PocoCapsule™

IoC & DSM framework for C++

Developer Guide

Release 1.0
October 17, 2007

Pocomatic Software LLC
Foster City, CA. USA

PocoCapsule™ IoC & DSM framework for C++, Developer Guide

Release 1.0
October 17, 2007
Foster City, CA. USA

This document is available at:
http://www.pocomatic.com/docs/pococpp-iocdsm-dev-guide.pdf

Examples used in this document are available at:
http://www.pocomatic.com/docs/cpp-examples

The installation packages of PocoCapsule/C++ IoC&DSM framework for Linux/Unix, Windows, and VxWorks are either downloadable at: http://www.pocomatic.com or available by sending request to prod@pocomatic.com

Table of Content

<This is a blank page>

Chapter 1 Introduction

This document is about PocoCapsule, an *inversion of control* (IoC) and *domain specific modeling* (DSM) framework for building component based C and C++ applications. The seamless combination of *non-invasive* IoC and *declarative* DSM reveals a software development paradigm shift to be explored in this document.

1.1 Why and What of IoC

The underlying principle of object-oriented design and development suggests that a large-scale complex application can be simplified by partitioning it into small, reusable, and loosely coupled objects. Hence, the primary focus of object-oriented modeling and/or programming languages, such as UML 1.0[1], C++, and Java, is to have abstraction, encapsulation, modularity, and hierarchy etc. as their intrinsic concepts and built-in grammar features.

However, the productivity in developing a software application, the customizability in deploying it, and the extendibility in maintaining/upgrading it are not solely determined by how well it is abstracted, partitioned, and implemented in objects. They also heavily rely on how effective and flexible these separated artifacts are able to be integrated and manipulated, namely, to be assembled back together into a seamlessly collaborated system as well as deployed/configured in a service or middleware adapting environment.

Although mainstream OO programming languages, such as C++ and Java, are fairly effective on general object-oriented decomposition, individual business logic abstraction, implementation, and low level constructions/configurations, they are rather primitive, stiff, and poorly expressive on integrating and manipulating domain/service level objects, referred to as *components* in this document. For instance, to facilitate life cycle control, to simplify dependency management, and to prevent tight coupling to particular service object implementations, common programming practices tend to avoid the explicit use of language intrinsic features such as object constructors and setters. Instead, applications developed in these classic OO languages often resort to various contrived low level plumbing structures and even kludges such as object factories, builders, directories, adaptors, singletons, configuration/property managers, managers of factories, managers of managers, and so on. For years, OO professionals have taken these design patterns for granted and even enthusiastically exercise them. The detrimental side of this practice, especially when applied to high level component based application development, has rarely been questioned[2].

Besides, the imperative nature of traditional OO languages makes it very hard and time consuming to extract the underlying high level architecture design out of the

[1] UML introduced the action diagram in 1.4 and the component wiring diagram in 2.0, hence has gone beyond primary class modeling.
[2] See more discussion by Matthias Felleisen, *On the Expressive Power of Programming Languges*, 1991

implementation code of a large-scale application. Maintenance engineers taking over an existing application have to rely on its out of date design/architecture documents and low level code patterns/comments.

The *component-based development* (CBD) addresses these issues by shifting these plumbing complexities into frameworks that manipulate components and set up applications based on users/developers provided *declarative* descriptions. Such a *declarative* description (also known as a *descriptor*) outlines *what* the desired application structure looks like without *imperatively* specifying *how* to build it step-by-step. Ideally, CBD frameworks should allow application developers to focus on business logic implementations without concerns of low level plumbing structures. The declarative nature of application descriptions should make the high level architectures of their applications self-documented.

During the past decade, numerous CBD frameworks have been standardized by committees and offered by vendors[3]. Unfortunately, most of them are heavily infrastructure oriented and awfully complex to average domain experts and even cumbersome to experienced system/framework gurus[4]. Furthermore, most of these traditional frameworks are specific to particular problem domains[5] and/or special component models. This largely prohibits the reusability of third party components and legacy implementations, rules out the transferability of developer/user skills and experiences, and discourages the interchangeability of development tools across frameworks. Hence, despite advocates keeping promise that CBD would be the hope of software development and tools would be the hope of the hopeless CBD, both of the CBD and its tools remain being hopelessly primitive, ad hoc, proprietary, and introduced more complexities and restrictions than the original issue to be addressed.

