Chapter 4

Independent Deployment

Published as: *Techniques for Independent Deployment to Build Product Populations*, Rob van Ommering, WICSA 2001: The Working IEEE/IFIP Conference on Software Architecture, Amsterdam, The Netherlands, August 28-31, 2000, p55-66.

Abstract: When building small product families, software should be shared between different members of the family, but the software can still be created as one system (with variation points) with a single architecture. For large and diverse product families (product populations), the software can no longer be developed in one context and at one moment in time. Instead, one must combine software components of which the development is separated in space and in time, each with their own evolution path. In other words, we need independent deployment of components.

We discuss four aspects of independent deployment in this paper. Two of these aspects - upward and downward compatibility - deal with variation in time. The other two - reusability and portability - deal with variation in space. For each aspect we indicate the relevance, provide some examples, and list several techniques to deal with it. The paper can thus be seen as a guide for product population development.

4.1 Introduction

If you are building a single product, then management of *complexity* is most likely your major concern. The obvious technique is to *divide and conquer*: split the system into well-defined components, build and test the components in isolation, and integrate the components into a system. Many engineers talk about components in this sense. There are of course other (or additional) ways to deal with complexity.

If you are building a *small* product family, then management of *diversity* will be your major concern. A proven technique is to build a variant-free architecture [88] with variation points [41]. Variation points can be implemented e.g. as compiler constants, a registry with properties, functions that can be 'plugged in' (like the

compare function in a sort function), or plug-in components. The latter solution results in a component framework, where diversity is handled by the plug-in components, and where complexity may be managed by splitting the framework itself into components.

If you are building a *large* product family, where the individual products have many commonalties but also many differences, then we believe that it is no longer feasible to create a single variant free architecture in which all software is developed in one context and at one moment in time. Instead, software developed for one product (a television) often needs to be combined with software created for a second product (a DVD player) to create a third product (a TV-DVD 'combi').

Hence, independent deployment of software becomes *the* major concern when architecting large product families (which we call product *populations* [73]). We use the term 'deployment' in the wide sense here; it includes design, development and evolution of the software.

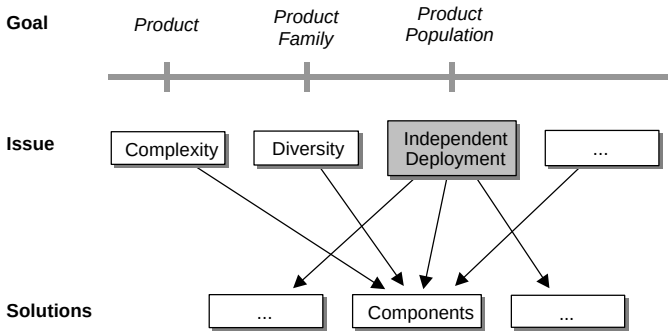

Figure 18. Independent deployment as problem, components as one of the solutions

Figure 18 shows the spectrum that ranges from single product development to product population development. It lists the major concern for each point in the spectrum as described above, and it indicates that there exists a multitude of solutions for each concern.

This paper is about independent deployment of software, and about techniques for achieving independent deployment. Some of these techniques can be called component technologies, others are just common sense engineering. The main questions we want to address are:

- Can I produce a non-trivial piece of software without knowing the specific client or the specific version of the client of that software in advance?
- Can I produce such a piece of software without knowing the specific underlying servers or the specific versions of such servers in advance?
- If so, what techniques do I have to apply?

This paper is organized as follows. In section 4.2, we define the notion of independent deployment and discuss its business relevance. It will turn out that we have to deal with *four* aspects, which we will baptize *upward compatibility*, *downward compatibility*, *reusability* and *portability*. These are discussed in sections 4.3-4.6, together with possible techniques and a number of examples. Section 4.7 puts yet a different light on the issues, and section 4.8 ends this paper with concluding remarks.

4.2 Independent Deployment

In this section we define independent deployment and we categorize it into four aspects that we can discuss separately. We briefly describe each of these aspects in this section - the next sections discuss them in more detail.

4.2.1 What is Independent Deployment?

Suppose we build systems that consist of two parts: a *client* C and a *server* S. Both are just pieces of software that call each other's functions - no distribution of computation is implied by the terms client and server. We can readily construct such systems if we develop the client and server *at the same time and in a single project*, by fine-tuning their mutual requirements. The resulting cooperation is indicated with arrow (1) in Figure 19.

