

A Model for Software Libraries

John M. Hunt
Clemson University
201 McAdams Hall
Clemson, SC

hunt2@cs.clemson.edu

John D. McGregor
Clemson University
312 McAdams Hall
Clemson, SC

johnmc@cs.clemson.edu

ABSTRACT

Software libraries have long been an integral element of software development. Recent advances in areas such as software product lines and extensibility mechanisms have focused renewed attention on collections, particularly heterogeneous collections, of software artifacts. The contribution of this paper is to propose a model for a *software library*. Our work creates a framework that is abstract enough to encompass many kinds of software libraries beyond those used for the sort of programming constructs normally considered. This allows quite disparate collections to be understood within the same framework. Notable to our work is a discussion of the role of context and deployment to libraries. A comparison of our model to existing models is provided. A number of different types of libraries are analyzed to demonstrate the power of our model and to show how it leads to better understanding of several types of software collections.

Categories and Subject Descriptors

D.2.2 [Design Tools and Techniques]: Software libraries; D.2.13 [Reusable Software]: Reusable Libraries; D.2.13 [Reusable Software]: Domain Engineering; D.2.1 [Requirements \ Specifications]: Methodologies

General Terms

Design, Standardization

Keywords

Modeling

1. INTRODUCTION

Software libraries have long been an integral element of software development. Recent advances in areas such as software product lines and extensibility mechanisms have focused renewed attention on collections, particularly heterogeneous collections, of software artifacts. The contribution of this paper is to propose a model for a *software library*,

briefly contrast our model to existing models, and show how it leads to better understanding of several types of software collections including: Dia Shape Sets, Eclipse plug-ins and software product line asset bases.

This model creates a framework that is abstract enough to encompass many kinds of software libraries beyond those used for the sort of programming constructs normally considered as libraries. The model allows quite disparate collections to be understood within the same framework. Notable to our work is a discussion of the role of context and deployment to libraries. A comparison of our model to existing models will be provided. A number of different types of libraries are analyzed including Dia Shape Sets, the Java Swing Library, Eclipse plug-ins and software product line asset bases.

2. THE IMPORTANCE OF MODELS

The development of a standard vocabulary, requirements, and supporting models is an important step in the maturity of a discipline. Having the common understanding of an area that a model can foster has many advantages including:

- Improved ability to discuss problems and solutions
- A common understanding of the available design space
- An ability to compare solutions and techniques
- Guidance for those new to an area

The OSI seven layer model of a computer network [11] is a classic example of how a model can support the evolution of an area.

A better understanding of a domain, in this case the domain of libraries, should allow development of better products. For example, our model explains the connection between a library and a number of issues including how its assets are deployed onto production systems and the relationship between deployment issues and an asset's binding times. This is an area that is frequently ignored in library development, but may greatly effect the usefulness of the library. By providing advice about these issues we enable the development of better libraries.

3. CURRENT USAGE

While libraries are referred to frequently in the literature, a comprehensive definition has been lacking. Most writers are content to use the formula: "A library is a collection of X." Where X could be almost anything: functions, classes, architectures, use cases, test cases, documentation, specifications, or other artifacts. Examining how the term library is currently used, we find two different perspectives on library use:

1. A collection of software artifacts used by a developer, who is normally in a different organization from the library creators, to assist in the development of a program. Here the key problem is how someone unfamiliar with the contents of the collection finds and selects useful items. This often assumes the need to adapt the items found. The actual mechanisms, by which products are composed, are not generally discussed.
2. A mechanism that holds a collection of artifacts for the purpose of facilitating composition with a product. The composition mechanism is often defined by the operating software or an intermediate runtime environment. Dynamically linked libraries (DLL) are an example. In this case, it is assumed that the desired items can be located and adaptation is not needed. The library users' problem of understanding and selecting assets is ignored.

While not contradictory, these two different uses of the term point to different aspects of libraries, both of which must be considered to gain a complete understanding.

4. WHY CALL IT A LIBRARY?

The general notion of a library has several characteristics that apply to software libraries including:

- Library refers to both a collection of items and a facility in which to house the collection.
- Libraries typically organize their collection in a systematic manner and may limit their collection to have a common focus.
- Items are gathered together into a library to improve access and management.
- The library makes the items more public or available.
- The container for the items, be it a book case or building, improves access to the items.
- The library differs from a storage warehouse in that items are intended to be accessed frequently and individually.
- The library differs from a repository by making items more public, where as a repository removes items from circulation, making the stored items more protected, and in the process more private.
- Modern libraries are collections of many types of elements such as books, videos, computer programs, and many other elements.

The goal of a library is improved access and use of the items in its collection.

Software libraries have the additional constraint that we create them for the purpose of helping to develop products. As a result, while items in the general case are typically free standing assets, most assets in a software library will be parts or modules that acquire their ultimate usefulness only when combined with other assets, either from another library or custom assets, to form a product.

With this background, our basic definition for a software library can be stated as:

- A collection of composable assets,
- that contribute to building a product,
- aggregated within a mechanism that holds the collection for the purpose of promoting reuse by providing greater accessibility,
- for use by others.

