

Software Deployment, Past, Present and Future

Alan Dearle



Alan Dearle is a Professor of Computer Science at St Andrews, Scotland's first University. His current research interests include programming languages, component deployment, operating systems, peer to peer systems (especially related to distributed storage), middleware and sensor networks. He previously held a Chair of Computer Science at The University of Stirling where he worked on Middleware and an exo-kernel operating system named Charm. This work followed on from the Grasshopper persistent operating system project which he co-founded with Professor John Rosenberg whilst an Associate Professor at The University of Adelaide. His PhD thesis work was conducted at The University of St Andrews under the supervision of Professor Ron Morrison. He was a co-designer and implementor of the persistent programming language Napier88 which supported strong typing, parametric polymorphism, a dynamically callable compiler and an integrated persistent run-time environment. He holds a PhD and BSc (Hons) both from the University of St Andrews.

Software Deployment, Past, Present and Future

Alan Dearle
School of Computer Science
University of St Andrews
St Andrews
Fife
Scotland
al@cs.st-and.ac.uk

Abstract

This paper examines the dimensions influencing the past and present and speculates on the future of software deployment. Software deployment is a post-production activity that is performed for or by the customer of a piece of software. Today's software often consists of a large number of components each offering and requiring services of other components. Such components are often deployed into distributed, heterogeneous environments adding to the complexity of software deployment. This paper sets out a standard terminology for the various deployment activities and the entities over which they operate. Six case studies of current deployment technologies are made to illustrate various approaches to the deployment problems. The paper then examines specific deployment issues in more detail before examining some of the future directions in which the field of deployment might take.

1. What is Software deployment?

Software deployment may be defined to be the processes between the acquisition and execution of software. This process is performed by a *software deployer* who is the agent that acquires software, prepares it for execution, and possibly executes the software [1]. Thus deployment is a post-production activity that is performed for or by the customer of a piece of software. It is at this point in time that all customer centric customization and configuration takes place. Software deployment may be considered to be a process consisting of a number of inter-related activities including the *release* of software at the end of the development cycle; the *configuration* of the software, the *installation* of software into the execution environment, and the *activation* of the software [2]. It also includes post installation activities including the *monitoring*, *deactivation*, *updating*, *reconfiguration*, *adaptation*, *redeploying* and *undeploying* of the software. We briefly expand on each of these activities in Section 1.2 below.

1.1. Concepts/Terminology

In this section we introduce common terminology and concepts which apply to many software deployment systems.

Most deployment systems incorporate the concept of a *component* which is defined in the UML2 specification [3] to be a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment. In [4] a component is defined to be a unit of composition with contractually specified interfaces and explicit context dependencies only. A component defines its behavior in terms of provided and required interfaces. In this context, an *assembly* is a set of interconnected components. An assembly can itself be viewed as a component made up of subcomponents and offering and requiring interfaces. This concept has been developed to its logical conclusion in the Fractal Component Model [5]. The required interfaces of the components in an assembly may be satisfied either by other components in the assembly or be required from the *environment* in which the assembly is deployed. The term resource is commonly used to refer to any artefact (both hardware, software and system artefacts) which a component requires in order to function. An *application* is simply a collection of components which performs some function. In order to *deploy* a component it must be instantiated, supplied with instances of components on which it depends and configured. A *version* of a component refers to time ordered revisions of a component or application and to platform-specific and/or functional variants [1].

Some systems provide the notion of a component *package* containing metadata and assemblies. The package may contain multiple implementations of components to satisfy the needs of different hardware and software environments. Thus when a package is deployed, the deployment system may need to choose the components that are most suitable for the environment. To do this appropriate and adequate

metadata must be included in the package to describe the components in it.

In OMG parlance the target environment is termed a *domain* and is comprised of nodes (computers), interconnects (network connections) and bridges (routes between interconnects). In many component models components are required to execute within a controlled environment known as a *container*. Containers serve many purposes including shielding components from the concrete environment on which they are hosted; providing lifetime management; enforcing policies on components; and providing mechanisms for the *discovery* and *binding* of other components in the environment.

Binding is the process by which a component obtains a reference to another component which it requires to operate correctly. Inter-component bindings may be made at a variety of times including when the components are built, when they are added to an assembly, when they are packaged, when they are deployed and when they are executed. Central to the binding task is how components are named and again a variety of mechanisms are in current use. In Unix environments, components are usually named using file system path names. In Windows environments components are often referred to using a globally unique identifier (GUID) which is used to index a system wide database – the *registry*. In Java environments a variety of mechanisms are used to resolve names including JNDI [6]. The most general naming scheme is found in the Web-services domain where components are referred to using a Uniform Resource Identifier (URI).

Another aspect to binding is the degree of type checking that is performed when names are resolved to values. In some environments, notably Unix, little type checking is performed when binding occurs, by contrast environments such as .Net and Web Services include sophisticated and expressive type systems and perform extensive checking.

1.2. The deployment lifecycle

Release is the interface between developers and the actors in the remainder of the software lifecycle. At the point of release the software is assembled into packages containing sufficient metadata to describe the resources on which it depends. *Installation* requires the software to be transferred to the customer and configured in preparation for activation. *Activation* is the process of starting the software executing or putting in place triggers that will execute the software at an appropriate time. This is sometimes achieved using graphical interfaces or with scripts or daemon processes. The opposite of activation is deactivation

and in many systems this is required prior to adaptation or reconfiguration where a piece of software must be passivated and rendered non-invocable. *Updating* is the process of changing a piece of installed software usually triggered by the release of a new version by the developers. Update is a special case of installation but may require installed software to be deactivated prior to update and reactivated after reconfiguration. Unlike updating and reconfiguration, *adaptation* is the process of modifying installed software in order to react to changes in the environment in which the software is installed. *Undeployment* is the process of removing the deployed software from the machine on which it was deployed. The process is also known as *deinstallation* of the software.

This software process requires specifications for:

- Packaging the software and associated metadata for delivery between the software producer and the deployer.
- Receiving and configuring the software into the deployer's environment *before* deployment decisions are made.
- Describing the facilities of the targeted execution environment.
- Planning how the software will be deployed onto the targeted distributed execution infrastructure.
- Performing the actual preparation of the application for execution, e.g., moving parts of the software to their location of execution.
- Launching, monitoring, and terminating the software.

The remainder of the paper is structured as follows, Section 2 presents six case studies that illustrate typical current approaches to deployment and highlights the differences between them. Section 3 examines some of the issues highlighted in Section 2 in more detail. Section 4 looks at the trends in deployment and the issues and challenges presented by those trends. Section 5 looks to the future and speculates on the future of deployment, Section 6 concludes.

