internal to the service communicate directly with each other within a single machine, not across a network connection.

3.2.8 Web Services and Service Oriented Architectures

What web services bring to the concept of an SOA is well accepted standards, something missing from previous approaches. In an SOA, we are now able to represent our assets as services - described using WSDL. This gives a standard way of describing these assets that is independent of language and platform. This means that we can now start to think of these services as building blocks that can be reused and assembled into larger structures - again independent of language and platform. This gives us the flexibility to respond to new business demands, and allows a significant degree of integration and communication both inside an enterprise, and between enterprises.

3.2.9 Engineering Web Services

In the commercial software market, this trend towards “software as a service” has been primarily in the business processes sector, including accounting, purchasing, human resources, supplier management, etc. Recently, engineering software vendors have begun implementing web-based versions of their applications. Such an implementation of software, while web-based, is technically not a web-service in the current sense of the term since the applications do not truly offer programmatic access, or they may require human interaction to complete a particular job.

The past decade has witnessed an obvious proliferation of software tools available for the various activities involved in engineering analysis and design...The mere existence of and improvements in these individual analysis tools does not mean they are improving the productivity of the engineer in fact, the opposite may very well be true. The reason for this is that there is usually no single entry point to these tools or a common way to interact with them, making it difficult for them to be used in a coordinated fashion. Moreover, once an ultimate design is arrived at, it is often very difficult, if not impossible, to determine what specific tools were used, in what combination, and what were the sources of the data supplied to them. The solution, therefore, is not more or even better design tools- the solution is a framework that provides a solid foundation for overcoming these problems[27].

3.3 Patterns

In the 1970's, Christopher Alexander wrote a number of books documenting patterns in civil engineering and architecture [56]. Alexander defined the term 'design pattern' in the following way: "Each pattern describes a problem which occurs over and over again in our environment, and describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice." On a general level, the principle of design patterns is well known in many engineering domains. They can be seen in the form of design catalogs and handbooks for mechanical, electrical, and software engineering. Patterns can be seen as a form of intellectual reuse.

Patterns are about communicating problems and solutions. Simply put, patterns enable us to document a known recurring problem and its solution in a particular context, and to communicate this knowledge to others. Each pattern expresses a relation between a certain context, a problem, and a solution.

Some of the common characteristics of patterns are as follows:

- Patterns are observed through experience
- Patterns are typically written in a structured format
- Patterns prevent reinventing the wheel
- Patterns exist at different levels of abstraction
- Patterns undergo continuous improvement
- Patterns are reusable artifacts
- Patterns communicate designs and best practices
- Patterns can be used together to solve a larger problem

In short, no pattern is an isolated entity. Each pattern can exist in the world, only to the extent that is supported by other patterns: the larger patterns in which it is embedded, the patterns of the same size that surround it, and the smaller patterns which are embedded in it [56].

An example of the use of patterns in engineering can be seen at Toyota, where many describe the widespread use of lessons-learned books. Books that provide the engineers with guidelines for designing different parts of a car, including company specific information on capabilities and feasibility ranges ensuring the manufacturability.

3.4 Product Architecture

Product architecture is one of the development decisions that most impacts a firms's ability to efficiently deliver high product variety. Products built around modular product architectures can be more easily varied. Ullrich and Eppinger state that a product can be thought of in both functional and physical terms. The functional elements of a product are the individual operations and transformations that contribute to the overall product performance. The physical elements of a product are the parts, components, and sub-assemblies that ultimately implement the product's functions. The physical elements of a product are typically organized into several major building blocks. Each physical element is then made up of a collection of components that implement the functions of the product, which we will call chunks. The architecture of a product is the scheme by which the functional elements of the product are arranged into physical chunks and by which the chunks interact [19]. The most important characteristic of a product's architecture is its modularity. A modular architecture has the following two properties: (1) chunks implement one or a few functional elements in their entirety and (2) the interactions between chunks are well defined and are generally fundamental to the primary functions of the product [19].

The most modular architecture is one in which each functional element of the product is implemented by exactly one chunk and in which there are a few well-defined interactions between the chunks. Such a modular architecture allows a design change to be made to one chunk without requiring a change to other chunks for the product to function correctly. The chunks may also be designed independently of one another. The opposite of a modular architecture is an integral architecture. An integral architecture exhibits the following properties: (1) functional elements of the product are implemented using more than one chunk, (2) a single chunk implements many functional elements, and (3) the interactions between chunks are ill defined and may be incidental to the primary functions of the products [19].

Decisions about how to divide the product into chunks and about how much modularity to impose on the architecture are tightly linked to several issues of importance to the entire enterprise: product change, product variety, component standardization, product performance, manufacturability, and product development management. The architecture of the product therefore is closely linked to decisions about how the product may be varied and changed.

Chunks are the physical building blocks of the product, but the architecture of the product defines how these blocks relate to the function of the product. The architecture therefore also defines how the product can be changed. Modular chunks allow changes to be made to a few isolated functional elements of the product without necessarily affecting the design of other chunks. These changes to the product allow the product to be adapted to a rapidly changing market. Some of the motives for product change are:

1. Upgrade: As technological capabilities or user needs evolve, some products can accommodate this evolution through upgrades.

2. Add-Ons: Many products are sold by a manufacturer as a basic unit, to which the user adds components, often produced by third parties, as needed.

3. Adaptation: Some long lived products may be used in several different use environments, requiring adaptation.

4. Wear: Physical elements of a product may deteriorate with use, necessitating replacement of the worn components to extend the useful life of the product.

5. Consumption: Some products consume materials, which can then be easily replenished.

6. Flexibility in use: Some products can be configured by the user to provide different capabilities.

7. Reuse: In creating subsequent products, the firm may wish to change only a few functional elements while retaining the rest of the product intact.

Modularity increases the range of product models the firm can produce within a particular time period in response to market demand. Products built around modular product architectures can be more easily varied without adding tremendous complexity to the manufacturing system.

3.4.1 Types of Modularity

Modular architectures typically comprise three types: slot, bus, and sectional. Each type embodies a one-to-one mapping from functional elements to chunks, and have well-defined interfaces.

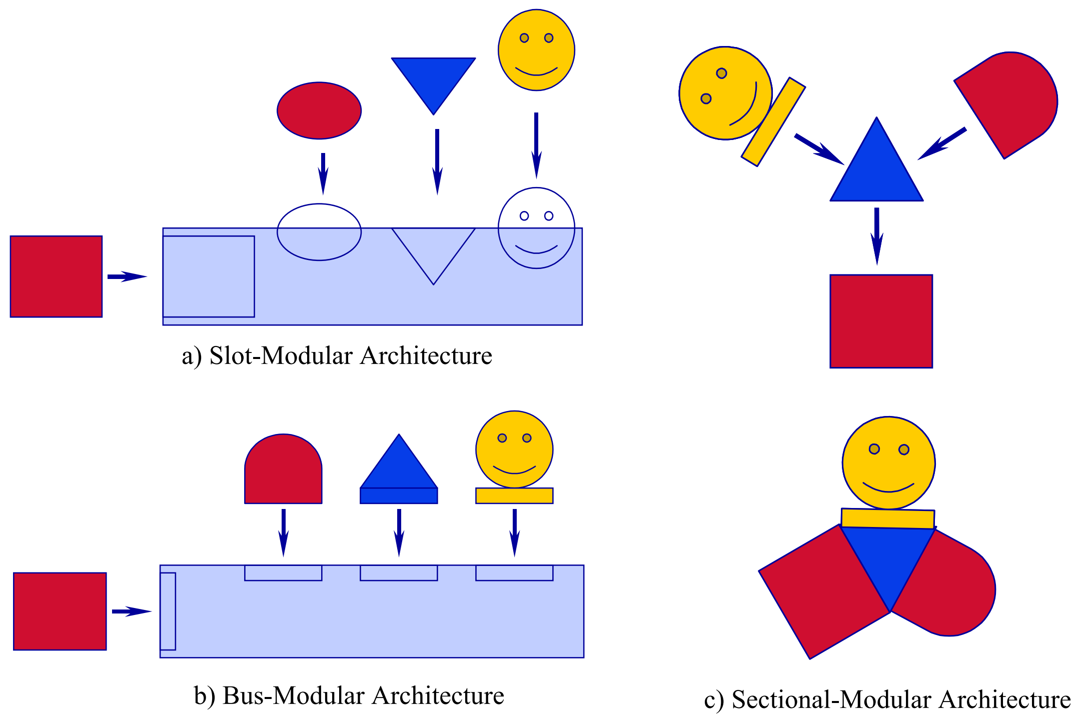

a) Slot-Modular Architecture
b) Bus-Modular Architecture
c) Sectional-Modular Architecture

Figure 3.12 Three types of Modular Architecture

3.4.1.1 Slot-Modular Architecture

Each of the interfaces between chunks in a slot-modular architecture is of a different type from the others, so that the various chunks in the product cannot be interchanged. An automobile radio is an example of a chunk in a slot-modular architecture. The radio implements exactly one function, but its interface is different from any of the other components in the vehicle (Figure 3.12a).

3.4.1.2 Bus-Modular Architecture

In a bus-modular architecture, there is a common bus to which the other chunks connect via the same type of interface. A common example of a chunk in a bus-modular architecture would be an expansion card for a personal computer (Figure 3.12b).

3.4.1.3 Sectional-Modular architecture

In a sectional-modular architecture, all interfaces are of the same type, but there is no single element to which all the other chunks attach. The assembly is built up by connecting the chunks to each other via identical interfaces. Many piping systems and office partitions adhere to a sectional-modular architecture (Figure 3.12c).

While a module refers to a physical or conceptual grouping of components that share some characteristics, modularity tries to separate a system into independent parts or modules that can be treated as logical units. Therefore, decomposition is a major concern in modularity analysis. In addition, to capture and represent product structures across the entire product development process, modularity is achieved from multiple viewpoints, including functionality, solution technologies, and physical structures; functional modularity, technical modularity, and physical modularity.

What is important in characterizing modularity is the interaction between modules. Modules are identified in such a way that between-module interactions are minimized whereas within-module interactions may be high.

3.4.1.4 Product Platforms and Product Families

Product platform and product family strategies take advantage of modularity through a shared set of modules. In this type of strategy, the modules form a set of subsystems and interfaces developed to form a common structure from which a stream of derivative products can be efficiently developed and produced. The typical Product Family Architecture (PFA) consists of three elements, namely the common base, the differentiation enabler, and the configuration mechanism.

Common bases refer to certain shared elements within a product family. The common base allows for economies of scale in the manufacturing and production processes.

Differentiation enablers are basic elements making products different from one another. They are the source of variety for a product family.

Configuration Mechanisms define the rules for and means of product variant derivation. Variety generation refers to the way in which the a distinct form of the product can be created. Such variety fulfillment is related to each differentiation enabler and is achieved through attaching, removing, swapping and scaling elements of the product.

These Product Family constructs are usually selected based on (a) current and future customer needs; (b) commonality in design and fulfillment; (c) ease of configuration, and (d) appropriate level of aggregation. If the construct is at too low a level of aggregation, such as at the nuts and bolts level, then the number of constructs may be too many and the configuration becomes too difficult. On the other hand, if the aggregation is at a very high level, such as complete modules or products, then the commonality may not be sufficient.

3.5 Fractals

It was the Polish mathematician Benoit B. Mandelbrot who first introduced the term ‘fractal’ in 1975 to characterize spatial or temporal phenomena that are continuous but not differentiable. The word fractal comes from a Latin word ‘fractus’, which means broken or fragmented. Unlike more familiar Euclidean constructs, splitting a fractal into smaller pieces results in the resolution of more structures. Each of these fragments or rather, fractals contains the basic characteristics of the whole structure [42].

Fractal properties include scale independence, self-similarity. complexity, and infinite length/detail. A fractal is a geometric object which can be divided into parts, each of which is similar to the original object. One of the implications of self-similarity is that each fractal must itself be a little fractal [31]. A normal Euclidean shape, such as a circle, looks flatter and flatter as it is magnified. At infinite magnification it would be impossible to tell the difference between a circle and a straight line. Fractals are not like this. Instead, with a fractal, increasing the magnification reveals more detail that was previously invisible [60].

3.6 Summary

This chapter explains the technologies and ideas necessary to understand the method that follows.

A foundation of existing PDG applications was presented to provide insight into the work that has been done. The internal structure was examined and was found to be incompatible with the idea of a reusable system.

The technologies critical to enabling a network of interconnected PDGs were explained. The idea of a Service Oriented Architecture and its corresponding benifits were introduced. We were acquainted with Web Services as an implementation technology for the SOA paradigm.

Patterns were presented as a form of intellectual reuse. A pattern is a documented and tested solution to a problem that is repeatedly seen.

Important concepts relating to product architecture were discussed including modularity and product family approaches to product design. These approaches are important to the idea of a PDG framework.

Fractals exhibit important properties that will be used in the following chapters as the method for breaking a PDG into reusable components is presented.

CHAPTER 4 METHOD

A formal method will be developed to provide direction for the building of a system level PDG. This method will provide a step by step process for building a system from components and sub-systems.

4.1 The PDG

The PDG methodology has shown tremendous promise both in academia and as it has been applied in industry. It has been shown to reduce design cycle time, enabling the engineer to focus on value added activities rather than repetitive tasks. It allows engineers to study more potential solutions and implement optimization techniques that were not previously possible. Issues formerly dealt with in the detailed design phases of product development can now be addressed in preliminary design because of the restructuring of the design process that the PDG supports. Manufacturing standards have been included in the various PDG implementations to help insure that a particular design may be fabricated. The PDG has also been shown to reduce the opportunity to introduce errors in the development process. The PDG allows the engineer to walk through a standardized, structured process for product design. This process represents the entire body of company knowledge and gives the engineer the ability to leverage that knowledge independently from individual experience. Each designer, therefore, has access to the collective knowledge of the group instead of being forced to re-learn problems that have been previously solved. Notwithstanding these advantages, the development of a PDG can be a significant investment. This investment is rapidly recovered, often in a single product development effort. A need has been shown, however, for a standardized framework for PDG development and deployment. This framework must reduce the development expense for new PDGs.

The PDGs that have been developed up to this point have aided in the design of a single part or simple system or subsystem. A typical product, however, is typically made up of many interacting parts and subsystems. A gas turbine engine for example, is made up of thousands of parts composing many interconnected sub-systems. Many specialized tools have been developed for the design of these parts and sub-systems. These tools do not communicate with each other, causing many manual transfers of information. Ridding the development process of these manual data transfers can significantly reduce design cycle time but there is a much larger problem caused by these isolated tools. It is the problem of system design. It is very difficult to create a detailed, system-level model because the tools used in the design are completely isolated from one another. They can not be executed in an organized system-level fashion. This difficulty shows a need for PDGs to be part of an interconnected design system. Not only does this aid in the creation of better system level models, but it also facilitates the rapid development of individual PDGs because certain functions can be shared and reused amongst the PDGs.

This work aims to provide strategies for the creation of a PDG framework which facilitates the rapid development and deployment of PDG applications as well as provide the ability to aggregate PDG components into a higher level system PDG.

The PDG methodology defines a transformation function for transforming customer requirements to a member of the product family. The PDG is an automated implementation of the transformation function for a product development process for creating all of the design artifacts and supporting information necessary for the design of a particular product. The complex transformation function is decomposed to intermediate transformations to deal with the complexity of the process. The intermediate transformations account for behavior predictions, company rules and best practices, the generation of design artifacts, data and artifact vaulting strategies, testing procedures and design artifact delivery procedures. The construction of the PDG as defined by Roach is constructed in three major steps: (1) selection of the product concept and embodiment, (2) development of the product transformation schematic, and (3) construction of the reusable intermediate functions and integration of them into the automated PDG application [1],[5].

This work will provide additional steps to the PDG construction process which aid in the creation of reusable PDG components. We will begin by introducing some new PDG concepts.

4.2 Units of Reuse and Units of Partition

In order to create reusable PDG components, we must first define reuse. For the purposes of this work, we will define reuse as the degree to which a software module, physical module or other work product can be used in more than one context. For a software module, reuse is the degree to which the module may be reused in a different computing program or software system. A physical module can be reused in a different product or application according to the principles of modularity as described previously. In order to build a completely flexible virtual product development environment, we must simultaneously consider the software and physical module reuse.

There are different levels of reuse and each level is equally important to the construction of a PDG. A flexible product development environment requires software, product, and process reuse. These different levels of reuse must be matched to see their full benefit. These different levels allow us to manage the complexity of the product development process and allows the decomposition of that process into manageable chunks.

4.2.1 The Object

The first unit of reuse that we will discuss is the object. An object can be considered to be a software bundle of variables and related methods. Everything that the software object knows (state) and can do (behavior) is expressed by the variables and the methods within that object. Figure 4.1 shows a “cellular” representation of an object where the object's variables make up the center, or nucleus, of the object. Methods surround and hide the object's nucleus from other objects in the program. This is referred to

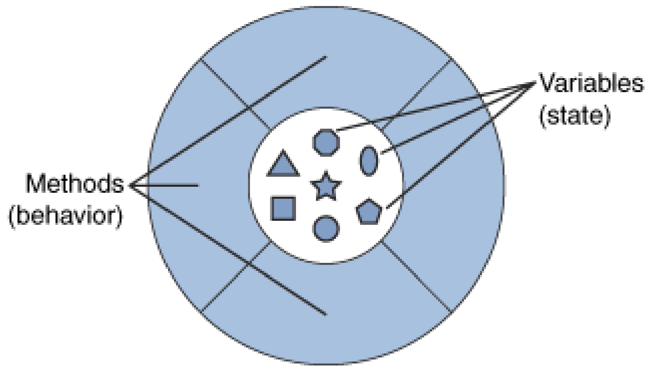


Figure 4.1 "Cellular" Representation of an Object [83]

as encapsulation or information hiding. Encapsulation promotes the idea of an object being self contained and the methods define a standard interface through which other objects can access the objects internal data. An object usually appears as a component of a larger software system that contains many objects. It is through the interaction of these objects that a system can achieve higher order functionality and more complex behavior. Objects interact with one another by sending messages to each other to request a service or piece of data (Figure 4.2). While objects promote reusability, they do have some disadvantages when it comes to a distributed PDG architecture. Objects exist and run on a single computer and the functionality that they produce can not easily be accessed from other machines or address spaces. Neither can they be easily accessed from objects written in another programming language. This poses a limitation because it is infeasible to execute an entire product development process on a single machine and it may be necessary to integrate applications written in different languages. CORBA and other technologies address these issues but require a tight coupling between the objects. For more informa-

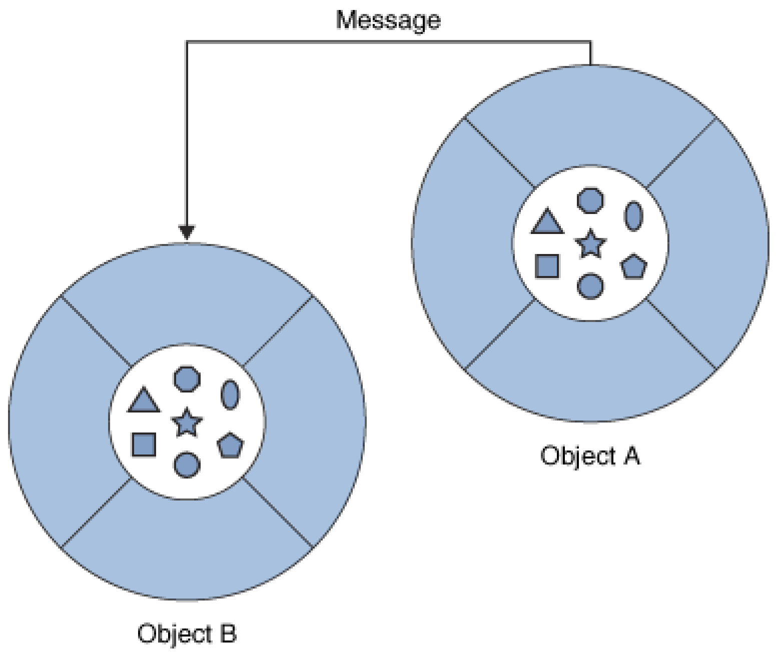


Figure 4.2 Message Passing Between Objects [84]

tion on CORBA see [85]. The object will be the most granular element of reuse in this PDG framework.