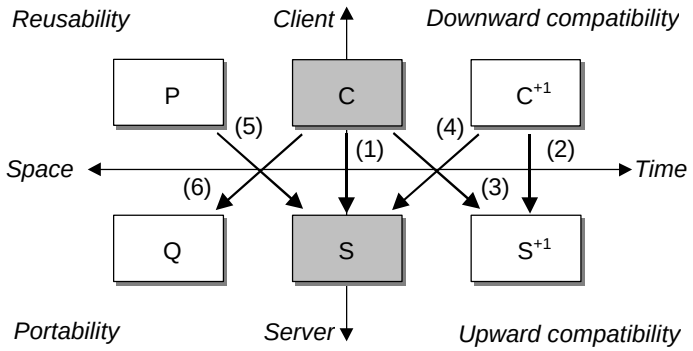

Figure 19. Dependent deployment (1-2) and independent deployment (3-6).

Now suppose we have to upgrade the system to satisfy new requirements. We create a new version of the client, C^{+1}, and a new version of the server, S^{+1}. Again we do so in a single project, and we succeed because we can fine-tune the (new) mutual requirements of the client and the server. The resulting cooperation is arrow (2) in Figure 19.

Both scenarios are state of practice. We shall now consider four other scenarios that deal with the construction of systems out of pieces of software that have *not* been developed together. Two of these scenarios concern a separation in *time* (the

right-hand side of Figure 19), the other two a separation in *space* (the left-hand side of Figure 19)[1].

Combining a new version of the server with an old version of the client (arrow (3)) is only possible if the client is *upward compatible* with respect to the new server. One way of achieving this is to make the new server *backward compatible* with the old server (as explained further in section 4.2.2). Although many people are familiar with this compatibility issue, achieving it is still difficult. Upward compatibility is discussed further in section 4.3.

Combining a new version of the client with an old version of the server (arrow (4)) is only possible if the client is *downward compatible* with respect to the server. Most software does not satisfy this criterion. One of the design goals of Microsoft's COM was to solve this problem. Downward compatibility is discussed further in section 4.4.

Combining a server with a different client (one that was not taken into account when creating the server) is called *reusability* (of the server). Arrow (5) illustrates this combination. Although it might look easy, in practice most software turns out to be applicable only in the specific context for which it has been designed. Reusability is discussed further in section 4.5.

Combining a client with a different server (one that was not taken into account when the client was built) is called *portability* (of the client). Arrow (6) illustrates this combination. Again, in practice, most software is not portable, but instead makes so many assumptions about the environment that it can only be used in the context in which it has been developed. Portability is discussed further in section 4.6.

Please note that the four scenarios sketched above need not occur in isolation. A software component is usually a client of certain servers, and at the same time a server for certain clients. If a component is reused in a new product, then it may have to work with the *same* versions of certain clients and servers, with *newer* versions of other clients and servers, and also with *other* clients and servers than in the previous product, all at the same time. We only describe pure scenarios in this paper to keep the discussion simple.

---

[1] Although time and space are both represented along the horizontal axis, they do *not* span the same dimension. A three-dimensional drawing, with space and time along the X and Y-axis, and the client-server dimension along the Z-axis, would have been better (but is more difficult to draw and comprehend).

4.2.2 Intermezzo on Compatibility

For people getting confused about the terms *backward*, *forward*, *upward* and *downward* compatibility, Figure 20 shows our definition of these terms (see also e.g. [111]).

A client is *upward compatible* with respect to a server if it also runs on a newer version of the server. Note that this cannot be fully guaranteed by the client if the features of the new server are not yet known at the time of building the client.

A client is *downward compatible* with respect to a server if it also runs on an older version of the server.

A server is *backward compatible* if *all* clients (even hypothetical ones) that ran on the old server also run on the new server. Note that this makes *all* clients of the old server upward compatible. The reverse is not necessarily true: clients can be made upward compatible without the server being fully backward compatible.[2]

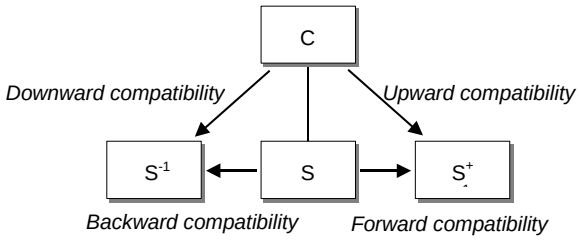

Figure 20. Types of compatibility.

