Model-driven development: Assets and reuse

G. Larsen

Several of the challenges that software organizations face today include an increase in the complexity of their information technology (IT) infrastructures and solutions, applications that may be difficult to use, and continued pressure to achieve tight time-to-market timelines. Organizations use many approaches to address these problems, including models. Models can embody critical solutions and insights and thus can be seen as assets for an organization. For example, a pattern that describes a recurring problem, its recurring solution, and the context in which it is relevant can be expressed as a model and shared with others. This paper presents some of the steps to identify reusable models, organize them for reuse, and package, publish, and ultimately reuse them—all with a focus on the benefit to the business and the alignment of IT to the needs of the business. Whereas the concepts associated with reuse are not new, organizing models for reuse in the manner described herein represents more recent techniques.

INTRODUCTION

There are many challenges in software development, but in this paper, we focus on mitigating complexity, improving consumability (the ease of use by which a model can be approached, its organization understood, and a determination made concerning how to apply it to one's needs), and reducing time to market.

Background

Several years ago, a consortium of software industry leaders—including IBM, Rational* Software (before its acquisition by IBM), and Microsoft—began exploring ways to help organizations repurpose software investments. In this exploration, it was determined that software entities need to be named, organized, reviewed, and reused to improve the return on the investment in them.

The consortium began to describe these software investments as *software assets*. This led us ultimately to create standards for asset packaging formats and then to develop tools and processes to work with assets throughout their life cycle. Although we found that assets take on many shapes and sizes and are used in multiple roles throughout the development life cycle, we found similarities

©Copyright 2006 by International Business Machines Corporation. Copying in printed form for private use is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the Journal reference and IBM copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free without further permission by computer-based and other information-service systems. Permission to republish any other portion of the paper must be obtained from the Editor. 0018-8670/06/$5.00 © 2006 IBM

among asset types. This made it possible to consider using automation with reusable assets.

Models, a reusable asset, provide a unique opportunity to mitigate complexity, improve consumability, and reduce time to market. We proceed by addressing each of these areas.

Complexity

The demand for more sophisticated solutions is one impetus to the increase in complexity of our software development projects. Increasing complexity is also driven by the pervasive nature of software in today's business and government processes. With software reaching into all aspects of business and government, software design and testing have become two of the critical factors to address the inherent complexity.

Some see complexity in software stemming from the variety that software organizations face—the variety of methods and tools available, the variety of software and hardware, and the variety in skill sets and organizational structures. The value of decomposing the technology stack that organizations use to deliver solutions can be argued, but with that comes an increasing number of elements and combinations. All of these factors—methods, tools, software, hardware, and organizational structures—add to the increasing complexity of software.

In *The Economist*,[1] an article on managing complexity outlined some of the well-known software project failures and posited that a fundamental reason for such failures is the lack of tools for developers and management that scale to the level of complexity required by today's solutions. Specifically, poor software design was identified as a fundamental element of this failure: As software has become more and more pervasive in business and government, and more complicated, the impact of poor software design has been steadily growing.

Consumability

From experience we learn that the longer it takes to locate, evaluate, use, and extend an asset the more difficult it is to preserve its value. Many of us have experienced the frustration of trying to find and understand work someone else performed. If it takes us longer than what seems reasonable, then we often look elsewhere or stop looking and re-create the content ourselves.

There are no hard-and-fast rules about how long that time is, but some common sense prevails. If we are looking for a model that describes the architecture for a system that our team will build for the organization, it is reasonable that we may take several days or more to find and evaluate it. Conversely, if we are looking for a sort routine, then we are less inclined to spend such a lengthy time. Finding the asset is only part of the problem. Understanding its purported solution and its relevance to the context at hand is another challenge to reusing it. The ability to comprehend what has been created and shared is the next major hurdle. There is a balance between complexity and comprehension, and often solutions are provided that are complex and difficult to understand. In the end, this can be summarized as *consumability* and *ease of use*.

Poulin points out that one aid to achieving consumability is to organize the assets in a consistent manner.[2] He notes from a study on the topic that using a standard layout lets a potential reuser quickly scan the important aspects of a component, such as text descriptions, pseudo-code, illustrations, and implementation information. More is said on this topic later in the paper.

Time to market

The notion of time to market is comprised of the individual daily wins an organization makes in its software development process. This includes the notion of time to value, a test that reusable assets must pass constantly to ensure their investment. *Time to value* means discovering and understanding the right model for the relevant context in a timely manner to achieve productivity improvements. It is directly impacted by the amount of time required to find the right model, but even more so by its reusability. Two factors that make an asset reusable, impacting its time to value and thereby affecting the organization's time to market, are its complexity and its comprehensibility.

There are few metrics dealing with either of these two factors. Quantitative metrics include the MacCabe Cyclomatic Complexity metric[3] and the Halstead Software Science metrics.[4] These metrics focus on program logic, structure, and lines of code, but they do not directly translate to the complexity of a model asset. Several studies that focused on comprehension concluded that if users were presented with consistently structured information

and artifacts regarding assets, the comprehensibility of those assets improved.[5,6] We can conclude that if an asset is truly comprehensible, offers minimal complexity, and solves a recurring problem, and if the consumer of the asset can discover it quickly, then that asset has its best chance at providing value in a timely manner. It is the aggregation of many time-to-value wins that can ultimately affect time to market.

