A Composable Software Architecture for Consumer Electronics Products

Rob van Ommering

A software architecture is always designed to serve one or more goals. Our goal is to produce a large variety of consumer electronics products in short development times—a necessity to stay alive as a company in these markets! We achieve this by building software components that can be combined in flexible ways to create products. This requires a component technology tuned to the domain, an overall product family design, attention for implementation details, and concern for issues that are traditionally not the domain of a software architect. The architecture as described in this paper is currently being applied in up-market television products.

The architecture of a system is often defined as *the set of subsystems and their mutual relations.* We think that this definition is far too limited:

- It is too '*specific*': it concentrates on facts alone (subsystems, interfaces) and not on, e.g., concepts.
- It is too '*complete*': a full definition of all subsystems and all interfaces is beyond what a single architect or small architecture team can achieve, especially for a product family.
- It is too '*high level*': it concentrates on subsystems alone, while in practice the choice of what some consider to be low-level implementation details (algorithms, data structures, communication mechanisms) can be critical for the success.
- It is too '*technical*': it does not address issues such as development environment, configuration management, process, organization, et cetera.

We define architecture bluntly as *everything a single person or small group of persons need(s) to do to let a large team develop a product or family of products successfully* (see [1] for a large set of definitions of architecture). Our definition is certainly too wide: it also includes fetching pizza on those long evenings just before a deadline! Still, it is a pragmatic definition, and we will illustrate it in this paper—without claiming any completeness—by recapitulating some of the steps we took to define a software architecture for a family of consumer electronics products.

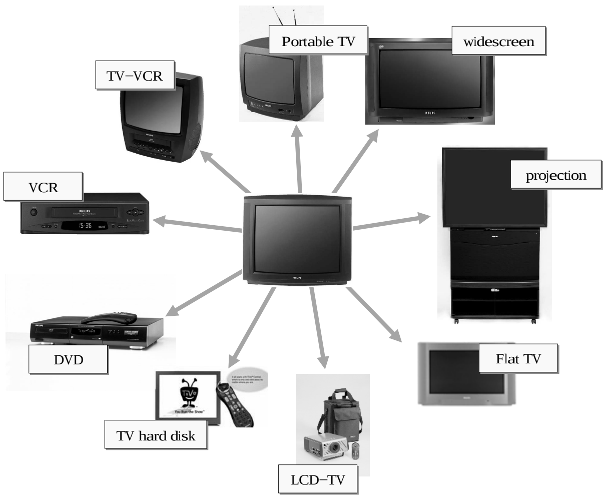

Figure 1: Diversity of products.

We do so in five sections. After a summary of our *requirements*, we introduce some of our *concepts* (our ‘component technology’). We then describe *product family design* issues: the differences between global and regional architecture. We delve into the *depths* of parts of our design, and discuss some ‘*non-technical*’ issues. We end with some concluding remarks.

Requirements

The main requirement for our software architecture is to enable the development of a *diverse* family of products, with for each product a *short development time* and a *high quality*.

Diversity, Lead Time and Quality

Our *product family* includes televisions with variation in price, (world) region, signal standards, image, sound and data features, output device (tube, flat, projection), and with a continuous evolution of the underlying hardware technology (see Figure 1). Soon, other products will be included as well, such as video recorders (VCR), digital versatile disc players and recorders (DVD), compact disc players (CD) and combinations of these products (e.g., TV-VCR). Note how diverse the family is; some products have hardly anything in common (e.g., a TV and CD player).

The size of the software embedded in consumer products grows exponentially, following Moore’s law closely. Current up-market televisions already have two Megabytes of ROM and two Megabytes of RAM. Software *development time* grows accordingly; it now takes over one hundred people more than two years to write the software for a new generation of televisions. This is no longer acceptable, since the market changes so fast that new products must be out in months, rather than years.

Quality is not generally understood to be a critical issue for consumer products, at least not compared to medical systems or rockets sent to Mars. Still, not one customer will be pleased with a television producing a *Fatal Error: Please Reboot your System,* while the same customer pays extra for bug fixes (usually called *new versions*) of the operating system of his PC. Add to this that service in the field is cumbersome for consumer products, and that errors are likely to be found due to the high quantity of products being sold.

