# Table of Contents

1. Introduction ........................................................................................................ 1
2. What Is a Reusable Component? ........................................................................ 1
3. Partitioning Into Domain Components ............................................................... 2
4. Component Interfacing ....................................................................................... 5
5. Conclusion ........................................................................................................... 8
6. References .......................................................................................................... 8
1. Introduction

Software complexity and time to market pressures are more challenging than ever for today’s development teams. New embedded products can have significant software components rivaling even desktop applications. Producing these systems without a powerful method of system partitioning designed for reuse can be daunting if not downright impossible.

This paper outlines effective methods for defining a stable partitioning of a system by subject matter. By effectively separating your software into manageable domain components you can substantially simplify internal component design, simplify interfaces between components, insulate your design from change, and provide reusable domain components for future products. The techniques described in this paper are applicable to systems developed using a functional or object-oriented approach.

2. What Is a Reusable Component?

Reusable software means different things to different people. For most, it means reusing implementation source code or binary components/libraries. While both have their adherents, implementation-level source and binary reuse have had limited success thus far. Ambler (1998) states, “At its worst, code reuse is accomplished by copying and then modifying existing code. A sad reality of our industry is that code copying is often the only form of reuse practiced by developers.” Also, the fine granularity and specificity of objects often defeats attempts at reuse in a different venue. Binary components and libraries are more often successfully reused but are limited by their conceptual specificity (binary components) and platform dependence (components and libraries). Large grained, subject matter based domain components, on the other hand, have significantly greater reuse potential. Ambler (1998) states, “domain components provide the greatest potential for reuse because they represent large-scale, cohesive bundles of business behaviors that are common to many applications.”

While software reuse is a fairly overloaded term, component is probably even more so. In this paper we define a component to be a collection of related elements that behave according to the policies of a specific subject matter and work together to support a cohesive set of responsibilities. A component’s elements do not rely on the existence of elements or other specific abstractions from other components. The capabilities of a component are accessible only via a published service interface. The key here is the term “specific subject matter”. In constructing a typical complex software system, one must deal with many different subject areas and issues: the main application features, user interface, external hardware, database issues, communications, as well as various design/implementation strategies. Clearly, it is difficult to comprehend all of these subjects together.
without some form of partitioning strategy. The strategy discussed here partitions a system based on the various subject matters without regard to system topology, process/thread allocations, or implementation language. Each component is a cohesive entity unto itself that may depend on other components to accomplish its purpose but is not intimately coupled to them. The benefit of this type of partitioning is two-fold. First, because a component embodies a consistent level of abstraction about its subject matter, its internal design is relatively simple and easy to maintain. Second, since most components are not dependent on the things that often change across product lines (hardware platform and system topology) but rather are based on their relatively stable problem space issues, the component’s reuse potential is increased. Lower level and design components, of course, are affected by platform changes. The usual goal applies here: to compartmentalize the platform dependent code as much as possible.

### 3. Partitioning Into Domain Components

Domain analysis is the process of establishing the software system boundary, partitioning the system by subject matter, and establishing requirements dependencies between the identified components. To begin this process the system under development must have already been characterized at a high level, typically through a Product Definition or Concepts document. Additionally, at least a preliminary attempt at the core of the System Level Requirements is needed. Once this level of requirements analysis has been accomplished the domain analysis steps can begin. Below are the basic steps to perform domain analysis. For a more detailed look at domain analysis see Duby (2001).

**Step 1 – Identify the Application Component:** The application component provides the services specific to this particular system. Almost all systems have a single application component named for the purpose of the system from the end-user’s point of view. For example, the application component for a medical instrument that measures the levels of substances in blood might be called “Blood Analysis”. The application component for a microwave oven could be called “Microwave Cooking”.

**Step 2 – Identify Key System Functions:** Examine the requirements and list the key system functions, which are high-level functions the system must perform to satisfy the end-user requirements.