5. A SOFTWARE LIBRARY MODEL

In this section we present the results of a domain analysis of a *software library*. We use this method to present several in-depth discussions of the concepts found in the domain. We use the Unified Modeling Language (UML) to describe the results of the analysis.

The model we present is intended to be abstract enough to cover a very broad range of libraries, not just those containing programming language artifacts. Take, as an example, a drawing program that lets its user build a complex form and then store it into a palette for future use in other drawings. Such a palette of objects, when implemented in software, can be considered a software library. We want to be careful that such an example can be described by our model. Once a model of this generality is established, it can then be specialized for more common examples, such as programming libraries, or even further specialized, perhaps for class libraries. We defer these more specialized cases to future work.

5.1 Use Cases

We begin with the use case diagram of the domain, shown in Figure 1

The domain has three actors:

- The Library Developer provides the contents of the library.
- The Library User creates or maintains a product and composes the library's assets into the product.
- The Client Product is the product composed using the library assets. For some composition mechanisms the product acts directly to compose itself with the library. For example, resolving and executing a branch to use a shared library. In other cases, the client product is passively assembled by the library user. Even in these

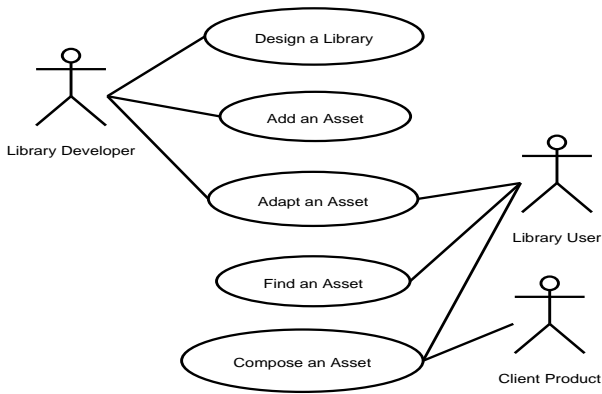


Figure 1: Use Case Diagram for Library.

passive cases, the assets in the library must conform to the mechanisms used to compose the product.

The diagram has the following high level use cases:

- Design a Library - Since we believe that a library has more coherence than simply a collection of assets it should be designed as such. One obvious design activity is scoping, as a particular library should focus on a set of related abstractions. Library design also establishes the context in which the assets are intended to be used. Libraries should be designed so that they are: complete, consistent, easy to use, and efficient [8]. A key to designing a library is understanding how it is used and how the division of roles between the library developer and the library user effects the design. This paper will focus on these use issues, rather than the design issues of a library.
- Add an Asset - The library developer creates the library by adding one asset at a time. Left implied is the ability to remove and modify assets already in the library. Adding an asset is singled out from other development activities as the result is visible outside the development environment.
- Adapt an Asset - By adaptation we mean the process of manually modifying a pre-existing asset for a new use. In the case of code, this is also referred to as code scavenging [9]. Adaptation may be applied by a developer, or a library user in the case that the library user has access to modifiable asset, typically source. Adaptation may be applied to any pre-existing modifiable asset, not just library assets. For example, code examples from a text book could serve as the source for adaptation. While library assets may be adapted there does not seem to be any unique role that the library plays in this process that distinguishes it from other asset sources, as such it is not considered further.
- Find an Asset - The library user must be able to find assets that are appropriate to his problem. The user must then be able to understand the asset in order to determine whether the correct asset has been found.

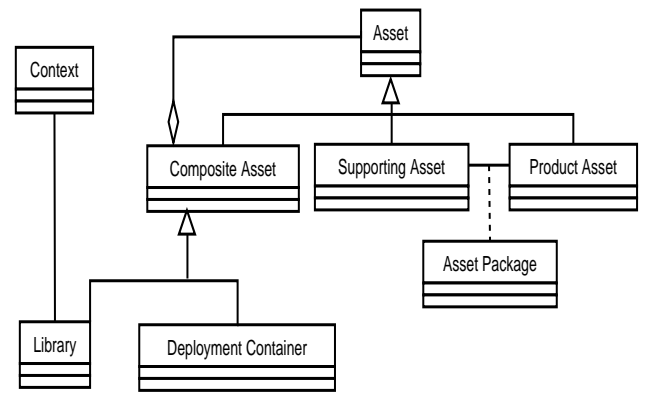


Figure 2: Top Level View of Library.

- Compose an Asset - The library user must be able to compose assets into products. The ability to compose an asset into a software product in an automated fashion is what distinguishes a library from other collection of software artifacts. Assets in the library must be designed to support composition. It should be noted that some composition mechanisms are more flexible than others. The C++ template mechanism allows the related code to be composed with a variety of variable types. This sort of flexibility, does not use manual intervention, does not change the original asset, and does not add additional assets into the library. These difference distinguish this sort of flexibility from adaptation.