Components and Architecture

Diversity increases, lead-time reduction, and quality improvement are in principle conflicting requirements. We feel that they can only be satisfied together by combining two approaches:

- The use and reuse of *components* from which a wide range of products can be constructed (bottom up).
- The definition of a *family architecture* that defines the context for the components to be developed (top down).

These approaches must be balanced carefully. Too much attention on the overall architecture may provide a rigid skeleton with too little flexibility to build a diversity of products. Too much attention on components may provide a set of building blocks that do not fit easily together to form a product. We’ll give some examples.

The *Microsoft Foundation Classes* (MFC, [2]) offer a framework that defines a skeleton application supporting the editing of single or multiple documents with single or multiple views and with file and printing support. A specific application is created mainly through inheritance. The disadvantage of MFC is that it is very difficult—if not impossible—to change the overall structure of the application. Try building an Internet browser (with Back and Forward buttons), with MFC!

Microsoft’s *Visual Basic* (VB, [3]) offers a component approach with which a large variety of interactive applications can be made. Given powerful reusable (and relatively context independent) ActiveX control components, simple applications can be quickly built. For more complicated applications, an architecture still has to be defined.

In electronic design, the components (transistors, core cells, chips, printed circuit boards) are surprisingly context independent hence reusable. There is a physical reason for this: all dependencies must be routed through wires and connectors. With little architectural effort, families of chips can be designed with which a large variety of products can be created.

Choosing an Approach

Microsoft's Component Object Model (COM, [4]) is—of all existing component models—the best candidate for our purposes. It offers language independence, location independence, and most importantly, various ways of handling evolution.

Evolution is very important to us. We want to be able to create new versions of components that still work in old applications, while new applications can take full advantage of the new functionality. Moreover, new applications should not break down if—for some reason—they are combined with old components. *Location independence*, i.e., transparency for calling functions in libraries, other executables, or at other processors, is becoming increasingly important for us, as our newest products have more than one embedded micro controller. *Language independence* (binary compatibility) will become important in the near future, when third party software is to be included in our products.

Unfortunately, COM is still a little too expensive for us. Binary compatibility results in extra code size and performance loss, requiring more powerful controllers and more memory, and in a consumer business this extra 'bill of material' cannot be afforded. Fortunately, we live in a *closed world*. All developers are part of one company, and we can exert some control over them. This allows us to create a component model that 'has the spirit of COM', yet uses a much more efficient implementation technology. This model is called Koala and is discussed in the next sections (for a more detailed description, see [6]). Evolution to COM in due time was an explicit design goal for this model.

The Component Technology

We shall now provide a brief overview of our component model.

Interfaces

Traditionally, a component's interface is described in the component specification document. But we want *different* components in our family to provide the *same* interface. There will be multiple tuners requiring different software drivers, and we want them to have the same interface. As another example, we have different micro controllers and different real time kernels on which we want the same API. So, we define such interfaces independently of components, as *first class citizens*!

Some components provide more functionality than others. Also, a new *version* of a component may provide more functionality than the old version. We therefore do not define an interface 'in one go', but rather use the COM notion of interfaces, as *small sets of semantically related functions.* This allows us to model variation in functionality between components in terms of absence or presence of such interfaces, rather than as implementation notes for the components.

A second advantage of using many small interfaces instead of one large API is that *evolution of interface definitions* can be managed better. No interface definition is perfect, so changes are very likely to occur in practice. Changing an API, while there are already implementations around that either provide or require it, is very confusing (as Java programmers will have experienced while using the AWT windowing classes). Our interface definitions are therefore *immutable* (as in COM). We'd rather define new interfaces (with a new name) than change existing ones.

A third advantage of using small interfaces is that we can also be very explicit on what a component *requires* of its environment. Making *requires interfaces* explicit allows third party binding (as will be explained below), and it also makes the architecture very visible. Advantage of the latter is that architects can spot undesired couplings between components at an instance, and thus have an early warning for spaghetti code arising.

