4.1 The Systematic Reuse of Software Components

The traditional approach to software development originated with the implicit notion that every product would be unique with fixed requirements. In reality, we know that many capabilities recur across products. This has led to the establishment of libraries of commonly needed capabilities in the form of previously built software parts that can be used beneficially in building new products. Existing parts can be used as-is or with modifications to serve as components that best fit the differing criteria of each product. (The potential for such modifications are best considered as parts are first built so that such modifications will be least costly to make later.)

Further arguing for the use of previously built parts, modern software-based products convey many complex capabilities that require substantial time and competence to build correctly. 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.

As-Is Component Reuse

The benefit of using an existing component may be justified in any case but is enhanced if the component is complete, consisting of not just code but proper documentation including its associated design and verification specifications as well. Otherwise additional effort is required to complete the encompassing component model in accordance with the project's development process.

Using an existing component built for another product has a potential downside: the requirements or design of the product being built may have to be modified to accommodate use of components that were built under different assumptions. Such use may be justified if the alterations do not diminish important capabilities or quality aspects of the product. However, this does reverse the traditional view that the requirements and design determine the implementation, whereas in this case the implementation becomes a constraint on them.

Another issue is when a component selected for use is only one element of a "framework", a set of components built to be used together. Dependencies among the components may make it infeasible to use any of them without the others. However, using the entire framework may mean that some of these components are redundant to, or possibly even have conflicting behavior with, other components providing similar capabilities in other parts of the product. Similarly, a component may be built to operate on a particular computational platform and require changes to work on a different platform.

Using a Library of Previously Built Components

A developer is nominally tasked with developing a component as defined by the product design but a previously built component having needed capabilities may suffice. A developer may obtain previously built components either from a program-maintained library of components used in building other of its products or from a sanctioned but separately maintained library of components developed elsewhere for such use. If a previously built component is to be used, then the product design may need to be modified to properly accommodate use of that component.

Having a general idea of what sort of component is needed, a developer must select among the set of available components to find the best fit to current needs. A best fit is a component that provides all essential behavior but no unwanted behavior. There may be multiple similar components from different sources and any given component 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. Choosing among a number of more or less similar components or versions of a component can be difficult and time consuming to do properly unless the specific differences among them and rationale for each are clearly documented.

Modifying a Previously Built Component

In the case that no existing component is an acceptable fit to the needs of a given product, it may still be of benefit to start development of the needed component using an existing component that is a good approximate fit (assuming that its source code is included or that it can be reasonably encapsulated to extend or limit its apparent

behavior). The consequence of modifying a component's source code is that the project may not be able to fully benefit from any improvements that other developers may later make to the component. The project's modified version of the component may accommodate such improvements or it may require additional work if at all.

In any case, at least the initial development effort may be significantly reduced by starting from an existing component. An important further caution is the need to be attentive to and not inadvertently violate assumptions that the existing component's previous developers may have made about how the component would work, in order to avoid making changes that could potentially introduce flaws in the behavior of either the modified component or of dependent components.

Whether developing a component from scratch or using a previously built component as-is or with modifications, the developer is responsible for completing all elements of the component model in accordance with project guidance.

The Problem with Naive Reuse

Finding parts for potential reuse, deciding which is best fit, and changing if necessary, adding yet another similar instance to the library (future searches get harder) ——

The traditional approach to software reuse relies on three simplifying assumptions: (1) when other products are known to have similar capabilities to those needed, developers will informally identify and use the relevant parts of those existing products rather than rewriting them from scratch; (2) if a part is an imperfect fit to the new product, the developer can just modify the part as needed and make it available for future reuse; and (3) when a product needs to be revised to meet changed customer needs, developers can easily identify how to change the parts of the product that account for any differences.

If an existing part is a good fit for a new use, reuse should be a given. However, fit is not simple to determine: even if a part is free of defects based on past uses, it may have been built with (unstated) assumptions that do not hold for the new use, it may lack some behavior that is needed, it may include some behavior that is unneeded or even unwanted, it may depend on other parts that are redundant or conflicting with other

parts otherwise being used, or it may follow conventions that differ from those that are preferred or required.

The developer may be reasonably inclined to chose a part for reuse that is a close fit to their needs and then modify it to be a proper fit. This has issues as well: repeated derivation of similar parts can result in a proliferation of hard-to-distinguish parts, making it difficult to identify a suitable fit for later reuse; a modified part may have defects or unforeseen behavioral differences with the original part; undiscovered defects in the original part and carried over to the new part will need to be fixed at least twice.

Building Components for Multiple and Changing Use

Useful products will be continuously revised not only because they may be flawed for their intended use but also because the needs that they are meant to serve change over time. This leads to previously built parts needing to be modified whether as existing products are modified or as new uses for those parts arise. This can lead to a proliferation in libraries of parts that are similar but differ in varying in aspects and degrees that need to be defined. Proliferation of similar parts increases the difficulty for developers in deciding which of those alternatives to reuse. This argues for a more systematic approach to managing the reuse of parts meant to serve more or less similar purposes.

A component built to support the specific needs of a single product is typically tailored to those specific needs. This raises two related concerns. The first concern is that no product can be assumed to meet only statically defined needs; over its useful life, a product will have to be repeatedly changed to meet changes in both needs and enabling technology. In building a product, consideration must be given to whether and how each component may need to be modified, whether during its initial development or during its useful life. Otherwise, as changes are repeatedly required, it can become increasingly difficult to maintain the conceptual integrity of the component. Each component should be built in a way that anticipates the more likely ways in which the component may be changed so that it can be easily modified in those ways.