Conventionally, we think of the library user as being a product developer, who selects particular pieces of a library to be composed into the product. In this case, the ability to search within the library is important. For products that have an open extension capability, the end user who is engaged in product composition, is the library user. For example, the end user might modify a web browser by composing it with a plug-in. In this case, the user typically does not search and select from among library assets, but instead decides whether to compose a feature.

Allowing a user to compose features has the effect of creating a domain specific language, a language that corresponds to the problem domain and does not require understanding of or access to the solution domain. This is an important distinction from adaptation, which requires access to and understand of the solution domain, and does not provide a new problem solving vocabulary.

5.2 Concepts

The concepts needed to describe a library and its assets are shown in a class diagram in Figure 2. We provide a glossary in Table 1 to give a brief definition of the classes in the diagram. The key abstractions in the diagram are assets of various types. Of particular note are two types of composite assets whose main purpose is to collect assets - the library and the deployment container.

5.2.1 Relationships between Assets

Table 1: Glossary

Asset	an item that has enough value for us to collect. Three types of assets - product, supporting, composite
Product Asset	an item that is composed into a product
Supporting Asset	an item that supports use of a product asset, such as documentation
Composite Asset	a collection of assets; Types of composite assets - library, deployment container
Asset Package	shows the relationship between a product asset and its supporting assets
Library	collects assets and deployment containers
Deployment Containers	collects product assets in a way that is composable with a client product

The dominant relationship among the assets, at a conceptual level, is described by the composite design pattern [6]. This recognizes that one relationship among assets is hierarchical, i.e., assets may be composed of other assets. This has different implications for the different asset types. In the case of simple assets - product and supporting - this means that existing assets in the library may be used to compose new library assets.

For deployment containers the composite pattern provides three types of relationships:

1. A deployment container could contain a collection of simple assets for which it provides a composition mechanism. This is the most common case.
2. A deployment container could contain other deployment containers as one way to supply additional composition mechanisms. An example is an Eclipse plug-in which uses a jar file to hold executable assets.
3. A deployment container could contain a library. This would provide a way to move the library as a unit to other development systems.

For a library the composite pattern provides three types of relationships:

1. A library contains a collection of all the simple assets, this is the reason we have a library.
2. A library contains a collection of deployment containers. Having more than one type of deployment container allows the library to offer more than one type of composition mechanism to client products.
3. A library contains a group of related libraries. The context of a contained library must inherit the context of a containing library. A library may have more than one parent library.

5.2.2 Assets

The items we collect in the library are assets. We call them assets because we assume that they have value or we would not bother to collect them. We divide assets into product and supporting assets. Product assets are composed into a product. A supporting asset helps us make use of a product asset. For example, a Javadoc page for the Swing GUI is a supporting asset for the Swing class library. A product asset is not necessarily an executable asset. A help file shipped with the product is an example of a non-executable product asset. A build script is an example of a supporting asset that is executable.

What distinguishes a software library from other collections of software assets is the ability to compose library assets with a product, identified in the compose an asset use case. The importance of supporting this use case is what motivates the distinction between product and supporting assets. The asset package relationship groups supporting assets with the product asset with which they assist. If an asset package does not include a product asset, it cannot be composed into a product.

While executable modules, such as program functions, are the oldest and most widely used asset type, every phase of software development can take advantage of reusable assets, and a library can assist in increasing the reuse those assets. During the specification phase, we might use a library that includes standard use cases for some category of product to compose the use cases for our product. During the high-level design phase, we might use a library that includes UML diagrams describing standard subsystems to compose the product architecture. During the detailed design phase, we might use a library that includes pattern languages to guide the completion of the design. During the implementation phase, a library that includes code fragments might be used by a program generator or an aspect weaver.

In most existing libraries, the product assets of a library tend to be of the same type or at least apply to the same development phase. However, the assets in a library do not need to be homogeneous. We can gain considerable power by including all of the assets needed to produce a particular product. For example, a library might include assets, such as UML diagrams, that can be used in the design phase along with the executable assets needed for that product.

The assets we set out to collect are product assets, those assets that become or produce part of a product. The most common example is a source file in some high level language that compiles into a linkable executable. However, other inputs to a build process, such as frames, meta-models, layers, etc. may play the same role. As might a collection of predefined shapes for a drawing program. The supporting assets are collected to assist with using a product asset. If the product asset is removed from the library, the supporting assets should be removed as well.

The idea that assets are collected to build products means the ability to use an asset in multiple products is a planned result. In this view there is no such thing as a truly “general purpose” asset, that can be used in every setting [10]. We build a particular thing and the thing we are building

imposes requirements on its parts [2]. The possibility of meeting these requirements by chance are quite low, this rules out several approaches to acquiring assets that have been commonly used.

5.2.3 Library

The term library refers not only to the contents of the asset collection but also the mechanism used to collect and manage the assets. The library has several attributes that are unique. The library is the level of abstraction where all three of the users we identified (library developer, library user, client product) are addressed.