2. Case Studies

In order to help understand the issues affecting software deployment, this section examines six technologies which include support for software deployment. Before exploring these technologies it is worth highlighting that looking at any one technology simplifies the problems of software deployment considerably since many of today's systems are heterogeneous with respect to hardware, the operating environment and the implementation technologies used to implement components and applications.

2.1. Java Beans

Enterprise JavaBeans (EJB) is a standard for building server-side components [7]. Java Beans were designed to simplify the development, deployment, and management of enterprise solutions. Java Beans are units of business logic contained within components that execute within containers. Once business logic is wrapped up in a Bean, it may be reused in a variety of contexts. The containers abstract the hosting environment and offer a variety of services including security, lifetime management and transactional database services in addition to low-level system aspects such as caching and memory pooling. Each Bean instance is implemented by (at least) a pair of objects and presents at least three different interfaces: the *home* interface used for lifecycle management purposes, the *remote* interface containing methods which may be called remotely by the client, and the *local* interface which defines the business methods callable by container-resident clients.

Java Beans must be packaged according to guidelines specified by Sun Microsystems. The standard bean packaging permits both management and deployment tools to be written which manipulate them. Java Beans are packaged in a standard Java JAR file along with a XML deployment descriptor file describing properties pertinent to the bean. This packaging may contain one or more Java Beans. However, once multiple Beans are packaged together they may not be separately managed since the containers manage them as a single unit. The deployment descriptor typically contains information about the bean's transactional, security and persistence requirements along with any bean specific properties. The manifest for the JAR file may contain a *Depends-on* attribute which specifies the components upon which the packaged component(s) depend. However, this is expressed in terms of the naming service name for the referenced components which is not unique.

The Bean lifecycle, as described by Sun, consists of four phases: development, deployment, service availability and undeployment. In the first phase the bean is written and packaged in a JAR file as described above. In the second phase an application assembler may adjust properties in the bean's deployment descriptor such as security attributes, persistence mechanisms, transactional properties and any bean specific properties that require tailoring to the deployment environment. This phase ends with the bean package being loaded into an appropriate directory where it may be activated by the container hosting it.

Support is especially lacking for the maintenance of inter-bean bindings in the face of updates. As described above, references to dependent beans are in terms of their non-unique name service name. Thus if the same name is used for two different beans those beans must be manually reloaded to ensure that bindings are up-to-date. This problem and a solution to it is described in more detail in [8].

To sum up, Enterprise Java Beans are relatively fine grained and language dependent. The EJB solution isolates the Bean from its environment by providing a standard container interface. This requires Beans to be written in a manner that it is compliant with the interfaces specified in the container interface. Enterprise Java Beans does not have any notion of remote installation of components. There are also problems with respect to the way in which Beans are named.

2.2. Linux

The most common method of deploying software for Linux is the Red Hat Package Manager (RPM). The manager supports a number of operations including installation, querying, verification, update and deletion of packages. These operations are supported by a database containing details of the packages that have been installed on a particular Linux installation. A RPM package typically contains binary executables, along with configuration files and documentation. Since binaries are contained in the packages, implicit dependencies exist between the packages and the host operating system and architecture. This is addressed by using a standard set of C libraries and by annotating packages with the architecture for which they have been compiled (i386 etc.). It is however also possible to create source packages containing source code which avoid this complexity but creates additional dependencies on build tools (such as *make* and *gcc* etc.).

Every RPM package is labelled with a package label which contains the name of the software, its version, the release (used to indicate the target Linux distribution) and the target architecture. This label is not contained within the package, instead being used to name the files containing the packages. Every RPM package contains four sections called the *lead*, the *signature*, the *header* and the *archive* [9]. The lead is used by Unix operations such as the command *file*. Although once used by RPM, the information in the lead has been superseded by the header due to inflexibility. Every RPM package contains one or more header structures which are represented as an indexed set of entries, each containing information on some datum. The signature contains cryptographic

information that may be used to verify the integrity, and in some cases, the authenticity of the header and archive contained in the package. The archive contains a Gzipped collection of files that comprise the package.

RPM files may also contain a number of scripts written using standard Unix scripting languages. The scripts are organized into sets responsible for building software, installation and erasure of software and verification. The building scripts are responsible for unpacking source files, building the software, installing and removing the software and performing post-installation tidying up. The installation and erasure scripts run at four times: before and after installation and before and after erasure. The post-installation script is responsible for verifying that the installation was correct; unlike the .Net installation managers (described below) there is no atomicity and roll back should errors be detected and this complexity is the responsibility of the script writer.

Higher level toolsets have been built using RPM, notably the Yellowdog Updater Modified (YUM). YUM is designed to determine inter-package dependencies and automate the installation of packages. It also provides the ability to manage collections of machines, such as in a server farm or University laboratory, without having to manually configure each machine using RPM.

To conclude, the Redhat Package Manager is widely used in the real world. It is a coarse-grain, language independent, operating system dependent approach. Its major failing is that not all dependencies are explicitly modelled and those that are, are not modelled in terms of packages but in terms of their contents.

2.3. .Net

In the .Net framework, the basic unit of deployment and versioning is the *Assembly*. The identity of each Assembly is encoded in its strong name which contains its simple text name, a four part version number, and culture information, together with a public key and a digital signature. Using a strong name for an Assembly ensures that the name is globally unique. In addition to uniqueness, the use of strong names ensures the integrity of version numbers and provides users with the assurance that Assemblies have not been tampered with. Assemblies contain a manifest describing the contents of the Assembly, type metadata, an optional set of resources and one or more CIL code modules. The metadata in the manifest describes the classes provided by the Assembly, versioning information, dependencies on other Assemblies and modules and security attributes. Importantly this metadata provides the necessary information for the .Net framework to

ensure type safety (supported by the Common Type System) and security. The type metadata describes the types implemented by the Common Interface Language (CIL) code contained in the Assembly. Several different types of Assemblies may be created in the .Net framework including Static, Dynamic, Private and Shared Assemblies. Here we will focus on Shared Assemblies which may be used by multiple applications and are therefore the most interesting.

Shared Assemblies typically reside in a per-machine data structure known as the Global Assembly Cache (GAC) which is a machine-wide store for the Assemblies used by more than one application. In Windows Assemblies are typically put in the GAC by one of the Windows installation programs. Multiple versions of an Assembly may exist contemporaneously within a GAC, with each being differentiated via their strong names. By default, applications bind to Assemblies in the GAC using the strong names (including version numbers) specified in the calling Assembly's manifest. This prevents the installation of a new version of a component from harming other currently installed applications and components. However, it is desirable for components to be updated when bugs are fixed and for versions to be discarded if, for example, security problems are discovered. To address this need the .Net framework provides version policies which are expressed in XML and may be specified in an application-specific, machine-specific or publisher specific manner. Clearly, the installation of a new (faulty) component can stop an existing application from working therefore the framework permits every application to bypass the policy specified by the publisher.