The *Inversion of Control* (IoC), sometimes referred to as *dependency injection* (DI), is an unorthodox but surprisingly simple, straightforward, and effective CBD paradigm that avoids all the pains above and turns out to be significantly superior to its predecessors.

The IoC technology is typically built and offered as a framework that comprises of a runtime engine known as the IoC container and a couple of instrument utilities. An IoC container can be a standalone executable and/or a lightweight runtime package/library to be embedded in user applications. Like its conventional predecessors, an IoC framework aims to facilitate macro-scope business or service level component application assembly,

[3] Examples of these component frameworks include EJB, OMG CORBA component model (CCM), a handful of robotic component frameworks (such as CARMEN, OROCOS, ORCA, MARIE, CLARAty, MIRO, and OMG RTC), the OASIS Open Composite Service Architecture (OpenCSA, also known as service component architecture (SCA) from OpenSOA) for Web Services/SOA, and the US Navy JTRS's Software Communications Architecture (SCA) for software defined radio (SDR).

[4] Measured by the length of a training course offered by an ORB provider, it takes more than 15 weeks for a C++/Java programmer to learn CORBA, CCM, and needed OMG services (Event/Notification, DDS etc.).

[5] For instance, various robotic applications or software defined radio applications specific component frameworks, and various component frameworks that mandate the component model (API) of valid components.

configuration, and deployment by abstracting away the plumbing complexities and boilerplate code of component instantiation, configuration, wiring, and lifecycle management, etc.

Comparing to their predecessors, IoC frameworks have the following distinct characters, to be explored in this chapter:

- IoC containers are *neutral to component programming models*.
- The *Hollywood Principle* of IoC technology allows component to be *container agnostic*.
- IoC containers are non-invasive to components.
- IoC technology is straightforward, lightweight, and has zero overhead.

1.1.1 Neutral to component programming models

Decent IoC containers are neutral to component programming models and accept virtually any type of objects of the underlying programming languages as components. For instance, in Java IoC containers, objects of any type extended from the type *java.lang.Object* are valid components, and are referred to as *plain old Java objects* (POJO). In PocoCapsule/C++ IoC framework, instances of arbitrary user or third party defined classes and template classes, classes generated from CASE or UML tools, old K&R C structs, unions, function pointers, arrays and so on are all qualified as components, and are referred to as *plain old C++ objects* (POCO)[6].

Being neutral to component programming models, IoC containers completely eliminate the compliance barriers and steep learning curves of traditional component programming models, significantly lower application development and maintenance complexities, costs, and risks.

The IoC technology is also OS and programming language neutral. For instance, PocoCapsule has Java and C++ editions, and could certainly be applied to other languages (such as Ada) later on. This document is about PocoCapsule IoC framework's C++ edition, therefore, is going to focus on components in *plain old C/C++ objects*, also referred to as "*component*", and/or "*bean"* hereafter.

1.1.2 Hollywood Principle, Container Agnostic, and Dependency Injection

Orthodox CBD frameworks, such as CCM, old (pre-3.0) EJB, and OASIS Open CSA (SCA)[7], tend to force component internal implementations tightly couple to the particular CBD frameworks by requiring them to retrieve property settings and dependency wiring configurations from global or component local scope contexts served by the runtime of the particular CBD framework.

---

[6] Many POCOs, such as STL containers, standard iostreams, are low level components. Although they can be manipulated and wired using IoC containers, it should be clarified that, not as a restriction but as a guideline, IoC frameworks are intended for business and/or service level objects integration and manipulation.

[7] SCA C++ binding 0.95.

In an IoC framework, components do not need to retrieve their settings and wiring configurations from the container. Instead, it is the container to perform such kind of provisioning by explicitly calling back setting/accessing functions of components. This is known as the *Hollywood Principle*, summarized as "*don't call me, I will call you*". This scenario simplifies component implementations, decouples components from runtime services or facilities of a particular container (this is referred to as *container agnostic*), allows components to be easily developed and unit tested without a container.

