A System Composition Pattern Language

Introduction

A modern software application seldom exists in isolation but instead within a set of related applications, such as a product line or corporate information system, in a shared context. Addressing such applications one at a time as the traditional application-oriented approach does, treats many of the common aspects of the context independently and repeatedly with each application. This approach results in higher costs for duplicated work, but more importantly it inevitably produces redundancy – multiple defined data and logic – and hinders the effective management of complexity and variation, all of which impair quality and reduce value. This System Composition Pattern Language addresses how to reduce such problems by recognizing sets of overlapping products and components as being a single system and optimizing the system as a whole. *Optimizing* means to reduce cost and increase value especially in terms of integrity, consistency, adaptability, and improved cycle time for new products.

Managing a system of related applications is a task too complex to be addressed by a single pattern, even in the abstract. Instead a set of related patterns, called a *pattern language*, is required. Table 1 summarizes the patterns of the System Composition Pattern Language.

Table 1. System Composition Pattern Language Summary

Problem	Solution	Pattern Name
How can you optimally build, maintain, and evolve a set of overlapping products (i.e., products with potentially common components)?	View all of the related products and components as a single system and strive to optimize the system as a whole instead of each product individually. Address redundancy, variation, and complexity at the system level, where these issues can be effectively managed.	*A.1. COMPOSABLE SYSTEM*
How can you ensure that components created by different groups at different times will inter-operate and not overlap?	Define a common authority and a set of global standards and policies.	*A.2. COMMON ARCHITECTURE*
Redundancy (i.e., multiple *definitions*) is the source of many quality problems, most notably impaired consistency, adaptability, and integrity. How can you eliminate redundancy across multiple products?	Partition the entire system into a common set of nonredundant primitives from which all the products can be synthesized.	*B.1. NONREDUNDANT PRIMITIVES*
	Segregate redundancy that cannot be eliminated by *NONREDUNDANT PRIMITIVES* into *sets* of primitive components and control the redundancy via common management of each set.	*B.2. MANAGED REDUNDANCY*
How can you combine the primitives into products without creating redundancy or unnecessary complexity?	Synthesize products in stages from nonredundant primitives via components at intermediate levels.	*B.3. INTERMEDIATE COMPONENTS*

How can you ensure components are not sub-optimized for a single product or parochial purpose?	Manage components separately from products. Charge component managers to support all products but with clear guidelines to prevent the de-optimizing of the overall system. Maintain the principle of component independence.	*C.1. INDEPENDENTLY MANAGED COMPONENTS*
How can you ensure that every component is managed effectively and the management of related components is well coordinated?	Group together components by relatedness for management. Assign every component to one and only one group. Group groups recursively into a hierarchy.	*C.2. DISJOINT COMPONENT MANAGEMENT GROUPS*
How can you provide all of the variation required across the system but prevent unnecessary variation?	Centrally manage variation. Define only the variants necessary to meet bona fide requirements across the system and no more.	*D1. SUFFICIENT VARIATION*
How can you minimize the number of variant components yet still support all required variation?	Identify and separate all orthogonal dimensions of variation. Define a single set of variants for each dimension and implement each variant with a separate component with a common interface. Allow variants of each dimension to be selected independently and combined dynamically.	*D2. SINGLE-DIMENSION SUBSTITUTABLE VARIANTS*

The *COMPOSABLE SYSTEM PATTERN* (A.1) describes the general approach and could be termed the *primary pattern*. The other patterns address in greater detail various aspects of that approach, and thus are dependent on the primary pattern and do not necessarily stand by themselves. If the *COMPOSABLE SYSTEM PATTERN* is not applicable to your problem, it is not likely that the other patterns of the language will be applicable either.

Most of the patterns, except for the primary pattern, are described in a single page. The primary pattern must establish a common context and is thus somewhat longer.

In many cases one pattern addresses the context resulting from the application of another pattern. For example, applying the *NONREDUNDANT PRIMITIVES PATTERN* produces many fine-grained components, while the *INTERMEDIATE COMPONENTS PATTERN* addresses one aspect of dealing with many components. This example illustrates pattern application *order*, which is a characteristic typical of pattern languages. Figure 1 depicts the dependencies among the patterns and the organization of this document. A double-headed arrow indicates alternative patterns, i.e., patterns that address the same issue in different ways. One set of alternative patterns is included, i.e., *NONREDUNDANT PRIMITIVES* and *MANAGED REDUNDANCY*; alternatives to other patterns in this language may also exist.