Show me the money

Walker Royce describes the context within which asset reuse flourishes[7]: In general, things get reused for economic reasons. Therefore, the key metric in identifying whether a component is truly reusable is to see whether some organization is making money on it.

In our corporate zeal to develop technologies and solutions, we often become enamored by technical brilliance. The cycle for innovation should always be tempered by value to the customer and to the business. In Model Driven Development** (MDD**) work, the same holds true for the use and reuse of assets. We use models as the basis for creating reusable assets; these models should provide value to the customer. The model used in this paper is introduced later.

ASSET-BASED DEVELOPMENT

Several years ago, we at Rational Software began an effort to describe how to leverage software investments for future use. Asset-based development (ABD) organizes the software-related investments, requirements, models, code, tests, and deployment scripts to be used for future software project activities. The four major areas of ABD are the following:

- *Process*—Describes the life cycle of assets, both assets relevant to a project and across projects
- *Standards*—Describes the standards to be used for assets, such as standard asset packaging, and for specific asset types like services
- *Tooling*—Describes the tools necessary to work with assets throughout their life cycles
- *Assets*—Describes the kinds of assets that are relevant to a particular organization

What is the relationship of ABD and component-based development? Component-based development focuses on the specification and implementation of software bits that can be reused. ABD broadens this to include a set of asset types that are useful to personnel in roles other than the developer role and to include other points in the development life cycle, such as during the inception and elaboration phases.

Some of the high-value assets we deal with are in the early stages of a project or application design. For example, a powerful mechanism to bridge the business and information technology (IT) gap and provide flexibility and productivity to an organization would be the ability to reuse a business process model as a template, customize it for a specific project or customer need, and then realize that business process with a set of reusable IT assets, such as use-case documents, services, and models.

Nature of assets

When we formally started working on ABD several years ago, we first sought agreement with several organizations on the definition of an asset. This is the result:

> An asset is a collection of artifacts that provides a solution to a problem. The asset has instructions on how it should be used and is reusable in one or more contexts, such as a development or a runtime context. The asset may also be extended and customized through variability points.

There are three key dimensions that describe reusable assets: granularity, variability, and articulation.

Granularity is the spectrum of asset size and shape. Assets may range from fine-grained, meaning they are small in size and purpose, to coarse-grained, providing larger size and purpose and often containing or referring to fine-grained assets.

Variability refers to the asset's degree of customization. This, too, is a spectrum. On one end, the asset is fixed, and on the other, it is widely visible and changeable.

Articulation is the level of completeness of artifacts in an asset. That provides a solution. We can again view it as a spectrum; some assets have very few artifacts that aid the consumer; whereas, others are fully articulated, providing artifacts that include, for instance, requirements and testing validation. Refer

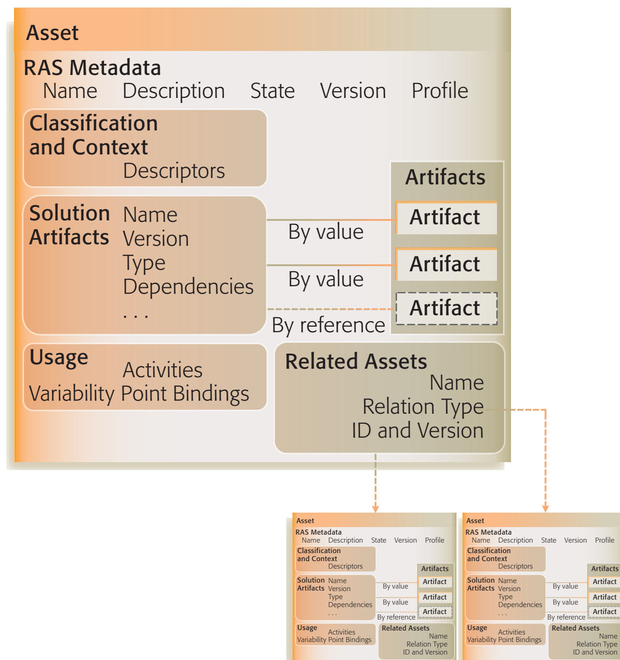

Figure 1
Assets and RAS metadata

to the Object Management Group, Inc. (OMG**) Reusable Asset Specification (RAS)[8] for further refinement on these dimensions. Models can cross degrees of granularity, variability, and articulation. It is important for us to describe the scope, customization, and supporting material for the model.

Models are assets

The general asset definition may be refined for various kinds of software assets. For example, a specific kind of asset, such as a service or in this case, a model, may specify the artifacts that must be in the asset. Some artifacts that are part of a model asset will be shown later. The following clarifies the terminology we use here.

- An *asset* is a collection of artifacts providing a solution to a problem (*Figure 1*).
- A *pattern* has a specification and one or more implementations. A pattern specification describes the solution to a recurring problem and may be implemented in many forms, such as a component or a model (using the term "implemented" loosely). The term pattern represents both the specification and the implementation when it is not necessary to distinguish between them.
- A *model* is a kind of asset, which may or may not implement a pattern specification.

