4.1 The Systematic Reuse of Software Assets

The traditional approach to software development originated with the implicit notion that every product would be unique with fixed requirements. In reality, most capabilities recur across many products and must be made to conform to differing and changing requirements for those products. This has led to the establishment of libraries of commonly needed capabilities in the form of previously built parts that can be used beneficially in building new products. Using such parts avoids the need to build equivalent parts from scratch as well as possibly needing specialized competence to do so. Furthermore, multiple uses of a part can induce investment in its improvement, including exposure and correction of residual flaws, increasing the value of the part for all current and future uses.

(For simplicity, this section focuses mainly on the reuse of components, as described in section 2.7, with associated design, implementation, and verification elements. This discussion applies analogously in practice to reuse of any product model content. Not only components but any content of a product model may be retained for potential reuse in building similar products. Examples include requirements and design specifications, user training and documentation, and testing materials. Libraries of hardware parts are also used in a similar fashion for computer-aided design and manufacture.)

Previously built parts can be used as-is or with modifications to fit the differing or changing needs of each product. If likely multiple uses of a part are considered and anticipated as it is being built, it can be built such that future modifications will be less costly to make when needed. With or without modification of a part being reused, the developer must ensure that as an element of the product model it is complete in accordance with project guidance and consistent with other elements.

As-Is Reuse of Existing Parts

Modern software-based products convey many complex capabilities that require substantial time and competence to build correctly. Furthermore, developers having the appropriate competence to build such capabilities are seldom readily available to every product development effort that needs them. As an alternative, the effort required to build software can be reduced when development practices support the effective use of previously built parts to provide such capabilities.

For an existing part to be reusable, it must have been built under assumptions that are consistent with its intended new use, specifically conforming to relevant product model specifications. Alternatively, aspects of the product model may need to be modified to allow as-is reuse of a given part. Such changes may be justified if the alterations do not diminish important capabilities or quality aspects of the product. (This reverses the traditional presumption that specifications determine realizations; in this case, the realization becomes a constraint on its specification instead.) If relevant specifications cannot be changed, the existing part would have to be modified to be reused.

There may be multiple existing parts that are reasonable candidates for a given reuse need. Selecting among multiple parts may be influenced both by perceived closeness of fit to the need and by the degree to which each candidate is fully realized (e.g., as a software component consisting of not just code but proper documentation including its associated design and verification specifications as well). A part that is only partially realized will require additional effort to have it conform either to particular needs or to project process guidance for the corresponding product model element.

Using a Library of Previously Built Parts

{reference 3.0 standardization-customization tradeoff: design around available parts or choose parts that fit design?}

Effective reuse of existing parts may be opportunistic or planned. For opportunistic reuse, a developer tasked with developing an element as defined by the product model may identify a previously built part that will suffice. Such parts may be held in either a program-maintained library of parts used in building other of its products or a sanctioned but separately maintained library of parts developed elsewhere for such use.

An opportunistic approach to reuse has five steps:

1. Define selection criteria for a part that would fit a need specified in the product model

2. Based on the organization or search capabilities of a selected reuse library, identify candidate parts that satisfy defined criteria
3. Evaluate the fit of each identified candidate to the specified need to determine which candidate is best, with or without changes, to align the part with the need
4. Select a suitable candidate part (or modify to create a new part) and integrate the part into the product model
5. For a new/modified part, optionally add it to the library with assumptions and guidance for it be reliably reused and/or modified for future uses

Having a general idea of what sort of part is needed, a developer must select among available parts to find the best fit to current needs. A best fit is a part that provides all needed capabilities without introducing any extraneous content. A library may offer multiple similar parts from different sources and any given part may exist in multiple versions, particularly if some versions were a result of modifications to other versions so as to improve the fit to particular needs. Properly choosing among a number of more or less similar parts or versions can be difficult and time consuming unless the specific differences among them and rationale for each are clearly documented. In any case, any differing assumptions must be resolved to achieve consistency of a reused part as an element in the product model.