Components

We quote Szyperski [5]: *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*

A Koala component (see Figure 2) offers functionality through a set of *provides interfaces* (drawn as squares with embedded triangles pointing into the component). A Koala component depends on its environment through an explicitly defined set of *requires interfaces* (drawn as squares with 'out-going' triangles). These interfaces must be bound to provides interfaces of other components by the *third party* that performs the composition.

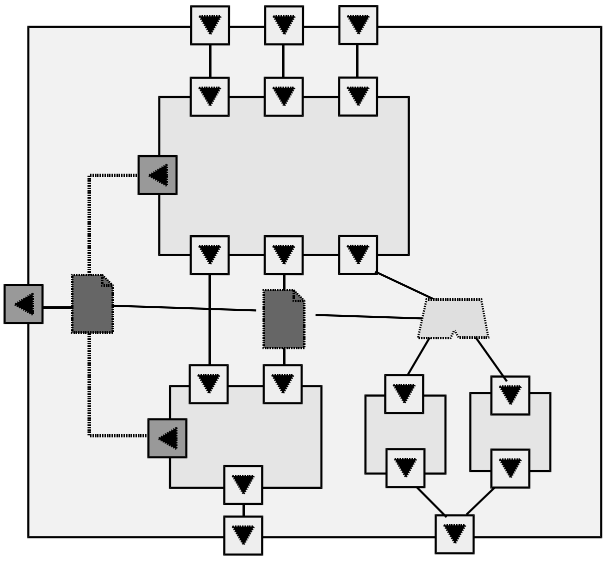

Figure 2: A Koala component.

Figure 2 also shows that our component model is *recursive*. The third party that instantiates and binds components can again be a Koala component! It is important to note that the *definitions* of the smaller components are reusable, but their *instances* are encapsulated in the outer component. In COM, the latter is called *aggregation* or *delegation*, where aggregation means directly passing interface pointers from inner components as if they belong to the outer components, and delegation involves some glue code at the outer level to call the inner components.

Glue code is indeed important, as we believe that the composition of components usually involves more than just clicking them together. In fact, the whole *raison d'être* of Visual Basic is its ability to *glue* components—VB cannot even bind directly! In Koala we have both options, either glue directly, or insert a glue module (the two 'document shapes' in Figure 2).

Requires Interfaces in COM. . .

COM has three ways to let a component 'depend on the environment':

- implicitly, e.g., by calling the Win32 API
- by using *CoCreateInstance*
- by offering connection points

The first option is used regularly in COM, making it necessary to have the full Win32 platform available in each product. For a PC, this is by definition true. For consumer products, we need much more control on what individual components require in order to minimize, e.g., the code size.

CoCreateInstance (and related functions) instantiates (sub) components in COM. It is regularly used to *access* services needed by a component, but since the component actually *instantiates* the service, it either gets its own private instance of the service, or it gets a proxy instance for a singleton class (sharing the service with all other clients). Both cases are not sufficient to deal with, e.g., a TV with two tuners, if two clients must be bound to one tuner and another client to the other tuner.

The third option is in our view the only 'explicit context dependency subject to composition by third parties' in COM. Unfortunately, it is only used in COM for notifications. We plead to make much more use of explicit requires interfaces, and to let the binding of such interfaces be done by encompassing components.

Describing Components and Interfaces

We define interfaces in an interface description language (IDL). We describe basic components in a component description language (CDL), where we list all provides and requires interfaces of the component. We describe compound components in the same CDL, by adding the list of subcomponents and the list of connections (hardware engineers would call these the *part list* and the *net list*, respectively). A small tool (also called Koala) generates header files from these descriptions that perform the actual connections.

We have a graphical representation for component descriptions (see Figure 2), showing all interfaces, subcomponents and connections of a component. These turn out to be valuable design diagrams, as they show the *actual* design of a component at the level of individual interfaces, instead of functions or components (see Figure 3 for a real-life example).

Late Binding

An important issue in product family design is to make the right decisions at the right moments in time. A component designer should not build product specific knowledge into his component, since that prevents the use of his component in other products. Instead, such decisions should be postponed to *product time*. Decisions that influence the component alone can of course be made at *component time*.