To assist library users, library developers provide a number of supporting assets to explain and guide using the library. These library level aids might include such things as tutorials on the use of the library and example programs that show how library assets might be used to solve a common problem. The library may contain one or more search mechanisms, whose primary purpose is to assist the library user in finding assets. To support client programs, libraries should make product assets available in a composable way, often through deployment containers.

The library should add value beyond the value of the contained assets. The services provided by a library include:

- Collection support. Allow assets to be used as a group or individually. For example, copy or move an entire library instead of each of the contained items.
- Access or composition support. Assets are intended for use by client programs outside of the library. It should be possible to compose an asset in a client program without copying it out of the library. The library may provide multiple composition mechanisms that support different binding mechanisms.
- Selection support. A recognized truism in reuse is that an asset must be found before it can be reused [9]. This problem is more obvious in libraries since library users are a separate group from library developers. The library should provide support to the user to find assets.

5.2.4 Deployment Containers

Deployment containers allow the library's product assets to be composed with client programs without access to the library's development environment. Independently deploying a subset of the library assets in a composable way is the key to a library's ability to share and make public its assets, in contrast to other deployment constructs, such as repositories¹. It is so fundamental to the library that the deployment container is often confused with the library. As can be seen from the use of the term "dynamic linked library" for what is actually only a deployment mechanism.

We can divide deployment containers into two categories:

¹Recent literature often uses repository as a synonym for library, but typically ignores the aspect of deploying assets away from the development system. It is not clear if this difference is meant to distinguish the two.

1. Those that deploy library assets to product development systems. An example is the statically linked library, which is composed during the development phase by a linker. Deployment to a product development system includes providing those supporting assets that assist the product developer.
2. Those that deploy library assets to production systems. Examples include the DLL (dynamic linked library) which the client links to at runtime. Putting deployment containers on the production system has the advantage that a single copy of the library assets can be used by multiple client programs. For example, most operating systems provide only a single I/O library for all of the hosted applications.

The composition mechanism supported by a deployment container determines the point in the software development process at which composition with the client program takes place; this is known as binding time. Libraries are usually assumed to have a single binding time, but this does not have to be the case. A library may have multiple deployment containers, each supporting a different binding time.

Once dispatched from the library development system, the deployment container and its enclosed assets are no longer under the management of the source control system. Therefore, deployment containers need a method of versioning independent from that of other development assets.

5.2.5 Context

All software artifacts are used in a particular setting or environment [10]. In this model, context represents the environment in which we intend to use the library assets. Context is included in our model to support the compose asset use case. As Alexander discussed in his classic teakettle example, the correctness of an artifact can only be understood in relation to the context into which we expect them to fit [1]. Alexander points out the dimensions in which an artifact must fit its environment are not enumerable, so a complete model of context is not possible. We show the major areas to be considered and discuss how context affects product development. Figure 3 shows our view of context as it applies to libraries.

We divide context into two parts [5]:

- the product domain, which describes what we are trying to build
- the solution domain, which describes how we can build a product

We further divide the solution domain into two parts:

- the platform, which specifies the library's dependences
- the architecture, which defines how the assets may be composed

If our library is contained within another library, its context includes that of its containing library. The contained library

may impose additional requirements but must continue to meet all the requirements imposed by the containing library.

We associate the context with the library rather than the individual assets. This differs from other proposals, notably the OMG Reusable Asset Specification (RAS), discussed in section 5.2. There are several advantages to our approach:

- Context can be used to group assets. For a given project, we are typically only interested in a particular context. For example, a particular hardware platform or language may have been chosen for the project. Placing the context at the library level allows the entire collection to be evaluated, at least at this coarse level of decomposition, for suitability to the project without examining each asset.
- One of the barriers that prevents a library user from using preexisting assets is the difficulty in understanding those assets. This difficulty can be considered an additional cost of using the library, which in turn may cause the library user to choose to develop a new purpose built asset as a substitute for a preexisting asset from the library. Placing context at the library level allows the cognitive effort of understanding an asset to be reused, at least in part, over the other assets in the library. Thus, lowering the average cost of using a library asset.
- A shared context makes it more likely that assets from the same library are compatible. If there is no common context the inter-operating components may place different interpretations on data values leading to incorrect results. An example, this occurred recently when one component in the Mars Climate Orbiter project used english units and another component used metric units, resulting in the loss of the space craft. The loss of the first Ariane 5 rocket was due to a component that expected a different size for a numeric parameter than what was provided by another component. We will often want to use more than one asset from a library in the same product, such that the output from one asset will become input to another. An example where many components from the same library typically inter-operate can be seen with GUI libraries. Think of the difficulty if each widget used a different unit of size (pixel, point, pica, inch, centimeter, etc.) and a different coordinate system for positioning.
- Many characteristics of a good library [8] depend on the assets in a library exhibiting similar behavior and usage characteristics. This similarity can most easily be achieved by placing context once at the library level, rather than trying to insure that contexts for each asset have the same values. Examples of such characteristics are: consistency, ease-of-learning and ease-of-use.