A server is *forward compatible* if *all* clients that run on the server will also run on the new version of the server. There are fewer examples of this kind of compatibility, but here's one: a prerelease of a certain bit of software is usually forward compatible with the final release.

4.2.3 Business Relevance

Why do we worry so much about independent deployment, and should we really? The bottom line is that most of us are still developing software in single projects and for single products. Sometimes, projects are *distributed*, but then different subprojects are *co-developing* instead of developing independently. We cannot easily transfer functionality from one product to another.

To give an example, we can create high-end analogue TVs, we can create set-top boxes and we can create DVD players. But we cannot easily create a high-end *digital* TV with *built-in* DVD player. Independent deployment *in space* helps us to

[2] In one of our architectures, the *golden rule* of compatibility is to make servers backward compatible. The *silver rule* of compatibility is to ensure that all existing clients are upward compatible.

reuse MPEG decoders and DVD players in TVs - this is the traditional goal of software reuse activities. Independent deployment *in time* helps us to decouple development projects and hence reduce risks.

As an example of the latter, suppose we are developing a new version of an Electronic Programming Guide (EPG) 'middleware' for inclusion in a new product. The EPG software is not ready in time, but the underlying software is, and so is the application on top of it. Both have additional features that can't be missed. Can we combine them with the old version of EPG?

More examples will be given in the next sections.

4.3 Upward Compatibility

We shall now go into the technical details of independent deployment. The first question we ask ourselves is: *can an old version of a client work with a new version of a server*? We call this upward compatibility of the client.

4.3.1 Relevance of Upward Compatibility

The relevance of the question above is sketched in Figure 21, where a product consists of a server S and two clients C_1 and C_2. To build a new product, a new version of S, called S^{+1}, may be needed together with a new version of C_2, called C_2^{+1}. But there is no time to update C_1, so C_1 must still run on S^{+1}.

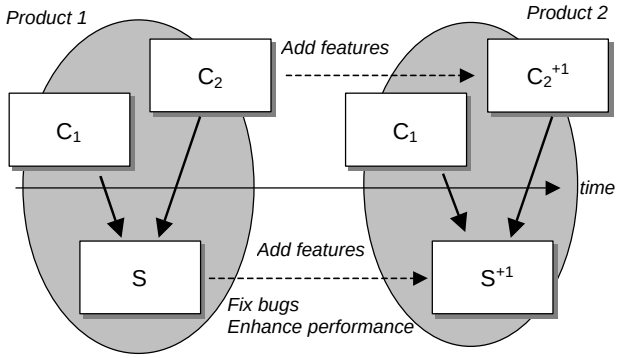

Figure 21. An upward compatibility scenario.

As a second example, suppose C_2 isn't there. Then still we might want to use S^{+1} because of bug fixes and performance enhancements, without being forced to update C_1.

4.3.2 Examples of Upward Compatibility

Windows 95 is designed to be backward compatible with Windows 3.1, so most Windows 3.11 programs will be upward compatible to Windows 95. See section 4.3.6 for some interesting exceptions.

Successive versions of Windows have seen subtle but also significant changes in the driver model. As a result, many hardware drivers only work on specific versions of the operating system. Some (older) hardware may even not be supported any more for new versions of Windows. Upgrading Windows on your PC is therefore rarely a simple job.

Microsoft's Internet Explorer has known several versions since its first release. Web pages that have been created for older versions of IE should still be viewable in IE5.5.

4.3.3 Backward Compatibility of the Server

The main technique for achieving upward compatibility of clients is to make the new version of the server *backward compatibility* with the old version (see section 4.2.2 for our definition of compatibility). This implies that one makes *conservative* extensions only to the component. For example, one can add functions to a component without changing the semantics of existing functions.

This form of compatibility is formalized in the field of *sub-typing theory*. In a nutshell, to be backward compatible, you should *provide more* and *require less*. This applies to data types (simple types, unions, structs), to functions (*in* and *out* parameters), to interfaces (functions, constants, types), and to components (provides and requires interfaces). A full discussion of this is outside the scope of this paper (see e.g. [89]).

The next two sections describe two ways of realizing backward compatibility.

4.3.4 Versioned Interfaces

The first way is to regard the interface of a component as a single interface, consisting of all the functions that the component exports. An extension of the component then changes the interface - a backward compatible extension is achieved if functions are added to but not removed from this interface. One still has to be careful: what is syntactically an extension can still be semantically incompatible, as illustrated by the following example.