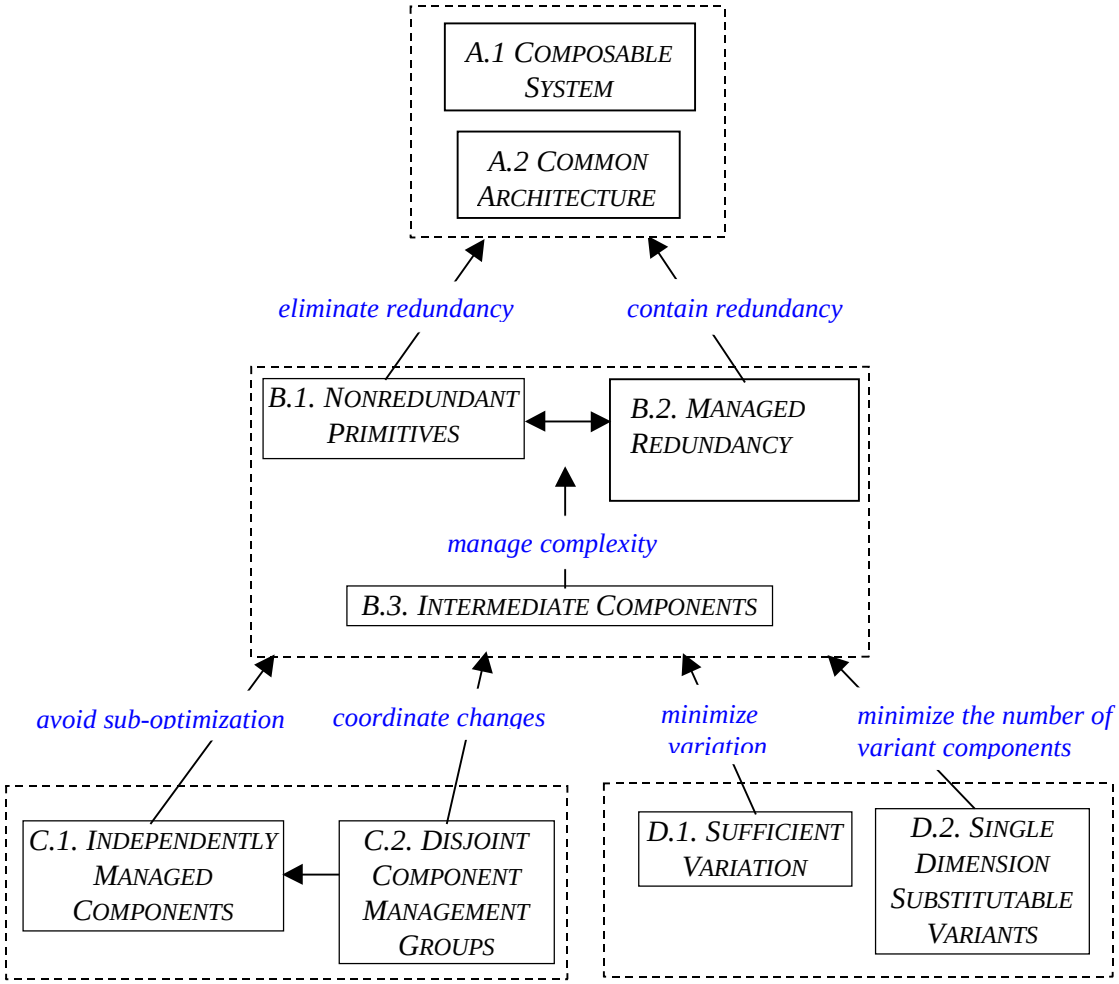

Figure 1. Composable System pattern relationships

This pattern language so far addresses only a few of the major issues of component-oriented systems. Opportunities abound for extensions and refinements; some candidate patterns are listed in Table 2, in Appendix A. The remainder of the article describes the patterns listed in Table 1.

Although this article specifically addresses software systems, the scope of the System Composition Pattern Language is very broad; it transcends software and can be applied in other domains as well, such as documentation and manufacturing. Some examples in other domains are discussed.

A running example from the documentation domain is used to show how the patterns work together to solve the larger problem. The example is introduced in the *COMPOSABLE SYSTEM PATTERN*, and a relevant part of it is described in each pattern. The example is in italics to distinguish it from the pattern text.

The definition of important terms, including *product*, *component*, and *variant* is provided in a glossary in Appendix B.

Composable System Patterns

A.1. COMPOSABLE SYSTEM PATTERN

Problem
How can you optimally build, maintain, and evolve a set of overlapping products, i.e., products with potentially common components?

Context
You have multiple products with a common context and overlapping content such as, a product line, corporate information system, or application family.

This pattern is most applicable when
- many overlapping products exist
- the product overlap is large
- consistency and integrity across the system is important
- the system is large and complex
- extensive variation is needed
- rapid response to change is needed without sacrificing consistency and integrity.

Forces

+ Driving Forces	*- Restraining Forces*
• The long-term economic success of the enterprise depends upon *overall* quality and productivity. • Without a holistic focus, it is difficult to tell whether a "solution" results in a net gain or just moves problems to a less visible location or costs to a future time period. • Important quality issues, such as redundancy, complexity, and variation, can be effectively managed only at a global level.	• The natural focus is on the short run delivery of individual products. • Urgent requirements may demand a "quick fix" precluding a holistic approach. • The whole is more challenging to deal with because it is larger, more complex, and controlled by less accessible managers. • Addressing the whole requires initial investment long before benefits are realized. • Existing parts of the system may exert constraints upon evolution.

***Example*:** *A vendor contracted to produce a family of about a dozen devices for a government agency. Government documentation standards were in effect, which specified a document for each device in a standard format. Although each document was unique, much of the required documentation was common or similar across documents, including the required format, vendor and contract information, general device requirements and context of use, the description of common communications protocols, and many common hardware and software components.*

Solution
Address the entire set of products and components as a single system and strive to optimize the whole system rather than sub-optimize products individually. Establish a *COMMON ARCHITECTURE*. Partition the system into *NONREDUNDANT PRIMITIVES* and synthesize *INTERMEDIATE COMPONENTS* in stages from the primitives. *INDEPENDENTLY MANAGE*

COMPONENTS separately from products to ensure components are not sub-optimized for a single product or parochial purpose, and that they adhere to the principle of component independence. Assign components to *DISJOINT COMPONENT MANAGEMENT GROUPS* so changes across products can be coordinated and variation can be controlled. Allow only *SUFFICIENT VARIATION*; partition needed variation into *SINGLE DIMENSION SUBSTITUTABLE VARIANTS* to allow configuration by simple selection.

Example Documentation Solution: *The whole set of documents was analyzed to determine the extent and location of the commonality.*

- *A few chapters and sections were entirely boilerplate or specific to the vendor or contract and so were identical for all documents.*
- *Several chapters were identical except for certain terms, such as the device names and other identifiers.*
- *Other chapters were significantly different, but contained sections which were either identical or varied only slightly; the analysis was applied recursively through several levels.*

In this way the entire system of documentation was decomposed into a set of components including chapters, sections, subsections, etc., terms and their allowable sets of values, a format, and the set of deliverable documents. Each primitive component was defined, and each composite component, up to and including complete documents, was defined as a combination of constituent components.