6. CONTRAST WITH OTHER MODELS

6.1 IEEE Standard 1420.1

Despite the longevity of the software libraries the only previous attempt we were able to find to provide a model was made by the Reuse Library Interoperability Group (RIG),

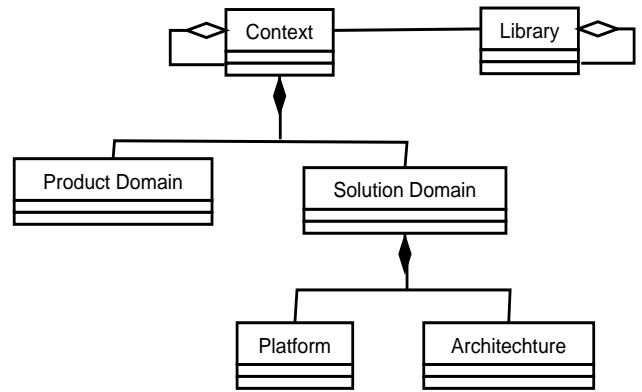


Figure 3: Context Detail.

an industry consortium [3]. Their work was published as IEEE standard 1420.1[7]. Their goal was to provide only an interface definition to exchange libraries, not a complete model. Much of this work is related to tracking software certification and specifying intellectual property rights, which was codified in standards 1420.1a and 1420.1b respectively. These issues are not of interest to us and will not be considered further.

The model provided includes only a class diagram; there is no use case diagram. This makes it difficult to be sure what they see as the overall role of the library. The existence of pre-existing libraries in the model implies the role of library developers. The use case they explicitly support is selecting an asset for reuse. There is nothing in the model to specifically support the composition of assets with client programs.

In the class model there is no equivalent to deployment container, which selects and supports assets for composition, this also fails to support a way to specify binding times. The library class lacks any attributes to assist in selection or limit searching without looking at all the assets contained. There is no way to specify a constraint on the asset to be included in a library. This is somewhat surprising as members of the RIG group stated that they believed libraries should be focused, specialized, collections, not general purpose[3]. Moving the domain attribute from the asset to the library class would help here. The role of architecture in reuse is completely ignored. There is no distinction between assets that are used in the product and supporting assets. In short, the role of context is ignored.

6.2 OMG Reusable Asset Specification (RAS)

OMG Reusable Asset Specification (RAS) [12] primarily models assets, however, it also models the relationships between assets, and even (briefly) discusses a repository for assets. In this model, the asset, which is often referred to as an *asset package* in the standard is composed of 5 parts: solution, profile, usage, classification, and related asset. The solution asset corresponds to our product asset. The other parts make it easier to work with the solution, which corresponds to our supporting assets.

While we provide a descriptive model, RAS is a proscriptive

approach, which imposes requirements related to OMG’s Model Driven Architecture (MDA). It is interesting that to provide automated assembly in MDA a large amount of context information is required. RAS puts its context information at the asset level. As has been noted, this means all use decisions must be made for each asset, instead of reusing information about the collection. In practice, this may be mitigated by a combination of the large granularity of the components intended for RAS and the planned development of detailed implementation profiles.

The RAS standard also defines RAS Repository Services. These define Java and HTTP methods to store and retrieve assets from a RAS Repository. However, no model is provided for the repository. The RAS glossary defines a repository as: “A centralized access and storage point for reusable assets.” It is not clear from this which use cases are envisioned. It does not seem to involve the actual composition of assets. The text mentions that these services are intended for small and medium repositories. However, it is not clear how the “centralized access and storage” is related to the multiple repositories. Also, there is no guidance on why an asset would be in a particular repository or how assets in a particular repository are related. The glossary also defines a reusable asset library as a “conceptual composite artifact that encompasses all possible reusable assets” which sounds much like the failed general purpose library paradigm. This concept of library is not otherwise referred to in the document and does not explain its relationship to reliable asset repositories.

7. ANALYZING SOME EXAMPLES

To validate our library model we analyzed a variety of libraries, to check if the model can describe them. Here we present as examples: Dia Shape Templates, the Java Swing Library, Eclipse plug-ins and Software Product Line asset bases. Swing will represent a typical class library. Eclipse and asset bases are not generally thought of as software libraries; however, they are collections that fit our library definition. Studying these examples can show how more extensive use can be made of libraries particularly in terms of improved asset composition. They illustrate the importance of context to library usage. They also provide an example of how a standardized model can help explain new material.

Both Eclipse plug-ins and Software Product Lines will show us something about the future directions of the software library domain. Domain choice drives the architecture and related design rules. Having a well defined context for product and solution domains supports the design of assets that are composable without modification.

7.1 Dia Shape Sets

The Dia drawing program allows users to define shape sets. The object diagram for Dia shapes is shown in Figure 4. Dia is a drawing program designed to draw different types of diagrams. It comes with shape sets for drawing such things as electronic circuit diagrams, UML diagrams, flowcharts, etc. The main abstractions the program works with are shape objects and connectors.