4.2.2 The Service

The second unit of reuse that will be used for the PDG framework is the service. Services offer significant advantages over simple objects. As stated previously, services are software components with well-defined interfaces that are implementation-independent. They separate the service interface (the what) from its implementation (the how). These services are consumed by clients that are not concerned with how these services will execute their requests. Services are self-contained and perform predetermined task, they by nature are modular and loosely coupled to the service requester. One of the powers of services and service oriented architectures is that services can be dynamically discovered without the service requester having any prior knowledge of the service or its location. Web services stress interoperability and provide a standard for the communication between applications written in different programming languages. Web services are accessible via standard network protocols such as HTTP which allow them to execute virtually anywhere in the world. The service will be the main unit of reuse for the PDG framework. In other words, the PDGs will be partitioned into various services. A single PDG will be an aggregation of these base services and a system PDG will be an aggregation of the underlying PDGs.

To help explain this partitioning of the PDG, we will explore the concept of the PDG as a fractal.

4.2.3 The PDG as a Fractal

As previously discussed, a fractal is a geometric object which can be divided into parts, each of which is similar to the original object. One of the implications of self-similarity is that each fractal must itself be a little fractal. A normal Euclidean shape, such as a circle, looks flatter and flatter as it is magnified. At infinite magnification it would be impossible to tell the difference between a circle and a straight line. Fractals are not like this. Instead, with a fractal, increasing the magnification reveals more detail that was previously invisible [60] (Figure 4.3).

The architecture of the PDG system can conceptually be thought of as a fractal. The system level PDG has the same conceptual structure as was laid out by Roach in [1].

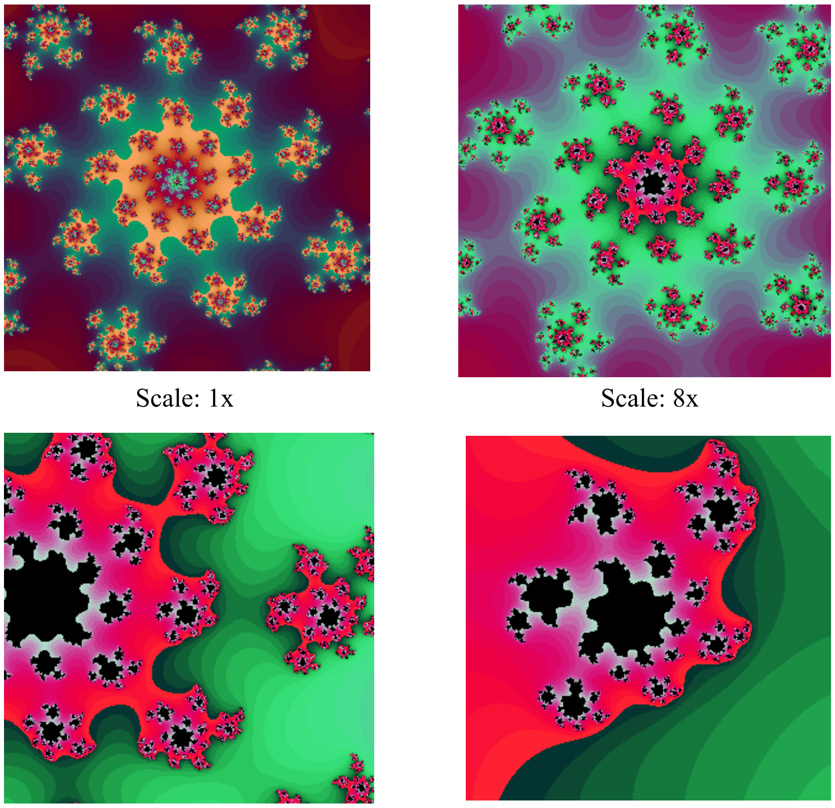


Figure 4.3 Illustration of Fractal Properties

Each of the sub-system PDGs would also have the same structure as do the components that make up the sub-system. This means that the primitive structure for the system is the PDG structure itself as defined by Roach. The system level PDG, therefore, has the C, M, K, B, A, V, T, and U sets, as well as the mappings between the sets. The subsystem PDGs are made up of those same sets. Figure 4.4 illustrates this principle. An example will help illuminate this concept. If we imagine a system level PDG for a turbofan engine family, we might find the set C to include: thrust, specific fuel consumption, weight, length and diameter. The set B would include behavioral predictions that would include these values as well as combustor temperatures, pressures, etc. The K, A, V, T and U sets would be

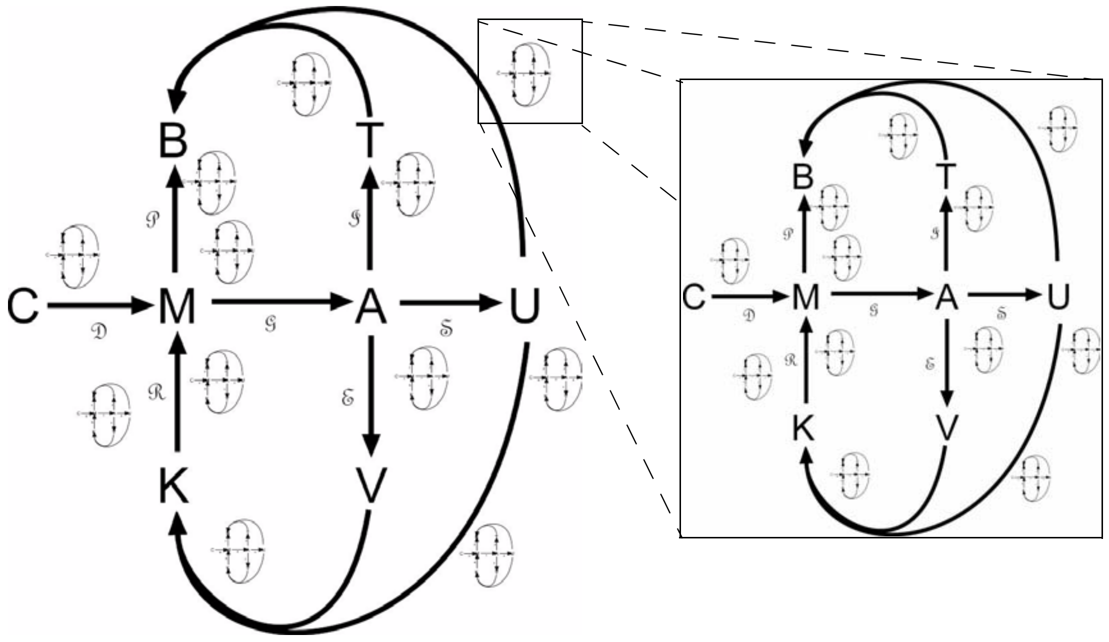


Figure 4.4 PDG Structure as a Fractal

similarly filled. A turbofan engine may be broken into various subsystems including: high pressure turbine, low pressure turbine, fan, low pressure compressor, high pressure compressor, combustor, nozzle, fuel system etc. The C set for the high pressure compressor might include: combustor temperature, efficiency, pressure ratio, etc. The high pressure turbine might further be broken down into a rotor, consisting of a turbine disk and a turbine blade. A PDG might also be defined for the turbine disk or turbine blade. As we can see, the turbine blade is as much of a PDG as the entire turbofan engine would be.

4.2.4 Patterns

Patterns are another element of reuse that will be used in the PDG system. Alexander, in his book *A Pattern Language*, provides profound insight into the design of complex systems. He observes that the design of a complex structure, like a building, can be viewed as the sequential application of a series of patterns. For example, one architectural pattern is that rooms should have light coming in from two different sides to prevent harsh shadows. Another pattern is that there should be a transition area between the street and the inside of the house. By applying these patterns to a design problem, you can create highly complex solutions to completely unique problems in a disciplined and organized way.

Patterns are elements of intellectual reuse. They have been widely used in the software community to convey solutions to commonly recurring problems and proven solutions to these problems. Patterns, then represent expert solutions to recurring problems in a context and thus are captured at various levels of abstraction and in numerous domains.

The PDG system will make use of proven software patterns and patterns specific to PDGs will be identified.

4.2.5 Process and System Reuse

Most companies' development processes are more a result of random evolution than conscious design. Such random process evolution can produce well-adapted solutions, but it is increasingly dangerous. Evolution is simply too slow when the external environment changes rapidly. Instead, we must shift to an approach of deliberate evolution, in which we analyze the process and make conscious choices.

The problem of designing processes is inherently different from most of the other design problems that we confront. In product development, we must design our process to be repeatable while still allowing new and innovative elements to be introduced into the process. The product development process must be flexible enough to adapt to changing market demand and the infusion of new technology and engineering knowledge. The product development process is one of the company's greatest assets and at the same time it is one of the most poorly managed. In order to create this flexible and responsive process the product development process itself warrants careful process design.

While flexibility is important, it must be controlled. We may generate new information during the product development process by making new mistakes, but we must protect ourselves against making the old mistakes again and again. We need to find some way to preserve what we have learned without discouraging people from doing new things. I believe that the secret to doing this is to concentrate at the right level of the process architecture.

Let us start with a familiar example. Consider the design architecture of the English language. At the letter level we have rigidly standardized on twenty six letters. Users of English cannot invent new letters, and if they did other users could not recognize them. If we move up to the word level we have about 450,000 words, of which a high school graduate might commonly recognize 40,000. We have almost complete standardization at this level. It is possible to introduce a new word into the language, but we cause some confusion when we do this. If we move up to the sentence level, there is little standardization. We have a small set of syntactic rules, such as the subject-verb-object word order, and enormous freedom everywhere else. We can produce an infinite number of well-formed, recognizable sentences in English. This infinite flexibility at the highest level of the architecture has been achieved by the standardization at the lower two levels.

This contains an important lesson for use when we design any process that needs to be flexible. Flexibility does not come from allowing all levels of the architecture to vary. This will only produce chaos. Flexibility at high levels in the architecture comes from standardization at the lower two levels in the architecture. Paradoxically, structure is the key to freedom.

The simplest approach to combining structure and flexibility is to build the development process out of modules. By altering the use and sequence of these modules we can produce millions of possible process configurations without losing control.

How can we develop such modular building blocks? One answer may be in the methods that we have discussed, namely objects, services and service oriented architectures. Web services exhibit the concept of information hiding and present a well structured external interface to the world, while preserving freedom in their internal structure. A well-planned external interface gives us a great deal of flexibility and is the key to reusing services.

When we design a development process we want to exploit the same properties. We want to create standardized building blocks that are defined primarily at their interfaces rather that by their internal procedures. If we standardize the interfaces, we can evolve the internal structure as necessary to meet changing requirements. Because the external properties are controlled, we can change internal methods and data without unraveling our entire development process. Once the inputs and outputs are defined the module owner can worry about the internal methods for the process.

What is powerful about this approach is that the choice of method is now hidden inside the module. This means that the method may be flexibly tailored to fit the needs of an individual program. Because we have standardized the external interface instead of the internal activities, we have created a module that can be reused by many projects.

We have also created an architecture that is tolerant of change, because most changes will only affect the internal structure of the module. Thus, we can easily introduce new methods without having to rewrite our development procedures. This means that modular development processes offer the desirable property of being both well-structured and flexible at the same time.

The Business Process Execution Language for Web Services (BPEL4WS) is a language for the formal specification of business processes and business interaction protocols. By doing so, it extends the Web Services interaction model and enables it to support business transactions. BPEL4WS defines an interoperable integration model that should facilitate the expansion of automated process integration in both the intra-corporate and the business-to-business spaces. The BPEL4WS specification is supported by many of the largest software developers including BEA Systems, IBM, Microsoft, SAP AG, Oracle, and Siebel Systems. IBM, for example, provides a product called *Process Choreographer* which is a BPEL4WS engine that facilitates the rapid development and deployment of business process. This new technology provides much of the flexibility and standardization needed for a continuously improving and adaptive product development process. As the name implies, the *Process Choreographer* can be used to choreograph or orchestrate the location and execution of web services. This can include the dynamic inclusion or exclusion of a particular service that represents a particular process step.

This means that we can now start to think of services as building blocks that can be reused and assembled into larger structures. This gives is the flexibility to respond to new business demands, and allows a significant degree of integration and communication for product development.

This section has discussed the basic building blocks for the PDG system. The next section details the method to break the PDG into these building blocks.

4.3 PDG Creation Process

4.3.1 Selection of the Product Concept and Embodiment

In a typical design process, the first two stages, generally concept generation and preliminary design, focus on the development of a solution context. This not only identifies the technologies to be used, but typically determines a product configuration as well. Consequently, in the development of a PDG, a solution context must be identified as the first step. Along with the chosen concept and embodiment, the best practice steps for designing the chosen product are identified so that they can be captured in a PDG that will be specific to a product class. This also includes the identification of all the design artifacts, performance predictions, knowledge, and other outputs from the design process.

For the purposes of this work, we will assume that the product concept and embodiment has been pre-determined. More detail about the selection of the product concept is found in [1].

4.3.2 Development of the Product Transformation Schematic

The Product Transformation Schematic (PTS) is a visual representation of the overall transformation function that will be used as a blueprint for the construction of the PDG. The development of the PTS is the process of defining the PDG as a transformation function and becomes a representation of the transformation function for the product development process. In it, the members of the domain and range sets for the intermediate transformations that comprise the overall product transformation are enumerated. The actual intermediate transformations between the sets, and their dependencies are identified and defined in detail. Also, the sequencing of the execution of the intermediate transformation functions is defined.

Roach proposed that the PTS be constructed in four phases. The four phases are (1) classification of product elements and intermediate transformations, (2) layout of plans for the intermediate transformations, (3) rectification of the master parameter list for the PDG, and (4) layout of the design and release cycles.

While these steps remain valid, additional steps are proposed to help in the design of a system level PDG. Generally speaking, mechanical design problems are too large to consider as a single system. Rather, they are aggregations of subsystems and subassemblies. Experience has shown that the classification of an entire system into a single set of

C, B, M, K, A, T, V, U sets is difficult and prone to error. We can imagine the number of members of the master parameter list for an entire turbofan engine when a single turbine disk was found to have hundreds of members in M. If these parameters are captured as a single list, that list will be enormous and difficult to manage. It also violates the encapsulation rules that we have discussed as being essential for reuse. The division of the master parameter list also has the advantage of apportioning the processing of the master parameter lists amongst the various services and helps to eliminate the sending of unnecessary data across the network. For these reasons, supplementary steps are necessary to help deal with system complexity.

The proposed method for service discovery is composed of the following steps: (1) recursive modular decomposition, (2) component feature decomposition, (3) recursive component aggregation, (4) classification of PDG module elements, (5) composition of the intermediate transformations, (6) rectify M for the PDG module, (7) aggregation of PDG modules into services, (8) compose services, and (9) layout the design and release cycles.

While this method shows how to decompose automated design modules into reusable services, it is important to note that experience is required to correctly decompose the system into PDG services. Just as in object-oriented design, these decomposition techniques require practice and experience. If the product is not decomposed properly into its constituent services, the penalty is the inability to reuse that service. If a mistake is made during the decomposition, the service may need to be refactored into a more reusable form.

We will now explore each of these steps in more detail beginning with recursive modular decomposition.

4.3.2.1 Recursive Modular Decomposition

The modular decomposition step refers to the breaking of the system into many smaller modules. Each module is responsible for a single, distinct function within the system. This is sometimes referred to as “top-down design”, in which the bigger problems are iteratively decomposed into smaller problems.

During this decomposition process, it is important to be acquainted with the product architecture. If a family type architecture is being used, it will be advantageous to break the PDG modules according to the product’s modular strategy. This allows the PDG to adapt easily to changes or additions to the modules used in the product family. A mismatch in this aspect, will cause headaches later in the product’s lifecycle as the product family evolves. Another reason for doing this is that the modules within a product family are usually designed to be shared and reused. This extra effort is rewarded as the modules are shared with other products within the family. If the PDG is structured to encapsulate these modules, that portion of the PDG can be used for the other products as well.

It is also preferable that those carrying out this decomposition be familiar with as many products within the company’s portfolio as possible. This allows them to assess the company’s portfolio as a whole when composing reusable services. A corporation that manufactures aircraft engines as well as turbochargers for automobiles and trucks may be able to create shared services for some types of compressor analysis. This provides a mechanism for a synergistic relationship between different divisions within a large corportation.

Every system is different and a generic all encompassing modular structure is not practical. Experience in this decomposition process, therefore, is invaluable. The modules that are chosen may not be right the first time but through experience, we will begin to see patterns. As these new patterns are discovered, it is important to document them so they may be reused.

The goal of this step is to recursively break the system into smaller and smaller pieces. We stop the anatomization of the system when we reach a logical level for that piece which many times corresponds with the part level, although it does not have to correspond to a part. This represents a component of the system. A component, however, does not represent the smallest unit, it is usually made up of a number of objects. In the turbofan system, a component that we find might be a turbine disk.

4.3.2.2 Component Feature Decomposition

Component feature decomposition is the process by which the component found in the previous step is broken into it's constituent features and parametric scheme. This is done in order to find any reusable features that may lie inside one of these components. These are typically represented by objects. We ask ourselves, can these objects stand alone?

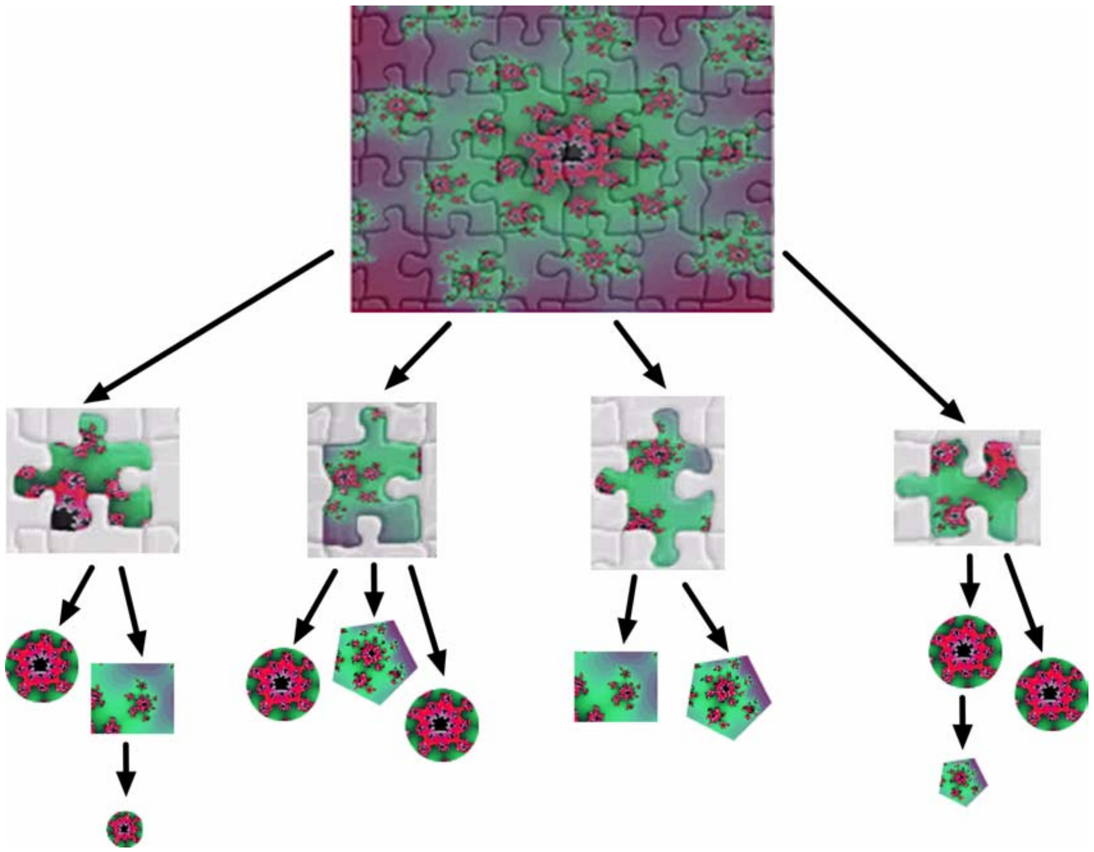


Figure 4.5 Recursive Modular Decomposition

As we look at the turbine disk component we might find an object that represents the firtree. The firtree is essentially a specialized slot in the disk into which the turbine blades are inserted. The function of this object is to hold the blade to the disk. We might spotlight this particular feature because it is a feature that we see elsewhere. It is also found in compressor disks and fan disks. The design and analysis process is essentially the same in any of these applications. In another example, we might identify the turbine blade as a component. Turbine blades exist in the system in the high pressure turbine and the low pressure turbine with some minor differences. We might find that as we break the turbine blade into objects, we recognize the airfoil object as a good candidate for reuse because it is used for turbine blades and vanes in both the high and low pressure turbine sections. If we do a really good job at defining a reusable airfoil, we may also be able to use the object in the fan etc. In these cases, it may make sense to demarcate the firtree or the airfoil as independent, ecapsulated services.