A technique for postponing decisions is *late binding*. A requires interface is an example of this. The component designer does not know which component will offer a service to him, so he just declares a requires interface for the service at component time. This interface is bound to a particular service at product time by a third party (the product engineer).

A second technique for postponing decisions concerns *diversity interfaces*, containing parameters to be filled in by the third party that instantiates the component. There is a natural tension between *reusability* and *usability* of a component. The larger the component is, the more usable it is, but also the more likely it is that product specific properties have crept into the component, making it less reusable. The ultimately reusable component is the empty component, which is therefore not very usable! The solution for this dilemma is to *parameterize* components. In Visual Basic, components (ActiveX controls) have a large list of properties with a default value mechanism that allows users of the component to fine-tune the component. In our model, these parameters are grouped into diversity interfaces, which can be bound to values at product time.

There are actually more decision moments than *component* and *product time*. Components are bound into *subsystems*, so there is also *subsystem time*, in which certain component properties are fixed, while other properties are defined in term of subsystem properties. There is also *factory time*, where properties of individual products are defined, e.g., to calibrate the deflection of a TV. And finally there is *user time*, where the user configures his television.

Late binding is often interpreted as *run-time binding*, and indeed this is one way of implementing it. Our component model allows to (re-)compile the software at product time, so that the compiler can optimize the code using decisions made only at product time (we call this *late compile time binding*; the technique is actually an instance of *partial evaluation*). The resulting code is much more efficient than had we only used run-time binding.

For decisions that cannot yet be made at product time, a little bit of code is generated that 'reads' the decision from non-volatile memory. This NVM can be programmed in the factory or by the user (using menus).

Handling Diversity

How do we handle diversity with the component model? Basically in three ways:

- through *selection* and *binding* of alternatives
- through *parameterized* components
- through *switches*

The first two choices are fundamental. If we want to implement drivers for two tuners, then either we create *two components* (of which one is selected and bound into each product), or we create a single *parameterized component*, where a diversity parameter is used to specify the underlying hardware. Rule of thumb is to create two components if the implementations are 80% different, to create one parameterized component if the implementations are 80% equal, and to split-up the component into common and specific parts otherwise.

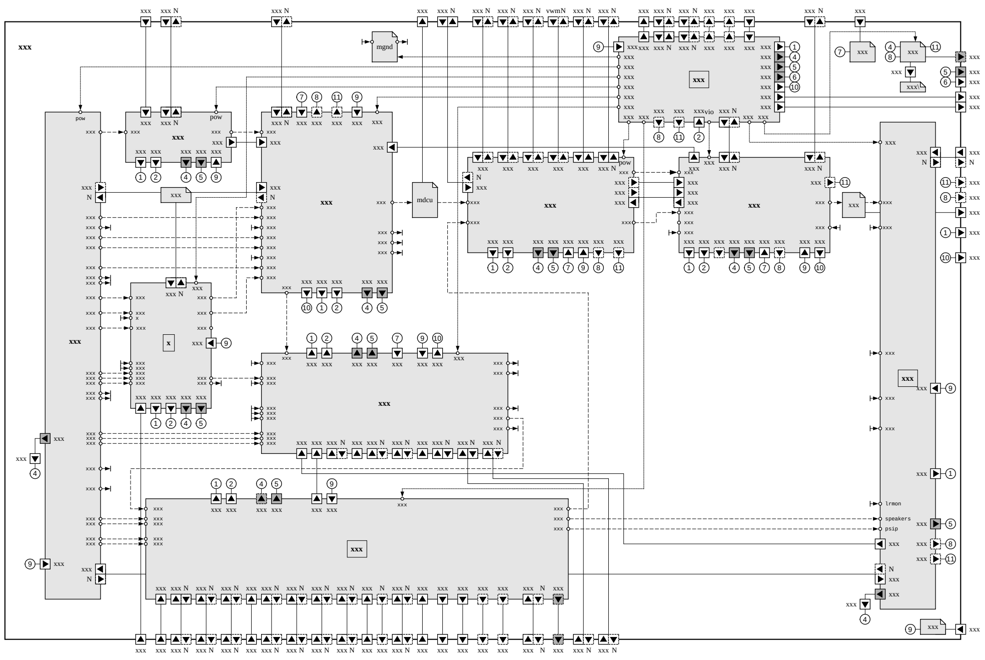
Figure 3: Real-life examples of Koala components 'at work'