For planned reuse, the product model is defined with the expectation that known parts will be a satisfactory fit for reuse, either as-is or with specific changes. Planned reuse often will entail reuse of other portions of a product model, in addition to the product components model, developed in building similar past products.

Modifying a Previously Built Part

If an existing part is a good fit for a new use, reuse should be a given. However, fit is not always simple to determine. Even if a part is free of defects based on past similar uses, a new use may diverge from the (possibly implicit) assumptions for which the part was built. Properly modifying an existing part (whether for reuse or just to meet changed needs) requires being aware of and adjusting for assumptions that the part's developer may have made about how the part ought to work. Otherwise, ill-considered changes may introduce flaws in the behavior of the modified part when used differently than previously assumed.

Even if a part cannot be reused as-is, initial effort may still be reduced by starting with an existing part that is a reasonably close fit and modifying it to provide a satisfactory fit to it new use (given that its source text is available or it can be reasonably encapsulated—or specialized in an object-oriented language— to extend or limit its native behavior).

Potential Issues in Reusing Existing Parts

Without due care, reuse of previously built parts, as-is or with change, has risks:

- Generally, an existing part was initially built to satisfy specific needs, usually of a given product or set of related products. A part built to satisfy any specific set of needs may not be a proper fit to a different set of needs and may not be easily changed to properly fit those needs. Reuse may require adding or modifying content or removing unwanted content. More superficially, an existing part may follow conventions that differ from those that are specified for a product. The effort to understand and change the assumptions on which a part is based can reduce benefits of reuse over developing a new part from scratch.
- If a part is modified to meet different needs, the modified part becomes a new candidate for future reuse. Even if the new part simply extends the existing part, the added complexity may make it a poor fit for use by other developers. Repeated derivation of modified parts can result in a proliferation of similar parts, that may not be easily distinguished, increasing the effort needed to select among them.
- Multiple parts may be available that provide an equivalent or similar fit to indicated needs. However, these parts may differ in breath or depth of content/capabilities, quality factors, assumptions, rationale, and tradeoffs that must be considered in selecting a part for reuse; these sources of difference may not be well documented. For example, a part built to operate on a particular computational platform may work without changes on a different platform, but exhibit differing behavior

requiring changes to exhibit the same behavior. A given part may be easier to modify for different needs than another similar part due to such differences.

- Responsibility for sustainment of reused parts can be problematic. If undiscovered defects in an original part are carried over to successor parts, all will need to be redundantly fixed. If a part being reused is later modified or a reuser's needs change, the reused part may need to be replaced by a modified part that better fits the reuser's current needs. The cost of maintaining derived parts may not benefit from changes in the original part (e.g., to correct defects or improve behavior). Similarly, successor parts may have defects or other substantive differences with the original part.
- A part selected for reuse may require use of other particular parts. For example, if a part is an element of a "framework", a set of parts built to be used together, dependencies among them may make it infeasible to use any of them without the others. However, using the entire related set of parts could mean that some of those parts have duplicate, or even conflicting, content with other elements of the product model that would need to be justified.

Building Parts for Multiple and Changing Use

The conventional approach to software reuse assumes that parts built for one purpose will either be useful for other purposes without change or can be changed for a different purpose with less effort than building similar parts from scratch. There is often good reason to make changes to obtain a part that better fits each different purpose, unless the original part was built with potential different uses explicitly considered. In reality, the only sound basis for reuse is an envisioned set of similar parts, each instance of the set being customized to fit specific needs for such a part.

Software reuse as traditionally practiced—opportunistically after a part has been built for a given purpose—is not well suited to building products that satisfy different needs. Whether a part built for one purpose is suitable for a different purpose is not easily determined after the part has been built without insight into the original developer's assumptions and tradeoffs made to suit the original purpose and the degree to which these conflict with other potential uses.

A part built to suit the needs of a single product, even considering potential changes that product may require, may not be easily changed to meet the needs of a different product. Building a part to meet the potential needs of other, possibly unknown products requires a broader awareness of what capabilities the part may need to provide across similar products and over time. With such broader scope, there are techniques for identifying and accommodating the factors that motivate how a part is built to be systematically changed to account for this concern. This can result in an ability to derive multiple customized versions of a part with only limited additional effort.