The Visual Studio .Net IDE contains tools to create Windows Installer files to install, update and repair .Net applications and components. Each installer may be digitally signed so that it may be authenticated. Such installer files are dependent upon the Windows Installer which is an operating system service which maintains a database of information on every installed application. This database records dependencies between installed components to support removal and repair. Windows supports four types of setup projects: standard, web-based, merge projects and CAB projects. The first two of these are compatible with the installation service, merge projects are used to package components rather than applications and CAB projects are used to package downloadable ActiveX controls. The Visual Studio environment includes support to detect dependencies on other .Net components. However, dependencies on legacy components such as COM components must be added manually to the project and hence to the installer and Assemblies.

Each standard setup project contains a setup routine responsible for performing installation and is capable of rolling the system back to the state in which it was prior to installation should the installation fail or if the installation is cancelled by the user. Installers can also contain custom actions which may be in the form of scripts, executable code or Assemblies, which are executed after the installer scripts have completed. The .Net framework supports conditional installation permitting installations to be customized according to local needs. This is supported via properties and launch conditions which permit installations to be customized according to which operating system is running, which files are present, what registry variables have been set, etc.

To summarise, like RPM, the .Net framework is widely used in the real world. It is also a coarse-grain, operating system dependent approach although a (high) degree of language neutrality is provided via the CIL. The tool support provided by Visual Studio is an example of the growing trend of development environments reaching out to the domain of deployment and lifecycle-management.

2.4. OMG, CCM and D & C

The OMG Deployment and Configuration specification attempts to “define the mechanisms by which component-based distributed applications are deployed” [1]. This specification is a replacement for the original Packaging and Deployment specification and XML DTD defined in the specification of the Corba Component Model V3 [10] (This specification has been superseded by version 4). The OMG process model consists of 5 steps: installation, configuration, planning, preparation and launch. Much of the specification focuses on abstraction over the environment. This is achieved by the definition of a Platform Independent Model (PIM) which is not only platform independent but also independent of middleware, programming languages and data formats (such as XML DTDs etc.). In the OMG PIM, a component has an interface containing operations, attributes, and ports which may be connected to other components. Component packages contain multiple alternative implementations of a component, each with different properties and suitable, for example, for deployment on different operating systems. Packages may be installed into Repositories where they may be configured. Each component may be either monolithic or be composed of sub-components and termed an *Assembly*. Within each Assembly, each sub-component is described using a *SubComponentInstantiation-Description* which either describes how to construct the component from its sub-components or describes a

suitably typed implementation. Packages may also contain *ComponentPackageReferences* which describe unbound references to required components and their type.

Monolithic components are described using a *MonolithicImplementationDescription* which contains information about what parameters must be passed to the target environment in order to instantiate the component and the requirements the component has of that environment. It also contains a description of the implementation artefact which is a complete concrete implementation of the component. This description includes an optional UUID, the most specific type of the component, the ports it offers, its properties and optionally configuration properties.

The PIM is divided into two independent dimensions each of which have a number of different aspects. The first dimension is known as the “Data versus Management/Runtime dimension” and is concerned with (a) the data model describing software artefacts and (b) the runtime management of those artefacts. The second dimension is based on the deployment process and is concerned with (a) the nature of the components; (b) the nature of the target environment and (c) how the software will execute in the target environment. Combining the data model aspect of dimension 1 with the three aspects of dimension 2 yields the *package descriptions* for components, the data model for the *domain* in which components are deployed and the *deployment plan* with which components are configured and connected on the target. Each of these models is described in UML2. Taking the second aspect of dimension 1 along with the nature of the components yields the *Component Management Model* which describes the interface to a repository manager for storing components. The second aspect describes a *Target Manager* for managing domains. Combining the last aspect of dimension 2 with the management dimension yields the *Execution Manager* which describes how to prepare components for execution, how to launch them using an *Application Manager* and how to manage their lifecycles using a *Node Manager*.

The 5 steps of installation, configuration, planning, preparation and launch map onto the three aspects of dimension 2 in the following manner. The first two steps interact with the repository manager. An installation tool supplies details to the repository manager which stores packages prior to being configured by a configuration tool. Planning relates to the target environment. A planning tool creates plans by interacting with both the Repository Manager and the Target Manager. The planning tool is responsible for the creation of a *Deployment Plan* which can be used by the Execution Manager to create a factory

object for the application. This plan is essentially isomorphic to the Assembly for the application with all the Assemblies recursively replaced by chosen concrete monolithic implementations. In the preparation phase, the plan is traversed and an Application factory is created by the Execution Manager for each component from which it may be instantiated. Next, the Node Manager(s) interact with the Application manager(s) to execute the application but not run it. This step returns references to ports provided by the applications. Finally, in order to start the applications running, node managers inject control into them along with the ports supplied in the previous step. In a distributed deployment, node managers are only responsible for the components on the nodes which they manage.

The PIM maps onto a concrete platform specific model using the CORBA Component Model (CCM) [10] in a number of transformation stages that ultimately yield Corba IDL or XML schema definitions. The CCM extends the distributed object computing model provided by CORBA with the concept of components. In CCM a component is an entity exposing a set of attributes and a set of ports, each of which contains *facets* defining a synchronous CORBA interface, *receptacles* providing connections to facets provided by other components, and asynchronous event sources and sinks. CCM components execute within containers called *executors* which provide the traditional container services. Many of the concepts from the PIM map directly onto the CCM without change. For example, the Node Manager uses the event consumer and facet ports provided by CCM components as interface providers, and the other ports as required interfaces.

The OMG D & C specification is perhaps the most complete attempt to define a deployment and configuration standard to date. Concerns surrounding the specification include its size and complexity, and its dependence on the underlying CORBA standards that it leverages.

2.5. Service-oriented Computing Paradigm

Using the Service-oriented computing paradigm, the service provider is responsible for the deployment and registration of the services offered. Those services may be provided using a variety of technologies including container based technologies such as Apache Axis [11] or Web-Services infrastructure such as IIS [12]. Thus the deployment of individual services offers many of the same challenges and involves similar processes and lifecycles as the other technologies discussed in this section. However, if one considers applications created from Web-Service components, the service-oriented

model has much to offer. In the Web-Service model, all services are identified using a collection of endpoints each of which specify a network address represented as a URI and a fully specified interface binding [13]. The interface binding specifies the protocol and/or data format for messages sent to that service. The service itself is specified in WSDL [14] which is language, architecture and machine independent and is extensible. Thus the Web-Service model presents an abstraction layer in which software may be written in different languages, run on different hardware and operating systems. The Web-Service model may be interpreted as a pure component model in which each component may have dependencies on others but only in terms of its WSDL interface and service endpoint. Furthermore, the individual services may be offered by different service providers and run in different protection regimes.