Assets are packaged according to the needs of the intended user—the asset consumer. Given a model that is built to a certain level of abstraction, granularity, variability, and articulation, an architect may determine that it is highly reusable in a certain business context to solve a specific engineering problem. This model may be packaged as an asset, or other packaging schemes may be used. A component with its attendant Java** Archive (JAR) file may be created, or it may be packaged with a model that describes the design of the component, and together, they comprise the asset.

For our purposes, models are treated as assets. Later, we cover the use of models to capture the implementation of IBM Patterns for e-business and organize and package these models as reusable assets that can be searched and reused for a specific development platform.

The OMG Model Driven Architecture** (MDA**)[9] describes a model organization for separating business and application logic from the platform technology. The approach described here for organizing models as reusable assets can be used with models organized according to MDA, but MDA is not a prerequisite for using the techniques described herein.

MDA describes several kinds of models, such as Platform Independent Models (PIM) and Platform Specific Models (PSM). These models can be packaged as reusable assets. If there is a situation in which some recurring business concepts can be applied across multiple applications and implementation technologies, it can be valuable to invest the effort to create the PIM and PSM models and any associated transformations, each of which may be a reusable asset and stored in an asset repository.

Life cycle of a model asset

The life cycle of assets includes the following major workflows:

- *Candidate asset identification*—Identify potential assets.
- *Asset production*—Harvest and create artifacts, packaging them into reusable assets.
- *Asset management*—Review, certify, manage version, and publish.

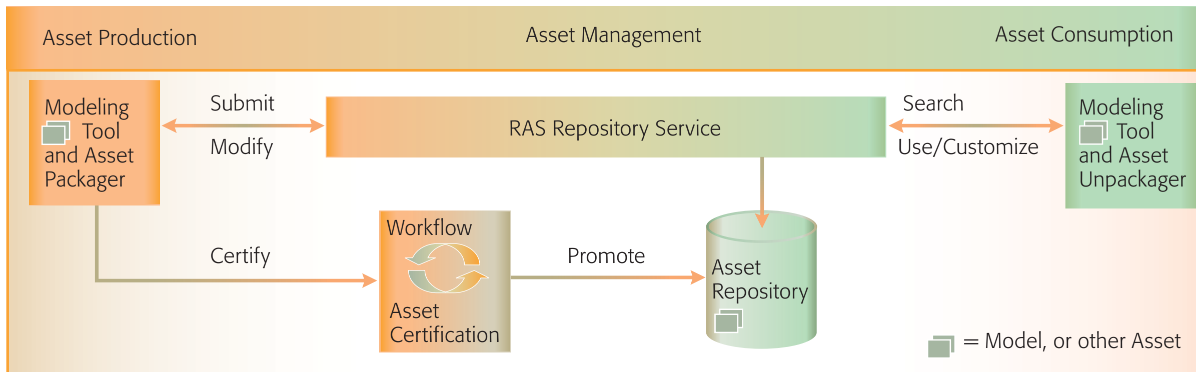

Figure 2
Tools that support the asset life cycle

- *Asset consumption*—Search, browse, reuse, and provide feedback.

A model is first identified as a candidate asset and then produced into a reusable asset for a specified context. It is then reviewed, the version is updated, and it is published as part of asset management. Finally, the model is searched, browsed, reused, and rated as part of asset consumption.

A challenge in working with assets of any kind—be they services, models, or components—is to identify which are reusable. As practitioners, we are often tempted to think that all the artifacts we create are candidates for reusable assets. This is a dangerous practice because, as assets grow, so do their attendant costs. Time needs to be invested to identify which assets are classed as reusable. This is one of the workflows in the asset life cycle. This is done by discovering recurring problems and their solutions and the context within which an asset might be reusable in solving the problems.

Although this may sound like identifying a pattern, which it is on one level because it is a recurring solution to a recurring problem, we are not conducting formal pattern-writing workshops to identify these recurring problems and solutions. In practice, we find that if candidate asset identification is made by looking for recurring problems and solutions rather than by relying on intuition, we improve our chances of making good investments.

Standards for assets

Earlier we cited Poulin's description of consumability and the impact of having a consistent approach to organizing assets. Several years ago, Rational Software began an effort to describe the metadata of assets. This structure was modeled, documented, and published as the OMG Reusable Asset Specification (RAS).[8] Today, RAS is expressed in Extensible Markup Language (XML) for each asset. The metadata that should be considered for packaging a model as a reusable asset is addressed in the section "Example: Patterns for e-business model."

For each asset, its name, several types of descriptions, its state, and its version can be described. Different sections of RAS metadata support various activities in the asset life cycle. For example, the classification section is targeted for asset consumption, giving one or more structures and perspectives through which a potential asset consumer can find the asset. The artifacts in the solution section can be included by value or by reference. By design, the majority of the elements described in RAS are optional to allow for a spectrum of reuse formalities. Thus, for informal sharing, only a few metadata elements are needed, but for formal sharing, a larger set of elements in the metadata can potentially be used. We use RAS to package the model assets in our example later, but before doing so, we review the kinds of tools that are necessary to support model assets through their life cycle.