Properly building a part for reuse means building it in a way that anticipates likely changes for other uses, as well as for needs changing over time. This requires the developer to be aware of assumptions about part content (e.g., component behavior) and how other potential uses would warrant different assumptions, leading to different realizations. At the least, the developer provides guidance to future reusers (e.g., annotations within or attached to the part) as to what assumptions characterize the part as provided and how it can be changed if assumptions are changed in particular ways to fit different needs.

The implication of this perspective is that a part built for reuse is not one-of-a-kind but is only one example of a set of similar parts (e.g., a family of software components). If an existing part is modified, the modified part becomes another candidate for future reuse. Retaining each such version can lead to a proliferation of parts that are similar but differ in ways and for reasons that need to be known to potential reusers. Proliferation of similar parts increases the difficulty for developers to decide which of those alternatives is best for them to reuse. This argues for a more systematic approach to managing the reuse of parts meant to serve multiple more or less similar purposes.

Techniques for Building a Component Family

A reusable component is a component that is built to be adapted as needed to different uses. Four techniques are considered for building reusable components that can be adapted.

The simplest technique for addressing potential changes is to document the assumptions on which a component's design and implementation have been based. Each assumption is elaborated in terms of how the code depends on that assumption and what changes to the code would be required if that assumption was changed. Associated alternatives considered and tradeoffs may further inform future developers. Based on this information, future developers are able to determine whether and how to change the code for the component to be suitable for a different use. Such changes may result in the existing component having a more general capability to continue to be used for its prior purpose or it may result in having to create a new version of the component if these changes make it unsuitable for its prior use.

Another technique is the use of programming language mechanisms that allow source code to be tailored as a step in its being translated into object code. Such mechanisms include macros, templates, and generics, or configuration to selectively include optional or alternative modules (or runtime libraries of related modules) that fit particular needs or computational platform. In object-oriented languages that provide for defining classes and subclasses (i.e., specialization), a choice among a set of alternative subclasses can result in differences in a product's behavior.

{build adaptive logic into component for runtime treatment of change}

A more flexible technique is the concept of "metaprogramming", the development of software that generates other software. In fact, this is a general technique for building not just software but customized instances of any type of structured artifact (e.g., system, software, or hardware specifications, documentation, test materials). This technique can be applied to build customized instances of any element of a product, including code, specifications, documentation, test scenarios, etc. In effect, this is a technique for uniformly representing alternative versions of any or all elements of a product (or instances of a product family), and providing the means to derive any version of an element or whole product.

The Nature of Descriptive Metaprogramming

Metaprogramming within DsE has taken a “descriptive” (as opposed to “prescriptive”) formulation. The particular form used defines reusable elements in an “adaptable” notation that is able to represent a set of similar elements in aggregate. This form defines a reusable element as a frame of fixed content within which changeable content is embedded, distinguishing between fixed and changeable portions of a reusable element. Software operates on such a form to replace the changeable portions with fixed content. Each changeable portion has associated criteria for its resolution into fixed content. Software customizes a reusable element to a specific use by applying its criteria to resolve each changeable portion into a fixed form.

This form is a generalization of more conventional code tailoring mechanisms. In this form, the structure of all resulting instances are explicit in the notation. The advantage of this form is that the form, structure, and content of an element is explicit. It is also a simple extrapolation from how any element is created manually, with a technique very analogous to how code is developed conventionally. Its limitation is that it cannot transform the essential structure of an element as defined by the developer, but relies on subsequent processing (e.g., a code compiler or document editor) to improve behavioral quality factors.

The descriptive form of metaprogramming has the advantage of providing a predictable result for any type of element. It can be used compatibly with other techniques (e.g., compiler optimization or transformational tools) to convert derived software code into more efficient forms.