2.6. Virtualisation

Much of the complexity in the deployment process described above stems from interactions between the product being installed, the environment and the execution policy constraints. One approach to avoiding this complexity is to abstract over these inter-dependencies; this is the approach followed in .Net, Java Beans and in the Web-Services approach. Another approach is to obviate these problems entirely by creating perfect custom environments into which applications and components may be installed. Whilst in the past this might have seemed a Utopian dream, it may now be achieved using virtualisation.

Virtualisation is a new trend in computing which is likely to have a large impact on software deployment. In a traditional operating system, a single operating system runs on a single piece of hardware offering a shared uniform environment to the programs which execute on that platform. By contrast, in a virtual environment a virtualisation layer running on a single piece of hardware offers multiple virtual hardware emulations on which multiple instances of potentially heterogeneous operating systems may be hosted. Thus multiple incarnations of WindowsXP, Linux and Windows 2000 could co-exist and run applications on a single physical piece of hardware. This technique is known as partitioning.

Using virtualisation software such as VMWare [15], an operating system along with its applications may be encapsulated into a single virtual machine. Typically such virtual machines are encoded in a single image file containing what was previously considered to be a server (CPU, disk, network interface, file system and applications). Such image files may be loaded into a

virtual environment and executed by the virtualisation software.

The ability to manage an operating system and an application as a single entity has profound implications for the deployment process. Schumate [16] describes the deployment process in a virtual environment as consisting of 5 steps: creation, generalization, testing, distribution and update. The first step involves creating a baseline image containing an operating system, data and application components. The baseline image is created on physical hardware and a copy of the software image (a file) is created using a tool such as Symantec Ghost. This copy is known as the *master image*. Next, the master image is customised according to the particular needs of the user. This image may be tested prior to distribution. In the distribution step the image is sent to the hosts which will execute the VM images. This is typically achieved using network boot technologies such as *Pre-boot Execution Environment* (PXE) or by distributing either physical CDs or ISO image files. Whenever an application or operating system needs to be updated, new images may be created from the master image or any of its derivatives. An advantage of this approach is that these images may be tested prior to (re) distribution without concern for hardware or software dependencies since the image represents an entire target environment.

2.6. Grid Services

Using the Service Oriented Approach, services are provided by providers who typically own or manage a collection of machines organised in clusters. The Grid Services deployment problem is significantly more complex since end-users (typically scientists) require the ability to deploy across institutional boundaries and different protection domains [17]. The current Globus Toolkit [18] differentiates between applications and services. Applications, typically monolithic Fortran programs, may be installed using the Grid Services infrastructure but services may not.

3. Deployment issues

3.1. Binding again

In the discussion in Section 1.1, inter-component binding was discussed. Components may be bound to other components at a variety of times:

1. at compile/link time by development environments,
2. at assembly creation time by a packager,
3. at configuration time pre run-time,
4. at run time.

There are subtle implications in choosing each of these binding times. In the first case, in a traditional programming environment, no extant components exist at compile/link time and the bindings that are created are between pure code fragments. When a name is used in the program it may only denote a pure code fragment such as a class. This creates tight coupling between classes at compile/link time. When this paradigm is used, all the code and the data objects that are created by the code will co-exist in the same addressing environment be it an address space or container.

In the Web-Services model, a URI may be used to generate a proxy for a, possibly stateful, component that exists at compile-time. This permits static code to bind to extant components early in the development lifecycle in a manner reminiscent of early Lisp environments [19], Smaltalk-80 [20] and Hyperprogramming [21]. Binding between code and extant entities is also supported in the Windows COM+ environment in which a GUID may be used to bind to a variety of entities.

When inter component references are resolved at assembly time, some language mechanism must be used to abstract over the names used in the program and the concrete instances provided in the assembly. Recently, this has been achieved using the factory pattern [22] which supports exactly this requirement (and is used in OMG D & C). Again this involves the binding of pure code fragments rather than possibly stateful components.

A rich variety of bindings may be created for references that are resolved at pre-run time, that is after components have been installed and instantiated. These include references to pure code and other locally installed components in the file system, references to other components installed in the same container or address space and remote references to extant code, data and components. A similar set of binding possibilities are also available at run time.

At run-time, the context in which these bindings are resolved is determined by the container or run-time environment. Some environments permit only local components to be referenced whereas others permit arbitrary components to be addressed. The COM+ environment is perhaps the most general in that it contains mechanisms for both dynamic and static binding and permits components to be instantiated in local or remote address spaces.

3.2. Containers and run-time environments

The J2EE and .Net environments have demonstrated the ability to abstract over local environments (including hardware and operating

systems) through the use of containers and machine independent run-time environments. Both these technologies isolate components from complexity at the cost of technology buy-in. In both cases the developer has no choice but to develop application components in a particular language (or language family in the case of .Net) and also write applications that obey the syntactic and semantic conventions imposed by the host environment be it a container or run-time environment. In this respect the introduction of containers and support environments do little to alleviate the hardware and operating system dependencies. Instead, the complexities of interacting with the hardware and the operating system have been traded with those of the container.

By contrast, the virtualisation approach permits each application to run in an environment that is perfectly suited to it. This includes: the hardware, the operating system, the set of data files installed in the file system; and other components and applications that are locally required.

3.3. Light Weight Containers and Inversion of Control

In Section 3.1, the problems of technology buy-in with container technologies is described. The concept of *lightweight containers* has emerged as another approach to avoiding this buy-in. Lightweight containers such as Spring [23] and PicoContainer [24] have been developed to make it easier to deploy and configure J2EE applications. Both of these systems utilize a technique known as *dependency injection* (as does OMG D&C as described above) [25].

Dependency injection or *inversion of control* (IOC) is a design pattern which is sometimes known as the “don’t call me, I’ll call you” pattern. The principle is that explicit dependencies on other components and the container are removed from the code. In many environments, components are responsible for finding the components which they require in order to run. This is sometimes known as the *service locator* pattern. Using IOC, instead of a component calling a run-time infrastructure to bind to other components, the infrastructure calls the component and supplies it with whatever resources are required for that component to execute. In Java Beans, Beans use the service locator pattern to call into the container to locate some component X they require. By contrast, in Spring, the environment would detect that the component required X and supply it.

Clearly the IOC pattern is much better suited to the deployment lifecycle than the service locator pattern. In both Spring and PicoContainer, plain old Java objects (POJOs) may be written and wired in this

manner by the run time environment. In Spring three types of IOC exist: *constructor injection* in which required components are supplied when a component is constructed; *setter injection* in which setter methods are provided to supply required components; and *interface injection* in which, for example, a Java interface is used to specify a contract between a component and a container or another component.