In one of our systems, the Real Time Kernel interface did not contain watchdog facilities. To add those, *IRealTimeKernel* was given an extra function to 'feed' the watchdog. The watchdog was *on* by default. This implies that existing applications that do not feed the watchdog are reset after a certain time has elapsed. The default should have been *off* to make the change upward compatible.

To manage changes to component interfaces properly, version numbers should be attached to the interface. A client can then be specified to require at least a certain version of that interface. Note that newer clients may not work together with servers that are too old. This is in fact a downward compatibility problem, as explained further in section 4.4.

Disadvantage of this approach is that it is not possible to make *incompatible* extensions to the component's interface without breaking existing clients. Some people increment the minor version number of an interface for compatible changes, and the major number for changes that are incompatible.

4.3.5 Immutable Interfaces

The second solution for achieving backward compatibility is to allow a component to provide multiple interfaces. If the interfaces are kept small, then in principle it is possible to regard interface definitions as immutable, i.e. once published, they never change anymore. Microsoft COM uses this approach. If interfaces are large, then immutability can probably not be achieved (see also Figure 22).

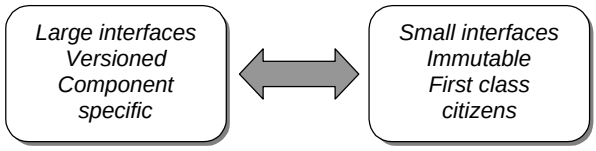

Figure 22. The interface granularity dilemma.

Microsoft COM allows component builders to realize backward compatibility by adding new interfaces while still supporting the old one. Old clients query for the old interface and will therefore still work, while new clients query for the new interface and can thus access the new functionality.

Interesting aspect of this approach is that it is possible to actually *change* the way a component is to be addressed, by adding an incompatible interface, without breaking down existing clients.

4.3.6 Client Detection by the Server

The design goal of Windows 95 was that Windows 3.11 programs were to be upward compatible. In practice, many Windows 3.11 programs do *not* run properly on Windows 95. One reason for this is that the programs make illegal use of the API (e.g., they still use memory *after* they have released it). To achieve some upward compatibility, Windows 95 recognizes such programs, and patches them on the fly when starting them.

4.3.7 Server Detection by the Client

Yet another solution is to let the client detect the version of the server and adapt it self accordingly. This is one of the approaches to achieve downward compatibility as described in section 4.4.6. Of course, this technique is not applicable if the new version of the server is not yet known at the time of creating the client. But it can be used when sufficient information is available with respect to the new version of the server. Note that if the new server is already available for a longer time, clients

are usually developed for the *new* server, and then face the *downward* compatibility problem instead of the upward compatibility problem.

4.4 Downward Compatibility

The second question we ask ourselves (see Figure 23) is: *can a new version of a client work with an old version of a server*? We actually recognize two cases here: the new version of the server already exists when the client is developed, or the new versions of the server and client are developed together.

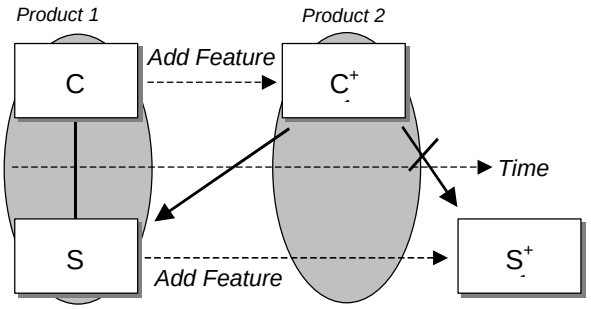

Figure 23. An example of required downward compatibility.

4.4.1 Relevance of Downward Compatibility

The relevance of downward compatibility may be less intuitive. State of practice is to ignore this problem. Let's think of some examples where it *is* useful.

If the new server is already available when the client is developed, it may still be desirable to be downward compatible with respect to the old server. Third parties that combine the client and server may choose for the old version of the server for qualitative (less buggy), economic (less expensive) or other (smaller footprint) reasons.

If the client and server are developed together, the following scenario can arise. Suppose that a new version of a client has *two* new features, one for which the new version of a server is needed, and one that also works with the old version of the server. The new server is not ready in time, or still buggy, and cannot be deployed in a product. But we do need the second feature. Can we use the new client with the old server?

4.4.2 Examples of Downward Compatibility

Suppose you haven't upgraded to Windows Millenium yet, but you want to use a new software application, released *after* W-ME. Can you use it with your old Windows 95, or do you have to upgrade your operating system as well? Most users would not appreciate the latter...