<——————— {discuss prescriptive form & optimization transformations in 5.4?}

The prescriptive form of metaprogramming is both more flexible and more opaque than the descriptive form, in that no portion of an element is inherently fixed. Software is written to operate on an intensional specification of an envisioned element; this specification is manipulated (derive, transform, and combine) to synthesize element content equivalent to the specification; content is represented as instances of the various semantic constructs of the element being generated. The advantage of this form is that it can effect more complex transformations of the element to iteratively improve the product's behavioral quality.

————-———>

Verifying a Component Family

<——-—— delete or move most of the following (notation, usage, examples) to an appendix?

Adaptable software is software that has been built to enable being mechanically customized to potential different uses. The source form of any software can be modified to change its behavior but software built to enable specific adaptability without having to directly modify its source form enables such changes to be made quickly and safely. The practice of creating a new instance as a copy that is then modified results in duplication of effort when changes in both are later needed; uncoordinated changes to the instances over time lead to the dilution of similarities and divergence lacking effort that would maintain common aspects to the greatest degree possible.

The following describes a particular notation for defining adaptable components as an example of the descriptive form of metaprogramming. (notation is an adaptation of the conventional prescriptive form of expressing behavior that is mechanically translated into a native hardware notation)

Building and Using Adaptable Components

A refinement of the idea of building individual components for multiple uses is to build adaptable components from which individual customized components can be mechanically derived. An adaptable component is a representation of a family of similar components, a family being a means of expressing a library of both previously built components and other similar components that may be needed but have not yet been built (Figure 4.1-1). A component family is characterized by an *abstraction*, its unifying concept corresponding to the ways in which instances are alike, and by the specific differences that distinguish each derivable component from other instances of the family. Representing a family with an adaptable component provides the means to standardize the realization of its instances, eliminating incidental differences while enabling accommodation of essential differences.