The basic asset type in Dia is a shape. Dia supports the hierarchical composition of shapes. Existing shapes can be

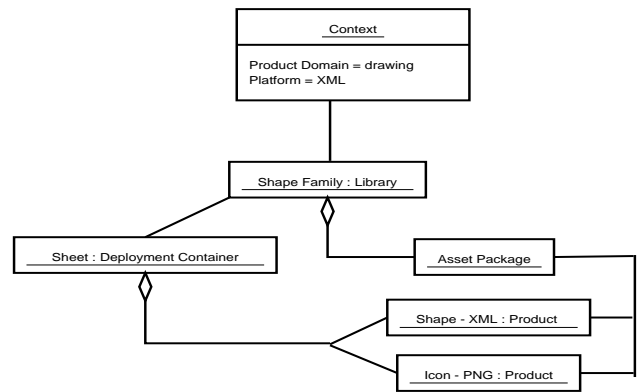


Figure 4: Dia Object Diagram.

used to draw a new compound shape, which can be saved as a shape. Shapes can be collected into sheets. This allows related shapes to be collected together. For example, AND, or, NAND, and XOR shapes are collected in a circuit sheet. Sheets allow a set of shapes to be pulled into the program as a group and made readily available to the user. Adding a shape set extends the programs capabilities.

A typical asset package, for a shape, has two product assets, a file which provides an icon to represent the shape on the palette menu, and a file which has an XML description of the shape which the Dia program can translate into a drawing. A sheet acts as a deployment container, grouping a collection of shapes together and making them available for composition. In this case the library user is working interactively with the library, so the search / selection mechanism is integrated into the client program. The user selects a shape by choosing a sheet from an alphabetized list of names. Choosing a sheet causes the icons for the sheet’s shapes to be displayed in a palette window. While the program comes with a large number of shapes, and allows new ones to be built, the ability to organize shapes into groups with the sheet mechanism keeps the number of shapes being worked with at a given time to a manageable level even with these simple search mechanisms.

Even though Dia shape sets are rather different from what is normally thought of as a software library, they meet both our understanding of a library, as well as the current usage of the term, that is a collection of assets. This object diagram shows that our model is able to accommodate them as well.

7.2 Java Swing Library

The Swing library is typical of Java class libraries. An object diagram for the Swing library is provided in Figure 5. It is similar in structure to many other class libraries, but provides a better documentation specification and supporting tools (such as Javadoc and Jar files) than most library systems.

A typical asset in a Java library is a class including the special comments that are used as input for Javadoc. Physically the class is defined in a single source file ending with a .java suffix. The class source code file is the input to the Java compiler to produce an class file and to the Javadoc tool

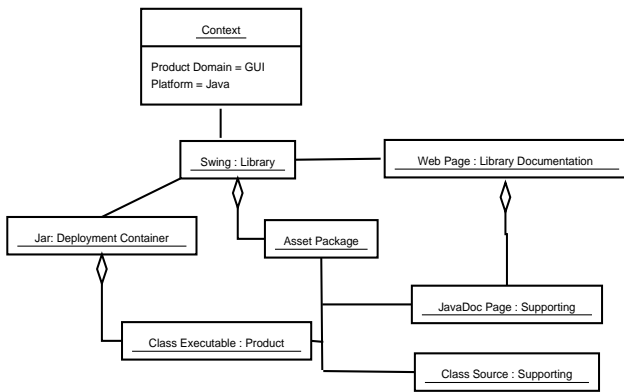


Figure 5: Swing Object Diagram.

which will produce a hyperlinked web page to document the class. The asset package for each class will bundle a Java source file, executable class file, and a web page.

The library provides documentation, a search mechanism, and a deployment container. Javadoc specifies a standardized format describing the library as whole. This page provides hypertext links to other library documentation, such as tutorials, and hypertext links to the generated web pages for each of the classes in the library. The links to the class web pages form the primary index for the library, based on an alphabetized list of class names. Since this documentation conforms to the standards for the web, any web-based search engine can provide an additional basis for search using text matching.

The Java language uses dynamic class loading rather than linking. While the executable class files can be used directly by a client program, Java also defines an archive file format, Jar, which serves as a deployment container. The Jar was designed to move a collection of class files around as a unit. Jar files have roles beyond deployment. A client program can load a class from a Jar file without unpacking it. If a program's main method is in the Jar file, it can be tagged to execute without unpacking the Jar file. Jar files can also include security information about a group of classes.

Much of Java's success is attributable to the support provided for libraries. Javadoc provides Java libraries with extensive, maintainable documentation that has a consistent look and feel and a consistent search mechanism. The Java virtual machine provides a hardware independent platform thus eliminating one major aspect of architectural mismatch. Many issues, such as memory management, normally left to applications are handled by the Java runtime environment, reducing the number of different context dependencies.