By addressing the system as a whole and decomposing it into disjoint components it was possible to create a system with almost no redundancy and thus totally consistent, with a single point of change for every modifiable aspect of the system, and of minimum total volume.

Rationale

Quality is dependent on minimizing redundancy and controlling complexity and variation. These objectives can be accomplished only by managing at a global level. Strategies exist to deal effectively with all of the restraining forces (see the other patterns in this pattern language).

The fundamental trade-off is between (1) the cost of coping with redundancy, complexity and unnecessary variation and (2) the cost of avoiding them through better planning, organization, analysis, and design. When considering a new approach, it is typical to underestimate the current costs, which seem only natural, and to overestimate any new costs, which appear to be extra and unnecessary. It is also typical to underestimate both the value of new benefits not yet experienced, and how well and inexpensively a new approach can work after it has been perfected.

"Optimize the whole" is a key principle of the Total Quality approach, which includes a great body of experience and supporting evidence; it is the whole that is of concern to the economic unit, not products individually. Without a holistic focus, many "results" are illusory, and are actually trade-offs against other goals that may leave the enterprise as a whole little better, and perhaps even worse, off. It is easy to be fooled when focusing on only one piece of a large puzzle. It is important to realize that the whole cannot be optimized by optimizing the parts individually.

Example Rationale: *Although a significant initial investment was required for analysis, structuring, and setup, the system was thereafter able to adapt quickly to changes in either content or format without any redundant updating while maintaining perfect consistency across*

all documents. The value of such qualities in the longer run exceeded by a large factor the initial investment. Moreover, it is difficult to see how such high adaptability together with perfect consistency could be provided at any price without global partitioning into non-redundant components. When the contract was later expanded to include several new devices, documents were quickly created for the new devices consistent with the original set of documents by reusing the existing system and the many already existing components.

Upon delivery, the client stated that he had never before received such a set of documents on time, up-to-date, and perfectly consistent.

Resulting Context

- **Many components are created** (and must be managed).
- **Component evolution is required** as well as new components to support new products and new product requirements.
- **Some components become common to multiple products**; changes to such components must be managed to avoid unintended effects.
- **A component-oriented as well as a product-oriented organization emerges.**
- **Effective communications becomes more important** both from product managers to component managers (*requirements pull*) and vice-versa (*capabilities push*).
- **More and earlier planning is required**; (e.g., to identify needed components and ensure they will be made available when needed for a product).
- **Redundancy is reduced or eliminated**. To the degree that data and logic is defined once and only once, consistency will be perfect, adaptability will be high, and bulk will be minimized.
- **Future investment in enhancements is highly leveraged**. All of the enhancements to a component are available to all of the system's products. Furthermore, more enhancements can be justified because greater value is realized and the cost is spread across more beneficiaries.
- **Related new products can be constructed more quickly**. The existence of components at many levels (assuming they are composable) along with a mechanism for combining and managing them allows new components and products to be constructed with less time and effort. In particular, operating prototypes can be assembled quickly from existing production-quality components.
- **New unplanned products can often be assembled from available components**. Even though they have been designed for the construction of a specific set of products, the components can be combined in other ways to construct products that had not been anticipated.
- **The availability of components suggests new products** that might otherwise not be considered and which often can be created inexpensively and quickly with the development or enhancement of relatively few components.
- **Parallel work is enabled**. Partitioning a system into loosely-coupled components allows each component to be addressed separately and work on multiple components to be done in parallel.
- **Division of labor is facilitated**. The components of a product are typically diverse (e.g., engines and brakes, processor chips and power supplies, operating systems and financial systems, excavation and steel erection) and require radically different aptitude, knowledge, skills, and experience to master. Organization by application usually requires developers to deal with many different specialties, and to learn new ones with each new application; this often produces jacks-of-all-trades but masters of none. Labor can be more effectively divided and specialization exploited around stable sets of related components.
- **Productivity is improved** both by producing more valuable output and by reducing costs.

- **Product evolution by incremental change is facilitated**. The existence of components facilitates incremental change and evolution.[1] Each component can be modified separately, with small and well-understood effects on clients (assuming proper encapsulation). This allows continuous evolution by many small changes, avoiding the trauma and risk of large replacements, and the waste of discarding entire applications.
- **Testing can be confined to the affected components.** Rigorous use of contractually specified component interfaces can reduce or eliminate the need for product regression testing.
- **Resources can be allocated more effectively**. Resources can be directed to where they will provide the greatest benefit to the whole over the long term, such as to components that support multiple current products or lay the foundation for future products, or to new components that will allow existing components to be reused for new products.
- **System structure is stabilized**. Domain abstractions tend to be very stable: businesses have customers, accounts, products and services -- they always have and always will; autos have engine, braking, and steering subsystems; all electrical appliances have a power supply. When systems are partitioned into components based upon stable domain abstractions, the structure of the system, especially the interfaces and high-level component collaborations, also becomes stable. While each component may evolve individually, the overall structure is unlikely to change much or frequently. A stable system structure in turn allows other facets of operations, e.g., team organization, to be stabilized.
- **Variation can be reduced and managed**. Unnecessary variation can be avoided. Necessary variation can be made coherent, available across the system, and easily selectable.
- **Complexity is reduced**. Partitioning a system into many components may seem to increase rather than reduce complexity. But on the contrary, a well-partitioned and organized system is less complex if it separates concerns, and allows a system to be understood at various levels while suppressing irrelevant detail. Also, a Composable System will be smaller, have little redundancy, coherent variation, and adhere to global organizing principles -- all of which facilitate human understanding and management. An automobile, for example, is understandable despite its approximately 10,000 parts because it follows a stable organization and has layers of intermediate components.