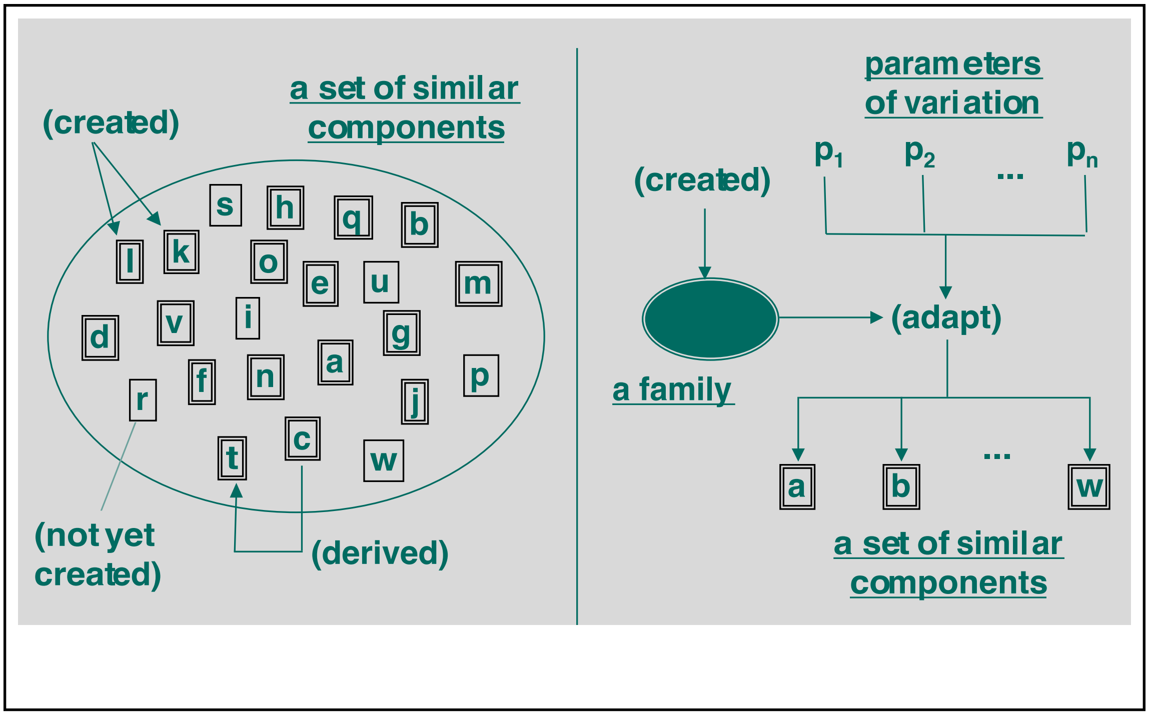

Figure 4.1-1. Two Views of a Component Family

A Notation for Adaptable Components

An example of a formal notation that can be used to represent the common and differing aspects of a component family, along with an associated mechanism for deriving customized instances, is described below. An advantage of this notation is that it can be used not only for describing code but also any other information associated with a component that can be expressed in text. This enables the representation of all the elements (i.e., design, implementation, and verification) of a complete component model. A later section describes how this text-based notation can also be the basis for representing adaptable content expressed in a graphical form.

The described “metatext” notation embeds relevant normal text within a metaconstruct that specifies how instances of that text will appear in a derived normal text form. This is similar to the (“macro”, “template”, or “generic”) capability associated with some programming languages, in that its associated mechanism works as a preprocessor of metatext to produce customized normal text. It can be used both in combination with or as a partial alternative to the object-oriented concept of specialization (subsets of a family being analogous to subclasses of a class).

An adaptable component is defined in an "mtp" (metatext processor) file. An mtp file contains normal component *target* text in which metatext constructs are embedded (Figure 4.1-2). These constructs define the ways in which instances of the family differ in their content.

- Target text (common parts of components)
- MetaPrograms (variant parts of components)
  - Name, to identify the abstraction
  - Parameters, with which reusers control tailoring
  - Definition (target text containing metaConstructs), to show how to extract tailored instances

```
« program F ( p1: text, p2: (p3?:text*, p4?:text) ) «
        some common text «p1»; «
        p2.p3
            ? «with custom text: «for p in p2.p3 with «, » «p»»»
            : p2.p4 ? «plus alternate text: «p2.p4»»
            »
»»
```

Figure 4.1-2a. Metatext Constructs

- Substitution: « p »
- Selective substitution:

  « p.q ? «with q» : «without q» »

- Repetitive substitution:

  « for i in p.pi «same but different due to «i».»»

- Metaprogram instantiation:

  « F (p1: «in all work products», p2: (p3:(«abc», «def», «ghi»)))»

      => "some common text in all work products; with custom text: abc, def, ghi"

Figure 4.1-2b. Metaprogram Operators

The primary metatext constructs, besides *target* text, are *metaprograms*, *metatypes*, *file-inclusions*, and *instantiations*. These constructs, in turn, contain fragments of target text that become elements of instantiated complete target text. Each metaprogram and metatype has an associated name that serves as a referenceable identifier of its definition. The content of an mtp files can be explicitly embedded in another mtp file using the *file-inclusion* construct. An optional "name space" identifier provided in a file-inclusion is used to qualify references to metaprogram constructs defined in the included file, avoiding naming conflicts with constructs defined in the including file.

A metaprogram has an associated name, metaparameters, and definition. Each metaparameter has an associated name and metatype. Each parameter can be designated as required or optional; the metatype of an optional parameter is optional (the parameter being only present or absent in a given instantiation).

There are three forms of metatype: target, tuple, and sequence. A metatype can also reference another equivalent metatype by name. A target metatype value is a literal string of target text. A tuple metatype is defined by a parenthesized set of named metatypes, each element having a value of the specified type. (A tuple whose members are all optional is the equivalent of an enumerated type.) A sequence metatype is defined by a single metatype, its value being a sequence of values of that type.