Such callbacks are referred to as *IoC invocations* in this document. IoC invocations with object pointers or references as input (or inject) arguments effectively wire up objects with their dependencies, therefore are constantly referred to as "*dependency injection*" (DI) or "*dependency wiring*".

1.1.3 Non-invasive to components

It was claimed[8] that the technique of inversion of control was merely a common character of all frameworks, just like having 4 wheels is a common nature of most vehicles. A key distinct character of IoC callbacks neglected in this false claim is the *non-invasiveness* (or *non-intrusiveness*). Traditional non-IoC frameworks might commonly use callbacks to perform component context injection, state notification, lifecycle controls, and retrieve facet references. However, they mandated components to support framework predefined callback interface/operation signatures. This undermined the POCO/POJO premise and is known to be "*invasive*" or "*intrusive*".

IoC containers, such as the PocoCapsule/C++, are able to controls and configure plain old programming objects *non-invasively* (or non-intrusively) through arbitrary callback interface/function signatures not previously defined and known by the given IoC container.

Java IoC containers typically support two types of non-invasive IoC invocations, namely *constructor/factory injection* and *setter injection* (the "*context injection*" is merely a variation of either of these two combining with lookup). Because of the nature of C and C++ programming languages, PocoCapsule/C++ IoC container supports a much richer set of non-invasive IoC invocations, including almost all public and/or global C and C++ POCO function invocations with arbitrary numbers of arguments, and data member accesses, such as:

- Constructors (and destructors in lifecycle control)
- Global C or C++ functions and operators
- Static and non-static member functions and operators
- Static and non-static data members.

Also, PocoCapsule/C++ seamlessly works with *generic programming* and other C++ programming practices by supporting template classes, template functions, macros, and

[8] http://martinfowler.com/articles/injection.html

so on. There is virtually no unreasonable restriction on class hierarchies and operation/function signatures.

1.1.4 Straightforwardness, lightweight, and zero overhead

Comparing to traditional CBD frameworks, IoC frameworks in general and PocoCapsule/C++ IoC in particular imply a significantly short learning curve, no invasive restrictions on interfaces, implementations, and packaging of components, no bloated code generation, and neutral to OS platforms, distributed middleware, component models, programming paradigms, technologies, and vendors.

Comparing to implementations of traditional claimed to be "lightweight" CBD frameworks, the 70Kbytes memory size of PocoCapsule/C++[9] and its straightforward component programming model imply one or even two order of magnitude reductions on runtime footprint, lines-of-code, associated vulnerabilities, and FAA/EASA safety certification cost of the container itself and applications of embedded, mission-critical, and/or safety-critical systems.

For performance critical (such as data mining, game or 3D visualization, image processing, robotic or motion control system, FFT, numerical computing) and real-time (such as VoIP and SDR) applications, PocoCapsule/C++ introduces zero CPU and I/O overhead and zero latency/jitter.

1.2 Why and What of DSM/DSL

IoC technology shifts the plumbing complexities of component based softwares into dynamically reconfigurable IoC frameworks and declarative application descriptions. This allows application developers to focus on implementing high level business logic and flexibly assemble up their applications without relying on contrived low level patterns, assuming application descriptions are reasonably expressive.

As going to be illustrated in the next three chapters, application descriptions in core schemas of IoC frameworks are more or less plain C/C++ application code expressed in XML. These IoC core schemas have the advantage of being very straightforward and generic to virtually all problem domains. However, this generic simplicity has the weakness of being less expressive, over verbose, and error prone for large scale applications[10].

To improve expressiveness and raise abstraction level, IoC frameworks would have to extend their core schemas. However, as will be discussed in chapter 4 of this document, a generic high level deployment schema applicable to vast diverse applications would be bloated and complex.