From a given version onwards, Windows 9x supports alpha blending in its graphic subsystem. Attempts to use this functionality on older versions of Windows results in strange artifacts. So, programs utilizing alpha blending should check the version of Windows.

Suppose you create a web site that should look stunning on IE5.5. Can you make it such that it still looks reasonable on older versions of IE?

4.4.3 Rely on Backward Compatibility

If the new version of the server is available at the time of building the client, the client may choose to only use the functionality as provided by the old server. This makes the client automatically work on both versions. This is not always a desirable solution, since features of the new server cannot be utilized.

4.4.4 The Multiple Implementation Technique

Another way of handling downward compatibility is to have specific client implementations for specific versions of the server. This technique is often used for drivers in Windows. The driver model is subtly different in subsequent versions of Windows, therefore many hardware manufacturers deliver specific drivers for specific versions of the operating system.

This is - of course - not an ideal solution. It puts a multiple maintenance problem at the client side, and it also burdens the creator of the system (in case of the Window driver example, the owner of the PC).

4.4.5 The Configuration Technique

Another technique is to specify the version of the underlying server as a configuration parameter to the client. This can be a compile-time parameter, which will work if all components are compiled into a product, but which results in the solution sketched in the previous section if components are deployed binary.

The disadvantage of this technique is still the burden that is put on the creator of the product combining the client and server.

4.4.6 The Version Checking Technique

Another technique is to let the client ask at run-time for the version number of the server. Depending on this number *and on knowledge about the server*, the client can then decide which functionality of the server it may use.

An example concerns web sites: it is quite common to start a web page with a query for the version of the underlying browser, and then have specific elements on the page, or even specific pages, for specific versions of the browser.

Disadvantage of the version checking technique is that knowledge about the server is built-in into the client. This hampers portability, as we will see in section 4.6.

4.4.7 The Capability Query Technique

The third and most powerful way of determining whether an underlying server implements a certain functionality is to explicitly ask the server for it. The query should be *function oriented*, and not *server version oriented*, as in the previous section. COM supports this from the ground up, using *QueryInterface* whenever functionality of a server is required.

If this technique is implemented properly, then ultimately, also the portability problem can be handled (see section 4.6).

4.5 Reusability

The third question we ask ourselves is: *can we reuse a server with a different client*? We suppose here that the server was not designed with the different client in mind.

4.5.1 Relevance of Reusability

This has been the goal of the reuse community for quite some time now. A more difficult reuse problem is actually the *portability* question as discussed in section 4.6, but let's first concentrate on what we call reusability in this paper.

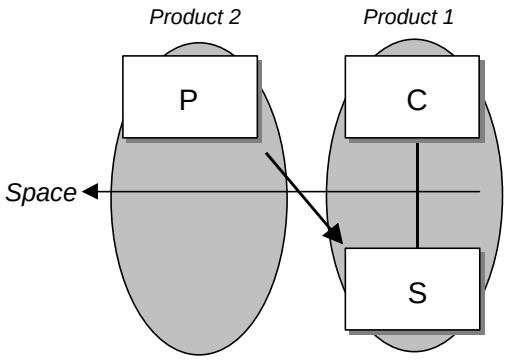

Figure 24. An example of reusability.

Figure 24 shows an example of reusability. A server S has been designed together with a client C. Can it now also be used together with a client P? The problem lies in the following dilemma (see also [103] section 4.1.8):

- To make software reusable, context specific decisions should not be taken in the software.
- To make software usable, a lot of decisions should be taken in the software.

Can we make software that is reusable and usable at the same time? We believe that the answer lies in parameterization.

4.5.2 Examples of Reusability

Windows is a platform and is therefore designed to be reusable, i.e. to create many applications on top of it.

The Internet Explorer is basically an ActiveX control that can be used for different purposes then just browsing the web. In fact, Microsoft uses it to browse through its new HTML Help format. The help pages are written in (dynamic) HTML. Seamless integration with help content on the web is also possible.

One can think of ample examples of desired reusability in our own (Philips) domain, the most urgent ones being the reuse of digital TV reception and decoding (from the set-top box domain) and reuse of storage facilities (tape, DVD, hard disk) in the classical analogue TV domain.

4.5.3 Configuration Interfaces

The most intuitive way to parameterize software is to provide one or more *configuration interfaces*. Such interfaces contain *Set* and *Get* functions through which properties of the software can be accessed. The properties influence the behavior of the software. The property mechanism makes the software reusable. If suitable default values are defined, the software can also be quite usable.