A metaprogram definition consists of instances of four metaprogram operators, all embedded within a base of target text: substitution, selective substitution, repetitive substitution, and metaprogram instantiation. This base target text defines the content that all instances of the metaprogram share whereas the operators determine the content that differs. Each operator is essentially "functional", simply replacing itself with its equivalent target text value within the text in which it is referenced. A substitution inserts its equivalent target text value. A selective substitution inserts one of alternative target text values. A repetitive substitution inserts each of a sequence of (optionally separated) target text values. A metaprogram instantiation inserts the equivalent target text value of the referenced metaprogram's definition based on specified metaparameter values. As an example of instantiation, instantiating the metaprogram in Figure 4.1-2a with «F (p1:«in all work products», p2:«(p3:(«abc», «def», «ghi»)))» would produce “some common text in all work products; with custom text: abc, def, ghi”.

Developing an Adaptable Component

Having identified the need for a family of similar components, representing the family as an adaptable component is similar to building a conventional instance component. Although these steps and the following example are written in terms of building a code component, the same approach works for building any type of text-based component such as developer or user document content or testing materials. (The next sections generalize beyond text-based component content.)

1. Choose or write a prototypical instance component as a base target text construct.
2. Identify a set of decisions that are sufficient to distinguish among the instances of the family.
3. Embed base target text within a top-level metaProgram, parameterized with decisions to represent how text is to be customized.
4. Build subordinate metaPrograms to organize the top-level metaProgram into a coherent structure.
5. Derive the prototypical and 2-3 other instance components using the metaProgram to verify that the adaptable component correctly expresses their content.
6. Refine and extend the metaProgram and its supported decisions to express the currently needed set of similar components.
7. Extend the metaProgram further to support other potential future needs for a complete component family as envisioned.
8. Generate prior and new instances to verify, revise, and extend the metaProgram, as needed.

A Progression of Simple Metaprogram Examples

The first example given (Figure 4.1-3a) is an example of a metaprogram that is a single fixed version with no accommodation for changes. It is a simple program for a simple fixed size stack with which integers can be added and retrieved in normal last-in, first-out order. The second example given (Figure 4.1-3b) is an extension of this example which can be customized to different programs, each supporting a stack of any size and any single type of value. The first example can be instantiated using the second example as "«stack (name:«integer», datatype:«int», maxsize:«1024»)»". Alternatively, "«stack (name:«string», datatype:«charstr»)" would derive a program for a (nominally) unlimited size stack of alphanumeric strings. This example can be easily generalized to for deriving a program that supports a stack for entries of multiple types, for data structures besides stacks (e.g., queues, deques, ordered lists) (Figure 4.1-3c), or for alternative implementations of a given structure.

Fixed-size, fixed-type stack

```
public class integerStack {
        static final int maxSize = 1024;
        int data [] = new int [maxSize];
        int size = 0;
public void add (int p1) throws stackFull {
        if (size == maxSize) throw new stackFull ();
        data [size++] = p1;
        }
public int get () throws stackEmpty {
        if (size == 0) throw new stackEmpty ();
        return data [--size];
        }
}
```

Figure 4.1-3a. Adaptable Component Example (1)

Figure 4.1-3a Adaptable Component Example (1)

Variant-size, variant-type stack

```
« program stack (name:text, datatype:text, maxsize?:text) «
public class «name»Stack {
        «maxsize?«
                «datatype» data [] = new «datatype» [«maxsize»]; int size = 0»
                : «Vector data = new Vector () »»;

        public void add («datatype» p1)«maxsize?« throws stackFull» {
                «maxsize?«if (size == «maxsize») throw new stackFull ();»
                data«maxsize?«[size++] = p1»:«.put (p1)»»;
                }

        public «datatype» get () throws stackEmpty {
                if («maxsize?«size»:«data.size()»» == 0) throw new stackEmpty ();
                return data«maxsize?« [--size]»:«.get ()»»;
                }
}
»»
```

Figure 4.1-3b. Adaptable Component Example (2)