**Step 3 – Assign Key System Functions to Components:** Group related key system functions together into components and select a meaningful component name. Allocate key system functions specific to this particular application to the application component. For example a MicrowaveCooking component would be responsible for executing cooking programs by switching the magnetron, lights, fan, and turntable on and off at the appropriate intervals while a SensorMonitoring component might be responsible for detecting whether the door is open or closed.
Step 4 – Write Component Descriptions: Write a component description that communicates its capabilities, the services it requires, and any potential reuse information concerning contexts that this component could operate in.

Step 5 – Build a Domain Chart: Create a Domain Chart diagram to show the components identified in the previous steps and their dependencies. In Figure 1, the components of a microwave oven controller are represented by UML packages and the inter-component dependencies by dotted arrows from client to server component. The application component is placed at the top. Any server components that the application component depends upon are shown below it. The application component is connected to the components it depends on with directed lines. The rest of the identified components are added in a similar fashion.

Figure 1. Domain Chart for Microwave Controller System
Step 6 – Review Domain Chart: Reviewers should first check the following:

- Are all key system functions assigned to a component?
- Are all names and descriptions clearly written? Is the description not overly specific to the needs of a particular client?
- Is breakout by subject matter and not by topology?
- If common abstractions appear in multiple components, have they been factored out into their own component? This point is crucial as subject-matter leakage between components can lead to similar elements in different components, causing undue complexity, higher maintenance costs, and reduced run-time efficiency.
- Are there any missing dependencies?

Next, let’s look at reuse possibilities. Component reuse opportunities occur due to two related but different phenomena. The first is the discipline of domain partitioning on the application software. Cleanly separating the system by subject matter naturally creates infrastructure/server components (e.g., UserInterface, HardwareControl, MessageServices, Alarms, NetworkCommunications) that are generic enough to be useful in either totally disparate applications or at least across different members of a product family. A server component’s reuse potential can be judged by exercising it in a future hypothetical context (or perhaps an actual, future reuse context). What happens if the component’s client(s) were replaced? Could it still be used as is? Or, could the component’s data be varied to accommodate different members of a product family? For example, say an Alarms component was designed to alert clients when certain incidents occur a specified number of times, or a measured value is outside a specified range. If the component is only concerned with such things as counting incidents, tracking measured values against thresholds, and supporting registration and invocation of client callbacks then this component could be used in multiple embedded contexts. A MessageServices component designed to interpret external control messages (with the result being an invocation of some other component’s service) could be used in multiple contexts merely by changing the message lookup data.

The second major opportunity occurs when we segregate design and implementation into its own component. Separating out implementation concerns from the analysis of the application problem-space offers more opportunity for reuse. The design strategies may be reused on a different analysis. The analysis may be reused by applying different design strategies. To attain this level of reuse, we need to adopt analysis model transformation technologies that automatically map a problem-space analysis to implementation by applying a chosen design strategy. Model transformation technology is the subject for an entire paper itself and will not be explored further here. For an overview of the transformational design approach to embedded systems development, see Fontana (2004).
### 4. Component Interfacing

How much components need to know about their servers’ internals can significantly affect design complexity and reuse. Simple interfaces, where the internals of the server component are hidden (and thus decoupled from the client component) make it easier to modify the server without affecting other parts of the system. We have found that layered, hierarchical subject matter partitioning promotes these simpler more robust interfaces. For example, in Figure 2 we show two possible breakouts for a circuit board manufacturing system where items are moved from station to station via a conveyor belt system. The subject-matter partitioning (a) has delegated all concepts concerning moving items down to a server component (Conveyance). This provides a couple of benefits over the hardware-based partitioning shown in (b). First, the interface provided by Conveyance is much simpler because the details of the conveyor belt system are hidden from the client component. Second, from a maintenance perspective we’re in better shape. If the system is re-implemented using robot arms rather than conveyor belts, the client CircuitBoardManufacturing component is not affected. The client merely requests that the server component move specified items to a specified location. It doesn’t care how the move is accomplished.

![Diagram showing subject-matter partitioning and hardware-based partitioning](image)

**Figure 2. Subject-Matter Partitioning vs. Hardware-Based Partitioning**

Another example of the simpler interfacing meant here is the way communication protocol stacks are typically described. While the protocol between two communicating entities is often complex, the interfaces between the protocol layer components within each entity are very simple – basically send() and receive().