Tool support for assets

Various kinds of tools play a role at points in the asset life cycle (*Figure 2*). To create a model as a reusable asset, a modeling tool is, of course, required, and there must be a capability in the tooling that can package assets according to RAS. The asset can be submitted to the asset certification

workflow, where it is reviewed. An asset repository is necessary for storage and versioning. Finally, the asset is made visible to the target asset consumers, who then use an asset unpackager tool and the relevant modeling tool. There are other ancillary scenarios through this life cycle (not covered in this paper), such as creating a new version of a model asset, submitting defects for an asset, and providing feedback and a rating for the model asset.

Having explained the fundamentals of ABD, we now present a sample model to use for creating and publishing a model asset.

EXAMPLE: PATTERNS FOR E-BUSINESS MODEL

IBM Patterns for e-business (P4eb) is used for our model.[10] The focus of the example is not on the patterns themselves; therefore, it is not necessary for the reader to be familiar with them. Rather, this model is used to identify candidate assets and to discuss how to organize and package the model to be reused by others. The techniques outlined here may be used with other models.

This model is built in IBM Rational Software Architect (RSA) 6.0.1.[11] There are four major packages that organize this model, which is a single file on the file system.

The IBM Patterns for e-business capture a layered set of architectural patterns that bridge the business and IT gap. They provide new levels of abstraction compared with traditional patterns (e.g., Gang of Four Design Patterns[12]), extending the use of patterns to earlier phases of solution design and development (e.g., inception and elaboration). A selected group of traditional design patterns become relevant within the context of an architectural pattern identified by P4eb. These patterns can be used to help design the architecture of an e-business application by reusing existing, tried-and-true approaches (patterns).

What is the value of this model? The model is intended to guide architects in the proper selection and structure of both their operational architecture and their application architecture. These patterns in particular describe how the architectures should be designed to meet the needs of the business. The operational architecture describes how business and IT services and components map onto a set of servers. The application architecture describes how the services and components are designed.

When these concepts are mapped to P4eb, the application patterns provide high-level application architectures, the runtime patterns provide high-level operational architectures, and, as we would expect, these are related. Finally, the runtime patterns are mapped to products, adding more context and refinement (though this is beyond the scope of this paper).

IDENTIFYING MODELS AS REUSABLE ASSETS

There are many views on how much time is spent in the software industry on projects in various phases and activities of the development life cycle. Borrowing a general profile from Poulin's work, he cites the following approximations[2]: 15 percent requirements definition, 15 percent design, 20 percent code generation, 30 percent test, and 20 percent administration. This sets the context for the parts of the software-development life cycle that are likely to be impacted through model use and reuse.

Among the many kinds of models, we are talking about models that aid software development, and within that group, we are discussing Unified Modeling Language** (UML**) models, which are useful in the requirements, design, code, and test activities. We identify a model as a reusable asset by first asking, "What problem is recurring? What is the recurring solution to that problem? What context are we talking about?"

In the case of P4eb, the problem stated by the authors was this: Systems in many organizations exist as islands but need to be knitted together to provide solutions that meet the changing needs of the business in the context of the digital economy. A set of prescribed architectures is needed that meets some general set of business needs in today's Internet world.[13] The solution is a set of models that describe several architectural views.

A subset of these patterns, which was captured in an RSA model, is the starting point in later sections for going through the steps of creating the model as a reusable asset. But first, we provide some background on models and their use to address some of the issues we face.

Reusable models to mitigate complexity, consumability, and time to market

Models have been used for quite some time. According to Schichl:

> The word 'modeling' comes from the Latin word modellus. It describes a typical human way of coping with reality. Anthropologists think that the ability to build abstract models is the most important feature which gave homo sapiens a competitive edge . . . . By 2000 BC at least three cultures (Babylon, Egypt, India) had a decent knowledge of mathematics and used mathematical models.[14]

Complexity: Addressed by well-formed models

Schichl identifies one of the early graphical models used in astronomy, in which Ptolemy created a model of the solar system in 150 A.D. by using cycles and epicycles. Apparently this model was used until 1619 when a better model was devised, the fundamentals of which are still used today.[14]

Models provide a means of communication, and when done well, they provide useful abstractions that can last for quite some time. How many of our models will last beyond two or three versions of a software application, let alone nearly 15 centuries? Selecting the proper abstractions in a model for a specific context is critical. However, abstraction is a double-edged sword.

Gabriel cautions against the overuse of abstractions:

> The problem is that people are taught to value abstraction above all else, and object-oriented languages and their philosophy of use emphasizes reuse, which is generally good. However, sometimes the passion for abstraction is so strong that it is used inappropriately. Abstractions must be carefully and expertly designed, especially when . . . reuse is intended.[15]

Abstractions in code as well as in models are used to hide detail, but by doing so, one may not be able to understand what the code does or what the model means. The question to ask is, "At what point in my modeling do I make heavy use of abstraction and at what point do I make light use of it?"

Part of the answer lies in the intended use of the model. When a model is intended to reach an audience that is making decisions which are not based upon the fine details of the model, then the heavy use of abstraction is justified. If the model is intended to communicate the essence of the solution and guide the user through the details, then less abstraction is justified. For instance, to whom is it valuable to reverse engineer some Java classes into a UML class diagram? The answer depends on what is being captured in the model and who expects to use it. If the essence of the class structure and relationships is communicated visually and if the intent is to communicate to software engineers and architects the static nature of the class design so that they can conduct impact analysis and review the overall software design, then this is likely to be a proper abstraction. However, one would never show this to a business analyst seeking an IT solution to a business problem.