Variant-size, variant-type, variant-access sequence (stacks, queues, deques)

```
« program lifoProcs (name:text, datatype:text, maxsize:text) «
    public «datatype» getFirst () throws «name»Empty {
        if («maxsize=«»?«size»:«data.size()»» == 0) throw new «name»Empty ();
        return data«maxsize?« [--size]»:«.get ()»»;
        }
»»
. . .
« program sequence (name:text, datatype:text, maxsize?:text, access:(fifo?,lifo?)) «
public class «name» {
    . . .
    public void add («datatype» value) { . . . }
«access.lifo ? ««lifoProcs (name:««name»», datatype:««datatype»»,
    maxsize:««maxsize?««maxsize»»:«»»» »»»
«access.fifo ? ««fifoProcs (name:««name»», datatype:««datatype»»,
    maxsize:««maxsize?««maxsize»»:«»»» »»»
    . . .
}
»»
```

Figure 4.1-3c. Adaptable Component Partial Example (3)

<------- omit/redundant ? ------

One approach for streamlining product development is to identify software capabilities that the enterprise will need in multiple future products and build software with those capabilities in the form of components that developers without that specialized competence can use to build individual products.

Assuming access to a collection of existing components, there are a series of challenges for cost-effective reuse:

8.1. Formulating criteria that characterizes a needed component

8.2. Finding existing components, as candidates for reuse, that are a sufficient fit to the criteria for the needed component

8.3. Determining which candidate component is the best fit to the needed component

8.4. Resolving differences, if any, between the selected component and the needed component, either by modifying the criteria for what is needed or by modifying the candidate component so it is a closer fit to the criteria

Reuse works best if previously built components can be used as-is, without change. Having to change prebuilt components undermines the intended benefits of reuse. It also leads to proliferation of similar components that then need to be separately maintained, diverging over time and further reducing expected savings. In addition, the replication of similar components, without clear definition of how they differ, will complicate the choice of future developers as to which version is best to use for their product, preferably without further change.

The anticipated savings with reuse over creating needed components from scratch will not be realized if a needed component is not closely matched by any existing component or if it takes too much effort to find a good fit. Every time a component must be modified to be reused, it increases the population of components having similar capabilities in the library, increasing future effort to find a fit. Building a new component may be less effort and better quality than trying to modify an existing component that was built to satisfy differing needs. In creating a new component, the ways in which it differs from similar existing components needs to be clearly defined as an aid to future reuse.

A well-organized library will associate components according to the degree that they offer similar capabilities. This helps reduce inadvertent duplication of equivalent components and the effort needed to find a good fit to current needs.

Ideally, the organization of an enterprise reuse library will parallel the architectural structure of existing and envisioned client products. For external libraries, additional effort is required to correlate between the two.

<figure: steps involved in reusing prebuilt components>

If a previously built component is in some way not suitable for use in a product, it would still be better not to discard the component and build a replacement from scratch. However, changing the component to be suitable may still require specialized competence to understand how and why the component was built as it was and how it needs to be changed to be suitable. Presumably, it was built under assumptions that are in some way in conflict with this product and changes need to be made with clear definition of the assumptions that will characterize the new version. Additional effort beyond modifying code will entail modifying developer documentation and verification materials associated with the component.

One way around this problem is to build the product with the presumption that specific existing components will be used. This may entail constraints on the product's behavior and appearance, in turn resulting in the product's customer having to modify their expectations and business practices to suit the product rather than building the product to fit their exact needs. However, by being able to deliver the product faster or at lower cost, the customer may consider that to be an acceptable tradeoff.

——————————>

<——-— (below is redundant to parts of above) ——-

A "Naive" Approach to Software Reuse

Although being able to reuse previously developed components can save time and effort in building a new product, it is not generally cost-effective to try to reuse software that was not built with an expectation that it would be used to solve multiple problems.

For someone building software in a given subject matter area, it makes no sense to always start over on building a new program if they have previously built solutions to similar problems. Even with significant differences, there may be parts of the previous solution that will address parts of the current problem.