We can use an interface *switch* to join these paradigms. A switch is a small predefined parameterized component that can route function calls—consider it a form of pseudo dynamic binding. We can for instance create a single parameterized component from two different ones by connecting the interface through switches (just like in hardware). Figure 3 shows another way how such a switch can be incorporated into a compound component.

The Product Family Design

We have explained the basic component model. This allows people to build components, and other people to use those components to build compound components, until ultimately products are obtained. If we organize all components in a *part-of graph,* showing the 'is an instance of' relation, we obtain a picture like Figure 4.

As architects, we have to manage the development of all of these components. Basic components cannot be designed without taking products into account, but they should also not be designed taking only a single product into account! We'll illustrate some of the steps we undertook to structure the development.

Subsystems

The most important aid is the notion of *subsystem.* Our precise definition of subsystem is a little complicated, due to the fact that we deal with product *families* rather than with single products. Before we start explaining this, please note that most programming languages do not provide much support for 'design in the large' (they are good at handling scope at the file level). Architects are usually forced to add 'design in the large' concepts in the development method that they prescribe. Here's our approach.

For a single product, a subsystem is a *large component that implements a particular subdomain of functionality.* The entire product consists of relatively few subsystems (10–20). In other words, subsystems form the first step of decomposition. It will not surprise you that we strive for maximum cohesion and minimal coupling when defining subsystems. This allows to create teams that develop individual subsystems with a minimum of communication between teams, thus enabling distributed development.

The ‘subsystem component’ itself is implemented using other (smaller) components. The subsystem component encapsulates *instances* of these smaller components. To simplify development, we also want encapsulation of the *definitions* of these smaller components, so that subsystem teams can change them without having to notify other teams. What a subsystem team develops is in fact not a single (large) component but a set of components, where only the large component definition is *public* and the smaller component definitions are *private*. Such a set is usually called a *package*. We can also include our interface definitions in the package: public interface definitions are definitions of interfaces that occur at the boundary of the public component; private interface definitions are only used for interfaces between private components.

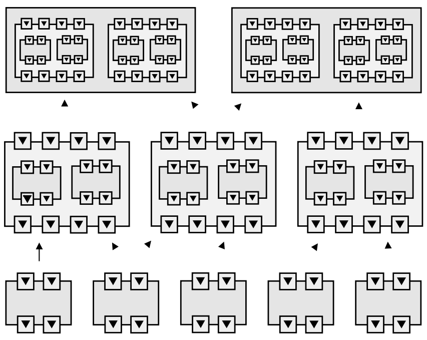
Figure 4: Part-of graph of components

But the subsystem is not used in a single product—it is intended to be used in multiple products of the family! It should be a *unit of composition* rather than a unit of decomposition. Different products will have different requirements for the subsystem, which can only partially be solved by creating a parameterized component. The other alternative for implementing diversity is to create *different compound components* (different selections and bindings from the same set of basic components), where preferably each compound component serves a group of products (it should not be product specific). So, a subsystem package can have more than one public component!

There is one extra way of handling diversity, and that is to offer small glue components (plug-ons) that product designers can use to glue a more generic subsystem into their product. An example is a UIMS, where the compound component requires an interface to draw a string and a bitmap, and where various small components implement these on different hardware devices. These glue components must also be public, as they are used by others.

What to remember from this? Well, from a *product* point of view, a subsystem is still a large component, possibly with some extra glue components. From a *family* point of view, a subsystem is a package with public and private component and interface definitions. The first notion structures the product design, the second notion the development process!