Known Uses:

- The documentation system that is described here in a running example.
- Unix provides many components along with simple yet powerful means to combine them, and could be considered a Composable System. Common architectural features, such as a standard data format, facilitate interoperability.
- Various *product line* and *domain engineering* strategies take a Composable System approach.
- Most complex hardware, such as automobiles, and indeed most complex artifacts of any type, are built with a Composable System approach.
- A pattern language, such as this one, is in some sense a Composable System.

Related Patterns: The *STOVEPIPE ENTERPRISE ANTIPATTERN* [Bro98] describes the symptoms of not using the *COMPOSABLE SYSTEM PATTERN* and is in some sense its antithesis. *SEAMLESSNESS* [DSo98] addresses reducing complexity by choosing the structure of a system to mirror a model of the problem domain.

A.2. COMMON ARCHITECTURE

Problem
How do you ensure that a system of multiple components of multiple products built by multiple groups over an extended period of time will be composable, interoperable, and non-redundant.

Context
You have a Composable System with multiple products and common components across products. Components will be built by multiple groups and used by multiple groups over an extended period.

Forces
- A Composable System implies multiple components that can be combined in some way. To be combinable requires at least a common combination mechanism which in turn compels each component to meet a common architectural standard.
- A system cannot be optimized by optimizing the parts individually but only by optimizing the system as a whole; this implies the need for system-level management.
- Important system characteristics, such as redundancy, complexity, variation, and scalability, can be effectively managed only at the system level.
- Centralized management can cause a bottleneck that impedes progress.

Solution
Establish a system-wide architecture governing authority and architecture governance process. The governing authority must define the rules and policies necessary at the global level for the set of products and components to function as a Composable System based upon the requirements and aims of the system; these include:
- Standard definitions and semantics (i.e., an ontology)
- Allowable component forms
- High level system organization including definition of the major interfaces
- Who will own, support, and manage each component, and how they will be organized (see *DISJOINT COMPONENT MANAGEMENT GROUPS PATTERN*)
- How components may be combined
- How variation will be managed (see *SUFFICIENT VARIATION PATTERN*)
- How components will be uniquely identified and versioned, and how they can be accessed
- How the architecture can evolve over time.

Develop streamlined processes and provide sufficient resources so that architecture governance does not become a bottleneck.

Example Documentation Solution: *An executive committee was established and sanctioned by the existing management. The committee established the necessary global standards:*
- *Form: Components would be in the form of standard text components, i.e., words, sentences, paragraphs, sections, chapters, etc., maintained in separate files on a common file system. The top-level document was created as a component and used to define all formatting.*
- *Naming: File (component) names would be based on the document structure (e.g.,"Preface", "Introduction", "Chapter 1") and device structure (e.g., "Subsytem X", "Capability Y", "Component Z").*
- *Combination: The available documentation system would act as the assembly mechanism via its capability to insert referenced files prior to formatting; this capability could be nested to any degree.*

- *Variation: Most variation would be implemented using M4, the Unix general macro processor. A symbolic name was defined globally for each variation point (e.g. $DeviceName), and a value defined for each variant (e.g., Device1, Device2). A configuration, i.e., set of variants selections, was defined for each component having variation points.*
- *Ownership: One and only person would own each component and variant set.*
- *Evolution: Existing mechanisms would be used to define new components and variants; new global issues would be brought to the committee to resolve globally.*

Rational:
An essential pillar of the Composable System Pattern is to view related products and their components as a single system thus allowing system issues to be identified and addressed where they can be dealt with most effectively and efficiently. System issues not dealt with at the system level will be dealt with sub-optimally and perhaps redundantly elsewhere.

Resulting Context:
- A *system* emerges from separate products and components through establishment of common definitions, standards, forms, and common global management.
- An authority is created with responsibility for each global aspect of the system that must be managed.
- A central authority must get involved with all system-level architectural changes.
- All mechanisms, organizations, and standards needed at the system level to support the development of new components and products are put into place.

B.1. NONREDUNDANT PRIMITIVES

Problem: How do you eliminate redundancy across a system of multiple products.

Context: You have a Composable System for which integrity, consistency and adaptability are important objectives, and you thus want to avoid redundant definitions of logic or data.

Forces:
- Many important qualities such as adaptability, consistency, and integrity depend on having only a single definition of logic and data (see *redundancy* in Glossary).
- Extra resources are required to design, build and support redundant software.
- Partitioning into non-redundant primitives creates many components that must be managed.
- Partitioning causes a unit of processing (i.e., a scenario) to be split across multiple components, making it more difficult to follow and understand as a whole, and possibly causing extra control transfers.
- To create components that are non-redundant within a domain requires addressing the entire domain.

Solution: Partition the System into primitive components of sufficiently fine granularity to eliminate multiple definitions of logic or data. Minimize the need to deal directly with fine-grained primitives, by combining the primitives in stages into *INTERMEDIATE COMPONENTS*, and ultimately synthesize products from these larger components. Assign every component to one and only one *DISJOINT COMPONENT MANAGEMENT GROUP* where it can be managed with related components and changes can be coordinated to avoid redundancy and maximize efficiency. Factor out any emerging redundancy into new *NONREDUNDANT PRIMITIVES*. Make enhancements to primitives, not products.

Example Documentation Solution: *The documents were decomposed into chapters. Each chapter that was unique, i.e., contained no significant redundancy with other sections, was designated to be a primitive component and implemented once for all of the documents. The chapters that were not unique but contained significant commonality were further decomposed and the common sections designated primitives. This process was continued recursively until no two components included significant redundancy.*

Rationale:
Redundancy severely limits quality, especially consistency and adaptability, and also impairs integrity and increases bulk; it should thus be considered a quality defect. Rather than attempting to remove defects later (or coping with them), it is better to avoid creating them in the first place.