Visual Basic is *the* example where components (ActiveX controls) are usable and reusable at the same time. Controls usually have a large list of properties, some of which are set at design-time, some at run-time, and others will retain their default value. The Visual Basic environment in fact cleverly integrates design-time and run-time; to change parameters at design-time, controls also *live* at design-time.

The configuration approach has its disadvantages. Parameters can only be changed programmatically, and quite some code (and time) may be spent on initializing the system at run-time. Also, there is no means to remove redundant code, i.e. code that is not reachable under the configuration settings in a specific product.

4.5.4 Diversity Interfaces

We speak of *diversity interfaces* if the software 'takes action' to retrieve values of certain parameters from its environment. There are various implementations of this idea. The simplest one is the use of the *#define* in the C/C^{++} preprocessor. This allows fine-tuning of the software *before* being compiled, so that all kinds of compile time optimizations are possible.

A variation of this is the use of a *registry* that contains key-value pairs with settings for a variety of software. Microsoft uses this technique intensively. The Windows

Explorer can be tweaked using all kinds of registry settings, e.g. to add context menus. Microsoft Word can be convinced to perform live scrolling when moving the scroll bar[3]. Of course, it can be argued that such settings are persistent information of the tools, and that this information should only be changed through option menus of the tools, but state of practice is that the registry needs tweaking now and then.

4.5.5 Plug-In Functions

A common technique is to parameterize a piece of software over a function. A classical example is a sort algorithm that is parameterized over a function that compares two elements of the set to be sorted. This allows the function to be written in a polymorphic way, so that it can handle elements of different types.

Templates (like in C^{++}) allow parameterization over data (see the previous section), over functions (as described in this section) and over classes (as described in the next section).

Another frequent example of the use of plug-in functions is in components that can notify other components of certain events occurring. Such components are usually parameterized over functions that handle the notifications. Another term for the handler functions is 'call-back' functions.

4.5.6 Plug-In Components

The ultimate flexibility may be achieved by parameterization over *components*. When using such software, fine tuning then takes place in the form of providing *plug-in components* that may implement quite some functionality. Examples can be found in the architecture of medical equipment [91], where for instance different types of geometry are handled by plug-in components.

We need to be careful in our vocabulary here. There is quite some consensus nowadays that components are containers of classes (e.g. a DLL or an EXE), and that classes can be instantiated into objects (see [103]). Usually it is the *objects* that have interfaces and can interact with each other.

In this terminology, the parameter is rarely a component, usually a class, but sometimes it is an actual object.

As another example, the standard Windows Explorer allows for a number of such plug-ins, to be defined for all or specific subsets of files. They can bring up context menus, add property pages, display yellow pop-up windows, or change icons depending on the contents of the file.

---

[3] Set *LiveScrolling* to "1" in "HKEY_CURRENT_USER\ Software\ Microsoft\ Office\ 8.0\ Word\ Options"

4.6 Portability

The fourth and last question we ask ourselves is: *can we reuse an existing client on top of a different server*? The name portability is sometimes confusing, as we will see in section 4.6.6.

4.6.1 Relevance of Portability

Independent deployment of the fourth kind - *portability* - is the most difficult to achieve. Still, if one achieves it, it is also the most powerful, as it subsumes downward compatibility as a special case. State of practice is (again) to ignore the question, or to solve it by standardizing on a platform (see section 4.6.3). Figure 25 shows schematically how a client C that has been originally designed to cooperate with server S, now has to be used in combination with server Q.

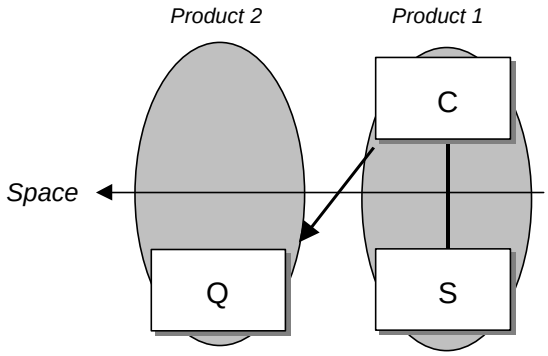

Figure 25. An example of portability.

4.6.2 Examples of Portability

Windows applications are rarely available on other platforms than Windows. In fact, it could be seen as a Microsoft strategy *not* to make their applications portable. There are some exceptions, however. Word is for instance also available on some Apple machines.

Can a web site that has been designed for Internet Explorer also be viewed with Netscape? Often, the solution sketched in section 4.6.5 is adopted.