Deciding how and to what extent to decompose the system is largely based on experience and patterns observed over time. There are some questions that can be asked to determine a reasonable level for the decomposition. First, can the object or module stand alone? Does it make logical sense for the object to exist independent of any other objects or modules? Is the object modularly understandable? The modular understandability refers to the ability of a person to understand the function of the object without having any knowledge of other services. If the object is not modularly understandable, is it an object that can be used to build a more logical reusable structure? It may make sense to combine this object with another object to form a modularly understandable and reusable function. Can the object be used in more than one context? The modules should map to a distinct problem. Does the module provide a specific service to its clients? Another instance where it is important to decompose the system further is when a replaceable component is found. In this case you want to decompose the system to this level so the system has the ability to swap this particular element for another that provides a similar function.

It is important to remember that the goal of the decomposition is not to find the smallest structures for the system. The purpose of the decomposition is to find objects and modules that can be combined into larger, coarser grained services. The next step in the process is to take the objects and modules found during the decomposition and to group

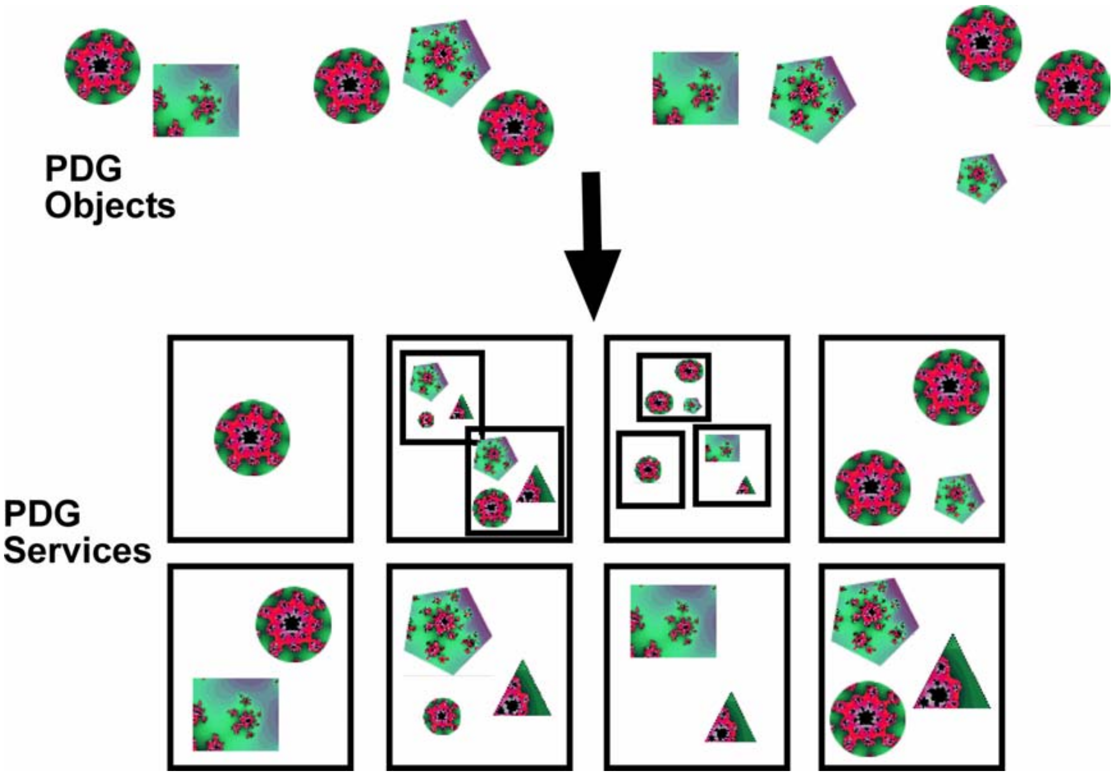


Figure 4.6 Recursive Component and Object Aggregation

them into coarse PDG services. These elements should be as coarse as possible while still maintaining flexibility and reuse.

4.3.2.3 Recursive Component and Object Aggregation

Recursive component and object aggregation refers to the conglomeration of the components and objects previously found into logical groupings. Here we are trying to assemble the components and objects into reusable structures for the PDG system. These form our lowest level of PDGs. This process is then recursively applied forming the sub-system PDGs and eventually the system level PDG. In an abstract sense each PDG can be seen as a service to the system.

There are some guidelines as we begin to demarcate these PDG services. We will discuss these guidelines one by one.

1. Modular Composability - The modular composability of a service refers to the production of services that may be freely combined as a whole with other services to produce new systems. Services should be created so that they are sufficiently independent to reuse in an entirely different context from the one for which they were originally intended. Part of this process is to identify the significant scenarios that the PDG module will need to support. This will help to describe how the PDG module will function within the system.

2. Modular Understandability - The modular understandability of a service is the ability of a person to understand the function of the service without having any knowledge of other services. The modular understandability of a service can also be limited if the service supports more than one distinct concept. The PDG services should be encapsulated at a level that promotes easy understanding of the function of that service.

3. Modular Continuity - The modular continuity of a service refers to the impact of a change in one service requiring a change in other services or in the consumers of the service. An interface that does not sufficiently hide the implementation details of the service creates a domino effet when changes are needed.

4. Direct Mapping - A service should map to a distinct problem domain function. The boundaries around the PDG service should map to a distinct area of the problem domain. This is important so the services are self-contained and independent. It is easy to pollute services interface with functions that: logically belong in another existing service,

belong in a new service, span multiple services and require a new composite service, are really internal knowledge that should not be exposed through an interface.

5. Information Hiding - The PDG service should never expose its internal data structures. Even the smallest amount of internal information known outside the service will cause unnecessary dependencies between the service and its consumers.

6. Loose Coupling - Coupling refers to the number of dependencies between modules. There are two types of coupling: loose and tight. Loosely coupled modules have a few will known dependencies. Tightly coupled modules have many unknown dependencies. A systems degree of coupling directly affects its modifiability. The more tightly coupled a system is, the more a change in a service will require changes in the service consumers. Coupling is increased when service consumers require a large amount of information about the service provider to use the service.

4.3.2.4 Classification of PDG Modules

In this phase, the results of the best-practice process for designing the particular aggregate component are classified as members of various domain and range sets for the intermediate transformations. The classifications are divided into eight major sets: customer specifications (***C***), product behavior predictions (***B***), company rules and best practices (***K***), governing master parameters (***M***), test results (***T***), product artifacts (***A***), vaulted artifacts (***V***), and final product deliverables (***U***).

This is done exactly as was defined by Roach in [1], the only difference being the scope of the effort. In this case sets are defined for each of the PDG services individually.

4.3.2.5 Composition of the Intermediate Transformations

In the PDG methodology, the intermediate transformations are defined for all of the domain and range sets. The intermediate transformations include predictive models, parametric CAD/CAE/CAM models, testing processes, delivery procedures, data vaulting procedures, parametric document models, etc. In Figure 2.1 on page 8 these intermediate transformations are represented by the straight line arrows between the various sets. For example, the set M, the governing master parameters is the domain set for the intermediate transformation which is constituted of predictive models (i.e. FEA, CFD, mathematical models etc.) that produce the range set *B*, the product behavior metrics.

Parametric models are identified to transform the various domain sets to range sets. For example, a CAD drawing is defined as an output design artifact. Thus, a parametric CAD solid model and drawing model represent the reusable transformation function that can be instantiated to produce the desired drawing artifact. Similarly, a mathematical equation may be used as a transformation function to produce predictions of product behavior from a specific set of inputs.

The second phase is to design the parametric intermediate transformations that were identified previously to produce each of the desired outputs. For design artifacts the transformation functions are reusable models capable of creating the artifacts. For behavior predictions, the transformation functions are simply the equations structured so they can be parametrically varied.

More information about this process can be found in [1] and [5].

As we begin to characterize the intermediate mappings, we must also be on the lookout for possible services that have reuse potential. At this stage, we might find two types of potential services. The first type is typically found in the *P* transformation. These are services where the analysis can be generalized enough to be readily used in more than one context. An example of this may be the encapsulation of flange sizing calculations for a PDG created for the design of industrial valves. The second type of service typically found at this stage is an integration service. These services typically provide an interface to other enterprise systems or applications used in the transformations. As we examine the the *E*, *R* and *I* mappings, for example, we see a need for an integration service to our PDM system, our knowledge management system, or our test request system. Upon analysis of the *G* and *S* mappings, we usually find integration services revolving around our CAD, CAM and document creation applications. These services provide a generic way to call these systems from any PDG component that may require the creation of a CAD model for example.

These generic integration services are usually more difficult to create because of the diversity of the potential service consumers. The benefit of creating them, however, is that they must only be created once because they can be shared by the entire system. Many of the commercial vendors of these types of tools are also now creating these types of web service interfaces as standard parts of their applications. The ubiquitous and standardized nature of web services has allowed these vendors to offer standardized programmatic a tic access to their applications. The full potential of these interfaces have yet to be seen because the tools used in typical product development do not take advantage of these standards. The good news is that we do not have to create many of these services for ourselves. The vendors are doing the hard work, we just need to create a framework that promotes their use.

4.3.2.6 Rectify M for the PDG Module

In this phase, the governing parameter for all of the intermediate transformations are gathered and rectified into a single independent list called the master parameter list. It is here that we eliminate redundant parameters and remove dependent parameters. Furthermore, identical parameters with different names must be reduced to a single parameter name. At the conclusion of this process, the master parameter list consists of a set of unique and independent parameters.

We must remember that the master parameter list referred to here applies only to the particular PDG module as we have defined it, not the entire system. Each PDG module has its own master parameter list along with its own definition for all of the other sets and mappings.

4.3.2.7 Aggregation of PDG Modules

The PDG modules are then collected into logical services. PDG services aggregate multiple PDG modules into a single interface and thus provide a coarser grained interface.

They may also be composite services. A composite service fronts other simple or composite services. The combination of functions into a composite service also gives the service consumer the ability to use each service separately or together.

The public interface for the PDG service is defined by gathering all of the members of the C, B, U sets for each of the constituent PDG modules.

At this time the services are ready to be written. The should be sufficiently encapsulated to the point where the interfaces are well defined. This allows the services to be developed independently of one another.

4.3.2.8 Compose Services

The system and subsystem PDGs are then composed and assembled from the existing PDG services. Using preexisting, tested services greatly enhances the system's quality and improves its return on investment because of the ease of reuse. Service composition refers to the ability to combine web services into a process. It is the ability to sequence and manage the conversations between web services into a larger transaction

A service may be composed in three ways: application composition, service federation, and service orchestration. An application is typically an assembly of services, components, and application logic that binds these functions together for a specific purpose. Service federations are collections of services managed together in a larger service domain. Service orchestration is the execution of a single transaction that impacts one or more services in an organization. It is sometimes referred to as a business process.

Because services communicate with one another across the network, it is important to design services to reduce unnecessary network chatter. This is done by using coarse public interfaces to the service. A service may still be implemented as a set of fine-grained objects, but the objects themselves are not accessible over a network connection. A service implemented as objects has one or more coarse-grained objects that act as distributed facades. These objects are accessible over the network and provide access to the internal object state from external consumers of the service. However, objects internal to the service communicate directly with each other within a single machine, not across a network connection.

4.3.2.9 Layout the Design and Release Cycles

The final phase in the construction of the PTS is the layout of the design and release cycles that must be instantiated with values to generate a specific design. Formalizing the design process is similar to scripting a movie, so it is referred to as storyboarding.

The storyboard is divided into two parts, corresponding to the design cycle and the release cycle. The goal of the storyboarding effort is to control the process by which values for the parameters in the master parameter list are determined and to control the sequence of execution of the PDG services. We begin the storyboard by choreographing the execution of the various services for the design and release cycles. This gives us the sequence of execution for each of the cycles including any iterations that may occur in the design cycle.

With the sequencing of the services complete, we now begin a separate storyboard for the look and feel of the user interface. It determines how the user enters the data for the C set and how they see the data contained in the B, A, and U sets. The screens that the user sees may not necessarily coincide with the services that are executed in the background. In fact, it is important that the UI is not coupled to the logic executed behind it. This will provide for better maintenance and reuse of the UI elements.

4.4 Summary

In order to divide the PDG system, we must define a unit of partition. The method explained above utilizes three different units of partition to achieve the optimal level of reuse. These units are the PDG object, the PDG module and the PDG service.

A general method for PDG decomposition was presented which builds on the methods developed by Roach in his dissertation.

CHAPTER 5 RESULTS

The method developed in the previous chapter will be implemented in two different examples. The Atmospheric Resistor and the turbofan engine systems will be used to show the application of the methods previously described. The architecture for a generic framework for PDG systems will also be shown.

5.1 Architecture of the PDG Framework

A generic framework was developed for interconnected systems of PDGs. This framework allows for “plug and play” additions to the services and allows for the easy addition or reordering of the user input screens. It is designed to be the base for future PDG implementations. This framework is based on industry standard patterns and technologies and provides application scalability. It is made up of four different layers: (1) the presentation layer, (2) the business logic layer, (3) the service layer, and (4) the data access layer. Each of these layers will be discussed in the sections that follow.

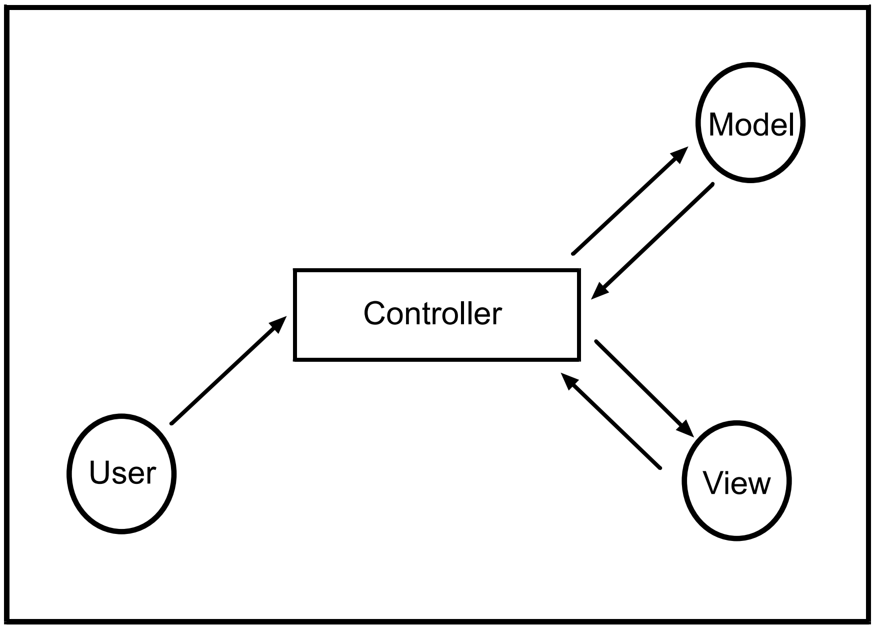


Figure 5.1 Model-View-Controller Pattern

5.1.1 The Presentation Layer

The presentation layer uses a standard Model-View-Controller (MVC) architecture. The MVC architecture is made up of three core components which allow the separation of the user interface and the business logic that it executes. The first of these components is the controller. When a user makes a request from the User Interface (UI) it is always intercepted be the controller. The controller acts as a traffic cop, examining the user's request and then invoking the logic necessary to carry out the requested action. The execution logic for a user request is encapsulated in the model. The model executes the business logic and returns the execution control back to the controller. Any data to be displayed to the user will be returned by the model via a standard interface. The controller will then look up how the data returned from the model is to be displayed to the end user. The code responsible for formatting the data to be displayed is called the view. When the view finishes formatting the output data returned from the model, it will return execution control to the controller. The controller, in turn, will then return control to the user.

The MVC pattern is a powerful model for building applications. The code for each screen in the application consists of a model and a view. Neither of these components has explicit knowledge of the other's existence. These two pieces are decoupled via the controller, which acts as an intermediary between these two components. At runtime, the controller assembles the required logic and the view associated with a particular request.

New functionality can be introduced into the application by writing a model and view and then registering these items to the controller of the application. If for example you want to provide a PDF format of the requested data in addition to the HTML format, you would only need to create a new view.

A MVC based framework offers a very flexible mechanism for building applicaitons. However, building a robust MVC framework infrastructure requires time and energy. For the purposes of this framework, we are able to take advantage of an existing MVC framework called Struts. Struts is an open source MVC framework developed by the Apache Software Foundation. It is readily available and generally accepted as a robust implementation of the MVC architecture.

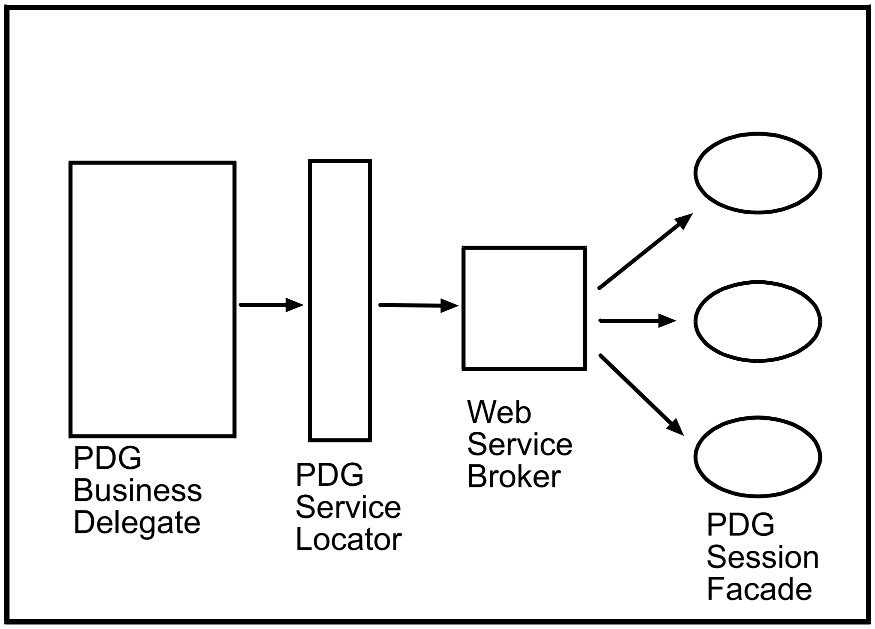


Figure 5.2 PDG Business Logic Layer

5.1.2 The Business Logic Layer

The business logic layer is made up of four distinct components. This layer implements the PDG's business logic and provides access to the various PDG services.