While different developers might solve the same problem differently, each developer would tend to produce similar solutions to different problems. The easiest way to do this is to just copy the previous solution and modify it as needed to create the new solution. From the perspective of the new software effort, this may make sense, certainly more than just discarding the prior solution, but it creates or fails to correct several difficulties:

- Software built to solve a specific problem will be narrowly tailored to that problem. It will embody certain assumptions that a different problem may not share. If the initial solution is applied to a new problem, it has to be changed if any of the original assumptions are violated.
- The original developer may understand the differences between the two problems and the details of the first solution well enough to modify it to be a proper solution to the second problem. Another developer may not adequately understand the prior problem and its solution or may just prefer a different solution.
- More broadly, there may have been multiple past solutions to such problems. A developer may have to determine which of those solutions is the best fit to solve a new problem. At a component level, there may exist libraries of solutions that are similar but vary in the degree to which they are a good fit to solving the new problem. It can be difficult and time-consuming to make such a determination without an existing deep knowledge of the various prior solutions and how the problems they solved differed from the new problem. Even if two prior problems had similar solutions, it may be difficult to determine if they were created by different developers using differing terminology and organization.
- Having chosen a particular prior solution as a starting point, the developer must determine how to change that solution to properly fit the new problem. This has the difficulty that tacit assumptions made in building the prior solution may in not obvious ways be in conflict with the new problem. Alternatively, the developer might argue that changes in how the problem is viewed might be better, allowing the solution to be used with less change needed. The difficulty with this is that the customer may not be receptive to changing how the problem is defined if that will conflict with how they envision how the solution will fit their preferred way of working.
- Having built a solution to a new problem, the developer may retain that solution in case there is a future opportunity to reuse it in solving a future problem. The difficulty this leads to is that there is now yet one more in the set of problem-solutions for developers to consider and evaluate. Furthermore, if a defect is

discovered in any one of those prior solutions, it becomes a challenge to identify and fix that defect in all the prior solutions that it affects. Every new solution leads to a proliferation of similar problem-solutions that exist, increasing the future effort required to maintain as well as evaluate among them for reuse.

When software is built to solve a specific problem, it is narrowly tailored to that problem. If a similar problem is later presented, it is natural to start with a copy of the prior solution and modify it to fit the new problem. Except in the simplest of cases, this is actually a very problematic approach to take, both in terms of the effort required to modify a copy correctly and in terms of the cost this imposes over the lifetime of these implicitly related software items.

(library scenario: choosing best-fit item (problem alignment/ assumptions uncertainty) to copy, how to change properly with differences in assumptions, proliferation of similar items)

(sustainment problem: duplication of change effort, unnecessary divergence that reduces common aspects, every changed item is a new clonable item)

(non-cloned single-owner/multiple-client challenge: make changes while keeping acceptable for different uses, design software to accept how software works without changes, change software to accommodate forced changes needed for other uses)

When Naive Reuse Does Work

Conventional Options for Adaptable Reuse

Limitations of Programming Language Constructs or Classes

(incomplete/fragmented solution, no whole-product applicability)

(PL: don’t apply to non-code elements: docum, verif materials; build all into code?)

(class/subclass for specialization: granularity at class level -> excessive redundant content to be managed)

Building and Using Adaptable Components

(represents potential for unified maint of family of similar components)

A more flexible alternative is to build components that are designed to be customized according to specific criteria. Using such components, a customer's needs can determine particulars of a product's behavior and appearance rather than having the product predetermine and impose these on the customer enterprise. However, such customization must be limited to reflect what customer's are likely to actually need; otherwise, costs can become excessive as components need to be changed in arbitrary ways. The cost of customization is bounded by building components that are easy to customize in limited expected ways and costly to change in unexpected ways. However, the limits on customization options are easily modified over time as circumstances change.

<figure: an example hierarchical taxonomy of abstractions (adaptable components)>

Assumptions of commonality and variability for a component family

Annotating a Component to indicate "safe" or assumption-breaking changes

A (Canonical -> textual & graphical) Notation for Building and Using Adaptable Components

<figure: constructs for a metaprogramming notation for a component family>

Examples of Adaptable Components

sw code, documents, test procs/data

The Economics of Building and Using Adaptable Components

(payoff requires reasonable projection concerning diversity in component future needs)

tradeoff is building in adaptability vs (1) cost of repeated search, selection, and manual modification by developer reusers and (2) cost of expanded library storage of instances

——————————>