Clear parallels exist between the IOC pattern and the deployment activities described above and the IOC pattern might seem to be the perfect solution to the problems of deployment and configuration. In both IOC and in deployment models such as the OMG D & C model, components have unresolved references to other components that are resolved by the infrastructure. In both models source code is freed from the complexities of having to bind to dependent components (with that responsibility devolved to other software). However, in lightweight frameworks such as Spring, the binding complexities that have been removed from the source code reappear in the form of a myriad of XML configuration files that specify inter-component bindings. Another failing of lightweight container models is that they have not to date addressed inter address space or inter machine component bindings.

3.4. IOC, Change Management, Reflection and Introspection

The IOC paradigm is applicable throughout the software lifecycle. It may be used in a programming context to prevent coupling between components under development. It may be used in packaging where some components are bound together to form Assemblies. It is also applicable in the deployment phase and its utility can be seen in the OMG D & C specification where the node manager injects control along with appropriate ports of other components into the components for which it is responsible. The IOC paradigm may also be used at run-time to dynamically unbind and rebind components. This functionality is demonstrated in the Fractal Component Model [5].

The Fractal Model is an extensible, programming language neutral component model which aims to “reduce design, deployment and maintenance costs of software”. The first main contribution of this model is a recursive data model in which components are composed of other components (hence the name Fractal). This data model is supported by a strongly-typed XML based Architectural Description Language (ADL) called Fractal ADL. This is not unlike other component models such as the OMG model described above. The second contribution of the model is that it is reflective; that is, it supports the ability to introspect

components and intercept calls. The Fractal Model draws heavily on the Aspect Oriented Programming (AOP) principle of *separation of concerns* [26]. Fractal components are each made up of a number of sub-components. The Fractal development methodology entails identifying services each of which is implemented by a component. The second step requires the dependencies between components and component hierarchies to be identified.

Each Fractal component contains a finite collection of subcomponents comprising its *content* together with a *controller* responsible for the composition of the components within it and the components' behaviour. In the Fractal Model, and arguably in all component models, components contain two kinds of references: internal references to sub-components and external references to other components. In the Fractal Model these are termed *internal* and *external* references. In order to reconfigure components it must be possible to access and update both of these references types. One way in which this may be achieved is using introspection and reflection. Using, for example, the Java reflection API, components may be decomposed and introspected and their (recursive) structure discovered. Alternatively, as in the Fractal Model, distinguished interfaces may be provided by components for the purpose of component introspection. The advantage of providing distinguished interfaces is that semantic consistency may be enforced in the face of change. Semantic checks include ensuring that components are consistently wired; that components are stopped and restarted before changes are made precluding internal inconsistencies; and that internal references are not wired to external components. Of particular importance is to ensure that bindings are consistently updated. Consider a sub-component *sc1* of component *C* which is referenced by sub-components *sc2* and *sc3*. If *sc1* is replaced with a new component *sc4*, to maintain semantic consistency both *sc2* and *sc3* must both refer to the new component *sc4*. In order to maintain such consistency components may offer a *BindingController* interface which permits changes to be made to bindings; a *ContentController* interface which permit sub-components to be added and removed from a component; and a *LifecycleController* which permits components to be started and stopped.

3.5. Maintaining metadata throughout the lifecycle

In the software engineering community many believe that it is critically important to retain design decisions throughout the software lifecycle. Such metadata may include descriptions of the components

(their interfaces etc.), the data, the source code (including languages used, revisions made, version history etc.), the workflows and the relationships between all of these. A new field of Application Lifecycle Management (ALM) has developed to address the management of software from acquisition through deployment and evolution to retirement. This clearly overlaps with the scope of deployment as described above. The overarching idea behind ALM is that a software architecture should exist which documents all the inputs to the software including its functional and non-functional requirements. The OMG have advocated the UML based Model Driven Architecture (MDA) approach [27] to address these needs. Using the MDA, software architectures are modelled as platform independent UML designs; these models may be progressively transformed and refined into an architectural framework. The underlying principle behind MDA is to have specialised modelling techniques for particular domains rather than having a single monolithic modelling framework. Different aspects may be modelled at different levels of abstraction using domain specific languages in a fractal approach to modelling. To unify these different languages a meta-model architecture is defined known as Meta Object Facility (MOF) which supports the interchange of modelling artefacts. MDA and the more generic Model Driven Development approach are both examples of a wider field of Model Driven Engineering (MDE). In MDE models (such as those defined in UML) are the primary software artefacts that are used in design, through implementation to deployment.

To store such metadata requires metadata repositories which in turn require schemata for describing metadata and tools for extracting metadata and keeping it in correspondence with the deployed system. Tools are beginning to appear that support this approach such as Borland Together 2006 and Eclipse MDDi. These tools permit models and code to be developed in which the code and the models are kept in step with each other. Although these tools are still mostly focussed on component development rather than the entire lifecycle; there is a clear trend of such tools reaching out from the development arena into the deployment and run-time environments.

4. Trends, Issues and Challenges

So far, we have examined the general issues surrounding software deployment and looked at the state of the art in deployment systems. We have also looked at some of the current issues relating to deployment and how they impact on the deployment process. In this section we examine some of barriers to

development in the field of deployment and how they might be tackled. The issues examined in this section are:

- Grain, of components containers and environments,
- Distributed deployment,
- Middleware,
- Adaptation and autonomies, and
- Architectural Specification.

4.1. Grain, of components containers and environments

There is currently little agreement over an appropriate granularity of either software components or the environment in which they are deployed. The grain of components varies enormously. In some systems (Java Beans for example) components are relatively fine grained programming language objects (or small collections of them). In other systems, notably Linux, the components are large grain binary executable file system objects. These two examples also highlight other differences – in one case the objects are programming language dependent typed objects, in the other they are language independent, un-typed binary objects. Grain is also an issue in terms of the dependencies between components. For example, .Net components check for dependencies in the Windows registry; such dependencies can potentially be at a very fine granularity - even down to a single byte. By contrast, in the SOA paradigm the grain of components tends to be relatively large (if it were not it would degenerate into a distributed object model). Similarly there is little agreement about the appropriate grain of packaged assemblies of components. Windows installers for example tend to be relatively monolithic and give the user little or no scope for deciding which of the packaged components to install, repair or uninstall.

The grain of the environment into which components are installed is also extremely variable. Installation into a narrow domain is supported in the Java Beans environment where only those services supported by a particular server instance are affected. Some components are installed into a machine wide domain. A good example of this is installation in Linux using RPM. An interesting hybrid described above is where some virtualisation technology is used. For example, using VMWare, the installation of components into a VM image is equivalent to the machine wide installation that might be performed using RPM or Windows installers. However, since the components are being installed into a VM image, they may have a perfect environment constructed for them. The VM

image creation process becomes akin to the creation of an assembly as described above. The difference between these assemblies and others is that the VM image is a totally self-contained and (virtually) machine independent executable image.