7.3 Eclipse plug-in

Eclipse is a modular, open-source product that provides an extensible Integrated Development Environment (IDE) [13]. Its goal is to provide a single user environment that supports the integration of development tools produced by different organizations. Eclipse allows the user to build a version of the product that fits their needs by installing appropriate

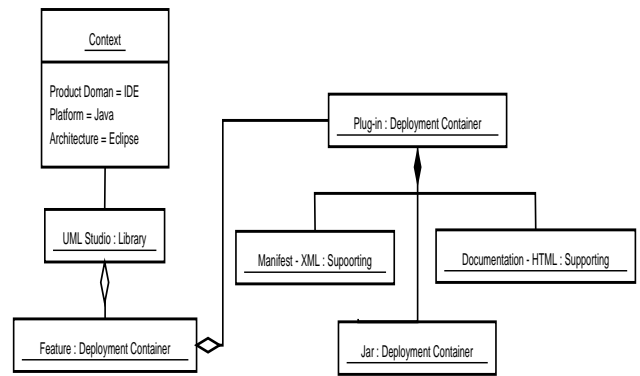


Figure 6: Eclipse Object Diagram.

modules. While Eclipse is very modular in many ways it is also designed to fit the needs of a specific domain (IDE), a specific platform (Java), and provides a specific architecture that modules must adhere to in order to be composed. An object diagram which presents Eclipse modules as a library is provided in Figure 6.

The typical Eclipse module provides a development tool or closely related tools. For example, to support a compiled language requires not only a compiler, but also a language specific editor, templates for the different file types in the language, debuggers, wizards, and a variety of documentation. Eclipse differs from most libraries by the emphasis it puts on supporting assets; and its support of open module composition after deployment.

Tools for Eclipse are deployed as features. A feature is a group of plug-ins that are deployed or upgraded together. For the user, the feature is the unit of both deployment and versioning. Physically a feature is composed into a compressed file, to allow it to be moved as a unit. Each plug-in in the feature has its own directory. Plug-ins provide or support different parts of a feature, as they extend different parts of the Eclipse platform. For a compiled language, we would expect the editor to be placed in one plug-in, while the compiler, which can be run without a user interface, would be in a different plug-in. Each plug-in must provide a specification, called the manifest, written in XML, that describes the plug-in to the Eclipse platform. Beyond the manifest, the assets provided for a plug-in vary. If a plug-in has an executable portion, it must consist of Java class files collected in a Jar file, located in the plug-in directory. This is an example of one deployment container, the plug-in, containing another, the Jar file. Documentation is stored in the plug-in directory as html files, and may be put into its own plug-in to make it easier to internationalize. Other assets found in plug-ins might include icons, images, web templates, etc.

The Eclipse model provides support for all three of our use cases. Adding assets is supported by a number of tools, such as the Plug-in Development Environment (PDE), which provides wizards to walk the asset developer through the process of adding assets. Searching and understanding assets is supported by search features and an expandable help sys-

tem. Plug-in assets can specify how they should be included in the table of contents for the help system. A number of ways to compose assets are provided. Plug-ins can modify the behavior of other plug-ins by extending them in an inheritance relationship or can use other plug-ins by specifying a dependency.

7.4 Software Product Line Asset Bases

“A software product line (SPL) is a set of software-intensive systems that share a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.” [4] SPL scopes which products will be included in the product line early in the analysis. This allows the development of a common architecture which has explicit variation points identifying where and how different product within the product line will vary from each other.

The core assets are collected in an asset base to be used in developing the products included in the software product line. The asset base is a heterogeneous collection, which includes assets for all development phases, from requirements to implementation. It is tightly scoped to include only assets that will be used in more than one product in the SPL. A SPL may use library mechanisms to group and prepare implementation assets for composition with products. Thus, an SPL’s core asset base is an example of a library containing libraries. While specific to a particular group of products, the size of an asset base may reach millions of lines of code.

The major differences between an SPL asset base and a traditional library can be seen in the context provided and the relationship between reusable asset and the products built with them. The SPL development process begins with a domain analysis which is further defined by a selection of features to be supported and a grouping of those features into products. So the product domain portion of the context is well defined for SPL. The product domain is further constrained by the selection of products that will be supported by the product line. In contrast, other than a tool domain, such as GUI development, we typically don’t know the bounds of the product domain for a reuse library. Within the product domain a traditional library is intended to be used in an open set of products.

In the solution domain, an SPL will typically define a single architecture to be supported. A typical reuse library will attempt to support multiple and undetermined architectures, often aiming for the difficult goal of being architecturally neutral. SPLs may support multiple platforms, but the platforms supported are made explicit. The influence of platform variants can be shown in the variation points of the SPL’s architecture. Reuse libraries typically support a single platform, but platform information is often implicit.

The relationship between the collected assets and the products produced also differs. SPL products are made primarily by assembling assets from the asset base. Ninety percent reuse levels are typical in product lines, with many reaching a hundred percent. All assets for a product line are collected in a single asset base. All of the assets in the asset base should be used in multiple products. Traditional reuse

libraries support a much lower frequency of reuse, typically not exceeding fifty percent. Achieving this involves finding, understanding, and using many different reuse libraries. A given product will use only a small percentage of a library. It is possible that many library assets will never be used in a product.