Layers

Many people see layers as *the* ultimate solution for the decomposition problem. In a single protocol stack, a layered design is indeed very useful, but in general the required relation between subsystems is more complicated than a one-dimensional layered architecture would indicate. Still, it is worthwhile to recognize at least three *categories* of software:

- software abstracting from computing hardware
- software abstracting from domain hardware
- application software

Software in the first category can be built in isolation, the second category needs the first, and the third category needs the first and second. We can view this as a *two-dimensional* layered architecture, as depicted in Figure 5.

We found it a good design rule to mirror the hardware structure in the software for the first two categories, while at the same time creating a software API (set of interfaces) that is hardware independent. Reuse of hardware blocks in different hardware platforms then results in reuse of corresponding software blocks in different software platforms. For new hardware blocks, new software components must be developed (using copy and edit), but software development time never exceeds hardware development time.

Note that the three categories of software have different evolution characteristics. On the long term, computing hardware abstraction software (operating systems) can be bought from independent vendors. Domain hardware abstraction software can be bought from the supplier of the domain hardware, and even application software will contain many third party elements (such as web browsers).

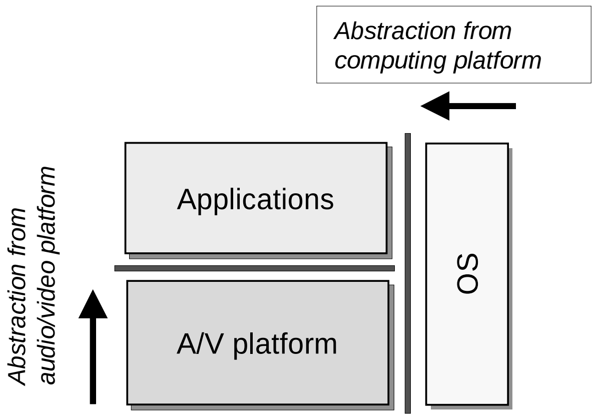
Abstraction from computing platform
Abstraction from audio/video platform
Applications
A/V platform
OS

Figure 5: Categories (layers) of software

Global Architectures

The global architecture contains all concepts, facts and rules that are relevant to all developers. As we are a large multi-site development organization, with many cultural differences, and each site having its own history, it is difficult to predefine a rigid global architecture. Instead, we only define the highly necessary items at the global level, and rely on regional architectures to fill in details that are only relevant to part of the system.

Example elements of the global architecture are:

- The *code architecture*, in terms of naming and coding conventions and a predefined directory structure.
- The *identification of all subsystems.* The actual definition of each subsystem is left to the subsystem architects.
- The *identification of key concepts.* Examples are rules on the use of the real time kernel (as discussed in the next section) and the non-volatile memory.
- The *identification (and sometimes definition) of key interfaces.*

Regional Architectures

The rest of the architecture is defined at regional levels. Note that different parts of the architecture may require different styles and solutions. This is especially true for software in the three layers as defined above:

- The computing platform requires an operating system approach, with device drivers and standard libraries.
- The A/V platform requires an approach that allows commonly occurring variations in hardware (replacing of a chip, change of signal routing) to result in local changes in the software.
- The services and applications require (amongst others) modern user interface technology.

Delving into the Depths

Sometimes, architects need to be concerned with what some consider to be low level details, if this is crucial for the success of the software. We shall discuss one example: the implementation of many small activities in the software.

Execution Architecture

The software in consumer electronics products typically consists of a large number of small activities that are relatively independent of each other. A real time kernel (RTK) can be used to program all of these activities, but there are two problems:

- Using an RTK task for each individual activity provides too much overhead, both in *time* (context switches) and in *space* (each task needs its own stack).
- Activities are then by definition *asynchronous*, and we therefore need to *synchronize* them explicitly, using, e.g., critical sections and semaphores.

Experience shows that unbridled use of asynchronous RTK tasks results in systems with many nasty bugs (dead locks, starvation, forgotten to synchronize...). Let us re-examine the characteristics of our activities. Many of our activities can be programmed in an RTK task as follows:

```
while(true){
  WaitForEvent();
  HandleEvent();
}
```

In other words, they are *stateless event handlers*. With stateless we mean that there is only one point in the task where the activity waits for an event. Of course, the response to an event may depend upon some state maintained in state variables.