The first of these components is the PDG business delegate. The PDG business delegate hides the complexity of instantiating and using the PDG services from the application consuming the services. It is designed to hide the complexity of remote communication with PDG services that may exist on many different machines distributed throughout the network. It also promotes reusability by abstracting the service from the service requestor. This abstraction effectively decouples the service from the service consumer.

The second component is the service locator. The service locator is designed to hide the details of looking up the service. The service may be implemented as a web service or an Enterprise Java Bean (EJB). The service locator provides a standard interface for service lookup regardless of how the service is actually implemented. It allows the PDG business delegate to transparently locate services in a uniform manner.

The third component is the web services broker. The web service broker is a course grained service that is also implemented as a web service. It functions to coordinate interactions among one or more PDG services, aggregates responses and provides transactional support for the PDG services.

The final component of the business logic layer is the session facade. The session facade is a coarse grained wrapper around finer-grained pieces of code. It is mainly used to wrap functionality that is not implemented as a web service.

5.1.3 The Service Layer

The service layer is the layer where the actual PDG services reside. It is essentially stand alone in the sense that the services exist on their own, independent of any particular application. The service layer provides the execution environment for the individual services.

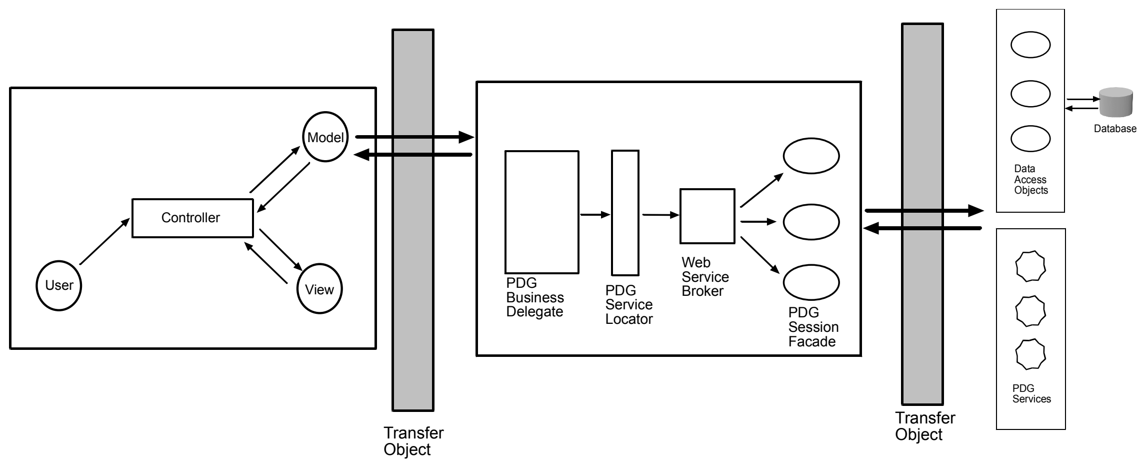


Figure 5.3 Generalized Architecture of the PDG framework

5.1.4 The Data Access Layer

The data access layer encapsulates the manipulation of data stored in a persistent database. Access mechanisms, supported APIs. and features vary among vendors of Relational Database Management Systems (RDBMS). Even with a single API, the underlying implementations might provide proprietary extensions in addition to standard features.

Mingling persistence logic with application logic creates a direct dependency between the application and the persistence storage implementation. These code dependencies make it difficult and tedious to migrate the application from one type of data source to another. When the data source changes, the components must be modified to handle the new type of data source.

The solution implemented in the PDG framework is the Data Access Object (DAO). The DAO is used to abstract and encapsulate all access to the persistent store. The DAO also manages the connection with the data source to obtain and store data.

5.1.5 Transfer Objects

Data is passed between these layers in the form of Transfer Objects (TO). Transfer objects are used to carry multiple data elements across a layer at the same time. A TO is designed to optimize data transfer across the layers and the network. Instead of sending or receiving individual data elements, a TO, contains all the data elements in a single structure required by the request or response.

5.1.6 Summary

This section outlines the architecture for a PDG framework that enables an interconnected system of PDG services. This base framework is generic enough to form the base for future PDG systems. The next section describes two example implementations of the techniques and methods described in this thesis.

5.2 Example Implementations

5.2.1 Implementation of the Atmospheric Resistor

The original Atmospheric Resistor PDG was built as a stand alone Visual Basic application. In this state, it does not have the ability to participate in a higher level system nor does it allow the reuse of the functionality within the PDG. We will show that by restructuring the PDG methodology, we can achieve an interconnected system of “little” resistor PDGs which provide greater potential for reuse. We will also show that these PDG services can then be accessed and used in different contexts.

5.2.1.1 Selection of the Product Concept and Embodiment

In this case, the problem that this product is trying to solve is to reduce the noise produced by gas venting into the atmosphere. This can be done with a silencer or an atmospheric resistor. For this example, we choose the atmospheric resistor as the product concept.

An atmospheric resistor is a fixed orifice "valve" that controls noise by controlling the expansion rate of the gas as it is released to the atmosphere. Resistors are used on a variety of oil and gas production facilities, pulp and paper mills and other process plants. The expansion of the gas is controlled by forcing the gas through a tortuous and circuitous path through a stack of thin disks. Figure 5.4 shows a typical resistor configuration.

5.2.1.2 Recursive Modular Decomposition

The purpose of recursive modular decomposition is to break the resistor system into smaller modules. We will continue to decompose the system into its constituent components until we reach a logical level.

Figure 5.5 shows how the resistor system was broken into modules. The six main modules chosen were the lifting attachment, the bottom flange, the top flange, the shroud, the disk stack and the standard parts. The shroud was broken down further into the shroud

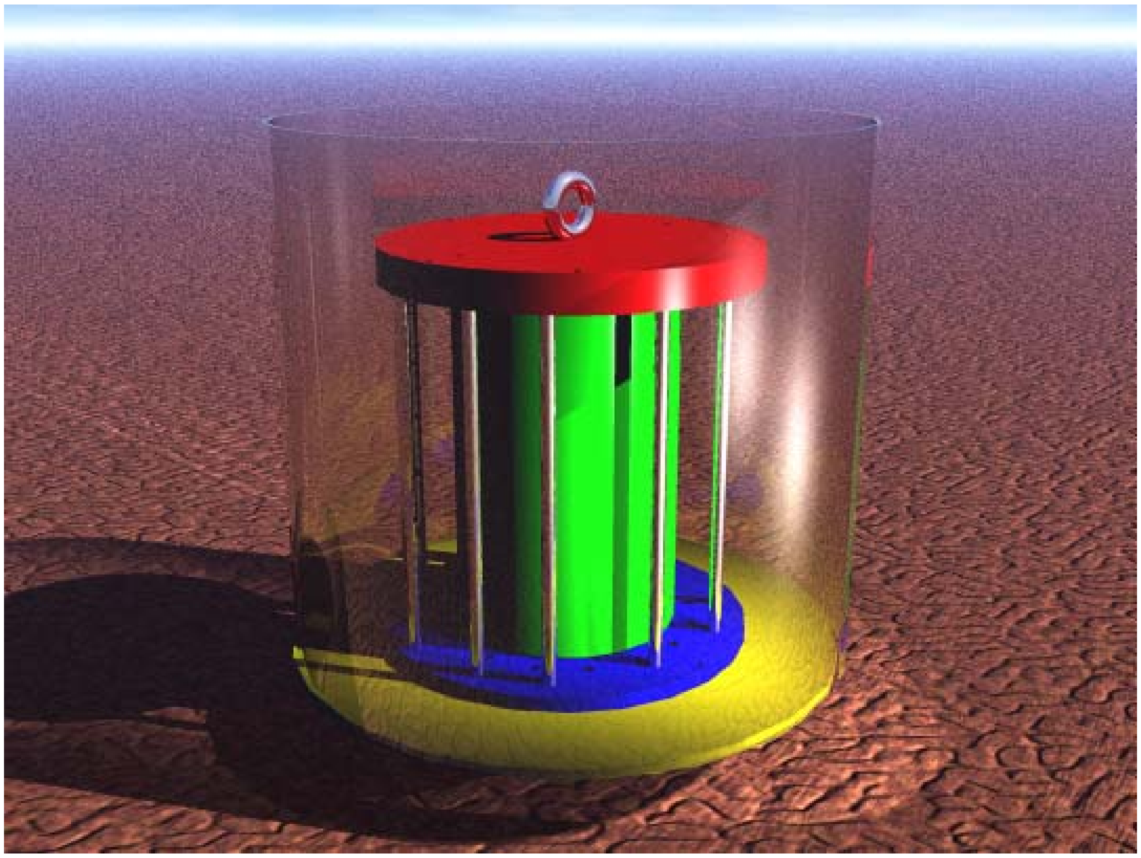


Figure 5.4 Atmospheric Resistor

bottom plate and the shroud cylinder. The disk stack was decomposed further into the top disk, the bottom disk, and the separator.

5.2.1.3 Component Feature Decomposition

In component feature decomposition we break the component found in the previous step into its features and parametric scheme. As we do this we try to identify any reusable features or parameterizations that may occur within these components. These are represented as objects.

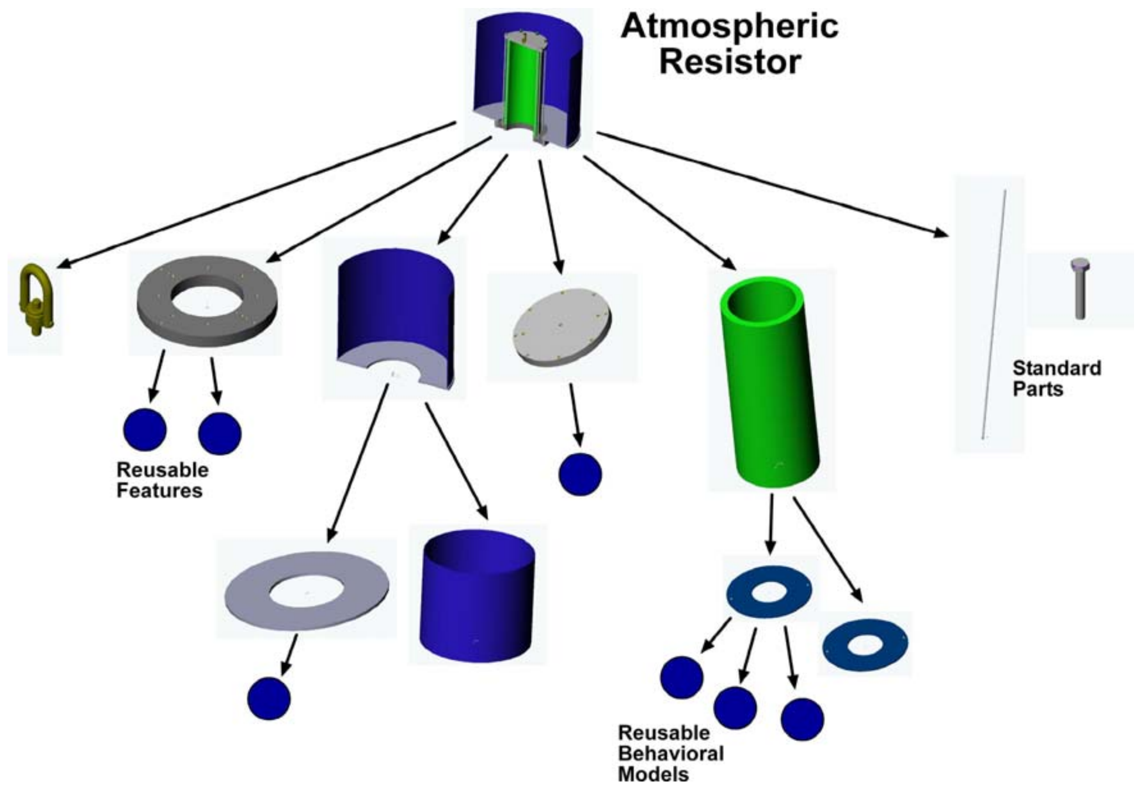


Figure 5.5 Modular Decomposition of the Atmospheric Resistor

In the atmospheric resistor we can identify a number of reusable objects. In the bottom flange, for example, we pinpoint a reusable flange feature, which represents the flange to connect the resistor to its upstream piping. Our knowledge of the product line tells us that this type of feature is used on many of the valves in our product line and therefore has reuse potential. The welded joint feature on the bottom flange also represents a potential reusable feature for the same reason. We also find a bolt hole circle feature in three of the modules. We find the expansion pattern in the disks that make up the disk stack is also used elsewhere.

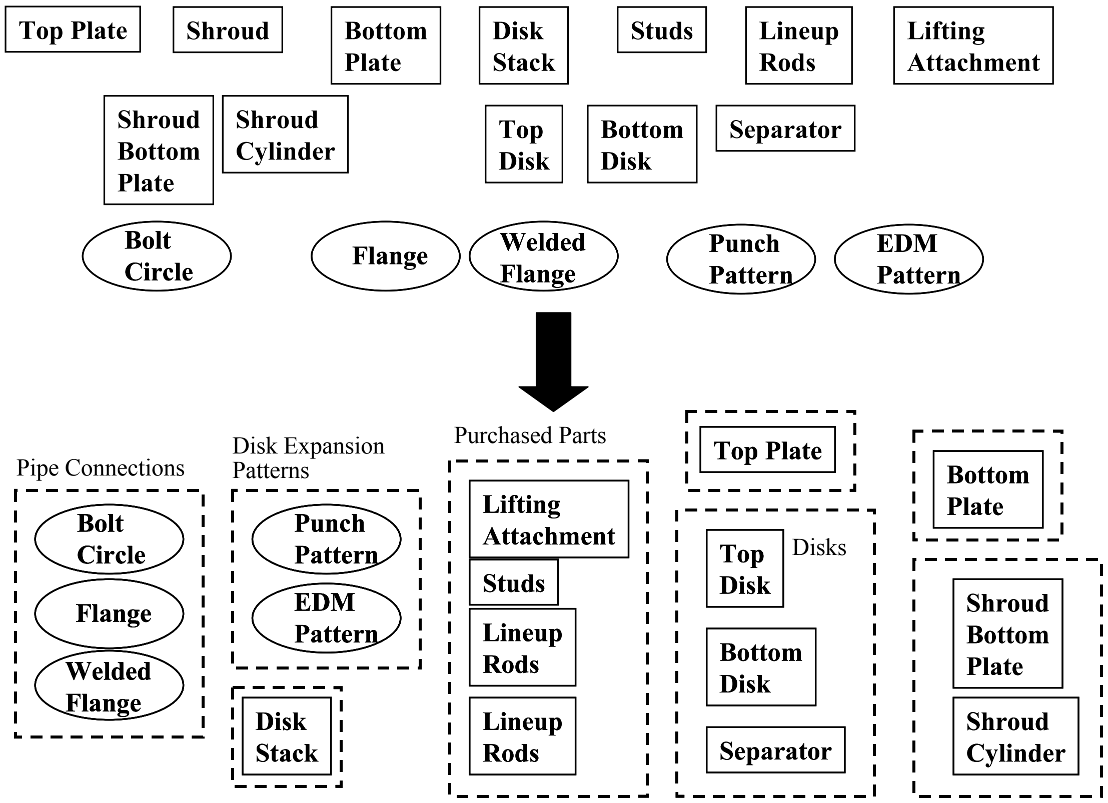
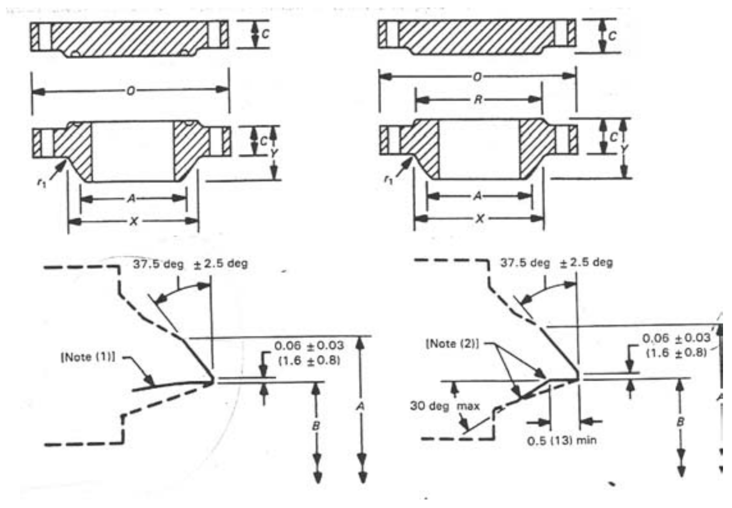


Figure 5.6 Recursive Component and Object Aggregation

5.2.1.4 Recursive Component and Object Aggregation

In this step we amalgamate the components and objects into logical reusable structures. Figure 5.6 shows how the components and objects were grouped together. They were combined into a pipe connections module, a disk expansion patterns module, a purchased part module, a top plate module, a bottom plate module, a shroud module, and a disk stack module. The pipe connections module, for example, contains the elements necessary to design the connections to the inlet and outlet piping for a valve. These groupings promote reuse across the product line and provide the ability to distribute the design of the resistor across the network.



M

- Flange OD
- Min Flange Thickness
- Hub Length
- Top Hub Dia
- Raised Face Dia
- Dia Bolt Circle
- # Bolt Holes
- Dia Bolt Holes
- Min Fillet Radius
- Weld Angle1
- Weld Angle2
- …

C

- NPS
- Schedule
- ANSI Class
- OD
- ID
- Thickness
- Material
- Temperature
- Pressure
- Welding Method

K

- Buttweld Dimensions ASME B16.47
- Flange Dimensions ASME B16.25

B

- Pipe Wall Thickness

A

- Flange Feature Model
- Flange Interface Dwg.
- Buttweld Feature Model
- Buttweld Interface Dwg.
- Interface Document
- NC Code

U

- Flange Feature Model
- Flange Interface Dwg
- Buttweld Feature Model
- Buttweld Interface Dwg.
- Interface Document
- NC Code

Figure 5.7 Classification of Pipe Connection Module into its Constituent Sets

5.2.1.5 Classification of PDG Modules

In this step the best practices for each of the defined modules are classified into the C, B, K, M, T, A, V, and U sets. Figure 5.7 shows the pipe connection module broken down into its sets. This is done in exactly the same way as Roach proposed in his work, the only difference being the scope. In order to take advantage of system modularity, the classification exercise is done for each module individually. This same exercise would be applied to the disk module, the purchased parts module, the bottom flange module etc.

5.2.1.6 Composition of the Intermediate Transformations

The intermediate transformations are used to map between the domain and range sets. They include predictive models, parametric CAD/CAE/CAM models, parametric document models etc. In the pipe connection module, for example, a CAD drawing is defined as an output design artifact. Thus, a parametric CAD solid model and drawing model represent the reusable transformation function that can be instantiated to create the desired design artifact. The mapping between the set K and the set M, known as the rules mapping (R), maps the dimensions for the pipe connections to the master parameter list using the C set as input. This is implemented as a simple lookup from a database table since these dimensions are defined by ANSI and ASME standards. The mapping between the set M and the set B, known as the P mapping, is defined as a simple calculation using the master parameter list as input and calculating the wall thickness of the pipe.

As was previously mentioned, we must be on the lookout for possible services with reuse potential during this phase. If we look carefully, we can find a number of reusable services that could be created for the pipe connection module.