Once a VM image has been created, it is like an assembly; it may be replicated and concurrently executed in multiple sites. The images may also be updated in situ or a new master image may be updated and disseminated. A major advantage of this approach is that an assembly may be both black box and white box tested at the time it is packaged and the customer will have a high degree of confidence that it will operate correctly once it is deployed, subject to external dependencies. This brings us back to the subject of how the contract between the assembly and the external world is expressed.

The VM approach offers hope of simplifying the deployment cycle following assembly (VM image creation). There are however costs and complexities with this approach. The first is the issue of grain and co-location. The virtualisation approach is appealing since it permits applications to be totally isolated from each other. Each application can run in its own hardware protected sandboxed environment which is perfectly suited to it. However, there may be performance implications to running a collection of VMs as apposed to a collection of traditional processes under an operating system.

Using the VM approach, each application is installed within a potentially perfect self-contained environment. However, such applications will still require configuration, for example to supply them with the addresses of external database servers, Web-Services that they might utilise, local registries etc. Another way of looking at this activity is that it is forming bindings between the application(s) running within the VM and applications and services running externally. This requires some form of injection of control. It also requires meta-information to be supplied to the configuration management system to enable it to perform the configuration – for example, is the information passed in using VM invocation parameters (akin to constructor injection) or via Web-Services, RMI etc. This takes us back to the issues of modelling and meta-modelling.

4.2. Distributed Deployment

The issue of distributed deployment has been discussed briefly in the Grid section above. Distributed deployment brings with it a host of technical issues that must be overcome. The primary barrier to distributed deployment is the lack of uniform security and trust models. As discussed by Brebner [17],

attempts to deploy Grid Services in a UK wide experiment were fraught with difficulties with respect to certificate management. The security mechanisms must be sufficiently powerful to prevent malicious parties from deploying and executing harmful agents, and prevent deployed components from interfering with each other, either accidentally or maliciously. This last point, that of logical isolation, is a particular pragmatic difficulty since there is a tension between sharing resources and isolation to ensure correct behaviour of both the deployed components and the underlying infrastructure.

In [28] we describe an architecture, called *Cingal*, which permitted code to be safely deployed on third-party machines. The system could deploy self-contained assemblies which contained closures of code and data and were called *bundles*. The bundles were signed, permitting their authenticity to be established by a small run-time kernel on each node, prior to being deployed and executed on each node to which they were sent. Post deployment, a wiring bundle was sent to each node to perform node-specific configuration and to establish inter-component inter-machine bindings. In *Cingal* the sandboxes were implemented as Java Virtual machines with the attendant undesirable container buy-in requirements described above. Using a virtual machine approach and using technologies such as Xen [29] and VMWare makes it possible to deploy code on a public infrastructure and still maintain application isolation. This approach motivated the Xenoserver project [30] and led to the construction of Xen. This approach is also now being followed commercially, for example, in the Tivoli Software Installation Service [31] and in the Amazon Web Services project [32].

Being able to deploy components and applications on distributed third party servers necessitates the ability to create and maintain inter-component bindings and monitor the running components. As described above, this task may be performed using light-weight frameworks and the inversion of control pattern. Currently, no distributed inversion of control frameworks exist; systems such as Pico-container and Spring focus on IOC in a single address space environment. The next logical step in this development is the development of distributed IOC frameworks.

What would a distributed IOC framework look like? Much of the functionality would be like the IOC environments that already exist. Components would remain the same – methods would be provided to manage inter-component bindings. However the fractal pattern would need extending to permit such bindings to be made between local components, between local address spaces (containers), and between machines. Such a model can be seen as an extension of the Fractal

Model where address spaces and the nodes hosting them become first class entities in the model. This requires the ability to expose bindings or interfaces to update at each of the three levels – intra-container, inter-container and inter-node. One way of achieving this is to expose binding manipulation interfaces in components and applications. The lack of ability to expose applications and components was one of the major criticisms by Brebner [17] of the Grid Services Model.

The ability to expose binding manipulation interfaces requires support at a number of levels. Middleware support is needed to permit bindings to be exposed and manipulated. Distributed IOC frameworks are required to permit distributed component based applications to be assembled and maintained. Lastly, architectural support (including Architectural Description Languages and meta-models) is required to describe the distributed architectural components.

4.3. Middleware

By definition, using the Inversion of Control pattern, control needs to be injected into components from the outside. If the Fractal/IOC model is extended to include address spaces and machines, mechanisms are required to support such injection and consequently, middleware support is required. Earlier it was argued that forcing applications and components to comply with the conventions of particular container architectures was a bad thing since it created an accidental problem of technology buy-in. The same technology traps should be avoided with respect to inversion of control. There is no point in having a distributed IOC model which locks users into particular programming languages or Middleware technologies.

In the RAFDA project [33] we have built middleware which permits arbitrary application components to be exposed as Web-Services. Using this technology, an arbitrary application component can be dynamically exposed irrespective of its type. This is in contrast to most other middleware technologies which require decisions early in the design cycle about which application components may participate in inter-address space activities. Using flexible middleware such as RAFDA, interfaces that permit bindings to be manipulated may be trivially added to applications and components without changing any existing source or executable code.

In addition to binding support, other middleware support is needed to make a distributed IOC framework viable. Such support includes event architectures to provide distributed asynchronous event busses, distributed registries and name services for

locating components, and the ability to instantiate components on remote nodes.

Event distribution architectures are required for two independent but equally important purposes. They are needed at application level to permit application components to communicate with each other. However, they are also needed by the deployment infrastructure to monitor the health and efficacy of application components. This leads naturally to the subject of adaptation and autonomies.

4.4. Adaptation and Autonomics

Installed software must evolve to address changes in both the environment in which it operates and the requirements placed upon it. Kephart [34] has proposed an autonomic lifecycle comprising a monitor, analyse, plan, and execute loop in which entities are monitored by collecting events which are analysed and corrective action is taken if required. To make such a process viable, a knowledge base is consulted by the components implementing the autonomic monitoring architecture. In the context of a distributed application, the corrective action and its causes are potentially extremely complex. The problems that might occur include applications and components not being able to satisfy their quality of service objectives, loss of network connectivity, over-demand for network bandwidth, failure of components or applications, failure of computational resources and unbalanced resource utilisation. The corrective actions for such problems may include the introduction of new servers, the deployment of new applications or components, changing the network topology and moving components between servers.

To enable such autonomic management requires a number of technical obstacles to be overcome including: the architectural specification of applications (described below), instrumenting applications or execution environments to record semantically meaningful events, the distribution of application events to where they are analysed, deciding what actions to take in response to those events and infrastructure to manage the individual architectural elements.