Resulting Context:
- Many fine-grained non-redundant components are produced.
- Applications cannot independently define components.

Related Patterns:
When it is difficult to apply *NONREDUNDANT PRIMITIVES, MANAGED REDUNDANCY* may be used as an alternate pattern. The *INTERMEDIATE COMPONENTS PATTERN* addresses the complexity of using many small components. *COMMON ARCHITECTURE* provides the foundation for a component-oriented system.

B.2. MANAGED REDUNDANCY

Problem
How do you eliminate redundancy across a system of multiple products.

Context: You have a Composable System for which integrity, consistency and adaptability are important objectives, and you thus want to have one and only one set of definitions. You cannot apply the *NONREDUNDANT PRIMITIVES PATTERN* in every instance.

Forces:
- Some dimensions, such as text semantics and style, are difficult to separate so that redundancy results if variants are required in one of the dimensions, i.e., variant styles may result in redundant semantics.
- The other forces are the same as those stated for *NONREDUNDANT PRIMITIVES*.

An example where it may be difficult to apply *NONREDUNDANT PRIMITIVES* is versions of documentation that have the same semantics but must be tailored for different purposes (e.g., versions that are legalistic, full detail, condensed, and summary) or audiences (e.g., technical and managerial; or in-house and external).

Solution: Segregate each instance of redundancy that cannot be eliminated by *NONREDUNDANT PRIMITIVES* into a minimal number of primitive components and provide common management for the set of components. Hold the set manager responsible to maintain the consistency of the redundant definitions across each component set.

Example Documentation Solution: *Some corresponding sections of different documents had identical intent and some identical text, but the text was interspersed with too many small sections of unique text to be worth partitioning into NONREDUNDANT COMPONENTS. Instead of partitioning these sections further, they were recognized as being mutual redundant and one person was given responsibility for maintaining all of them while ensuring that the redundant parts remained consistent across the set of components.*

Rationale:
The best way to deal with redundancy is to avoid it completely via *NONREDUNDANT PRIMITIVES*. The next best alternative is to confine it to minimal sets of components and control each set under common management. To be nonredundant means to have one and only one definition within the system. In the former case the single nonredundant definition is the nonredundant primitive; in the latter case the nonredundant definition must exist in the head of the component set manager, and he must ensure that all of the components reflect that single definition.

Resulting Context:
- Many components are produced.
- Applications cannot independently define components.
- Component sets are created which require common management.

Related Patterns:
NONREDUNDANT PRIMITIVES, when it can be applied, is a preferred alternate pattern to *MANAGED REDUNDANCY*.

B.3. INTERMEDIATE COMPONENTS

Problem: How can you best synthesize multiple products from many small primitive components with managed complexity and minimal redundancy.

Context: You have a Composable System and have applied *NONREDUNDANT PRIMITIVES* and/or *COMMONLY MANAGED COMPONENT SETS* and have created many primitive components.

Forces:

- Many primitives may be required to synthesize a product (e.g., an auto has about 10,000 components, a processor chip millions of transistors, an application many instructions).
- It is tedious and time-consuming to deal individually with many fine-grained primitives.
- It is difficult to understand the working of many primitives without intermediate structure.
- Many problem state abstractions occur at intermediate levels and each should have a corresponding component.
- The existence of intermediate components facilitates evolution[1].
- The same combinations of primitives tend to reoccur within and across products.
- Coordination logic is most easily understood when distributed close to the components directly coordinated (i.e., within intermediate components).

Solution: Combine the primitives in stages into larger components. Create a leveled structure with *NONREDUNDANT PRIMITIVES* at the bottom, products at the top, and layers of intermediate components in between. Create intermediate components that relate closely to problem state abstractions. Encapsulate details at each level to control complexity. *INTERMEDIATE COMPONENTS* can be identified before as well as after primitives (i.e., the identification process may proceed top-down as well as bottom-up).

Example Documentation Solution*: The most primitive components, such as bare sentences and variables, were combined into complete sections, the sections into larger sections, chapters, etc. Each conceptual level directly combined the components of the next lower level; thus each document (product) combined the title, all of the chapters, and the variants specific to it. Each component was named for a well-defined domain entity, such as a part of the standard document format or a hardware component.*

Rationale:
It is necessary to partition a system into fine-grained components to eliminate the quality defects of redundancy. But it is difficult to deal with many fine-grained primitives, and with products built directly from primitive components without intermediate structure. Intermediate components that mirror components in the problem domain make a system more understandable, and supports more efficient handling, both mentally and physically, of the primitives. Structure is a fundamental device for managing complexity, and the more stable the structure the better.

Resulting Context:

- Many intermediate components are created (in addition to the many primitive components)
- Larger, more capable components are created, allowing products to be assembled (directly) from fewer components.
- Coordination logic can be built up in stages and is distributed over multiple intermediate levels rather than centralized at the top.

- Products acquire a leveled structure that reflects, and becomes as stable as, the problem domain.
- The structure submerges details and allows the system to be understood at multiple levels of abstraction.
- The intermediate component interfaces further isolate the effects of change.

Related Patterns:
"The *WHOLE-PART* design pattern helps with the aggregation of components that together form a semantic unit." [Bus96] *WHOLE-PART* addresses the following three types of relationships: assembly-parts, container-contents, and collection-members.

COMPOSITE [Gam94] describes how to organize component hierarchies so that Wholes and Parts can be handled uniformly. "*CASCADE* is a generic pattern for layering and ordering the parts of a complex whole. Each layer is itself a *COMPOSITE PATTERN*." [Fos99]

C.1. INDEPENDENTLY MANAGED COMPONENTS

Problem: How can you ensure components are not sub-optimized for a single product or parochial purpose.