One of the challenges we face is increasing complexity. Abstraction in models is a powerful means to rise above complexity, but it must be used in a manner appropriate to a specific audience. Properly organized models reduce the effort to understand the abstractions they communicate.

Consumability: Model organization and metadata

Consumability is the ease of use by which a model can be approached, its organization understood, and a determination made concerning how to apply it to one's needs. To be consumable, both a model's structure and its diagrams should be well organized, and it should be packaged with metadata that further describe its intent and intended reuse context.

Time to market: Finding the model, getting at the value

Value is created by discovering and understanding the right model for the relevant context in a timely manner to achieve productivity improvements. For this to happen, a model must be organized, packaged, and shared as an asset with minimal effort. The focus now shifts to examining how to organize models as reusable assets and the impact that can have on models and their users.

ORGANIZING MODELS TO BE REUSED

The structure of our model is shown in *Figure 3*. It is organized with two top-level elements: a documentation folder and the model file itself, `p4eb_patterns.emx`. A document that discusses the modeling conventions to which the model adheres should be included in the documentation folder to aid in the reuse of the model. It can be seen that we use a UML package called `zAssocations`. In it, we place the associations exposed by the RSA Model

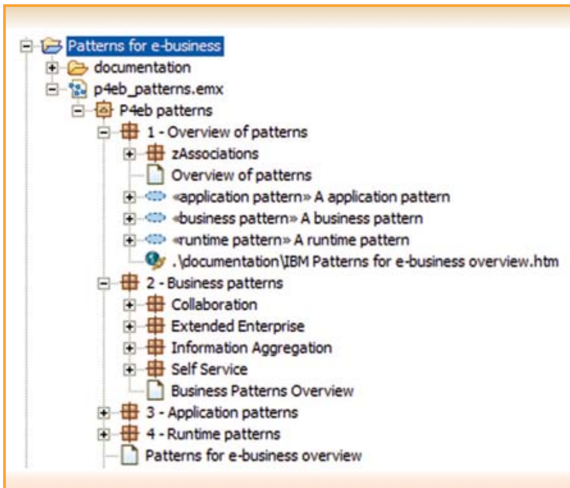

Figure 3
Initial P4eb model structure

Explorer view to make navigating the Model Explorer easier.

Under the `2 - Business Patterns` package is another set of packages that follow the major categorization of the P4eb patterns: `Collaboration`, `Extended Enterprise`, `Information Aggregation`, and `Self Service`. This structure is used throughout the model in each of the major pattern packages: Business Pattern package, under which are Application Pattern packages, under which are Runtime Pattern packages. We chose this structure to focus the user on the relevant subset of patterns, thereby reinforcing previous decisions and not overwhelming the user with all possible patterns from which to choose.

One of the first questions to ask is, "With whom do we intend to share this candidate asset?" It is very important to understand the anticipated skill level of the target consumer. The answer for our example model is that it is intended to be shared with architects.

The next question is, "How do we expect the architects to approach the problem of determining which pattern to use in the model?"

The following are the assumptions we made about the use of this model:

1. We assume that architects will be fairly new to these patterns; as such, we expect they will first review high-level information about the patterns to become familiar with them. Hence, some documentation or pointers to the P4eb site should be included.
2. Next, we assume that architects will determine the nature of the business problem to be solved, evaluate the business patterns, and select the relevant one. From there, the constituent application and runtime patterns will be selected.
3. Finally, we assume architects will use the runtime pattern models as a template to refine for their environment and to map products that will be used.

Understanding the cascading structure of this set of patterns is important because once a business pattern is chosen, we immediately ignore a whole set of other application and runtime patterns that are not relevant. Thus, this is a guide on how to organize the model. Right now our model is organized according to the preceding categories, but our objective is to align the model closer to the reuse boundaries. Thus, we adjust the model organization with the following packages:

- `Overview of patterns`
- `Collaboration`
- `Extended Enterprise`
- `Information Aggregation`
- `Self Service`

The `Business pattern` packages have been brought to the top as peers of the `Overview of patterns` package. Our premise here is that because of the first assumption, the architect will start with the `Overview of patterns` package and review the patterns. According to our next assumption, the architect will then select a relevant business pattern. Once this decision is made, the pattern cascades into application patterns and finally, into runtime patterns. This structure allows for a mixture of opportunistic navigation through the patterns, as well as a prescriptive, hierarchical structure for reviews of the constituent application and runtime patterns.

When we built the original set of models for P4eb, we organized the patterns according to their primary classification, meaning that all business patterns were in a business pattern package, and so forth. This created some production challenges, as many diagram and model relationships had to be created.

This also created some reuse challenges, as it was awkward to break the model up into multiple assets along those boundaries—having one asset for just business patterns and another one for just application patterns. In the end, the affinity among the patterns in terms of how they were reused became the determining packaging structure and is reflected in what is presented in this paper.

A key principle in packaging for reuse is to always evaluate the skills of the expected asset consumers, the context in which they will reuse, and the approach they will take to reusing the assets. In short, know your audience.