The first example of a reusable service is found in the mapping to the CAD model artifact. We know that this requires an integration with our CAD software. Experience has shown that this type of integration service can readily be reused in multiple PDG contexts. Nearly all PDGs require some sort of CAD output in the form of drawings or solid models. With this in mind, we identify a potential integration service that could be developed as an interface between the PDG system and the CAD system. This service must only be developed once. Once it is developed any element within the PDG system can use it whenever a CAD artifact is required.

A second example of a reusable service is found in the R mapping. In this case, the R mapping is a simple lookup from a set of tables compiled by the ANSI and ASME standards organizations. These standards are designed to ensure that the dimensional characteristics of pipe manufactured by different vendors were the same. Since all connections to the resistor are made to piping that meets these standards, we can use tables to look up the dimensions for the standard flange or welded connections to this piping. These lookups are identical for a resistor, or any of the other control valves in this particular manufacturers product line. A service is therefore identified to perform these lookups.

Another example of a reusable service is in the mapping that maps the master parameter list to the Interface Document artifact. This is a report document that contains the information that specifies the interfaces to the resistor and how it should be joined to the upstream piping. The Interface Document is typically a Word document made up of parametric text and images. The document is instantiated for each design to provide the interface information germane to that particular design. We have seen that nearly every PDG requires some sort of document creation mechanism. This experience has shown that an integration service to integrate Word into the PDG system would be valuable. As with other integration type services this would only need to be created once and may be provided by the vendor of the application.

5.2.1.7 Rectify M for the PDG module

At this stage, the master parameter list is reconciled to eliminate redundancies and dependencies. At the conclusion of this step, we have a single list of unique and independent parameters.

5.2.1.8 Aggregation of PDG Modules

The PDG modules found in the previous steps are collected and united under a single interface. This creates a composite service which is made up of two or more finer grained services. This composite service simply becomes a single interface point for more than one related services.

In the resistor PDG system, we may decide to provide a composite service which includes the disk stack and the disk services. This does not mean that they no longer exist as individual services. It simply means that, for the sake of simplifying the interface point, we choose present these as a single service because they will be used together the majority of the time. If we need to access them individually, we are still able to do so.

5.2.1.9 Compose Services

The services can now be built and then composed and assembled into the system and sub-system applications. Because the services are completely encapsulated, they may be created independently of one another and individually registered in the UDDI.

In simple terms, the web service is created by developing the code that defines the functionality of the service and describing this functionality in a WSDL document. The