Context: You have a Composable System and have created components used in multiple products.

Forces:
- Each product manager, by charter, strives to optimize an individual product (i.e., *a part*).
- The best interest of the whole requires optimizing across all products (i.e., *the whole*).
- Products may have differing or even conflicting requirements for a common component.
- Component-orientation requires additional organization and risks creating a bureaucracy.

Solution: Manage components separately from products. Charge component managers to support the products but without de-optimizing the whole, and to maintain the *component independence principle* (see Glossary). Ensure that every component is assigned to one and only one *DISJOINT COMPONENT MANAGEMENT GROUP*. Channel all requirements for a component to the component owner so that they can be most effectively and efficiently supported as a whole. Forbid product managers to create components themselves or to own existing components. Avoid bottlenecks by providing the component organization with sufficient resources to be responsive to the product organization, and by requiring the product organization to plan ahead to identify requirements. Establish an effective mechanism to communicate the existence and capabilities of components.

Example Documentation Solution: *Every component was assigned to an owner independently of the documents (although in some cases one person wore two hats). Permission to modify a component was given only to the owner; all others were required to funnel their requirements to the owner. Each owner periodically provided the current status of the owned components, and announced new planned and available capabilities. Differing requirements were resolved in several ways without creating redundancy, such as by eliminating a misunderstanding, reaching a compromise, or creating component variants.*

Rationale:
Product managers and developers are not well positioned to own and manage common components, by virtue of their perspective and motivation, and often also by their knowledge and skills[2]. A separate component organization with a different charter and focus is thus needed. Gathering and analyzing all related requirements together is a key to optimizing the whole. Many options exist to resolve differing and conflicting requirements, but they are not likely to be found unless someone is in position to view all relevant requirements and charged to find common optimized solutions.

Resulting Context:
- The organization becomes component-oriented as well as product-oriented.
- A flow of requirements emerges from product managers to component managers, and a flow of information about capabilities, plans and status emerges in the reverse direction.

Related Patterns:
DISJOINT COMPONENT MANAGEMENT GROUPS addresses how to manage related components.
SUFFICIENT VARIATION addresses one way to deal with differing or conflicting requirements.

C.2. DISJOINT COMPONENT MANAGEMENT GROUPS PATTERN

Problem: How can you ensure that every component is managed effectively and the management of related components is well coordinated?

Context: You have a Composable System, have applied *NONREDUNDANT PRIMITIVES*, *COMMONLY MANAGED REDUNDANT SETS*, and *INTERMEDIATE COMPONENTS*, and have many components to manage.

Forces:
- Components must continue to change and evolve to support current and new products.
- Some sets of components are closely related and may require coordinated changes, or at times may need to be refactored to avoid redundancy, for example:
  - a superclass and its subclasses
  - sets of variants of the same dimension
  - components and their specialized subcomponents
- In a component-oriented system it is necessary to be able to find a component if it exists, or to determine its absence if it doesn't, and to know where to direct requirements.

Solution: Assign every component to one and only one component management group. Assign the most closely related components, i.e., those most likely to require coordinated changes, to the same group. Define the group organization as part of *COMMON ARCHITECTURE*. Partition component responsibility disjointly, i.e., so that group responsibility does not overlap and one and only one group is responsible for each component. Publicize the group organization and responsibilities.

Example Documentation Solution: V*ariants for device and document identification were assigned to the same group since device and document names and numbers are closely related, and if one changes, the others were likely to as well. When a new device was defined, a new variant was required for each of the related components.*

Rationale:
Every component must be managed, and related components should be managed together.

The distinction between grouping for use and grouping for management is an important one. It is a good thing for a component to be included in multiple use groups (e.g., products, intermediate components); this represents reuse that leverages the component's value. But it would be a bad thing for a component to be in multiple management groups, as this would represent redundant definition or ownership, which would reduce quality and value. Thus *use may overlap*, but *ownership must be disjoint*.

Resulting Context:
- A hierarchical component organization based upon component *ownership* is created independent of the product organization based upon component *use*.
- A natural home is created for every component, and components are managed in groups whose proximity reflects their relatedness.
- The hierarchical component ownership organization provides a structure for component searching and for collecting related requirements.

D.1. SUFFICIENT VARIATION

Problem: How can you provide all of the variation required across the system but prevent unnecessary variation.

Context: You have a Composable System that requires variation (as virtually all do).

Forces:
- Variation is valuable and necessary in almost all systems.
- Unnecessary variation creates extra complexity and costs.
- Insufficient variation encourages redundant development.
- Arbitrary variation is a natural result of independent redundant development, i.e., independent developers can be expected to introduce arbitrary differences into common functionality (above and beyond any required variation).
- Variation is a global phenomenon and can only be managed at the global level.

Solution: Establish a manager for each variation dimension (see *DISJOINT COMPONENT MANAGEMENT GROUPS PATTERN*) with the sole authority to define variants for that dimension. Collect centrally all of the variation requirements across the entire system by variation dimension. Define only the minimum number of variants necessary to support the combined requirements for each dimension. Direct all requests for new variation to the cognizant manager(s). Define new variants only when bona fide requirements cannot be satisfied by existing variants. Introduce only the minimum amount of new variation needed to support each new requirement.

Example Documentation Solution: *Variants were identified where more than one version of a component was needed across the set of documents, and each required variant was implemented. For example, since each device required a unique name, a set of strings was defined as variants of 'Device Name'. One manager controlled this set of names; when additional names were needed, the requirements were given to the manager and he created additional names (variants).*

Rationale: Variation must be carefully controlled to provide high quality and low cost. The availability of needed variation makes a system more valuable and obviates redundant development. But variation beyond what is needed makes a system more difficult to understand and expensive to use and support. Variation, like redundancy and complexity, is a global characteristic and can be effectively managed only at the system level. The benefits of reducing variation have been recognized by the quality community for many decades and is a key objective of quality improvement programs.