There are ample examples of required portability in Philips again: for instance the reuse of Teletext and Electronic Program Guide (EPG) software, where the problem mainly lies in the porting of this software to the different infrastructures of the various products.

4.6.3 Platforms

The usual 'solution' is to *standardize* on a platform (cf. Windows). If everyone uses the same underlying platform, then we do not have a portability problem. There are various reasons why this approach does not work well in all cases.

First of all, no choice of any platform is eternal. You may for instance select a real time kernel (RTK), and then discover one day that the company has been bought by another company, and the RTK is being discontinued. Or, you might have selected the perfect RTK and then customers insist that you provide the software also on another RTK.

Secondly, platform versus application is only *one* division of the software, but the portability issue also arises *within* the platform and *within* the application.

4.6.4 Adaptation Layers

Mismatch between the required interface of a client and the provided interface of a server is often solved by inserting an adaptation layer. However, this is a patch, and not a fundamental solution. It is worrying how large a role adaptation layers play in current architectural discussions. Some people see them as a fact-of-life.

4.6.5 Server specific Clients

The solution that many web sites deploy for working well with Netscape *and* Internet Explorer is to query for the identity of the server, and then have specific pages for specific servers. This solution is quite similar to the one described in section 4.4.6

4.6.6 A Client Server Architecture

Another solution is to standardize on an interface or an interface suite between client and server. Any client that supports this interface suite can then be used with any server that requires it.

A good example of this is Microsoft's ActiveX compound document functionality (previously called OLE [17]). A large set of interfaces is defined that allows OLE containers to embed instances of OLE servers. Examples of OLE containers are Word, Excel and PowerPoint. The same applications are also examples of OLE servers.

The compound document functionality is successful with respect to the *servers*. It is relatively easy to create another server of which objects can be embedded in one of the OLE containers. Creating an OLE *container* is much more difficult. There are not many examples of OLE containers, so in our terminology, the *reuse* of OLE servers is limited.

As a final note, we admit that the word *portability* - one of our four aspects - does not optimally cover the client server architecture described here. When we talk about portability we usually mean the deployment of a client with a server that have *not* been designed under a single architecture. The next sections will elaborate on this.

4.6.7 Explicit Context Dependencies

The real solution to the portability problem might lie in the author's *Rule of Portability*:

> *80% of the software does not **need** more than 20% of the underlying software.*

Unfortunately, a variant of Murphy's law states:

> *Only 20% of the software does not **use** more than 20% of the underlying software.*

The only way to make sure that 80% of the software actually does not use more than 20% of the underlying software is to *make* all context dependencies explicit and to *manage* them explicitly. We believe that it is not sufficient to study link maps for context dependencies. Instead, software developers must make explicit choices in using certain servers, and software architects should monitor such explicit choices.

Interfaces to the environment are sometimes called *requires interfaces*. To really make them work, the build environment should not allow a build if functionality of the environment is used without having specified it explicitly.

4.6.8 Third Party Binding

One step further is explicit third party binding. Now it's no longer the client specifying which *server* is to be used, but only which *service*, and a third party (the product creator) selects a particular server and binds it to the client. The binding is done *late* in the production process, i.e. when the product is created, *after* the clients and servers have been built. But it can still be done *early* in the compile/link/run process (called *static* binding, and explained in the next section), or *late* (called *dynamic* binding, and explained in section 4.6.10).

4.6.9 Static Binding Techniques

The most common static binding technique is to use parameters in the configuration management system to select between component variants. Such parameters may for instance describe the underlying hardware platform or the product to be created. Disadvantage of this approach is that it hides architectural

information in the CM system, and that it is impossible to postpone the choice to run-time.

The technique deployed in Koala [72] is to use *#define* to map names of required functions to names of provided functions. In fact, Koala is more general since it also allows to postpone the binding to run-time when desired.

Yet another technique is to map names of required functions to names of provided functions in the linker, via symbol mapping. This technique also allows for postponing binding decisions until run-time, by inserting selector functions in between.

Note that in all cases, the conceptual level of indirection between the required and the provided function is mostly absent at run-time.

4.6.10 Dynamic Binding Techniques

Dynamic binding techniques rely on a level of indirection that is present at run-time. The simplest solution is to use *function pointers*; a technique often used to bind callback functions. To make the binding easier, function pointer *tables* can be used.