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://threadengage.pdg.cci.com"
xmlns:impl="http://threadengage.pdg.cci.com" xmlns:intf="http://thre
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <wsdl:types>
  <schema elementFormDefault="qualified"
targetNamespace="http://threadengage.pdg.cci.com"
xmlns="http://www.w3.org/2001/XMLSchema" xmlns:impl="http://threaden
xmlns:intf="http://threadengage.pdg.cci.com"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <complexType name="PrelimEngageVO">
    <sequence>
     <element name="a" type="xsd:double"/>
     <element name="dmi" type="xsd:double"/>
     <element name="dpe" type="xsd:double"/>
     <element name="tpi" type="xsd:double"/>
    </sequence>
   </complexType>
   <element name="calcShearAreaExt">
```

Figure 5.8 Sample WSDL Document from Resistor PDG

WSDL document tells the service consumer what services are available and how to access them

The source code and WSDL file for a simple Web Service are found in Appendix A. This is a simplified version of a service to illustrate the mechanics of creating a Web Service. Further information on Web Services can be found in [30].

The services can then be plugged into the framework and composed into a higher level application.

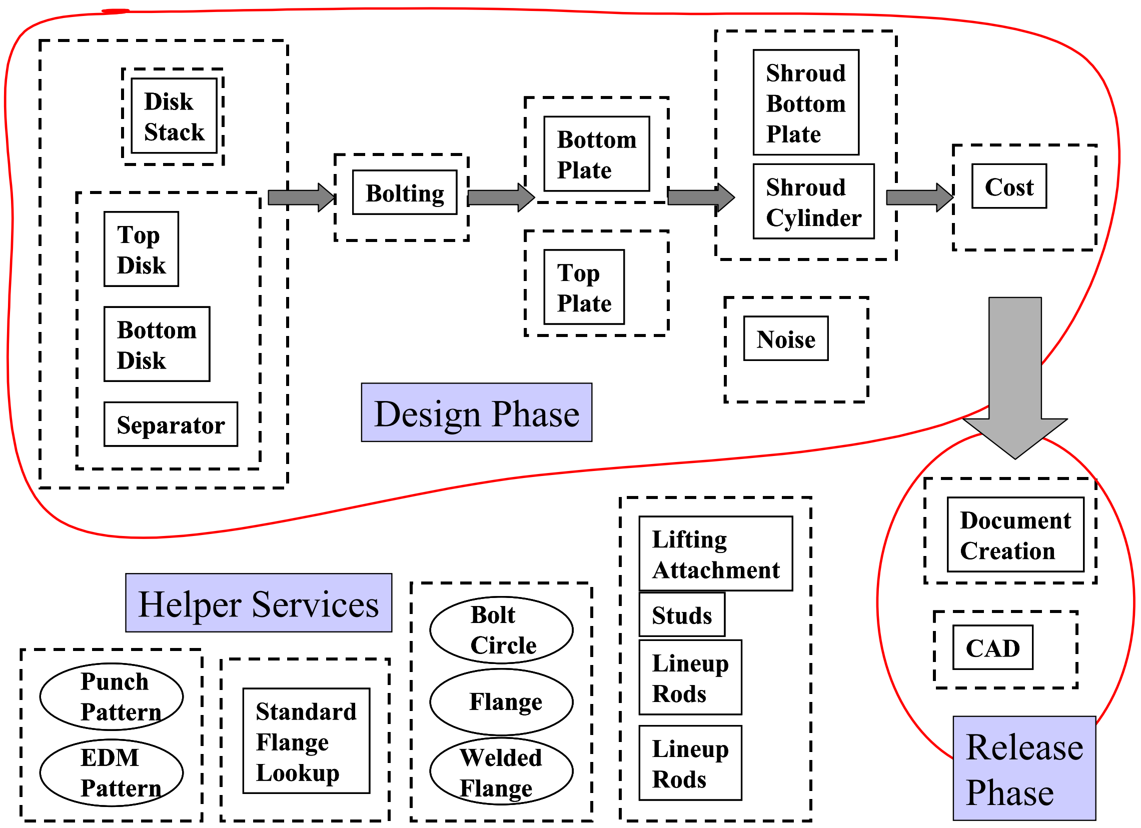


Figure 5.9 Service Choreography

5.2.1.10 Layout the Design and Release Cycles

We begin the storyboard by choreographing the execution of the PDG services. Figure 5.9 shows an example of service choreography. In its most simple sense service choreography is the sequencing of the services that the application needs to call in order to provide the required functionality. The diagram shows the design and release phases as well as the helper services. The helper services encapsulate shared functionality needed by the main services but are shared amongst the main services.

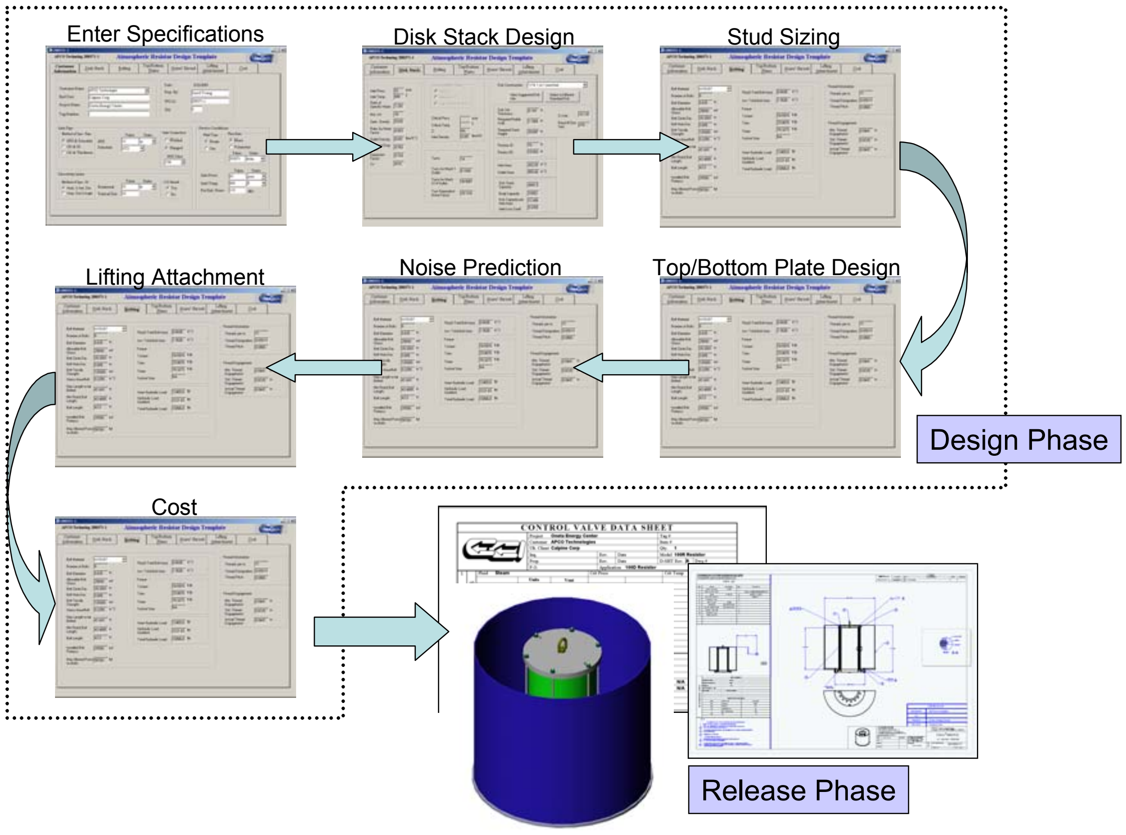


Figure 5.10 Resistor User Interface Storyboard

With the sequencing of the services complete, we create a separate storyboard for the look and feel of the user interface. This is mainly to determine how the user interacts with the application and how the resulting data is presented. Figure 5.10 shows the storyboard for the user interface portion of the application.

5.2.1.11 Results

The atmospheric resistor system described above has shown that a PDG can be broken up into individual services and those services can take part in an interconnected

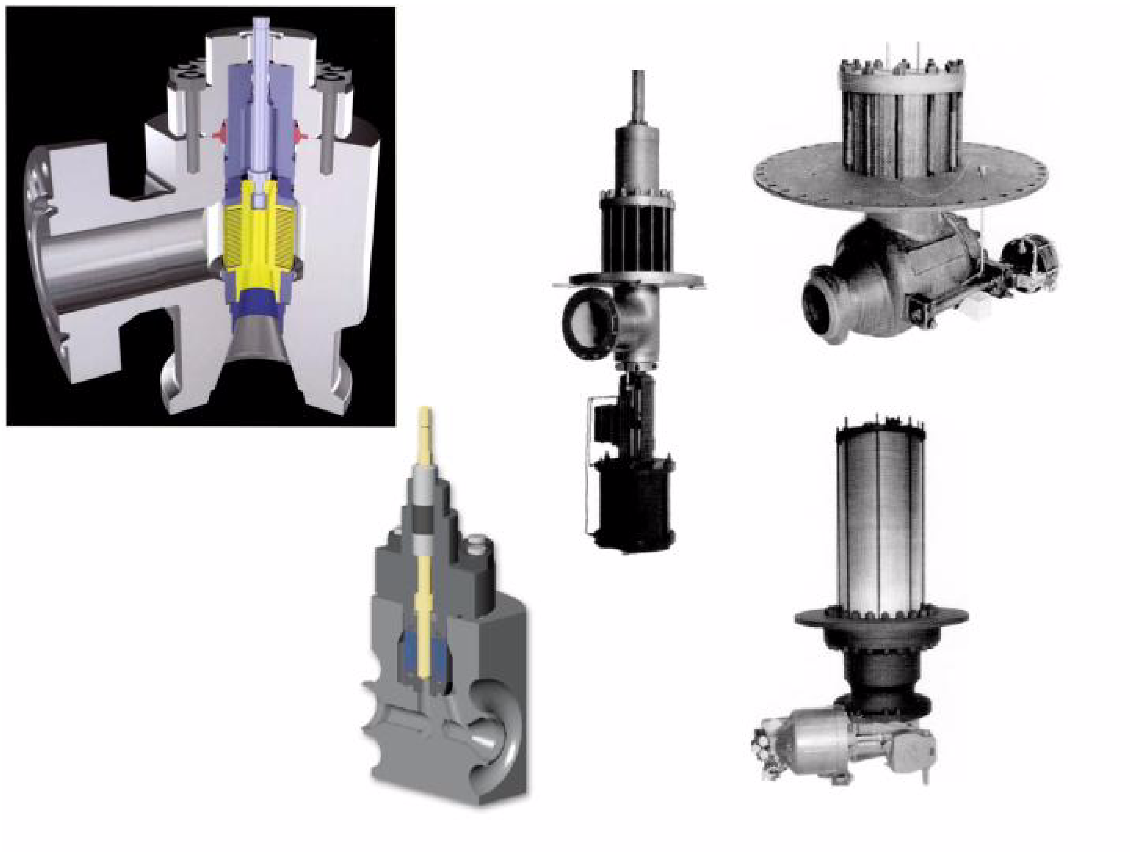


Figure 5.11 Other Products Where Resistor PDG Services Could Be Used

system of other PDGs. The services provide a much more reusable framework for PDG creation and allow PDGs to be built from existing and pre-tested PDG services.

The PDG services can also be used in the context of other product lines. Figure 5.11 shows some of the other products where these same resistor PDG services could be used. These products include choke valves, valves for atmospheric venting, valves for boiler feed pump recirculation as well as sprinter valves. While each of these products is different, there are some very important similarities that can be used to our advantage. Some of these similarities are due to a product family architecture that was not explicitly designed but rather came from the use of the same technology across these products. Other similarities come from the fact that in general all valves are performing a similar function although it may be under vastly different conditions. It is easy to see, therefore, how a service based environment for product development can create an interconnected system of shared PDG services which can be used over and over again. Many of the services created in the resistor PDG system are directly applicable to other products.

The integration services can obviously be used for these other products since they provide access to needed applications. The disk stack services are especially useful since all of these products use a similar disk stack technology to control fluid flow. This particular company currently has many duplicate copies of the disk stack calculations in various spreadsheets and other stand alone programs. Each of these programs and spreadsheets must be maintained separately in their current paradigm. We have seen that there are significant advantages that could be seen if the disk stack portion of these spreadsheets and programs were to be consolidated into a PDG service. The pipe connection service would also be reused across these products as they all must be joined to the piping systems of the process plant. Other services such as the thread engagement service would prove useful in the PDG system as they too would be used over and over again. Since many of the purchased parts such as nuts, bolts, rods, actuators etc., are used in every product, the purchased part service would provide a much needed standardized lookup service for these parts.

The example of the Atmospheric Resistor PDG system shows that the method developed does indeed provide a valuable technique for building a better PDG infrastructure. This method provides the ability to reuse PDG modules in many different contexts without recompiling or even touching a single line of code. This has been shown to be the key to building a system level PDG without creating a single all ecompassing PDG program. The functionality is encapsulated and can be combined into the relevant process at runtime. The encapsulation provided by this method allows for easier maintenance of the system and provides the ability to distribute processing. This distributed processing is takes advantage of the processing power of all of the computers attached to the network and also provides a mechanism to divide the work geographically.

5.2.2 Implementation of the Turbofan Engine

The turbine disk PDG, which sparked many of the ideas for this work, proved to be a much more complicated problem. The turbine disk PDG was designed to improve upon many of the weaknesses shown in the resistor PDG. It was built as a web based application to take advantage of economies in license usage for expensive 3rd party software and takes advantage of a server based environment. It too was shown to provide significant reductions in design cycle time, allowing the engineer to focus on the more value added activities in the design process. It also helped to eliminate the need to run intermediate reduced order models because of the speed with which the higher level models could be built and executed. The turbine disk also provided the ability for an engineer to leverage the entire body of company knowledge.

Notwithstanding these significant advantages, several difficulties were encountered in the development of the turbine disk PDG that were not seen with the Atmospheric

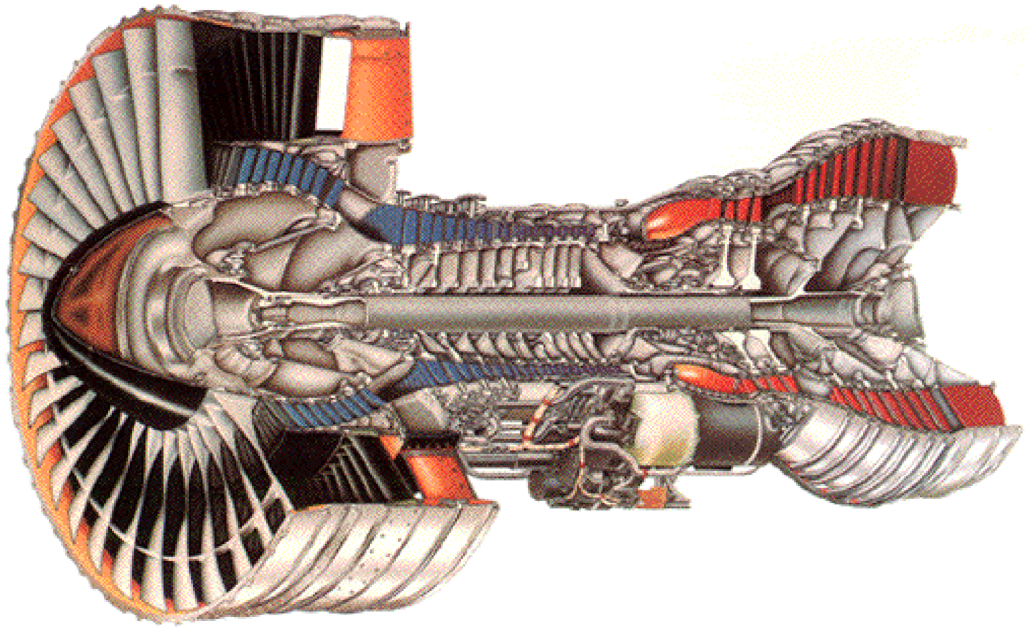


Figure 5.12 The Pratt and Whitney PW4000 Turbofan Engine

Resistor. The turbofan engine is a much more complicated product than the resistor is. Not only are the analytical models more difficult and complex but the system level interactions are orders of magnitude more difficult. A need was identified for the turbine disk to be able to take part in a higher level system.

The implementation of the turbine disk, front frame, fan, vane and turbine blade PDGs will be used to illustrate an interconnected system of PDG services.

5.2.2.1 Selection of the Product Concept and Embodiment

The product concept chosen for this exercise is that of a turbofan engine. These engines are typically used on commercial aircraft and other business and commuter type aircraft. A typical configuration for a turbofan engine is shown in Figure 5.12. The main part of a turbofan engine is made up of a fan, a compressor, a combustor and a turbine.

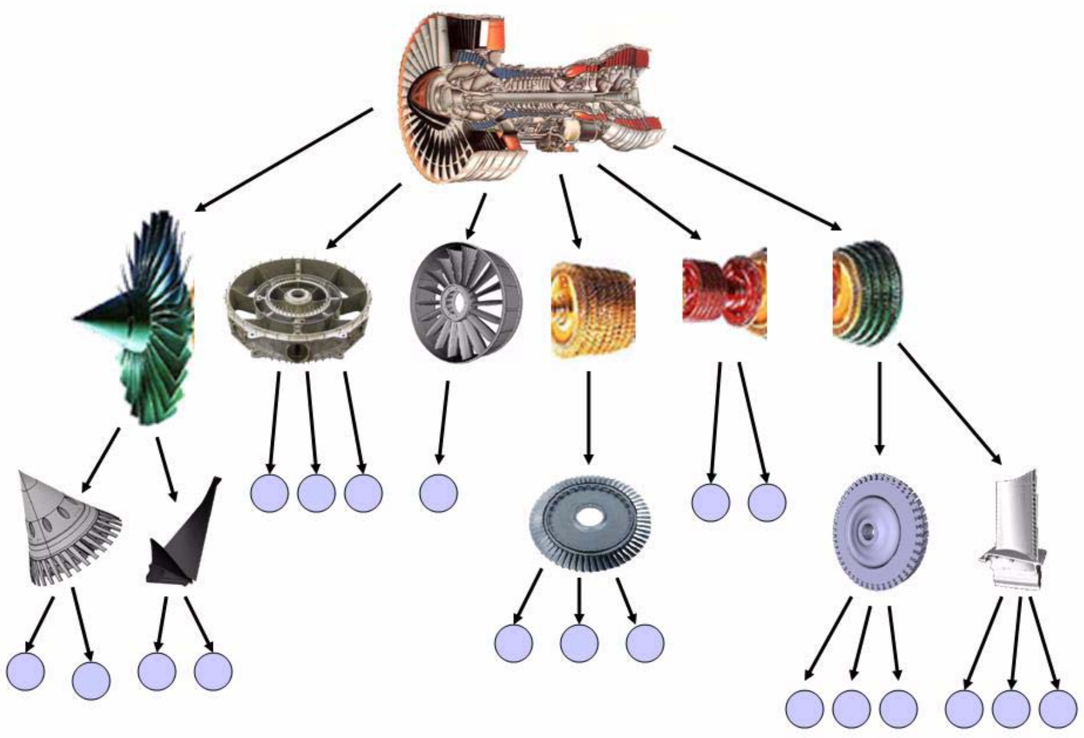


Figure 5.13 Simplified Decomposition of a Turbofan Engine

5.2.2.2 Recursive Modular Decomposition

Figure 5.13 and Figure 5.14 are simplified graphical representations showing how the turbine engine system was broken down into modules. While the turbofan engine is made up of thousands of parts, we will only consider five of the major engine modules in this discussion. These modules are the fan, the front frame, the compressor, the combustor, and the turbine.

The fan module can be further broken down into its constituent components. The fan blade, the fan disk and the fan stator all make up the higher level fan module.

The Front Frame is a single entity and will not be broken down any further at this time. It will, however, be decomposed into objects when we reach that step.

The compressor module can be broken down into the low pressure section and the high pressure section. Each of these sections is made up of a number of rotors. The rotors are made up of compressor disks, compressor blades, and in some cases a single entity called a centrifugal compressor.

A combustor may be further decomposed into a number of sheet metal components that make up the combustor casing and fuel nozzles.

The turbine section is very similar in structure to the compressor section. It is made up of a high pressure turbine section and a low pressure turbine section. Each of these sections is also made up of a number of rotors, each of which could be made up of a turbine disk and a turbine blade.

While there are many more components that make up these turbofan engine modules, we will consider only these for the sake of simplifying the discussion. These components will be further broken up into their corresponding objects in the next section.

5.2.2.3 Component Feature Decomposition

In the previous section, we broke the fan module into the fan blade, the fan disk and the fan stator. Using experience, we would break the fan blade into an airfoil object, an attachment object, and a platform object. The fan disk may be made up of a disk object,

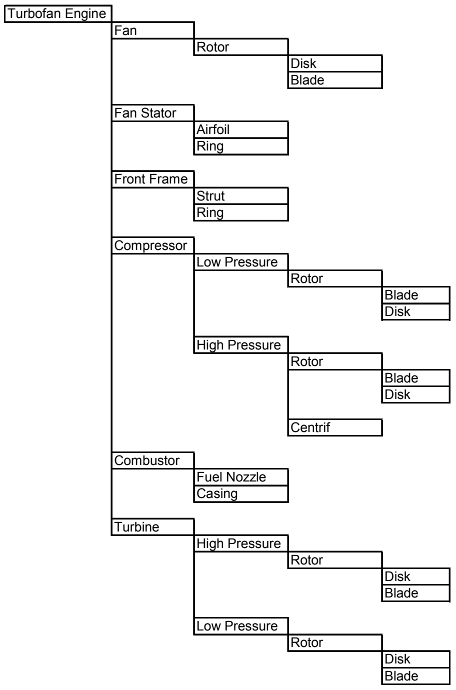


Figure 5.14 Tree Representation of Engine Decomposition

an attachment object, and a flange object. The fan stator is made up of an airfoil object, a hub object and an outer ring object.

The front frame module was not broken down any further in the previous step. There are, however, some potentially reusable objects contained within the front frame. We would consider the regular strut and the king strut to be objects. These objects would be made up of airfoil objects and internal passage objects. The core struts, the hub, the bypass ring, the outer ring and the mount pads are also designated as objects.

The compressor section was divided into the low pressure section and the high pressure section. Each section was made up of a rotor which contains a disk and a blade. The disk would be broken down into a flange object, a curvic object, and an attachment object. A compressor blade might be divided into and airfoil object, an attachment object, tip shroud object and a platform object. A centrifugal compressor could be divided into a hub object and a blade object.

The combustor section was broken into the casing and the fuel nozzles. It could be divided further into objects that represent the sheet metal pieces that comprise the combustor casing.

The turbine module was divided into high and low pressure sections, each made up of a number of rotors. The rotors were further broken into a turbine disk and a turbine blade. The disk would be comprised of the disk object, the curvic object, the flange object, the attachment object, the discourager object, and the anti-rotation slot object. The blade would be broken down into the airfoil object, the attachment object, the platform object and the tip shroud objects.

5.2.2.4 Recursive Component and Object Aggregation

We now can take the results from our previous two decomposition activities and join together the modules and objects into logical reusable structures.

We will combine all of the airfoil objects into a single module. Even though the airfoil parameterizations may be slightly different, this functionality will be centralized into a single PDG module. The disk objects will similarly be grouped into a single PDG module as will the platform object. The attachment objects for the disks and the attachment objects for the blades will also be consolidated into a single module despite their differences. In this case the design of the disk attachment is so tightly coupled to the design of the blade attachment that we will aggregate them into a single PDG module. We will also combine the disk flange objects and the curvic objects into a single disk coupling module.

The hub object and the outer ring objects found in the fan stator will be joined back together under the fan stator module. While this may seem odd that we would join these objects after decomposing them into individual objects, the process of breaking the modules apart allowed the discovery of a reusable airfoil object that was previously hidden from view. The hub and outer ring objects will be combined under the fan stator module.

The three different types of strut objects from the front frame module will be compiled into a single strut module, this strut module will in turn be combined into an aggregated service with the internal passage module and the mount pad module.

The hub and blade object for the compressor will be combined in a service aggregation as will the sheet metal objects that make up the combustor casing. The disk module will contain the discourager object, the anti-rotation slot object and will be a client to the coupling service and the attachment service.

With the aggregation of the modules and objects complete, we are now ready to classify the intermediate transformations for each of the PDG services.

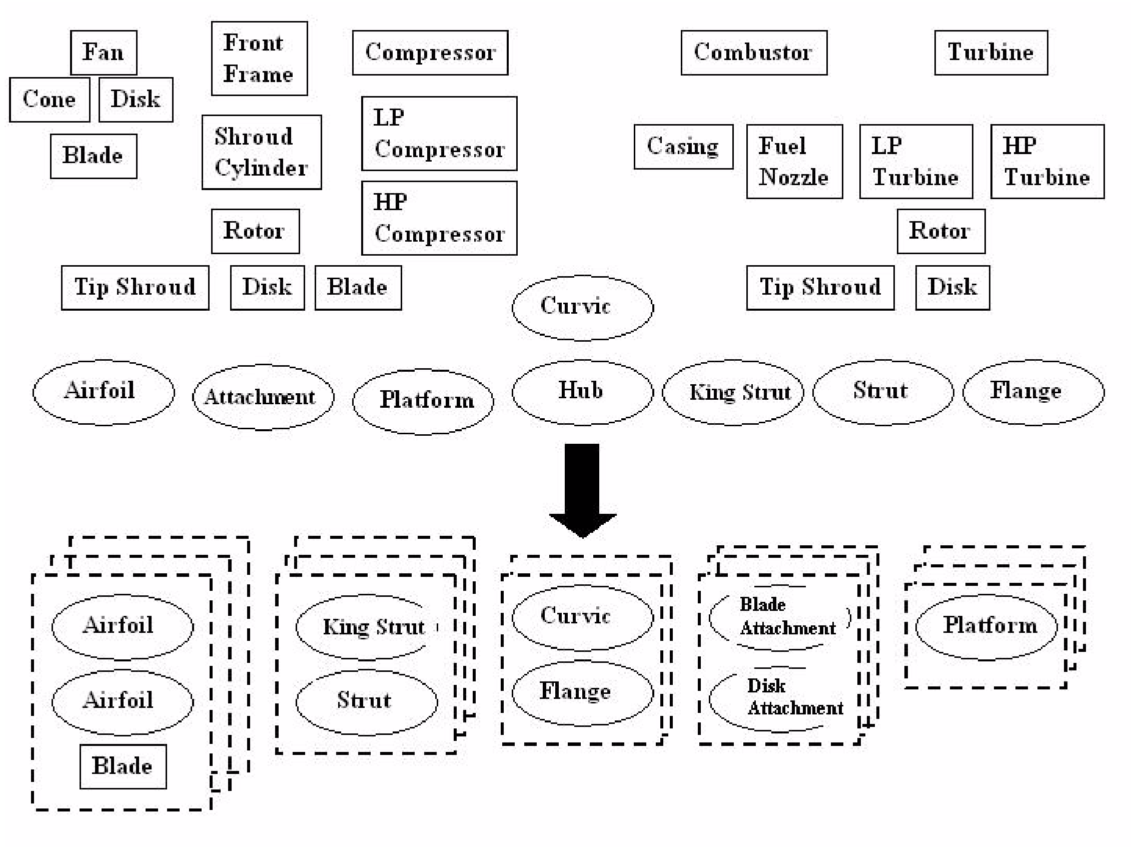


Figure 5.15 Sample Aggregation from Turbofan Engine PDG System

5.2.2.5 Classification of the Intermediate Transformations

The intermediate transformations themselves are defined using the method developed by Roach. In this case we will focus mainly on identifying reusable mappings.

If we look at the G mapping we can find a number of potential integration services. These are a CAD service, a document creation service and a service that can be used to create plots of our data.

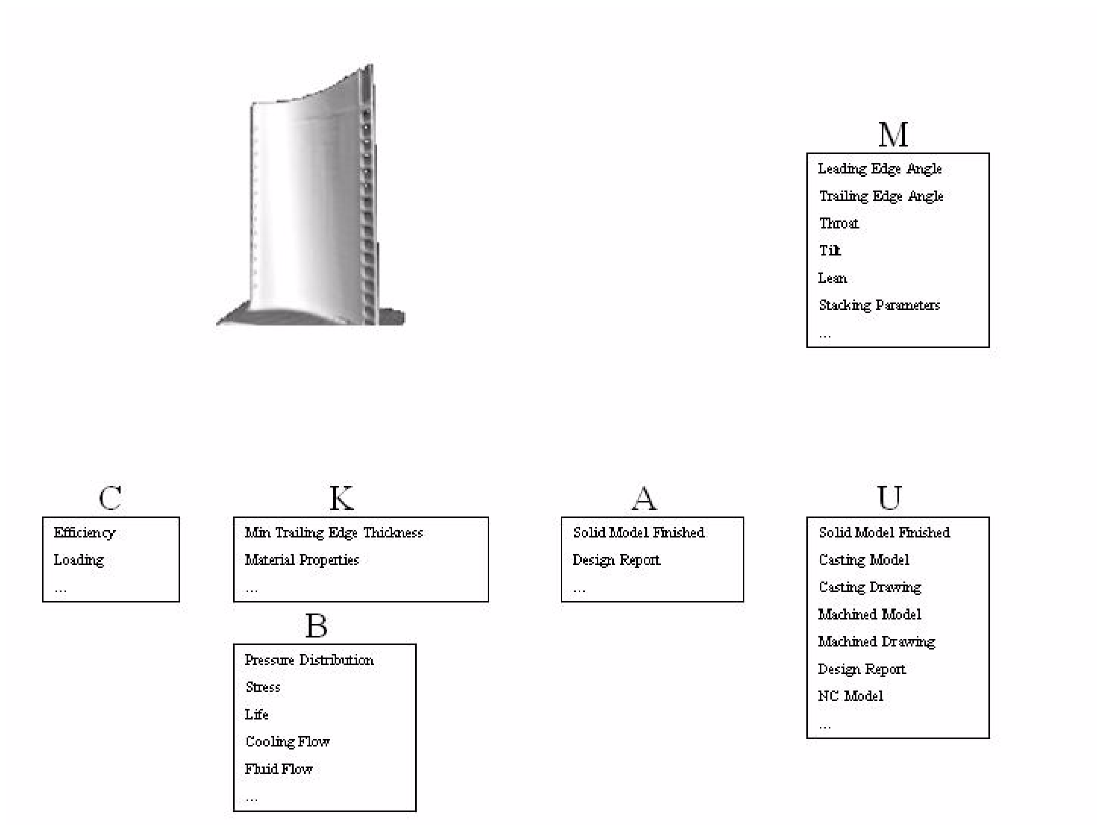


Figure 5.16 Set Definitions for Turbine Airfoil

In the P mappings we find a potential integration service with our CFD and FEA software packages. We also find some simple 2D aerodynamic calculations that we can demarcate as a PDG service.

The R mapping identifies a service to look up material data from the materials database. Another useful service is a service to look up rules from the design for manufacturing database.

The vaulting map identifies an integration service to our design repository database. It is here that the designs are collected and stored for historical record keeping.

5.2.2.6 Rectify M for the PDG Module

The master parameter list is then reconciled to eliminate duplicate parameters and to eliminate dependent parameters from the list. Again, this is done for each PDG module individually.

5.2.2.7 Aggregation of PDG Modules

At this point we are ready to aggregate the modules that we have defined into PDG services. These services may be constructed as composite services where the higher level service calls other services to perform its function.

The disk module might be made from the attachment service, the coupling service, the disk object, and the discourager object. This does not mean that the functionality of the attachment service lies inside of the disk module, only that the disk module requests the services of the attachment service.

A rotor subsystem service might be made up of the disk service discussed above as well as the blade service. This subsystem service would then be called a number of times to do the analysis of the low pressure turbine system. This service the becomes a client to the turbine section service which becomes a client for the turbofan engine service. This process is repeated until the entire system is built from these reusable services.

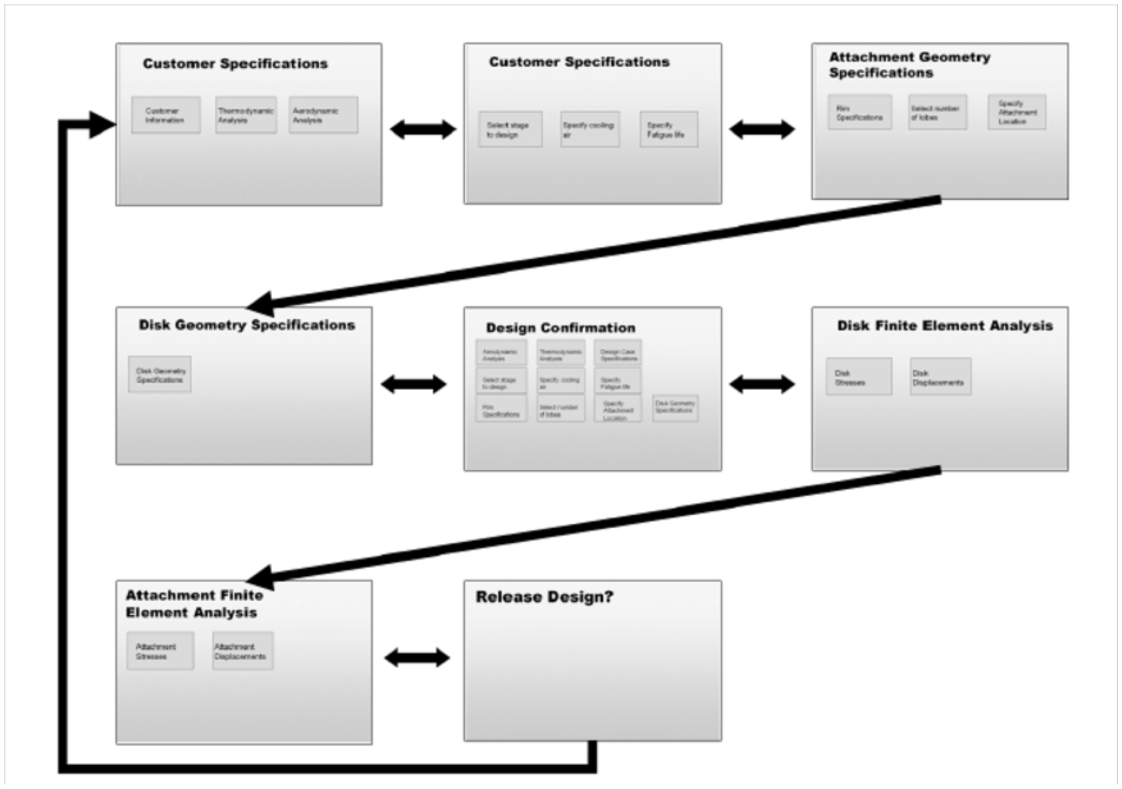


Figure 5.17 Storyboard of the Turbine Disk User Interface Design Phase

5.2.2.8 Compose Services

Service composition is the process of sequencing the services into a workflow. This workflow essentially exists only at runtime because the services are never permanently connected to one another. They are simply discovered in the UDDI registry and bound to at the time of execution.

5.2.2.9 Layout the Design and Release Cycles

Figure 5.17 and 5.18 show the layout of the design and release phases for the turbine disk portion of the PDG system.

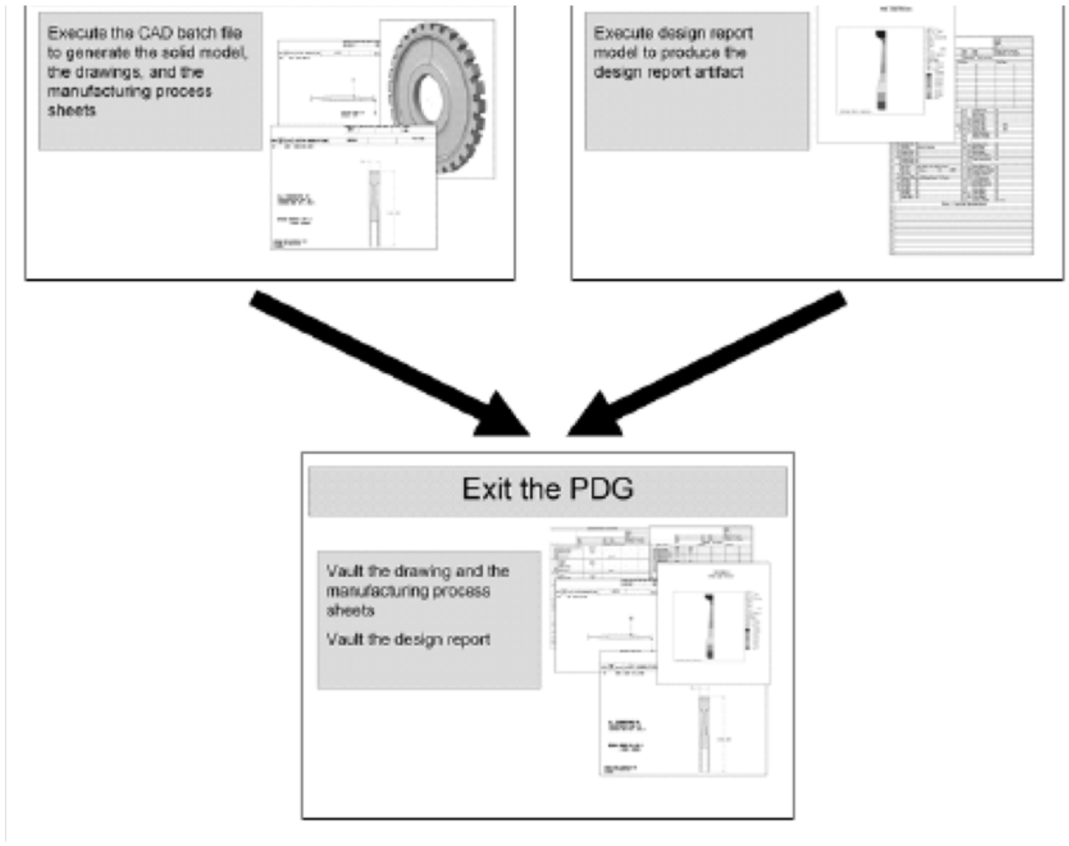

Execute the CAD batch file to generate the solid model, the drawings, and the manufacturing process sheets
Execute design report model to produce the design report artifact
Exit the PDG
Vault the drawing and the manufacturing process sheets
Vault the design report

Figure 5.18 Storyboard of the Turbine Disk User Interface Release Phase

5.2.2.10 Results

The turbofan engine system shows that through the method developed in this work, a complicated system may be decomposed and divided into reusable services. The fluid nature of service composition allows the process to easily adapt to changes in the product development process. Different services can be dynamically added or excluded in the design process at the time of execution. This essentially creates a service toolbox for product development. The correct service can be pulled from the toolbox in order to help solve the problem at hand. A service based architecture also has many advantages when it comes to communications between diverse design groups. Since all services by definition have a standardized interface, the passing of data between design groups becomes much easier. The data passed does not need to be converted into a different format for the application consuming the data since the services are based on industry standard integration technologies such as SOAP and WSDL.

CHAPTER 6

SUMMARY, CONCLUSIONS AND RECOMMENDATIONS

This thesis has extended the PDG methodology originally developed by Roach to include a method for developing an interconnected system of PDGs. This chapter presents a summary of the work followed by a discussion of the conclusions and recommendations for future research.

6.1 Summary

The concept of a service based system for PDG development represents an extension of the existing PDG methodology and provides an efficient method to implement a PDG for an entire system. These additions to the PDG methods provide a generalized approach for constructing PDGs with reuse in mind. It takes advantage of the standardized Web Services interfaces, allowing services to be easily plugged into the system. A framework for a generic PDG architecture was also presented which takes advantage of a number of proven software patterns. The combination of this framework and methods provides a mechanism for rapid PDG development and integration. PDGs built using this approach are easier to maintain and because each service is clearly demarcated, PDGs are easier to build in a distributed team. The following sections review the contributions to the general body of knowledge.

6.1.1 A Method For Decomposing a Product into PDG Services

In Chapter 4, the PDG methods developed by Roach were extended to facilitate reuse. The object, the PDG module and the PDG service were presented as general units of reuse in a PDG system.

The object is the lowest level of reuse within the PDG system. It can be seen as the smallest bundle of reusable functionality. Encapsulation of this functionality is the key to enable the object to be used in more than one context. Objects are identified by recursively decomposing the system until the smallest logical elements of reuse are identified. Examples of reusable objects were the disk object in the Atmospheric Resistor and the airfoil object in the turbofan engine.

These objects are then collected into larger units known as PDG modules. A PDG module is merely a collection of related objects. The turbine disk module, for example is made up of the disk object and the discourager object. While the PDG module is reusable at a higher level than the object, it is not necessarily exposed in the service layer.

The modules are then aggregated into PDG services. A PDG service might be made up of one or more PDG modules. The pipe connections service, for example is made up of the flange module, the welded flange module and the bolt circle module. A service may also become the client to another service. The attachment service would be a client to both the disk service and the blade service.

In order to effectively break the system into these reusable units, the PDG method was extended to include some additional steps. These steps include (1) recursive modular decomposition, (2) component feature decomposition, (3) recursive component aggregation, (4) classification of PDG module elements, (5) composition of the intermediate transformations, (6) rectification of M for the PDG module, (7) aggregation of PDG modules into services, (8) composition of services, and (9) the layout the design and release cycles.