Resulting Context:
- Unnecessary variation is suppressed without preventing needed variation.
- A manager is created for each variation dimension.

Related Patterns:
SINGLE-DIMENSION SUBSTITUTABLE VARIANTS addresses how to minimize the number of variant components.

D.2 SINGLE-DIMENSION SUBSTITUTABLE VARIANTS

Problem: How can you minimize the number of variant components yet still support all required variation?

Context: You have a Composable System with multiple dimensions of variation.

Forces:
- Variation occurs in multiple dimensions (e.g., font, size, color), many of which are orthogonal.
- The number of variant combinations is potentially very large (i.e., the product of the number of variants in each dimension), and rises geometrically with new dimensions and variants.
- The number of variant combinations is larger than the number of dimension-variants, and is much larger with many dimensions and variants; for example, three dimensions each with ten variants would have 1,000 variant combinations but only thirty individual dimension-variants (i.e., the difference of the product and the sum). Permanently coupling variants of separate dimensions within a single component spawns unnecessary additional variant components.
- Permanently coupling variants may also create redundancy by defining the same variation within more than one component (*redundancy* is defined as "multiply defined" – see Glossary).

Solution: Identify and separate all orthogonal dimensions of variation. Define a single set of variants, i.e., a *variant set*, for each dimension that provides *SUFFICIENT VARIATION*. Implement each member of the set by a separate component. Factor out all commonality from variant components to avoid redundancy. Enable variants to be selected independently for each dimension and combined dynamically. Manage all of the variants of a dimension jointly as a *DISJOINT COMPONENT MANAGEMENT GROUP*.

Rationale: Variation is easier to comprehend and manage if orthogonal dimensions are separated so they can be dealt with independently. The number of components can be drastically reduced by creating components for each dimension and allowing them to be combined as needed.

Resulting Context:
- The number of variant components is minimized and grows only at the rate of new variants rather than geometrically.
- Redundancy is reduced.
- Once the variant set for each dimension has been created along with the mechanism for combining them, every possible combination can be produced; i.e., new combinations of existing variants do not have to be created explicitly.

Related Patterns:
SUFFICIENT VARIATION addresses how to provide all necessary variation but no more. *DISJOINT COMPONENT MANAGEMENT GROUPS* describes how to manage sets of related components such as variant sets. *PLUG-POINTS AND PLUG-INS* [DSo98] addresses how to provide "modular and extensible variety in an objects' behavior".

Acknowledgments

The following have provided helpful comments on one or more drafts of this document: the Vanguard Pattern Study Group (especially Ed Forbes, Ed Bacon, Dave Bartlett, Bob Maksimchuk and Bruce Lierman), the OOPSLA'98 Pattern Writing Tutorial and Workshop (especially Jim Coplien and Frank Buschmann), Larry Ackley, Lester Shuda, and Rob Weening. Special thanks is due to Liping Zhao for providing many helpful suggestions as part of the PloP sheperding process.

In writing this pattern language, the following patterns were used from *A Pattern Language for Pattern Writing* by Gerard Meszaros and Jim Doble [PloP3].

- *PATTERN LANGUAGE*
- *PATTERN LANGUAGE SUMMARY*
- *PROBLEM/SOLUTION SUMMARY*
- *COMMON PROBLEMS HIGHLIGHTED*
- *RUNNING EXAMPLE*
- *DISTINCTIVE HEADINGS CONVEY STRUCTURE*
- *GLOSSARY*

The pattern documentation in this document generally follows the Coplien form.

References

[Bus96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, *Pattern-Oriented Software Architecture, A System of Patterns*, John Wiley and Sons, 1996.

[Bus98] F. Buschmann, J. Coplien, R. Gabriel, and C. Schwaninger, OOPSLA'98 Tutorial Notes*: Pattern Writing*

[Bro98] Brown, Malveau, McCormick, and Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crises*, Wiley, New York, 1998.

[DSo98] D. F. D'Souza and A. Wills. *Object, Components, and Frameworks with UML, The Catalysis Approach*, Addison Wesley Longman, Inc., Reading MA, 1998.

[Gam94] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading MA, 1995.

[Fos99] T. Foster and L. Zhao, *Cascade*, Journal of Object-Oriented Programming, February 1999.

[PloP3] *Pattern Language of Program Design 3*, edited by Martin/Riehle/Buschmann, Addison Wesley Longman, Inc., Reading MA, 1998.

[Sim69] H. A. Simon, *The Sciences of the Artificial*, The M.I.T. Press, Cambridge, Massachusetts, 1969.

[Was95] M. Wasmund, *The Spin-off Illusion: Reuse Is Not A By-Product*, Software Engineering Notes, August, 1995 (proceedings of the Symposium on Software Reusability SSO'95), p 219.

Appendix A: Candidate Patterns

The items in Table 2 describe ideas for potential additions, extensions, or refinements to this pattern language, but they have not been extensively analyzed.