As seen at the top of ***Figure 4***, a new project was created in RSA called `Patterns for e-business2`, as we wanted to retain the migration of this model. The figure shows the realignment of the model based on the preceding assumptions. Following the `Directly Integrated Single Channel Pattern` all the way through, we see the organization of business pattern to constituent application patterns to constituent runtime patterns.

Another modeling convention that we have found useful is to allow for several forms of model navigation. ***Figure 5*** illustrates this by showing the Self Service business pattern connected to supporting documentation. Next, the business pattern is connected by diagram links to its constituent application patterns. Thus, model organization is more than how it appears in the Model Explorer; it is also about its navigability on the diagrams themselves. We seek to minimize the amount of time and effort required to understand the asset. By selecting one of the application pattern diagram links on Figure 5, we can then navigate to the application diagram, which itself has links to constituent runtime patterns, and so on.

It is critical, once the model has been properly organized and packaged, for it to be validated. The assumptions we make are critical and must be proved or disproved before broad delivery of the asset.

With this model organization in place, we now turn to packaging the model for reuse by others.

PACKAGING MODELS TO BE REUSED

Our current P4eb model is now close to being ready to be packaged as a reusable asset. As we think about sharing this model as a reusable asset, we are reminded of our expected use scenario; namely, the architect will review the pattern overview, then select a business pattern, and then the constituent patterns. There are well over 100 patterns for e-business, and not all of them are yet captured in this model. The life cycle of the model elements also needs to be considered, as there will be iterations on the model and new versions.

Given the number of patterns, the need for iteration on the model, and the expected use style, the model is broken up into the five following P4eb RAS assets: Overview, Collaboration, Extended Enterprise, Information Aggregation, and Self Service. To do this,