The idea of a PDG fractal was also developed to help convey the idea that each service that makes up the PDG system is itself a small PDG. This provides insight into how we leverage the proven PDG methodology to create a network of PDG services.

The application of patterns to the development of PDGs was also proposed as a form of intellectual reuse. This enables PDG creators to build on proven practices and methods for PDG development. The insight that the patterns provide can help to eliminate many known pitfalls in this type of architecture and builds on proven integration techniques.

6.1.2 A System Level PDG

A method for creating a system level PDG was also presented. The system level PDG is formed by orchestrating a group of well defined PDG services. The standardization at the service level can be shown to provide great flexibility at the process level. A change in the process is simply made by changing the sequencing of the service executions. This process is defined using the BPEL4WS standard. This ability to define the service choreography using a standard business process language allows us to abstract the process from the functional code. We do not have to re-code the process within our applications. The BPEL document is deployed to a workflow service and can be instantiated at will. The services are never permanently connected to one another. They are dynamically bound at runtime and execution logic can change the process as it executes based on feedback that the workflow service captures.

6.1.3 A Web Based Framework for PDG Systems

The architecture for a web based PDG environment was presented. This environment allows the easy plug and play of PDG services and provides a potential base for future PDG environments. It provides ample separation between the user interface, the process choreography and the PDG services to enable general reuse of the system.

6.2 Conclusions

The purpose of this research was to provide a method to nest PDGs in an interconnected framework. Evaluation of the results of this research leads to the following conclusions:

1. Encapsulation is the key to creating reusable PDG elements. If the functionality is not encapsulated properly, the system will become too tightly coupled.

2. A nested PDG framework requires a standard communication protocol. The protocol used for this research is the industry standard Web Services protocol of SOAP over HTTP.

3. A service based approach to this framework provides enormous flexibility at the system level. As the process changes, additional services may be added or a different combination of existing services may be used. This provides standardization at the service and object levels while still maintaining flexibility at the process level. This provides an easy mechanism for new methods and tools to be integrated into the system. The service based approach can take further advantage of best in class process and optimization software.

4. PDG services can be classified into three groups, the integration services, the calculation services and the helper services.

5. It is possible to create a generic framework for PDGs. This framework takes advantage of the latest software technologies and is structured around reuse.

6. Flexibility is achieved by standardizing the lower levels of the architecture.

The next section provides recommendations for future work.

6.3 Recommendations

The purpose of this work was to develop a method for nesting PDG elements into an interconnected system. While this objective was achieved, future work can be done in order to improve upon this framework. The following are recommendations for future work:

1. Develop robust methods and strategies for the storage of the knowledge contained in K. At this time there is no standard structure nor robust strategy for storing PDG knowledge elements.

2. Apply the extended PDG method to more industrial products to further evaluate the method and to build a larger set of PDG patterns.

CHAPTER 7

REFERENCES

[1] Roach, Gregory M., 2003, *The Product Design Generator--A Next Generation Approach to Detailed Design*, Doctoral Thesis, Brigham Young University, Provo, UT.

[2] Davis, S., 1987, *Future Perfect*, Addison-Wesley Publishing Company Inc., Reading MA.

[3] Pine II, J.B. and Boyton, 1993, "Making Mass Customization Work", *Harvard Business Review*, Vol. 71, No. 5, pp. 108-111.

[4] Roach, Gregory M., Cox, Jordan J. and Teare, Shawn, S., 2001, "Reconfigurable Models and Product Templates as a Means to Increasing Productivity in the Product Development Process", *Proceedings of the 2001 Mass Customization and Personalization Conference*.

[5] Roach, Gregory M., Cox, Jordan J. and Young, Jared M., 2003, "A New Strategy for Automating the Generation of Product Family Members and Artifacts to an Aerospace Application", *Proceedings of the 2003 Design Engineering Technical Conferences*, Chicago, IL, CIE-2003-85.

[6] Baker, Albert D., Parunak, Van Dyke H., and Erol, Kutluhan. 1999. "Agents and the Internet: Infrastructure for Mass Customization", *IEEE Internet Computing*.

[7] DaSilveria, Giovani, Borenstein, Denis, Fogliatto, Flavio S., 2001, "Mass Customization: Literature Review and Research Directions", *Production Economics*, Vol. 72, pp. 1-13.

[8] Aziz, El-Sayed, and Chassapis, C., 2002, “Development of an Interactive Web-Based Support System for Gear Design”, *Proceedings of the 2002 Design Engineering Technical Conferences*, Montreal, Canada, DETC2002/DAC-34114.

[9] Szykman, Simon, 2002, “Architecture and Implementation of a Design Repository System”, *Proceedings of the 2002 Design Engineering Technical Conferences*, Montreal, Canada, DETC2002/CIE-34463.

[10] Shen, Weiming, Ghenniwa, 2002, “A Distributed Multidisciplinary Design Optimization Framework based on Web and Agents. *Proceedings of the 2002 Design Engineering Technical Conferences*, Montreal, Canada, DETC2002/CIE-34461.

[11] Karne, Ramesh K et al, 1998, “Web-It-Man: A Web Based Integrated Tool for Manufacturing Environment”, *Proceedings of the 1998 Design Engineering Technical Conferences*, Atlanta, GA, DETC98/CIE-5524.

[12] Rangel, Fernando, and Shah, Jami, J., 2002, “Integration of Commercial CAD/CAM System with Custom CAPP Using Orbix Middleware and CORBA Standard”, *Proceedings of the 2002 Design Engineering Technical Conferences*, Montreal, Canada, DETC2002/DAC-34069.

[13] Wong, Lee Ming, Friesen, Charles, Foo, Ken How, Wang, Gary G., and Pang, Lucas. 2002, “Development of an Automated Design and Optimization System for Industrial Silencers”. *Proceedings of the 2002 Design Engineering Technical Conferences*, Montreal, Canada, DETC2002/DAC-34118.

[14] Siddique, Zahed, and Yanjiang, Zhou., 2002, “Automatic Generation of Product Family Member CAD Models Supported by a Platform Using a Template Approach”, *Proceedings of the 2002 Design Engineering Technical Conferences*, Montreal, Canada, DETC2002/CIE-34407.

[15] Siddique, Zahed, and Ninan, Jiju, 2003, “Internet Based Framework to Perform Automated FEA on User Customized Products”, *Proceedings of the 2003 Design Engineering Technical Conferences,* Chicago, Illinois, USA, DETC2003/DAC-48719.

[16] Ninan, Jiju and Siddique, Zahed, 2004, “Finite Element Analysis Template Approach to Support Web-Based Customer Centric Design”, *Design Engineering Technical Conferences*, Salt Lake City, UT, USA, DETC2004-57697.

[17] D. Xue, F. Zhang, 2002, “Distributed Database and Knowledge Base Modeling for Concurrent Design”, *Computer Aided Design*, Vol. 34, pp 27-40.

[18] Flores, Rogelio, Jensen, C. Greg, and Shelley, Jon, “A Web Enabled Process for Accessing Customized Parametric Designs”, *Proceedings of the 2002 Design Engineering Technical Conferences*, Montreal, Canada, DETC2002/DAC-34078.

[19] Ulrich, K.T., and Eppinger, S.D, 2000, *Product Design and Development*, McGraw-Hill, Inc., New York.

[20] Otis, C.E., and Vosbury, P.A., 2001, *Aircraft Gas Turbine Powerplants*, Jeppesen Sanderson, Inc., Englewood, CO.

[21] Bergin, Jr., T.J., and Gibson, R.G., 1996, *History of Programming Languages - II*, ACM Press, New York NY, and Addison-Wesley Publ. Co., Reading MA.

[22] Northrup, Matthew M., 1997, *The Application of Object-Oriented System Analysis to the Early Stages of Discrete Product Design*, Masters Thesis, Brigham Young University, Provo, UT.

[23] Montgomery, Stephen L., 1994, *Object-Oriented Information Engineering: Analysis, Design, and Implementation*, AP Professional, New York, New York.

[24] Beiter, Kurt A. and Ishii, Kosuke, 2003, ”Integrating Producibility and Product Performance Tools Within a Web-Service Environment”, *Proceedings of the Design Engineering Technical Conferences*, Chicago, Illinois, USA, DETC2003/CIE-48281.

[25] Reichwald, Ralf, Piller, Frank T. and Moslein, Kathrin, 2000. “Mass Customization Concepts for the E-Conomy: Four Strategies to create competitive Advantage with Customized Goods and Services on the Internet*”*, *International NAISO Congress on Information Science Innovations*, Dubai, UAE.

[26] Kao, Kevin J., Seeley, Charles E., Yin, Su and Kolonay, Raymond M., 2003. “Business-to-Business Virtual Collaboration of Aircraft Engine Combustor Design*”, Proceedings of the Design Engineering Technical Conferences*, Chicago, Illinois, USA, DETC2003/CIE-48282.

[27] Wujek, Brett A. Koch, Patrick N., McMillan, Mark and Chiang, Wei-Shan, 2002, “A Distributed, Component-Based Integration Environment

for Multidisciplinary Optimal and Quality Design*", American Institute of Aeronautics and Astronautics*, Cary, NC, USA.

[28] Johansson, Bjorn and Krus, Petter, 2003, "A Web Service Approach for Model Integration in Computational Design*", Proceedings of the Design Engineering Technical Conferences*, Chicago, Illinois, USA, DETC2003/ CIE-48196.

[29] Radcliffe, Clark J and Sticklen Jon, 2002, "The Internet Engineering Design Agent System: iEDA*", Proceedings of IMECE ASME International Mechanical Engineering Congress and Exposition*, New Orleans, Louisiana, USA, IMECE2002-39286.

[30] McGovern, James, Tyagi, Sameer, Stevens, Michael E., Mathew, Sunil, 2003, *Java Web Services Architecture,* Morgan Kaufmann Publishers, New York.

[31] Anosike, Anthony and Zhang, David, 2000, "An Agent-Oriented Modelling Approach for Agile Manufacturing*"*, *School of Engineering and Computer Science, University of Exeter*, Exeter, United Kingdom.

[32] Wang, S.L, Xia, H., Liu, F., Tao, G.B., and Zhang, Z., 2002, "Agent-Based Modeling and Mapping of a Manufacturing System*"*, *Journal of Materials Processing Technology*, Vol. 129, pp. 518-523.

[33] Shen, Weiming, *2002,* "Distributed Manufacturing Scheduling Using Intelligent Agents", *IEE Intelligent Systems*, January/February 2002, pp. 88 - 94.

[34] Tian, Gui Yun, Yin, Guofu, and Taylor, David, 2002, "Internet-Based Manufacturing: A Review and a New Infrastructure for Distributed Intelligent Manufacturing", *Journal of Intelligent Manufacturing*, Vol. 13, pp. 323-338.

[35] Tumkor, Serdar, 2000, "Internet-Based Design Catalogue for the Shaft and Bearing*"*, *Research in Engineering Design*, Vol. 12, pp. 163-171.

[36] Whitfield, Robert Ian, Duffy, Alex H.B., Coates, Graham, Hills, William, 2002, "Distributed Design Coordination*", Research in Engineering Design*, Vol. 13, pp. 243-252.

[37] Aldous, Kenneth, and Lintott, Andrew B., 2003, "A Web Platform for the Exchange and Transformation of Business Objects*"*, *Design Engineering Technical Conferences*, Chicago, Illinois, USA, DETC2003/CIE-48266.

[38] Skolicki, Zbigniew and Arciszewski, Tomasz, 2003, "Intelligent Agents in Design", *Design Engineering Technical Conferences*, Chicago, Illinois, USA, DETC2003/DTM-48671.

[39] Wang, Peijun, Bjarnemo, Robert, and Motte, Damien, 2003, "Development of a Web-Based Customer-Oriented Interactive Virtual Environment for Mobile Phone Design", *Design Engineering Technical Conferences*, Chicago, Illinois, USA, DETC2003/CIE-48300.

[40] Rosenman, Mike, and Wang, Fujun, 1999, "CADOM: A Component Agent-Based Design-Oriented Model for Collaborative Design", *Research in Engineering Design*, Vol. 11, pp. 193-205.

[41] Raj, Ajoy R., and Ramani, Karthik, 2004, "Enabling 'Self-Service' Data Management: Distributed Product Data Management Architecture", *Design Engineering Technical Conferences*, Salt Lake City, UT, USA, DETC2004-57788.

[42] Liao, Xiaoyun, and Wang, G. Gary, 2004, "Variation Analysis of Non-Rigid Assembly Using FEM and Fractals", *Design Engineering Technical Conferences*, Salt Lake City, UT, USA, DETC2004-57753.

[43] Shooter, Steven B., Stone, Robert B., Simpson, Timothy W., Kumara, Soundar R.T., Terpenny, Janis P., 2004, "Toward an Information Management Infrastructure for Product Family Planning and Mass Customization", *Design Engineering Technical Conferences*, Salt Lake City, UT, USA, DETC2004-57430.

[44] Robertson, D. and Ulrich, K., 1998, "Planning Product Platforms," *Sloan Management Review,* 39(4), pp. 19-31.

[45] Araque, Rafael, Bailey, Trevor, Bonilha, Murilo W., and Fletcher, Jay, 2004, "A Systems Framework For Platform Architecture Analysis", *Design Engineering Technical Conferences*, Salt Lake City, UT, USA, DETC2004-57299.

[46] Yang, QZ, and Lu, W.F., 2004, "Development of a J2EE Web Application for STEP-Based Design Conformance Checking*",* Design Engineering Technical Conferences*, Salt Lake City, UT, USA, DETC2004-57522.*

[47] Bidarra, Rafael, van Bunnik, Andre, and Bronsvoort, Willem, 2004, "Direct Manipulation of Feature Models in Web-Based Collaborative Design", *Design Engineering Technical Conferences*, Salt Lake City, UT, USA, DETC2004-57716.

[48] Zha, Xuan F., Li, Ling L., and Lim, Samuel Y.E., 2004, “A Multi-Agent Intelligent Environment For Rapid Assembly Design*”, Planning and Simulation*, Design Engineering Technical Conferences, Salt Lake City, UT, USA, DETC2004-57713.

[49] Tseng, Mitchell M., and Piller, Frank T., 2003, *The Customer Centric Enterprise: Advances in Mass Customization and Personalization*, Springer, New York.

[50] Nanda, Jyotirmaya, Thevenot, Henri J., Simpson, Timothy W., Kumara, Soundar R.T., 2004, “Exploring Semantic Web Technologies for Product Family Modeling*”*, *Design Engineering Technology Conferences*, Salt Lake City, UT, DETC2004-57683.

[51] Li, W.D., and Lu, W.F., 2004, “Development of a Web-Based Process Planning Optimization System*”*, *Design Engineering Technical Conferences*, Salt Lake City, UT, DETC2004-57675.

[52] Mulberger, Jessica L., and Simpson, Timothy W., 2004, “Advancements in a Web-Based Framework in Product Family Optimization and Visualization*”*, *Design Engineering Technical Conferences*, Salt Lake City, UT, DETC2004-57688.

[53] Hao, Qi, Shen, Weiming, Zhang, Zhan, Park, Seong-Whan, Lee, and Jai-Kyung, 2004, “A Multi-Agent Framework for Collaborative Engineering Design and Optimization*”*, *Design Engineering Technical Conferences*, Salt Lake City, UT, DETC2004-57686.

[54] Shen, W., Norrie, D.H. and Barthes, J.P., 2001, “Multi-Agent Systems for Concurrent Intelligent Design and Manufacturing*”*, Taylor and Francis, London, UK.

[55] Wang, L., Shen, W., Xie, H., Neelamkavil, J, and Pardasani, A., 2002, “Collaborative Conceptual Design: A State-of-the-Art Survey”, *CAD*, 34(13), pp. 981-996.

[56] Alur, Deepak, Crupi, John, and Malks, Dan, 2003, Core J2EE Patterns: Best Practices and Design Strategies, Prentice Hall PTR, Upper Saddle River, NJ.

[57] Panchal, Jitesh H., Chamberlain, Matthew k., Rosen, David W., Allen, Janet K., and Mistree, Farrokh, 2002, “A Service Based Architecture for Information and Asset Utilization in Distributed Product Realization*”, American Institute of Aeronautics and Astronautics*.

[58] Gerhard, Jonathan F., Rosen, David, Allen, Janet K., and Mistree, Farrokh, 2000, "A Distributed Product Realization Environment for Design and Manufacturing", *Design Engineering Technical Conferences*, Baltimore, Maryland, USA, DETC2000/CIE-14624.

[59] Han, Charles, Kunz, John C., and Law, Kincho H., 1999, "An Internet-Based Distributed Service Architecture", *Design Engineering Technical Conferences*, Las Vegas, NV, USA, DETC99/CIE-9077.

[60] Xiao, Angran, Choi, Hae-Jin, Kulkarni, Rahul, Allen, Janet K., Rosen, David, and Mistree, Farrokh, 2001, "A Web-Based Distributed Product Realization Environment", *Design Engineering Technical Conferences*, Pittsburgh, PA, USA, DETC2001/CIE-21766.

[61] Brock, Rebecca Wirfs and McKean, Alan, 2003, *Object Design: Roles, Responsibilities and Collaborations*, Addison Wesley, NY.

[62] Knapik, Michael and Johnson, Jay, 1998, *Developing Intelligent Agents for Distributed Systems*, McGraw-Hill, NY.

[63] Murch, Richard and Johnson, Tony, 1999, *Intelligent Software Agents*, Prentice Hall PTR, Upper Saddle River, NJ.

[64] Shooter, S. B., Keirouz, W. T., Szykman, S., Fenves, S. J., 2000, "A Model for the Flow of Design Information in Product Development", *Engineering with Computers*, Vol. 16: pp 178-194.

[65] Zhang, Mike Tao and Goldberg, Ken, 2002, "*I*nternet-Based CAD Tool for Design of Gripper Jaws", *Design Engineering Technical Conferences*, Montreal, Canada, DETC2002/CIE-34460.

[66] Whitney, Daniel E, and Dong, Qi, 1999, "Introducing Knowledge-Based Engineering into an Interconnected Product Development Process", *Design Engineering Technical Conferences*, Las Vegas, NV, DETC99/DTM-8741.

[67] Roy, U., and Kodkani, S. S., 1999, "Product Modeling Within the Framework of the World Wide Web", *IIE Transactions*, Vol. 31, pp 667-677.

[68] Paydarfar, Saeed, 2001, "An Integration Maturity Model for the Digital Enterprise", *The Digital Enterprise*, Fall 2001, pp. 29-44.

[69] Gonzales-Zugasti, Javier P., Otto, Kevin N., and Baker, John D., 2000, "A Method for Architecting Product Platforms", *Research in Engineering Design*, Vol. 12, pp 61-72.

[70] Reich, Yoram, Konda, Suresh, Subrahmanian, Cunningham, Douglas, Dutoit, Allen, Patrick, Robert, Thomas, Mark, and Westerberg, Arthur W., 1999, "Building Agility for Developing Agile Design Information Systems*", Research in Engineering Design*, Vol. 11, pp. 67-83.

[71] Martin, Mark V., and Ishii, Kosuke, 2002, "Design for Variety: Developing Standardized and Modularized Product Platform Architectures*", Research in Engineering Design*, Vol. 13, pp. 213-235.

[72] Ullman, David G., 2002, *"*Toward the Ideal Mechanical Engineering Design Support System*", Research in Engineering Design*, Vol. 13, pp. 55-64.

[73] Ahmed, Saeema, Wallace, Ken M., and Blessing, Lucienne T.M., 2003, "Understanding the Differences Between How Novice and Experienced Designers Approach Design Tasks*"*, *Research in Engineering Design*, Vol., 14, pp. 1-11.

[74] Rohl, Peter J., Kolonay, Raymond M., Paradis, Michael J., and Bailty Michael W., 2001, "Intelligent Compressor Design in a Network Centric Environment*"*, *General Electric Corporate Research and Development.*

[75] Tumkor, Serdar, 2000, "Internet-Based Design Catalogue for the Shaft and Bearing*"*, *Research in Engineering Design*, Vol. 12, pp 163-171.

[76] Simpson, Timothy W., and D'Souza, Brayan, 2002, "Assessing Variable Levels of Platform Commonality Within a Product Family Using a Multi-objective Genetic Algorithm", *Symposium on Multidisciplinary Analysis and Optimization*, Atlanta, Georgia, AIAA 2002-5427.

[77] Messac, Achille, Martinez, Michael P., and Simpson, Timothy W., 2002, "Effective Product Family Design Using Physical Programming*"*, *Engineering Optimization*, Vol. 34, pp 245-261.

[78] Lampel, J., and Pine, J.B., 1997, "Customizing Customization", *Sloan Management Review*, Vol. 38, pp. 21-30.

[79] Erens, F., and Verhulst, K., 1997, "Architectures for Product Families", *Computers in Industry,* Vol. 33 pp. 165-178.

[80] Erens, F., and Wortman, H.C., 1996, "Generic Product Modeling for Mass Customization", *Implementation Road Map*, 1996, Ann Arbor, MI.

[81] Jiao, J., 1998, *Design for Mass Customization by Developing Product Family Architecture*, Doctoral Thesis, Hong Kong University of Science and Technology, Kowloon, Hong Kong.

[82] Jiao, J., and Tseng, M.M., 1999, “A Methodology of Developing Product Family Architecture for Mass Customization”, *Journal of Intelligent Manufacturing*, Vol. 10, pp. 3-20.

[83] Sun Microsystems, “What is an Object?”, URL: http://java.sun.com/docs/books/tutorial/java/concepts/object.html, valid as of June, 2005.

[84] Sun Microsystems, “What is a Message?”, URL: http://java.sun.com/docs/books/tutorial/java/concepts/message.html, valid as of June, 2005.

[85] Flores-Zubillaga, Rogelio, 2002, *Distribued CAD Components*, Masters Thesis, Brigham Young University, Provo, UT.

[86] IBM Developerworks, “Business Process Execution Language for Web Services”, URL: http://www-128.ibm.com/developerworks/library/specification/ws-bpel/, valid as of June, 2005.

APPENDIX A SIMPLE WEB SERVICE SOURCE CODE

7.1 Capacity.java