The problem of instrumenting applications to record semantically meaningful events is akin to the problem described above with respect to the establishment and maintenance of inter-component, inter-machine bindings. Pieces of reporting functionality must either be installed in components a priori or be dynamically added to them. Forcing applications and components to adopt particular frameworks is unlikely to be successful in a heterogeneous business environment. Mechanisms that allow monitoring probes to be added

to software after it has been written and potentially dynamically on demand will be needed.

The encoding of the information necessary for the autonomic management of applications places yet more burden on architectural specification languages. This additional information includes that necessary to rebind application components, to express which components may be shared and those that may not, the domains in which components and applications may be executed and quality of service information. It is likely that such concerns will need to be divided into separate architectural meta-aspects as proposed by the OMG in order to keep complexity under control.

If we assume that a suitable architectural description of an application exists, in order for autonomics to be applied to the distributed deployment problem, two separate but closely related problems must be addressed: how to initially deploy an application, and how to manage its subsequent evolution in the face of host failures and other perturbations. Both these problems may be addressed by specifying a high-level configuration goal which is used to drive the autonomic process. In order to describe how an application is intended to be structured, we have proposed the use of domain-specific constraint-based languages [35]. Using a constraint-language, it is possible to describe configuration goals in terms of resources including software components and physical hosts, relationships between hosts and components, and constraints over these. From such configuration goals, plans may be developed for the deployment of components using the available physical resources. These goals could also be used to configure monitoring software to assess whether the executing application continues to obey the constraints specified in the description as described above.

Such constraint specifications could also be used to evolve the application in response to constraint violations arising from changes in the environment. There are several levels at which a deployed application may be evolved. The simplest involves the evolution of the configuration in order to maintain a previously specified goal. Thus the configuration evolves whilst the high-level configuration goal remains the same. We term this *autonomic evolution*, and consider it to be fundamental to the autonomic management of distributed applications. A second level of evolution is needed when the high-level goal itself changes, due to a change in application requirements. Both kinds of evolution may be handled in the same way, treating the first as a special case of the second in which the goal remains fixed. In both cases an ongoing autonomic cycle could be employed to solve the current constraint problem, deploy the

resulting configuration, and monitor the deployment to determine when to repeat the sequence.

The development of domain specific constraint-based languages to express the constraints over architectural specifications remains a difficult and complex task. The implementation of constraint-solvers to solve deployment problems also remains a daunting task requiring research from both software engineers and the constraint satisfaction/optimization communities.

4.5. Architectural Specification

Many of the current architectural specification languages do not currently cope with or describe dynamic change. A specification expressed in an ADL describes a future intended state of a system. Clearly to be of more general applicability, it must be possible for architects to express changes to software systems using ADLs. This is being addressed by the *dynamic software architecture* community. The ability to change a software specification written in an ADL introduces many problems which are closely related to the area of deployment. Some of these include which operations are permissible: for example, may components be created and destroyed, can the architectural topology be changed, and may nested components be unnested?; and can a component's internal state be preserved and transferred to a new instance and if so how?

Medvidovic [36] suggests it should be possible to accommodate new components, upgrade components, reconfigure the architecture and modify the mappings of components to machines. He argues that ADLs do not currently have the right set of features to accommodate such dynamic changes and that (ADL) language mechanisms are required to carry out such changes. Medvidovic calls such a language an Architecture Construction Notation (ACN).

The operational semantics of such ACNs are akin to the set of operations that are required to perform injection of control (bind, unbind, instantiate etc.). Clearly, an ACN may be viewed as a meta-program which operates on the ADL architectural representations. This is obviously related to the meta-programming ideas in the OMG MOF and a rich area for future architectural research.

If we assume that all other problems related to autonomic management have been solved (and some would argue that they have), the problem of coordination remains. Deployment, redeployment, wiring and configuration of distributed components and applications often requires components to be passivated, for their state to be stored, and then restored following reconfiguration. Furthermore such

actions need to be coordinated across multiple sites and with transactional support to prevent inconsistencies from arising. Update actions may sometimes only be performed at certain times, for example at night when demand is low; this introduces further temporal complexities to the update task. Such activities will not only require languages to express these operations (as discussed above), those languages will need to be capable of expressing the complex temporal and transactional state space that occurs during reconfiguration.

Management processes that manage the deployment and evolution of systems are subject to the same failures as the systems they manage. Consequently, some fault tolerance must be built into the control system, for example, the provision of a failure resistant collection of managers. This introduces a plethora of problems including the engineering of the fault tolerance of the management infrastructure, reliable distribution mechanisms for events from applications to the control system and determining when changes that have been instructed have been completed (distributed termination). Each of these areas is a major research challenge in its own right.

5. Future Directions

Much of the current focus in both industry and academia is on the Internet domain. However, two areas have not been discussed, but are of increasing interest in both communities are the mobile domain and the sensor-net domains.

Whilst the mobile domain is likely to be dominated by IP based protocols, the deployment problems are complex due to the heterogeneity and number of the devices, intermittent network connections, the lack of centralised control and that the mobile devices tend to be owned by individuals. The mobile arena is a rich ground for future research in this domain.

The field of wireless sensor-nets is rapidly growing. Deployment in the field of sensor-nets is currently extremely simple – sensor-net nodes are directly connected desktop machines and monolithic programs (generally including the operating system) are loaded into them. As sensor-nets are more widely deployed and used, there will be an increasing need to update the software on nodes in situ. This will require individual components to be deployed over the wireless network. The deployment problems raised by this prospect are similar to those in the mobile domain but exacerbated by low bandwidth communications and the need to conserve energy.

6. Conclusions

This paper attempts to define what is meant by software deployment and clarify some of the terminology used in this field. State-of-the-art deployment systems from various fields have been examined and shown to operate on a variety of different grained objects and make different assumptions about the environment. The different systems have different approaches to the typing and type checking of objects and to the way in which components are named and the scope of names.

The lack of uniform agreement on what constitutes a component, an assembly or a package and what meta-data they might have associated with them is a hindrance to development in this field, as is the technology buy-in associated with different languages, operating systems, containers and methodologies.

Deployment is an inherently complex area; there is a danger of introducing accidental complexity through the introduction of overly elaborate and complex architectures and meta-architectures and the languages, tools and mappings that they introduce.

The use of virtualisation offers hope of a silver bullet to avoid much of this complexity permitting the entire environment to be encapsulated within a virtual image and deployed easily. The use of the various Web-Services technologies presents for the first time a machine and language independent standard for describing typed service interfaces. It also presents a standard uniform naming scheme which removes complexity and ambiguity from the deployment lifecycle. Finally, distributed inversion of control might permit virtualised services to be interconnected using Web-Services standards.

7. Acknowledgements

I would like to thank Graham Kirby for his helpful comments throughout the preparation of this paper.