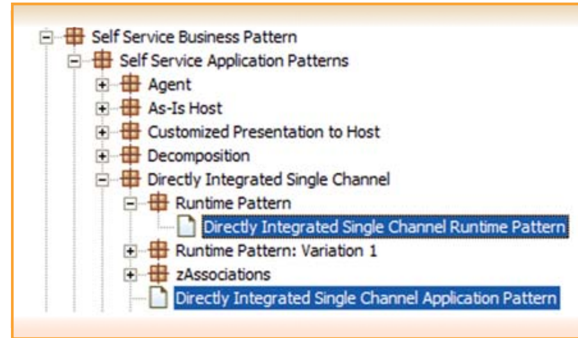

Figure 4
P4eb model reorganization: Self Service

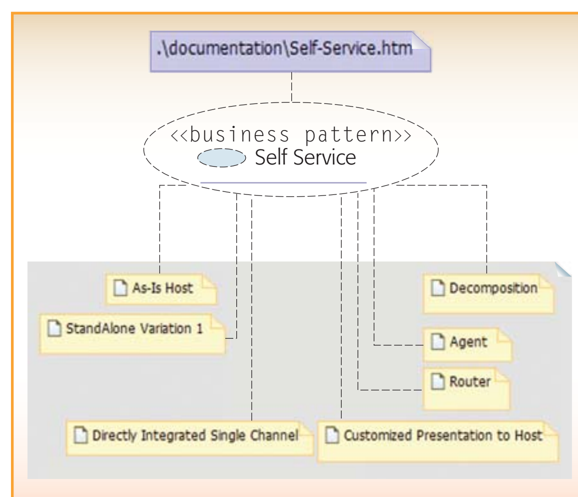

Figure 5
Organization of the business pattern diagram navigation

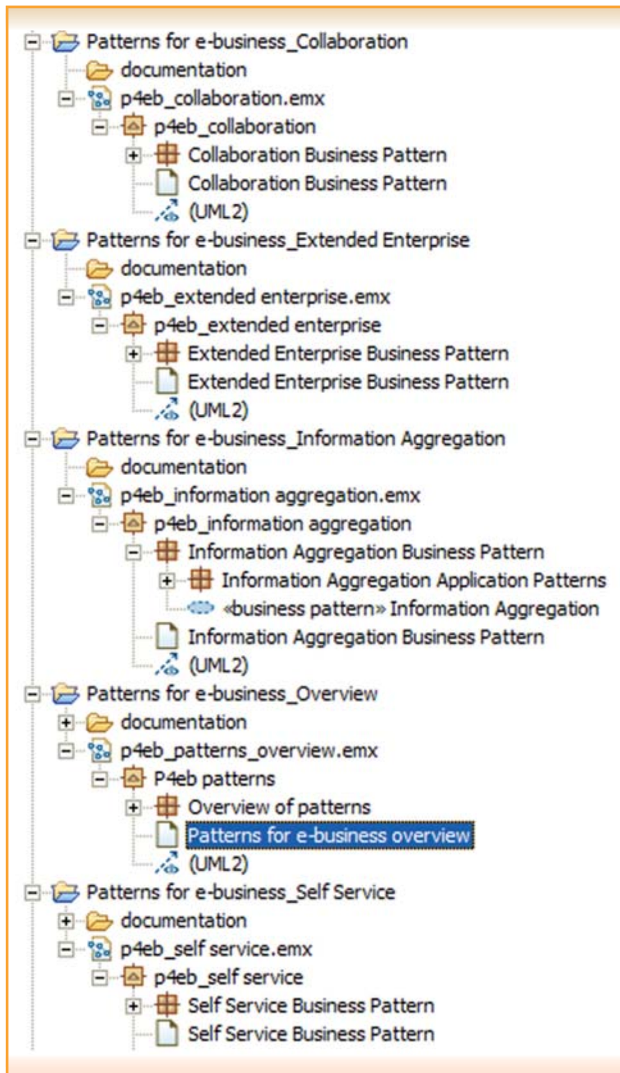

Figure 6
Restructuring the RSA/Eclipse projects for model reuse

we refine the model one more time and create five RSA projects, each containing a documentation directory and a UML model with the relevant constituent patterns and other model elements. We then package the models and their RSA/Eclipse platform projects as assets, classifying them with the RAS metadata structure, and declare the relationships among the assets. This makes it possible to search, browse, and retrieve the overview asset and then to find the business pattern asset of interest. This also improves our ability to iterate on the models, minimizing the impact to other pattern models. This strategy could be refined into more fine-grained assets as needed.

Figure 6 shows the RSA projects and models that were created based on the decisions we made. Note that the Model Explorer structure, meaning the constituent application patterns and their constituent runtime patterns, are still in place.

RSA provides several mechanisms to package the assets. We create five RAS assets, one for each of the RSA projects in Figure 6. To begin, we create a RAS manifest file and fill in the relevant metadata for the `Patterns for e-business Overview` RSA project. The RAS manifest file is an XML document that contains information about a reusable asset. Assets can also be packaged in the RAS format without using RSA. *Table 1* shows a subset of the metadata entered for the `Patterns for e-business Overview` asset.

An asset can have multiple classifications. In Table 1, we created multiple descriptors; some describe the context for which this asset is reusable, namely the Development Environment and the phase in the Rational Unified Process* (RUP*)[16] for which this asset is intended to be used.

We are now ready to package the assets. This is done using the File > Export > RAS Asset options. For each asset, we select the respective `.rmd` file (RAS manifest file), and the wizard does the rest. We published each of these assets in the local RAS repository.

In this example, we published the assets in the repository rather quickly. In practice, the assets should go through extensive review and validation. The goal is not to produce as many assets as possible; rather, it is to produce the right set of assets to positively impact the business. This speaks to the need for development-time governance of assets, which is beyond the scope of this paper. In general, a customizable workflow and a set of policies that can be enforced throughout that workflow are necessary for the review, certification, and publishing of an asset.

FINDING AND REUSING MODEL ASSETS

There are two major styles of searching for assets: opportunistic and systematic. Assets may be displayed in folders in the repositories, rather opportunistically. In opportunistic searches, we browse the repositories or conduct searches based on keywords and phrases. Opportunistic-style tech-

Table 1 RAS metadata for P4eb Overview asset

Element Name	Value
Name	P4eb Overview
Short description	This asset provides documentation and overview models of the Patterns for e-business.
Version	1.0
Description	This model should be used to help select which Pattern for e-business Business Pattern should be used. Below are the assumptions for using these assets. • Architects will be fairly new to these patterns; as such, we expect they will review high-level information about the patterns first to become familiar with them. Hence, we should include some documentation or pointers to the P4eb site. • Next architects will determine the nature of the business problem to be solved, evaluate the business patterns, and select the relevant one. From there, the constituent application and runtime patterns will be selected. • The architects will use the runtime pattern models as a template to refine for their environment and to map products that will be used.
Classification Section	
Author	IBM
Keyword	Pattern, P4eb, architecture
Known uses	Guide architects to selection of architecture
IDE	RSA 6.0.1
Modeling language	UML
RUP phase	Inception, Elaboration
Solution Section	
Artifact.Name	IBM Patterns for e-business Overview (other documents here were omitted for space consideration)
Artifact.Name	p4eb_patterns_overview
Artifact.Type	UML model
Artifact.Reference	Patterns for e-business_Overview/p4eb_patterns_overview.emx
Usage Section	
Activity.Name	Start with Patterns for e-business Overview diagram
Activity.Name	Read pattern overviews and select top-level P4eb pattern
Activity.Name	Import selected P4eb pattern asset and follow instructions
Related Assets Section	
Asset.Name	Pattern for e-business — Collaboration patterns
Asset.Name	Pattern for e-business — Extended Enterprise patterns
Asset.Name	Pattern for e-business — Information Aggregation patterns
Asset.Name	Pattern for e-business — Self Service patterns

niques are used in reuse, but we have found that searching for assets in this manner can erode the value proposition of the assets if reuse is scaled to larger groups of people and across boundaries, teams, time zones, and skill sets.

The other style of searching is systematic; this is a more prescriptive form of reuse. It dictates associations among assets and identifies the assets to be used. Recipes are a good metaphor for this style of searching: We have a list of ingredients (assets) and the guidance to "mix" them. Recipes also have the benefit of being customizable. Rather than searching for all the ingredients, a solution (the recipe) can be sought that points to all the ingredients (assets) needed. It offers the advantage of saving time.

More value can be created by producing a set of recipes that mix multiple assets together to form larger-grained, yet customizable solutions. We have concluded that the prescriptive reuse of models and other assets holds some promise for our asset-based development efforts and most notably, for the business.