---

5
The concept of shielding clients via simple component interfaces is similar to the Facade Pattern discussed in Gamma, et. al. (1995). All of the benefits mentioned for the Facade, ease of use, weak coupling, parallel development enablement, and reduced compilation dependencies are present here as well. Unlike the Facade pattern, however, an alternative (detailed) interface is not allowed here.

So let’s define what we mean specifically by a component interface. Each individual service that makes up the interface performs some activity or communication within the context of the component and is the only part of the component exposed to the rest of the system. Services can be either synchronous, where the client waits for a response, or asynchronous where the client may expect a response back at a later time. There are a couple of interfacing techniques that increase reuse potential. First, services should be abstracted from the server component’s perspective. It is easy to fall into the trap of letting the client component define the interface it wants (naturally from its perspective). For example, a high-level automotive control component might desire a RegisterOverTemp service from a lower level Alarms component to predefine what callback should be made when some over-temperature condition occurs. However, to maximize reuse, the interface should be constrained to perspective of the component offering the service. In this way, the server is more likely to present a generic interface suitable for multiple clients rather than being geared to a particular client. If named from the Alarms component’s perspective we might have come up with a RegisterThresholdCrossing service that could satisfy the automotive application as well as a high-speed communications switch application.

Another important technique useful for promoting component reuse is the service handle. A service handle, similar in concept to a function pointer, can be provided to a component as an atomic data item. The receiving component can store this handle and then at the appropriate time call the handle without explicitly knowing which service it is invoking. This can simplify the design and promote reuse when asynchronous clients desire an indication when a requested service has completed. Since components can only communicate with other components via their service interface, replying requires that the server component invoke a published client service to indicate service completion. In the best case, the server component has just been coupled with its client by explicitly invoking a specific client service. In the worst case, the server has multiple clients and must now somehow know which client to call back when it completes its service. To alleviate this coupling and complexity, use a service handle instead. Now the server component does not need to explicitly know the client service to invoke or which client to respond to. Instead, it can “call” the provided service handle. Figures 3 and 4 show the difference between the explicit or fixed interface and the more flexible service handle interface. Figures 3 and 4 show two domain components of the circuit board manufacturing system. In Figure 3 the CircuitBoardManufacturing component invokes the Move service of the Conveyance component (for simplicity, the “item” and “location” parameters are not shown). When the Conveyance component
satisfies the requested move it explicitly invokes the `MoveComplete` service of `CircuitBoardManufacturing`.

Contrast this with Figure 4. In this case, `CircuitBoardManufacturing` invokes the `Move` service and passes a service handle to its `MoveComplete` service. Now when the `Conveyance` component satisfies the request it simply calls the service handle provided by the client domain without explicitly knowing which service it is calling.
Because of the relatively simple decoupled interfaces, both reuse and parallel development are promoted using these techniques. Additionally, there may be additional development time savings due to reduced compilation dependencies.

5. Conclusion

Domain component reuse is the most powerful method of software reuse. It offers large-grain reuse vs. the single function orientation of binary components. Domain components are more stable in the face of change since they are partitioned based on problem space concepts rather than topology and/or platform choices. In addition to its reuse advantages, domain componentization tends to provide simpler internal structures and inter-component interfaces. The subject matter focus of domain components creates low coupling between components and the increased opportunity for parallel development. Two main techniques are available for creating reusable domain components. First, performing domain analysis on an application tends to identify server components that can be used in multiple contexts, whether across members of a product family or possibly in totally unrelated products. Second, with the help of analysis model transformation technology, nearly complete separation of implementation issues from the problem-space can be achieved. This provides both application non-specific design components, usable for various products with similar performance requirements, as well as straightforward porting of an application to a different target platform, language, or even to a different processor topology.

Experience with these techniques has shown that successful organizations tend to perform domain analysis. Going forward, consciously think about your system partitioning techniques and you too can enjoy the benefits of reusable domain components.

6. References


Gamma, et. al. Design Patterns, Elements of Reusable Object-Oriented Software, 1995.