The second concern is that a component 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 components to meet the potential needs of other, possibly unknown products requires a broader awareness of what capabilities each component may need to provide across similar products and over time. With such broader scope, there are techniques for identifying and accommodating the change factors that motivate how a component can be built to be systematically changed to account for this concern. This can result in an ability to build multiple customized versions of a component with only limited additional effort.

Three techniques are useful in accommodating potential changes to a component as part of its development. All of these rely on the principle of separation of concerns in which every component of a product has clearly delineated responsibilities with well-defined interfaces between it and other components (e.g., following information hiding per Parnas[1]); this allows determining whether a change can be isolated to the component or if it will affect related components as well. A change that does not require changing a component's interface can be made unilaterally whereas changes to an interface have to be coordinated across multiple components.

The first technique for addressing potential changes is to document what assumptions a component's design and implementation have been based. Each assumption is elaborated in terms of what parts of the code depend on that assumption and what changes to the code would be required if that assumption was changed. 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.

---

[1] D.L.Parnas, "Designing software for ease of extension and contraction", ACM ICSE '78: Proceedings of the 3rd international conference on Software engineering, May 1978, 264–277. <https://dl.acm.org/doi/abs/10.5555/800099.803218>

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 set of alternative subclasses can serve a similar purpose.

A final technique is the concept of "metaprogramming", the use of software to generate other software. In fact, this is a general technique for building not just software but customized instances of any type of structured artifact. 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 all versions of any or all elements of a product or product family, and providing the means to derive any version of a product or instance of a product family.

The Nature of Metaprogramming

Metaprogramming can take one of two general forms: descriptive or prescriptive. The descriptive form is software that manipulates the syntactic form of a product element, represented in an "adaptable" notation to express a family of elements. It operates within the frame of fixed fragments of an element to derive embedded customized fragments. This form is a generalization of code tailoring mechanisms from above. 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 prescriptive form of metaprogramming is both more flexible and more opaque than the descriptive form. Prescriptive metaprogramming is software written to operate on an intensional specification of an envisioned element to manipulate (derive, transform,

and combine) element fragments as data; data elements correspond to instances of the various semantic constructs of the software 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 (though potentially at some cost in developmental quality).

The use of these techniques for consistent customization of the content of all elements of a product is addressed in later sections. A particular notation for defining adaptable components as an example of the descriptive form of metaprogramming is described to complete this section.

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.

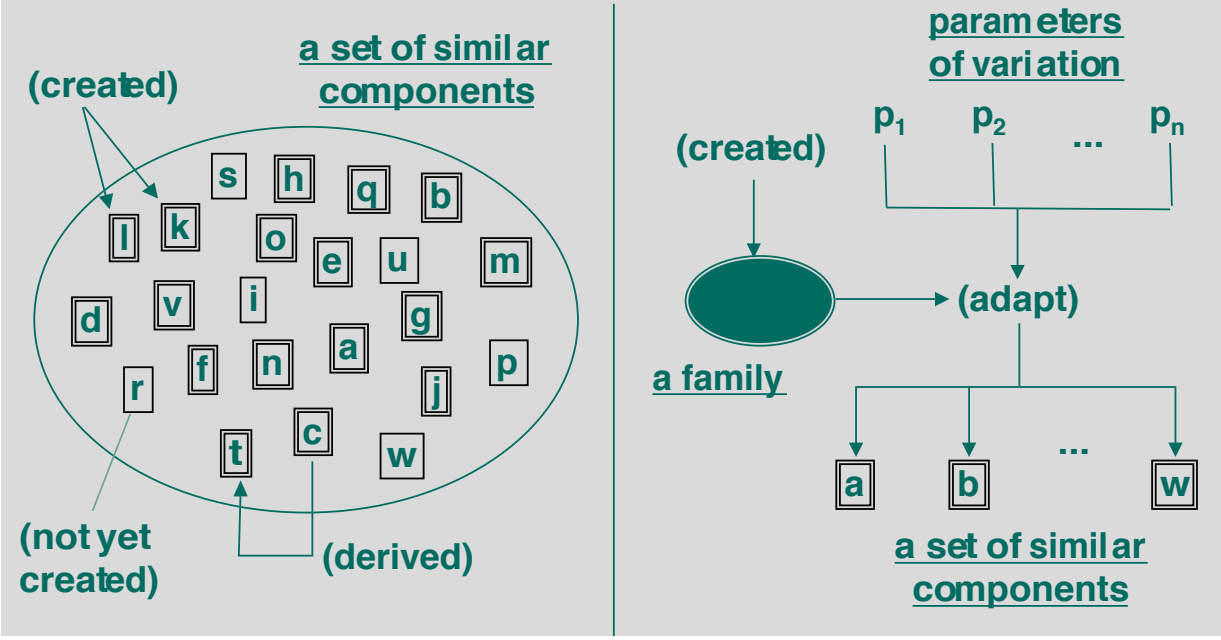

Figure 4.1-1. Two Views of a Component Family

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)

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 ? ------

One answer is to identify software capabilities that the enterprise will need in multiple 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.

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.

Even 40 years ago, it was apparent to someone building software in a given subject matter area that it made no sense to always start over on building a new program if they had previously built a program solving a similar problem. Even with significant differences, there were often parts of the problem that could use corresponding parts of the previous solution.

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, particularly in the then-preferred linear process of 40 years ago, was to just copy all or parts of the previous solution and modify them as needed to create the new solution. From the perspective of the new software effort, this made sense, certainly more than just discarding the prior solution, but it created or failed 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 disagree on how to solve such problems.
- 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 will 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