		<n> objects	polymorphic
static	f(x, y)	f(o, x, y)	
dynamic	*fp(x, y)	*fp(o, x, y)	o->f(o, x, y)
dynamic	t->f(x, y)	t->fp(o, x, y)	o->t->f(o,x,y)

Figure 26. Various kinds of binding.

Often, it is not a *function* or *class* that must be bound, but rather a specific *object* of a specific class. The solution is to add the object handle to the functions, usually as first parameter. Finally, if a specific object of *any* class is to be used (as long as it supports a certain function or interface), then a polymorphic solution has to be used, resulting in yet another level of indirection.

Microsoft's COM deploys polymorphic interface binding.

4.7 The Quality Dilemma

We have mostly discussed *technical* issues up to now. We shall now discuss one *quality* issue with respect to independent deployment (see Figure 27).

Suppose client C has been developed together with a server S, and the combination P = C + S has been tested and released as a product. Since then, a new version of the server, S^{+1}, has been released with some bug fixes and added features. Somewhat later, a new product P' has to be released where client C needs a minor

update, C'. Do you use the old S in the product, or the new S^{+1}? In the latter case, you profit from the fact that some bugs have been solved, but have other bugs been introduced when adding features to S?

Now suppose that the new product also requires a slight modification of S. Do you add this to S^{+1}, or do you create a new derivative of S, resulting in a branch S'? In the latter case, what happens if you want to create yet another product P" that requires a slight extension of C' and S'?

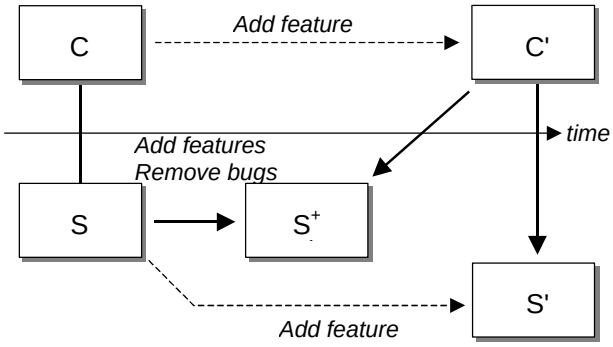

Figure 27. The quality dilemma.

The message may be clear: if the (short-term) advantages of using S^{+1} in P' do not outweigh the risks, branching will occur, and the development team of S will fall into a (long-term) multiple maintenance trap. Careful management is desired.

4.8 Concluding Remarks

4.8.1 Independent Deployment Revisited

We have discussed independent deployment of software in terms of four issues: upward compatibility, downward compatibility, reusability and portability. We have provided a number of techniques (see Figure 28) to deal with these issues, and have given examples. We do not claim that our inventory of techniques is complete.

Note that in practice, combinations of these issues occur, as software usually has multiple clients and uses multiple servers. We have discussed the issues in isolation to keep the discussion clear. Note also that some of the techniques address more than one issue. We have not tried to give the reverse mapping here.

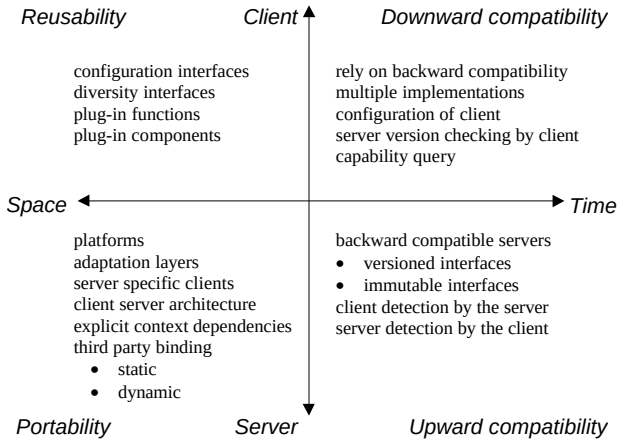

Figure 28. Summary of techniques

4.8.2 Related Work

Perry [89] also discusses compatibility, and defines the notions more formally than we found necessary to do for this paper. He concentrates on the use of configuration management and version control systems to manage compatibility, whereas we are interested in 'compatibility in the large', i.e. for product populations. See [76] for a more elaborate discussion on these topics.

Some examples have been drawn from Microsoft's COM as designed for OLE (see [17] chapter 1), where one of the design goals was to allow clients to work with servers even if they do not know of each other's existence. See [113] for a seminal paper on this topic.

4.8.3 Acknowledgements

I want to thank Hugh Maaskant and Chritiene Aarts for discussing and reviewing this paper.