Many techniques are used to search for assets. Recipes, taxonomies, ontologies, and classification schemas can be used. A classification schema provides a structure that classifies assets. The values from the classification schema are stored in the asset's classification section. The most difficult and often least valuable approach for searching for assets is using keywords. Expecting someone to enter a keyword that is exactly what the asset producer used when packaging the asset decreases the likelihood that the appropriate asset will be found.

When the reuse scope and community is broad and crosses organizational boundaries, time zones, skill sets, and other elements, and as the number of assets grows in a repository, it is helpful to have a searching mechanism that lets the consumer navigate the structure itself (the classification schema or the taxonomy) from which to select values. Another key technique is to use ontologies to model the grammar of a domain and create relationships among terms. Then the asset consumer searches by using terms from the domain but is not required to know which terms were used to package the asset because the model of the grammar provides the association of terms. A formal language for describing ontologies is Web Ontology Language (OWL).[17]

Earlier we stated that keyword-style searching for assets was often the least valuable. However, many times customers ask for Google-style searches. This approach provides a nice balance of searching where unstructured, opportunistic searching can take place but the search engine can estimate the context for the terms and apply some structured benefits to the search.

It is difficult enough, even with these structured searching mechanisms, to find and evaluate an asset, but it is even more difficult to understand how multiple assets can be used together when one is not familiar with them. Again the notion of recipes may offer a technique to mitigate reuse costs. Asset producers can capture this knowledge and identify the assets themselves or the categories of assets that should be used as the "ingredients."

SUMMARY

This paper introduced model-driven development, including a brief history of models and a caution concerning the use of abstraction. We then proposed that models be seen as assets and discussed the fundamentals of asset-based development. An example for organizing and packaging models as reusable assets was given.

There are many challenges that face software organizations today. These include increasing complexity, solutions that are hard to use, and time-to-market constraints. If we select the right models for others to use and make them accessible for easy reuse, then we can mitigate the impact of challenges, and we can positively affect all phases of the software development life cycle.

*Trademark, service mark, or registered trademark of International Business Machines Corporation.

**Trademark, service mark, or registered trademark of Object Management Group, Inc. or Sun Microsystems Inc. in the United States, other countries, or both.

CITED REFERENCES

1. Managing Complexity, *The Economist* online (November 25, 2004), http://www.economist.com/printedition/displayStory.cfm?Story_ID=3423238.

2. J. S. Poulin, *Measuring Software Reuse: Principles, Practices, and Economic Models*, Addison-Wesley Professional, New York (1996).
3. T. J. McCabe and A. H. Watson, “Software Complexity,” *Crosstalk, Journal of Defense Software Engineering* **7**, No. 12 (December 1994), http://www.stsc.hill.af.mil/crosstalk/1994/12/xt94d12b.asp.
4. M. H. Halstead, *Elements of Software Science*, Elsevier North-Holland Publishing Co., New York (1977).
5. M. C. Linn and M. J. Clancy, “The Case for Case Studies of Programming Problems,” *Communications of the ACM* **35**, No. 3, pp. 121–132 (March 1992).
6. David R. Musser and Alexander A. Stepanov, *The ADA Generic Library: Linear List Processing Packages*, Springer-Verlag, New York (1989).
7. W. Royce, *Software Project Management: A Unified Framework*, Addison-Wesley Professional, New York (1998), p. 38.
8. *Reusable Asset Specification, Version 2.2*, Object Management Group, Inc., http://www.omg.org/technology/documents/formal/ras.htm.
9. Model Driven Architecture, Object Management Group, Inc., http://www.omg.org/mda/.
10. IBM Patterns for e-business, IBM Corporation, http://www-128.ibm.com/developerworks/patterns/index.html.
11. IBM Rational Software Architect, IBM Corporation, http://www-306.ibm.com/software/awdtools/architect/swarchitect/index.html.
12. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, New York (1995).
13. J. Adams, S. Koushik, G. Vasudeva, and G. Galambos, *Patterns for e-business: A Strategy for Reuse*, IBM Press, Big Sandy, TX (2001), pp. 1–2.
14. H. Schichl, “Models and the History of Modeling,” in *Modeling Languages in Mathematical Optimization*, J. Kallrath, Editor, Springer, New York (2004), pp. 25–26.
15. R. P. Gabriel, *Patterns of Software: Tales from the Software Community*, Oxford University Press, New York (1998), p. 19.
16. M. Aked, *Risk Reduction with the RUP Phase Plan*, IBM Corporation, http://www-128.ibm.com/developerworks/rational/library/1826.html.
17. Web Ontology Language (OWL), World Wide Web Consortium, http://www.w3.org/2004/OWL/.

Accepted for publication December 16, 2005.
Published online July 12, 2006.

Grant Larsen
IBM Rational Software, 10632 W. Ontario Avenue, Littleton, Colorado 80127 (gjlarsen@us.ibm.com). Mr. Larsen is currently the chief architect for asset management for IBM Rational Software. He received a B.S. degree from Brigham Young University in 1988. He works with the asset-based development strategies through process, standards, tooling, and reusable assets. Mr. Larsen has been a member of the group that developed the Reusable Asset Specification (RAS), recently adopted as a standard by Object Management Group, Inc. At Rational Software he was a member of the UML committee for Rational and composed portions of that specification. Mr. Larsen has published several journal articles, has been a guest editor, and has contributed to other books on frameworks and related technologies. ■