A group of such activities can be handled by a single RTK task. Such a task waits for a group of events, and calls the corresponding handler. The handlers are now mutually synchronized, which means that we do not need critical sections for communication between the handlers!

In our implementation, an activity is implemented as a *pump*: a queue of messages with one function that processes the message. Pumps are created on *pump engines*. A pump engine is an RTK task that manages a set of pumps and calls the appropriate pump function whenever there is a message in the queue for one of the pumps.

Each activity has a characteristic time interval at which the event occurs, and the handler for the event has a characteristic duration (which of course must be smaller than the time interval!). Of course, two events with handlers that have significantly different timing requirements cannot be handled by one RTK task: the ‘slower’ handler will block the ‘faster’ handler for too long a time.

So we still need a few RTK tasks running at different ‘heart beats’ to serve all of the event handlers. We assign the activities to these tasks based upon two grouping principles:

- *Cohesion in time*: as explained above, two activities can only be grouped if they share the same timing requirements
- *Cohesion in space*: it is advantageous to group activities with a lot of mutual communication as the implicit synchronization makes the use of semaphores and critical sections superfluous.

But when do we decide which activity runs on which task? Remember that a component builder does not have product specific information. One solution is to define all RTK tasks in the overall architecture. We choose for a different solution.

This allows a component designer to decide that certain activities run on the same RTK task (pump engine), hence need no internal synchronization. The product designer decides how activities in different components are mapped to pump engines. By default, we do not include synchronization mechanisms into our components. If two components operate on different pump engines in a product, then the product designer must also ensure synchronization between the components.

More Architectural Issues

We shall now highlight some of the less technical architectural decisions that we took to setup our product family development. Some architects limit themselves to technical issues only—we feel that to be a severe shortcoming.

Process

First of all, let’s discuss the development *process*. When developing a single product, the selected process is often a derivative of the waterfall model: architecture, global design, detailed design, implementation, testing. For developing product families, *three* types of processes are relevant (see Figure 6, the ideas were derived from [7]):

- defining and evolving the *family architecture*,
- developing and evolving *subsystems*,
- developing and evolving *products*.

These processes run concurrently and relatively independent of each other, but of course with explicit synchronization points. Each process is executed in a *project*. At any point in time, there is only one architecture project, but there are many subsystem and product development projects. Each project has a clear start and end point in time. The architecture, subsystems and products have a longer life span than a project; they are the assets of the organization. The responsibility for maintaining them can be handled by a sequence of projects.

As subsystems are intended to be used in multiple products, they must be developed independently of products. For that reason, we never allow subsystems and products to be developed in a single project. For reduction of overhead, we do allow multiple subsystems to be developed in a single project.

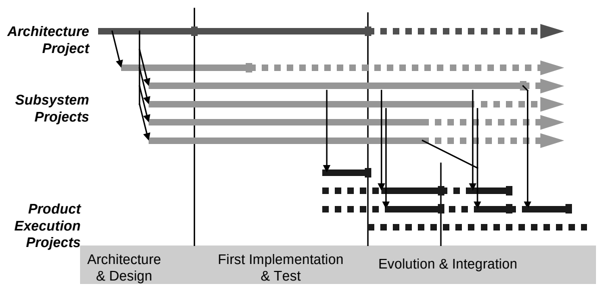

Figure 6: Three types of development processes

Organization

We already explained that architecture, subsystems and products are developed independently of each other (but of course with some synchronization). These development activities are mapped to different departments in the organization, which are usually located at different sites. Often, such departments have specific capabilities, in alignment with the subsystems or products that they develop.

We feel strongly about the rule of developing a subsystem in one project at one site. Ideally, the organization is brought in alignment with the technical architecture. In practice, there are often mismatches between technical architecture and organization. In such cases, we tend to let the technical architecture be influenced by the organization! In other words, we'd rather have a less optimal technical architecture that matches the organization, than an optimal architecture that doesn't match the organization!