```
/*
 * Created on Jul 7, 2005
 *
 * TODO To change the template for this generated file go to
 * Window - Preferences - Java - Code Style - Code Templates
 */
package com.cci.services.noise;

/**
 * @author Jared
 *
 * TODO To change the template for this generated type comment go to
 * Window - Preferences - Java - Code Style - Code Templates
 */
public class Capacity {

public double calcCapacityConst( CapacityVO vo )
{
double flowRate = vo.getFlowRate();
double compressFlowFactor = vo.getCompressFlowFactor();
double density = vo.getDensity();
double pressure = vo.getInletPressure();
double atmPressure = vo.getAtmPressure();
```

```
        return (0.016 * flowRate) / ( compressFlowFactor * Math.sqrt( density * ( pressure
- atmPressure ) ) );
        }
        }
```

7.2 CapacityVO.java

```
/*
 * Created on Jul 7, 2005
 *
 * TODO To change the template for this generated file go to
 * Window - Preferences - Java - Code Style - Code Templates
 */
package com.cci.services.noise;

/**
 * @author Jared
 *
 * TODO To change the template for this generated type comment go to
 * Window - Preferences - Java - Code Style - Code Templates
 */
public class CapacityVO {
private double flowRate;
private double density;
private double inletPressure;
private double atmPressure;
private double compressFlowFactor;
/**
 * @return Returns the atmPressure.
 */
public double getAtmPressure() {
return atmPressure;
}
/**
 * @param atmPressure The atmPressure to set.
 */
public void setAtmPressure(double atmPressure) {
this.atmPressure = atmPressure;
}
/**
 * @return Returns the compressFlowFactor.
 */
```

```
public double getCompressFlowFactor() {
return compressFlowFactor;
}
/**
 * @param compressFlowFactor The compressFlowFactor to set.
 */
public void setCompressFlowFactor(double compressFlowFactor) {
this.compressFlowFactor = compressFlowFactor;
}
/**
 * @return Returns the density.
 */
public double getDensity() {
return density;
}
/**
 * @param density The density to set.
 */
public void setDensity(double density) {
this.density = density;
}
/**
 * @return Returns the flowRate.
 */
public double getFlowRate() {
return flowRate;
}
/**
 * @param flowRate The flowRate to set.
 */
public void setFlowRate(double flowRate) {
this.flowRate = flowRate;
}
/**
 * @return Returns the inletPressure.
 */
public double getInletPressure() {
return inletPressure;
}
/**
 * @param inletPressure The inletPressure to set.
 */
public void setInletPressure(double inletPressure) {
this.inletPressure = inletPressure;
}
```

```
}
```

7.3 Capacity_SEI.java

```
package com.cci.services.noise;

public interface Capacity_SEI extends java.rmi.Remote
{
 public double calcCapacityConst(com.cci.services.noise.CapacityVO vo);
}
```

7.4 CapacityVO_Deser.java

```
/**
 * CapacityVO_Deser.java
 */

package com.cci.services.noise;

public class CapacityVO_Deser extends com.ibm.ws.webservices.engine.encod-
ing.ser.BeanDeserializer {
   /**
    * Constructor
    */
   public CapacityVO_Deser(
          java.lang.Class _javaType,
          javax.xml.namespace.QName _xmlType,
          com.ibm.ws.webservices.engine.description.TypeDesc _typeDesc) {
       super(_javaType, _xmlType, _typeDesc);
   }
}
```

7.5 CapacityVO_Helper.java

```
/**
 * CapacityVO_Helper.java
 *
 */

package com.cci.services.noise;

public class CapacityVO_Helper {
   // Type metadata
   private static com.ibm.ws.webservices.engine.description.TypeDesc typeDesc =
                new  com.ibm.ws.webservices.engine.description.TypeDesc(Capaci-
tyVO.class);

   static {
               com.ibm.ws.webservices.engine.description.FieldDesc  field  =  new
com.ibm.ws.webservices.engine.description.ElementDesc();
      field.setFieldName("atmPressure");
      field.setXmlName(com.ibm.ws.webservices.engine.utils.QNameTable.create-
QName("http://noise.services.cci.com", "atmPressure"));
      field.setXmlType(com.ibm.ws.webservices.engine.utils.QNameTable.create-
QName("http://www.w3.org/2001/XMLSchema", "double"));
      typeDesc.addFieldDesc(field);
      field = new com.ibm.ws.webservices.engine.description.ElementDesc();
      field.setFieldName("compressFlowFactor");
      field.setXmlName(com.ibm.ws.webservices.engine.utils.QNameTable.create-
QName("http://noise.services.cci.com", "compressFlowFactor"));
      field.setXmlType(com.ibm.ws.webservices.engine.utils.QNameTable.create-
QName("http://www.w3.org/2001/XMLSchema", "double"));
      typeDesc.addFieldDesc(field);
      field = new com.ibm.ws.webservices.engine.description.ElementDesc();
      field.setFieldName("density");
      field.setXmlName(com.ibm.ws.webservices.engine.utils.QNameTable.create-
QName("http://noise.services.cci.com", "density"));
      field.setXmlType(com.ibm.ws.webservices.engine.utils.QNameTable.create-
QName("http://www.w3.org/2001/XMLSchema", "double"));
      typeDesc.addFieldDesc(field);
      field = new com.ibm.ws.webservices.engine.description.ElementDesc();
      field.setFieldName("flowRate");
      field.setXmlName(com.ibm.ws.webservices.engine.utils.QNameTable.create-
QName("http://noise.services.cci.com", "flowRate"));
      field.setXmlType(com.ibm.ws.webservices.engine.utils.QNameTable.create-
QName("http://www.w3.org/2001/XMLSchema", "double"));
```

```
                typeDesc.addFieldDesc(field);
                field = new com.ibm.ws.webservices.engine.description.ElementDesc();
                field.setFieldName("inletPressure");
                field.setXmlName(com.ibm.ws.webservices.engine.utils.QNameTable.create-
QName("http://noise.services.cci.com", "inletPressure"));
                field.setXmlType(com.ibm.ws.webservices.engine.utils.QNameTable.create-
QName("http://www.w3.org/2001/XMLSchema", "double"));
                typeDesc.addFieldDesc(field);
            };

            /**
             * Return type metadata object
             */
               public static com.ibm.ws.webservices.engine.description.TypeDesc getType-
Desc() {
                return typeDesc;
            }

            /**
             * Get Custom Serializer
             */
            public static com.ibm.ws.webservices.engine.encoding.Serializer getSerializer(
                  java.lang.String mechType,
                  java.lang.Class javaType,
                  javax.xml.namespace.QName xmlType) {
                return
                  new CapacityVO_Ser(
                    javaType, xmlType, typeDesc);
            };

            /**
             * Get Custom Deserializer
             */
            public static com.ibm.ws.webservices.engine.encoding.Deserializer getDeserial-
izer(
                  java.lang.String mechType,
                  java.lang.Class javaType,
                  javax.xml.namespace.QName xmlType) {
                return
                  new CapacityVO_Deser(
                    javaType, xmlType, typeDesc);
            };

        }
```

7.6 CapacityVO_Ser.java

```
/**
 * CapacityVO_Ser.java
 *
 */

package com.cci.services.noise;

public class CapacityVO_Ser extends com.ibm.ws.webservices.engine.encod-
ing.ser.BeanSerializer {
    /**
     * Constructor
     */
    public CapacityVO_Ser(
           java.lang.Class _javaType,
           javax.xml.namespace.QName _xmlType,
           com.ibm.ws.webservices.engine.description.TypeDesc _typeDesc) {
        super(_javaType, _xmlType, _typeDesc);
    }
    public void serialize(
        javax.xml.namespace.QName name,
        org.xml.sax.Attributes attributes,
        java.lang.Object value,
        com.ibm.ws.webservices.engine.encoding.SerializationContext context)
        throws java.io.IOException
    {
        context.startElement(name, addAttributes(attributes,value,context));
        addElements(value,context);
        context.endElement();
    }
    protected org.xml.sax.Attributes addAttributes(
        org.xml.sax.Attributes attributes,
        java.lang.Object value,
        com.ibm.ws.webservices.engine.encoding.SerializationContext context)
        throws java.io.IOException
    {
        return attributes;
    }
    protected void addElements(
        java.lang.Object value,
        com.ibm.ws.webservices.engine.encoding.SerializationContext context)
        throws java.io.IOException
    {
```

```
    CapacityVO bean = (CapacityVO) value;
    java.lang.Object propValue;
    javax.xml.namespace.QName propQName;
    {
      propQName = QName_0_0;
      propValue = new Double(bean.getAtmPressure());
      context.serialize(propQName, null,
          propValue,
          QName_1_5,
          true,null);
      propQName = QName_0_1;
      propValue = new Double(bean.getCompressFlowFactor());
      context.serialize(propQName, null,
          propValue,
          QName_1_5,
          true,null);
      propQName = QName_0_2;
      propValue = new Double(bean.getDensity());
      context.serialize(propQName, null,
          propValue,
          QName_1_5,
          true,null);
      propQName = QName_0_3;
      propValue = new Double(bean.getFlowRate());
      context.serialize(propQName, null,
          propValue,
          QName_1_5,
          true,null);
      propQName = QName_0_4;
      propValue = new Double(bean.getInletPressure());
      context.serialize(propQName, null,
          propValue,
          QName_1_5,
          true,null);
    }
}
    public final static javax.xml.namespace.QName QName_0_4 =
          com.ibm.ws.webservices.engine.utils.QNameTable.createQName(
                "http://noise.services.cci.com",
                "inletPressure");
    public final static javax.xml.namespace.QName QName_0_3 =
          com.ibm.ws.webservices.engine.utils.QNameTable.createQName(
                "http://noise.services.cci.com",
                "flowRate");
    public final static javax.xml.namespace.QName QName_1_5 =
                com.ibm.ws.webservices.engine.utils.QNameTableTable.createQName(
                    "http://www.w3.org/2001/XMLSchema",
                    "double");
            public final static javax.xml.namespace.QName QName_0_0 =
                com.ibm.ws.webservices.engine.utils.QNameTable.createQName(
                    "http://noise.services.cci.com",
                    "atmPressure");
            public final static javax.xml.namespace.QName QName_0_2 =
                com.ibm.ws.webservices.engine.utils.QNameTable.createQName(
                    "http://noise.services.cci.com",
                    "density");
            public final static javax.xml.namespace.QName QName_0_1 =
                com.ibm.ws.webservices.engine.utils.QNameTable.createQName(
                    "http://noise.services.cci.com",
                    "compressFlowFactor");
        }
```

7.7 Capacity.wsdl

```
        <?xml version="1.0" encoding="UTF-8"?>
        <wsdl:definitions targetNamespace="http://noise.services.cci.com" xmlns:impl="http://noise.ser-
vices.cci.com" xmlns:intf="http://noise.services.cci.com" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:xsd="http://www.w3.org/2001/
XMLSchema">
         <wsdl:types>
           <schema elementFormDefault="qualified" targetNamespace="http://noise.services.cci.com"
xmlns="http://www.w3.org/2001/XMLSchema" xmlns:impl="http://noise.services.cci.com"
xmlns:intf="http://noise.services.cci.com" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
          <complexType name="CapacityVO">
           <sequence>
            <element name="atmPressure" type="xsd:double"/>
            <element name="compressFlowFactor" type="xsd:double"/>
            <element name="density" type="xsd:double"/>
            <element name="flowRate" type="xsd:double"/>
            <element name="inletPressure" type="xsd:double"/>
           </sequence>
          </complexType>
          <element name="calcCapacityConst">
           <complexType>
            <sequence>
             <element name="vo" nillable="true" type="impl:CapacityVO"/>
            </sequence>
           </complexType>
          </element>
```

```
  <element name="calcCapacityConstResponse">
   <complexType>
    <sequence>
     <element name="calcCapacityConstReturn" type="xsd:double"/>
    </sequence>
   </complexType>
  </element>
 </schema>
</wsdl:types>

  <wsdl:message name="calcCapacityConstRequest">

     <wsdl:part element="intf:calcCapacityConst" name="parameters"/>

  </wsdl:message>

  <wsdl:message name="calcCapacityConstResponse">

     <wsdl:part element="intf:calcCapacityConstResponse" name="parameters"/>

  </wsdl:message>

  <wsdl:portType name="Capacity">

     <wsdl:operation name="calcCapacityConst">

        <wsdl:input message="intf:calcCapacityConstRequest" name="calcCapacityConstRequest"/>

              <wsdl:output  message="intf:calcCapacityConstResponse"  name="calcCapacityConstRe-
sponse"/>

     </wsdl:operation>

  </wsdl:portType>

  <wsdl:binding name="CapacitySoapBinding" type="intf:Capacity">

     <wsdlsoap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>

     <wsdl:operation name="calcCapacityConst">

        <wsdlsoap:operation soapAction=""/>

        <wsdl:input name="calcCapacityConstRequest">

           <wsdlsoap:body use="literal"/>

        </wsdl:input>

        <wsdl:output name="calcCapacityConstResponse">

           <wsdlsoap:body use="literal"/>
```

```
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="CapacityService">
    <wsdl:port binding="intf:CapacitySoapBinding" name="Capacity">
      <wsdlsoap:address location="http://localhost:9080/CapWeb/services/Capacity"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```