Table 2. Summary of Candidate System Composition Patterns

Problem	Solution	Pattern Name
How can you exploit the value of the components in a composable system?	Expand the domain to include more overlapping products, i.e., to more organizations within an enterprise, to the entire enterprise, to the industry, or beyond.	*BROAD DOMAIN*
	Remove all barriers to finding, understanding, and assembling components.	*FACILITATE COMPONENT ASSEMBLY*
	Identify new products that can take advantage of existing components.	*NEW PRODUCTS FROM EXISTING COMPONENTS*
How can you maximize investment in new components?	Select components to build or enhance that will provide value to multiple existing or future products.	*NEW COMPONENTS TO SUPPORT MULTIPLE PRODUCTS*
	Build new components that have synergy with existing components.	*SYNERGISTIC COMPONENTS*
How do you decide the proper time for components to be assembled?	Assembly requires processing time depending on requirements and context. Later assembly time provides more flexibility. When assembly occurs is orthogonal to many other considerations.	*COMPONENT ASSEMBLY TIME*
How can you create the most stable system structure?	Define interfaces around domain abstractions. Provide abstract interfaces that will support multiple variants.	*STABLE DOMAIN ABSTRACTIONS*
How can you make a Composable System most understandable in terms of its problem domain?	Define components that relate closely to problem domain objects. Use problem domain component names.	*PROBLEM DOMAIN COMPONENTS AND NAMES*
How can you ensure that a change to a common component will not have unintended effects without full regression testing of all clients?	Rigorously specify the services (contract) that each component provides and do not allow unspecified side effects. Base the use of a component only on the services specified. Following a change to a component, test to ensure that the component still fully supports its specification.	*DESIGN BY CONTRACT*
How can you ensure that all related requirements are considered when designing a component, and redundancy is	Make a change only to the component with the relevant responsibility. Capture requirements as soon as stated. Direct each requirement downward	*REQUIREMENTS PULL*

not created when product enhancements and modifications are made?	from the originating product to the group responsible for components most closely related to the requirement. When appropriate, decompose the requirement and direct subordinate requirements separately.	
How can you ensure that product managers are aware of all of the capabilities available from the components?	Maintain a directory of capabilities. Advertise each new capability as soon as it is planned and when it becomes available.	*CAPABILITIES PUSH*
How can you incorporate into your Composable System a component that doesn't follow the *COMMON ARCHITECTURE*?	Encapsulate the foreign component with interfaces that do follow the *COMMON ARCHITECTURE;* eliminate any side effects.	*WRAPPED COMPONENT*
How can you coordinate the management of software components and the management of the problem domain components they represent?	Align component management groups and the management organization of the problem domain with respect to the components under management.	*PROBLEM DOMAIN ALIGNED GROUPS*
How can you make a Composable System most flexible and least redundant?	Partition to the finest granularity, and define a component for every intermediate component level.	*FULL PARTITIONING*
How can you ensure that common functionality will be packaged in comparable forms.	Define a standard form(s) into which all components of a specific type must be mapped.	*CANONICAL COMPONENT FORMS*
How can you make component assembly most flexible and adaptable?	Separate the directions of how to assemble the components from the components themselves. Automate the assembly process if possible.	*SEPARATE ASSEMBLY DIRECTIONS*

Appendix B: Glossary

The following are definitions of the most important Composable System terms.

Component - a cohesive unit with a well-defined interface by which it can be combined, unchanged, with other components to form a larger system; i.e., a building block, a unit of composability[3].

The notion of a component is fractal, i.e., scale-invariant. A component may be of any size or complexity. Components may be combined in one of many ways with other components to form any number of larger and more complex components, which may in turn be combined in the same or different ways to form other components, ad infinitum.

From a *component perspective*, each component is viewed as a disjoint unit of functionality packaged in a sufficiently generic way to support requirements for that functionality across the system.

Component Independence Principle – a component should be independent of its users and managed in the best interests of all of its clients, not a single particular one.

Product - a component which is delivered, and presumably provides value, to an end user, i.e. someone outside the system.

From a *product perspective* each product is viewed as a composite of components packaged for delivery outside the system; each product is unique, yet related to and consistent in many ways with other members of the system.

Note that a *product* is defined relative to a *system*. A product of one system thus may be a component of a product of another system elsewhere in the food chain. (If so, this would imply a larger system to which the *COMPOSABLE SYSTEM PATTERN* potentially may also be applied.)

Redundancy - one of the key concepts that motivates this pattern and is used here to mean "defined more than once". The acid test for redundancy: when a system change is needed, must you make the change *once*, or *more than once*. A copy may or may not be redundancy by this definition. If the a copy is made through a well controlled mechanism from a single definition, as by a macro expansion or a data base replication, it is not considered to be redundancy because a change need only be made one time for all copies. Similarly, two composites that incorporate a common component are not redundant if the common component can be changed and propagated to all of the composites that use it. *Clones*, i.e., modified copies, on the other hand, because they require a change to each clone, are considered to be redundancy. (Most redundancy is created not by cloning per se, but by overlapping and uncoordinated development.)

The System - the totality of all of the products and components under consideration, along with the rules and mechanisms for creating the products from the components.

From a *system perspective*, all products and components are viewed as a single system to be

optimized as a whole.

Variant Set - a set of components with a common function and interface, but differing in some way. A *Variant Set* is a general mechanism for dealing with variation by segregating it into a set of components in such a way that each distinct variant resides in a single discrete component. This reduces the selection of a variant to the selection of one of a set of components.

---

[1] Simon discusses how the existence of "intermediate stable forms" facilitates evolution: "The time required for the evolution of a complex form from simple elements depends critically on the numbers and distribution of potential stable intermediate forms." Applications (complex forms) can evolve more quickly from good components (stable intermediate forms). Software systems, like living organisms, must evolve and adapt to survive; most systems don't survive because they can't easily be adapted. [Sim69]

[2] Wasmund calls the idea that reusable components can be produced as a by-product of a traditional application development *the spin-off illusion*. He states "There is no 'soft' way of building up reusable assets, because the short-term goals of product development groups fundamentally conflicts with the goal of building up an asset library". [Was95]

[3] This definition owes much to the following definition of *component* by D'Souza and Wills: "A coherent package of software artifacts that can be independently developed and delivered as a unit, and that can be composed, unchanged, with other components to build something larger." [DSo98].