8. References

- [1] OMG, "Specification for Deployment and Configuration of Component-based Distributed Applications", 2003 <http://www.omg.org/docs/mars/03-05-08.pdf>
- [2] A. Carzaniga, A. Fuggetta, R. S. Hall, D. Heimbigner, A. v. d. Hoek, and A. L. Wolf, "A Characterization Framework for Software Deployment Technologies", Technical Report Department of Computer Science, University of Colorado, Boulder, Colorado, April 1998.

- [3] OMG, "Unified Modelling Language: Superstructure version 2.0", August 2005 2005 <http://www.omg.org/cgi-bin/apps/doc?formal/05-07-04.pdf>

- [4] C. Szyperski, "Component technology: what, where, and how?" Proc. 25th International Conference on Software Engineering, Portland, Oregon, pp. 684 - 693, 2003.

- [5] E. Bruneton, T. Coupaye, and J. B. Stefani, "The Fractal Component Model", ObjectWeb February 5, 2004 2004 <http://fractal.objectweb.org/specification/index.html>

- [6] Sun Microsystems, "JNDI 1.2 Documentation", <http://java.sun.com/products/jndi/docs.html>

- [7] Sun Microsystems, "JSR-000220 Enterprise JavaBeans 3.0", <http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html>

- [8] M. Rutherford, K. Anderson, A. Carzaniga, D. Heimbigner, and A. Wolf, "Reconfiguration in the Enterprise JavaBean Component Model", Proc. IFIP/ACM Working Conference on Component Deployment, pp. 67-81, 2002.

- [9] E. C. Bailey, Maximum RPM: Sams, 1997.

- [10] OMG, "CORBA Components formal/02-06-65", OMG <http://www.omg.org/docs/formal/02-06-65.pdf>

- [11] Apache Software Foundation, "Webservices - Axis", 2005 <http://ws.apache.org/axis/skin/images/pdfdoc.gif>

- [12] Microsoft, "IIS 6.0 Technical Reference (IIS 6.0)", 2006 <http://www.microsoft.com/technet/prodtechnol/WindowsServer2003/Library/IIS/848968f3-baa0-46f9-b1e6-ef81dd09b015.mspx?mfr=true>

- [13] J. C. Schlimmer, "Web Services Description Requirements ", 2002 <http://www.w3.org/TR/ws-desc-reqs/>

- [14] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, "Web Services Description Language (WSDL) 1.1", 2001 <http://www.w3.org/TR/2001/NOTE-wsdl-20010315>

- [15] VMWare, "Building the Virtualized Enterprise with VMware Infrastructure", Technical Report 2006.

http://www.vmware.com/pdf/vmware_infrastructure_wp.pdf

[16] S. Schumate, "Implications of Virtualization", Technical Report 2004. www.dell.com/downloads/global/power/ps4q04-20040152-Shumate.pdf

[17] P. Brebner and W. Emmerich, "Deployment of Infrastructure and Services in the Open Grid Services Architecture (OGSA)", Proc. Component Deployment 2005, pp. 181-195, 2005.

[18] Globus, "Globus Toolkit 4.0 Release Manuals", 2006 <http://www.globus.org/toolkit/docs/4.0/>

[19] B. Smith, "Reflection and Semantics in LISP", Proc. 11th ACM Symposium on Principles of Programming Languages, New York, pp. 23-35, 1984.

[20] A. Goldberg and D. Robson, Smalltalk-80: The Language and its Implementation. Reading, Massachusetts: Addison Wesley, 1983.

[21] G. N. C. Kirby, R. C. H. Connor, Q. I. Cutts, A. Dearle, A. M. Farkas, and R. Morrison, "Persistent Hyper-Programs", in Persistent Object Systems, Workshops in Computing, A. Albano and R. Morrison, Eds.: Springer-Verlag, pp. 86-106, 1992.

[22] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides., Design Patterns: Elements of Reusable Object-Oriented Software: Addison Wesley, 1994.

[23] R. Johnson, J. Hoeller, A. Arendsen, C. Sampaleanu, R. Harrop, T. Risberg, D. Davison, D. Kopylenko, M. Pollack, T. Templier, E. Vervae, P. Tung, B. Hale, A. Colyer, J. Lewis, C. Leau, and R. Evans, "The Spring Framework - Reference Documentation", 2006 <http://static.springframework.org/spring/docs/2.0.x/reference/index.html>

[24] "Picocontainer", 2006 <http://www.picocontainer.org/>

[25] M. Fowler, "Inversion of Control Containers and the Dependency Injection pattern", 2004 <http://www.martinfowler.com/articles/injection.html>

[26] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming.", Proc. European Conference on Object-Oriented Programming, pp. 220-242, 1997.

[27] OMG, "OMG Model Driven Architecture", 2006 <http://www.omg.org/mda/>

[28] A. Dearle, G. Kirby, A. McCarthy, and J. Diaz y Carballo, "A Flexible and Secure Deployment Framework for Distributed Applications", in Lecture Notes in Computer Science 3083, (eds), Proc. 2nd International Working Conference on Component Deployment (CD 2004), Edinburgh, Scotland,, W. Emmerich, Wolf, AL Ed.: Springer,, pp. 219-233, 2004.

[29] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer, "Xen and the Art of Virtualization", Proceedings of the ACM Symposium on Operating Systems Principles, 2003.

[30] D. Reed, I. Pratt, S. Early, P. Menage, and N. Stratford, "Xenoservers: Accountable execution of untrusted programs", Proc. Proceedings of the 7th Workshop on Hot Topics in Operating Systems (HotOS-VII), 1999.

[31] M. Foster, J. Ilgen, and N. Kirkwood, Tivoli Software Installation Service: IBM, 2000.

[32] Amazon, "Amazon Web Services", 2006 <http://developer.amazonwebservices.com/connect/entry.jspa?externalID=123&categoryID=48>

[33] G. Kirby, S. Walker, S. Norcross, and A. Dearle, "A Methodology for Developing and Deploying Distributed Applications", in Lecture Notes in Computer Science 3798, A. Dearle and S. Eisenbach, Eds., pp. 37-51.

[34] J. Kephart and D. Chess, "The Vision of Autonomic Computing", IEEE Computer, vol. 36 no. 1, pp. 41-50, 2003.

[35] A. Dearle, G. N. C. Kirby, and A. J. McCarthy, "A Framework for Constraint-Based Deployment and Autonomic Management of Distributed Applications", Proc. First International Conference of Autonomic Computing (ICAC), pp. 300-301, 2004.

[36] N. Medvidovic, "ADLs and dynamic architecture changes", Proc. Joint Proceedings of the Second International software Architecture Workshop (ISAW-2) and International Workshop on Multiple Perspectives in Software Development (Viewpoints '96), San Francisco, California, pp. 24 - 27, 1996.