These differences are summed up by Clements “Software product lines represent a significant departure from software re-use schemes in which attempts are made to make assets as general as possible without the context provided by an architecture and a scope definition, and from opportunistic reuse schemes in which low-payoff assets are scavenged ad hoc from a reuse repository.” [4].

7.5 Summary

These brief examples show very different collections. The type and strength of relationships among the elements in the libraries are different. In the Eclipse example, the elements would be expected to be consistent with one another. On the other hand, a product line asset base may contain assets where choosing one asset excludes choosing another, an exclusive-or relationship. The Eclipse example has a very clear need for completeness - the plug-in needs to work - while the product line asset base may be quite incomplete. There are several directions in which this work needs to be extended:

1. The existing model should be applied to additional types of libraries. The examples in this paper, Dia shapes sets, Eclipse plugins, and software product line asset bases, in addition to the programming libraries normally considered, suggest the diversity of libraries that should be addressed.
2. The model is presented at a very abstract level to allow it to cover the maximum range of libraries. Specializations of the model should be developed for important categories of libraries, the most obvious being programming libraries. This more specific model could consider common programming issues such as error handling and memory management. A programming library model could be further specialized to handle common cases such as class libraries, active libraries, etc.

8. CONCLUSIONS

Software libraries are one of the oldest, most used approaches to software reuse. Despite their long past and interesting future, there has been almost no research on libraries as a product domain. Based on experiences with other product domains, a thorough domain analysis advances the state of the practice.

We have presented an analysis of libraries as a product domain, beginning with the definition that a library is: A collection of composable assets, that contribute to building a product, aggregated within a mechanism that holds the collection for the purpose of promoting reuse by providing greater accessibility, for use by others. While this covers many different types of asset collections it excludes many as well. A contrast can be seen in the World Wide Web.

The web is a collection of assets, including software, and it provides search mechanisms to assist in finding the desired asset. Yet, it is not a software library, because most of the assets are not intended to be composed into products and because the web does not provide a composition mechanism.

We have provided simple, but comprehensive, use cases. We have three actors (library developer, library user, and client program) and three essential use cases (add an asset, find an asset, and compose an asset). With these use cases we avoid focusing exclusively on either the search problem or the composition problem. As a result, we highlight the need to be able to compose the assets found into a product.

In our model by making the deployment container a separate entity and allowing multiple instances, we open the way for support of multi-binding time libraries. By making a clear provision for a multi-binding time library we provide the opportunity for better library support for product lines, where multiple binding times is a significant issue.

Finally, we have clarified the relationship between a library and its context. There is growing acceptance that reusable software requires an explicit context. This applies to libraries as well. The library context includes both product domain and the solution domain of both platform and architecture. Our model identifies the appropriate concept with which to associate context is the library, not the individual asset, as has been the case in other work. Placing context at the library level allows assets to be grouped by context, allows better reuse of developers understanding, and produces the situation where a library's assets are compatible with each other.

Libraries continue to be an important means of providing reusable software; however, they are still understood, designed and built in an ad-hoc manner. This paper by providing the top level requirements and model is intended to provide a starting point for a comprehensive approach to library development and use.

9. REFERENCES

- [1] C. Alexander. *Notes on the Synthesis of Form*. Harvard University Press, Cambridge, Massachusetts, 1964.
- [2] C. Baldwin and K. Clark. *Design Rules: The Power of Modularity*. MIT Press, Cambridge, Massachusetts, 2000.
- [3] S. V. Browne and J. W. Moore. Reuse library interoperability and the world wide web. In *Proceedings of the 19th International Conference Software Engineering*, pages 684–691. ACM Press(New York), May 17-23 1997.
- [4] P. Clements and L. Northrop. *Software Product Lines, Practices and Patterns*. Addison-Wesley, Boston, Massachusetts, 2001.
- [5] J. O. Coplien. *Multi-Paradigm Design for C++*. Addison Wesley Longman, Inc., Reading, Massachusetts, 1999.
- [6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley Longman, Inc., Reading, Massachusetts, 1995.
- [7] IEEE. Data model for reuse library interoperability: Basic interoperability data model (bidm). Standard 1420.1, IEEE, 1982. Standard for Information Technology - Software Reuse.
- [8] T. Korson and J. McGregor. Technical criteria for the specification and evaluation of object-oriented libraries. *Software Engineering Journal*, 7(2):85–94, 1992.
- [9] C. W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, June 1992.
- [10] J. McGregor. Context. *Journal of Object Technology*, 4(7):35–44, September-October 2005.
- [11] L. J. Miller. The iso reference model of open system interconnection: A first tutorial. In *Proceedings of the ACM '81 conference*, pages 283–288. ACM Press (New York), 1981.
- [12] OMG. Reusable asset specification. Standard RAS, Object Management Group, 2005. <http://www.omg.org/docs/ptc/04-06-06.pdf> accessed July 14, 2005.
- [13] OTI. Eclipse platform technical overview. White paper, Object Technology International, Inc., 2003. Paper <http://www.eclipse.org/whitepapers/eclipse-overview.pdf> accessed July 14, 2005.