Documentation

Traditionally, software is documented with a client requirement specification, a software requirement specification, a global design, a detailed design, and then implementation and test information. In a reuse organization, things need to be a little different.

The most obvious difference is that we concentrate on writing *component data sheets* (i.e., user manuals) rather than requirement specifications. Such data sheets are written in advance, and explain the component to the user of that component, rather than serving as contract for the *builder*. The ideas are borrowed from the hardware domain, where ICs are described in data sheets. Note that these data sheets do not contain any internal design information—this has to be documented separately!

The second difference is that we document interfaces separately of components. This is very useful for generic interfaces: they need only be documented once, and component data sheets can just refer to the *interface data sheets*. It is also useful for more specific interfaces that are provided by a small set of components. It is less useful for interfaces that are really specific to one particular component, but to make things conceptually simple, we follow the same paradigm there.

Configuration Management

Any serious product development activity uses a configuration management (CM) system to maintain the sources of the product. CM systems are typically used for:

- version management
- variant management
- build management
- distributed development

In the context of building product families in a multi site organization, we have a specific opinion on each of these topics. They are all concerned with the move from a single large product organization to a large set of smaller, relatively independent, subsystem and product development 'companies.'

For *version management*, a CM system is invaluable. Each subsystem and product organization should have a CM system in which they keep track of this history of their source files.

We do not want management of *product variation* to be handled by the CM system. Our original reason for this is that we want to handle some of the product variation at compile time, some at run time, without making this distinction in advance. CM systems necessarily operate at compile/link time only. A second and more important reason is that we want to make product variation explicit in the architecture, instead of hiding it in a CM system.

Many CM systems provide *build support* to create executables from the source files. This build support is integrated with the CM system's capability of handling product variation. Since we solve the latter in our architecture, we need not use the CM system's build support, but can instead choose 'the best of class.' As a side effect, we can build our products outside of the CM system, e.g., at home or in the plane.

Finally, CM systems are often use to manage *distributed development*, but we find it a better solution to have the different sites develop and deliver software as if they were separate companies. Our sites deliver releases of their software as ZIP files that are distributed through the company intranet.

Concluding Remarks

We started this article with some remarks on the definition of architecture as a set of subsystems and their mutual relation. We found this definition to be *too specific*, and have spent some time explaining the concepts in our architecture. We found the definition to be *too complete* and *too high level*, and have shown at which level of detail we operate as architects. We found the definition to be *too technical*, and have discussed a number of issues normally not tackled by software architects.

The Koala component model which is part of this architecture is inspired by Microsoft's COM. It was an explicit design goal to enable evolution to COM in the future. However, Koala introduces some concepts not readily found in COM (such as an explicit notion of requires interfaces). We are currently investigating how to model these in COM.

The software architecture as described in this article is now being applied by over 100 people in five different sites.

References

[1] *How do you define software architecture?* Software Engineering Institute, Carnegie Mellon, http://www.sei.cmu.edu/architecture/-definitions.html

[2] *Programming Windows with MFC*, Jeff Prosise, Microsoft Press; ISBN 1572316950.

[3] *Microsoft Visual Basic*, http://msdn.microsoft.com/-vbasic/

[4] *Microsoft COM*, http://www.microsoft.com/com

[5] Clemens Szyperski, *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley, 1998, ISBN 0-201-17888-5.

[6] Rob van Ommering, *Koala, a Component Model for Consumer Electronics Product Software*, Proceedings of the Second International ESPRIT ARES Workshop, LNCS 1429, Springer Verlag, Berlin Heidelberg, 1998, p76–86.

[7] Ivar Jacobson, Martin Griss, Patrick Jonsson, *Software Reuse—Architecture, Process and Organization for Business Success*, Addison Wesley, New York, 1997, ISBN 0-201-92476-5.

About the Author

Rob van Ommering is working at Philips Research on software architectures for product families of resource constrained products in the consumer electronics domain. Key areas of research are the use of software component technologies and the use of explicit architectural descriptions. He is one of the creators of the component oriented software architecture that is currently being used in up-market television products of Philips Consumer Electronics.