A solution to the dilemma of *expressiveness and diversity vs. intuitiveness and simplicity* is the *domain specific modeling* or *domain specific language* (DSM or DSL). Namely,

[9] Assuming encoded descriptors are used.
[10] This is one example of the so-called "XML Hell".

instead of having a bloating one-size-fits-all assembly/configuration/deployment model (schema) expressive for generic applications[11], an DSM/DSL integrated IoC framework allows users or third parties to easily and quickly customize and/or extend the simple but less expressive core IoC deployment model (schema) into different user defined models (schemas). As each of these models (schemas) is only designed for one specific problem domain, it is able to be simple, intuitive while still highly expressive with a relatively small set of problem specific vocabularies and high level syntax structures.

1.2.1 Building DSM/DSL declaratively and portably

A conventional approach to support user defined/extend schemas in IoC frameworks is using XML DOM element plug-in[12]. The programming model of these callback handlers is imperative, proprietary, and relies on low level functions such as XML DOM API. Therefore, this approach and its native callback handlers are non-portable, costly to develop/maintain, not able to be automated/tooled, and tightly locked to the particular IoC container.

In contrast, instead of depending on *imperative* and hard coded DOM parsers or DOM element handlers to process user defined DOM elements, PocoCapsule leverages the declarative, component model neutral, and container agnostic IoC infrastructure itself to facility DSM/DSL. Based on the standardized, portable, and ubiquitous mainstream XSLT technology, this solution is completely free from any proprietary plug-in API or any low level XML DOM or SAX programming. Comparing to the proprietary plug-in handler approach, this IoC based solution is straightforward for domain experts, easily transformable across decent IoC containers, and open to third party tool vendors[13].

1.2.2 IoC + DSM: A paradigm shift

The implication of IoC is far more profound than "a lousy constructor" and the DI design pattern/principle. The following two aspects of this technology make it a very effective and neutral bridge between legacy *imperative* programming paradigm and the *declarative modeling* realm where high level abstraction, customization, and tooling become much more convenient.

The first aspect of this paradigm is *declarative* descriptions. A traditional *imperative* program instructs "*how*" to proceed step by step in performing given operations. A *declarative* description in the new paradigm, on the other hand, expresses "*what"* are the intended structures and dependencies without specifying procedures to construct them.

The second aspect of this paradigm emphasizes large scale high level application modeling in domain specific languages (DSM/DSL), rather than low level object oriented modeling (interfaces/claases) in traditional OO programming/design languages (such as C++, Java, and UML).

[11] As the UML 2.0 attempted, namely to be all thing for all men.
[12] Such as the Spring 2.0 extensible XML configuration.
[13] In fact, there are many third party XSLT tools available today.

Therefore, instead of merely focusing on the "dependency injection" (DI) pattern or component wiring, PocoCapsule and this document emphasize the open, declarative, and user extended/defined modeling aspects of these technologies.

1.2.3 IoC + DSM: A framework of frameworks

A DSM/DSL integrated IoC framework, such as PocoCapsule, not only allows flexabily application description/modeling changes/enhancements, but also becomes a highly effective platform for building other standardized or user defined high level CBD frameworks. Typically, such a CBD framework would imply thousands or even tens of thousands lines of code and weeks or even months of efforts of a skillful framework developer, if it was built from scratch[14]. In a significant contrast, based on an IoC+DSM/DSL framework such as PocoCapsule, an average developer or even a domain expert without framework building experience is able to comfortably construct the same high level container in just a couple of days or even hours with less than a few hundreds lines of highly descriptive and self-documenting code.

The following standardized or user defined component frameworks, built from IoC and DSM technologies, are available out-of-the-box in PocoCapsule.

- The OASIS Open Composite Service Architecture (***Open-CSA***) for web service and service oriented applications.
- The US Navy JTRS Software Communication Architecture (***JTRS-SCA***) core framework (CF) for software defined radio (SDR) applications.
- The OMG Robotic Technology Component (***OMG-RTC***) framework for robotic applications.
- A CORBA server application assembly, configuration and deployment framework supporting POA server, OMG Event/Notification, OMG DDS, and OMG RTC applications.
- Other user defined deployment frameworks for WS, SOA, SDR, and robotic component applications.

These frameworks and their relations to the PocoCapsule core framework are illustrated in the diagram in the next page. Their usages are described in appendixes A, B, C, D, and demonstrated in PocoCapsule user examples[15].

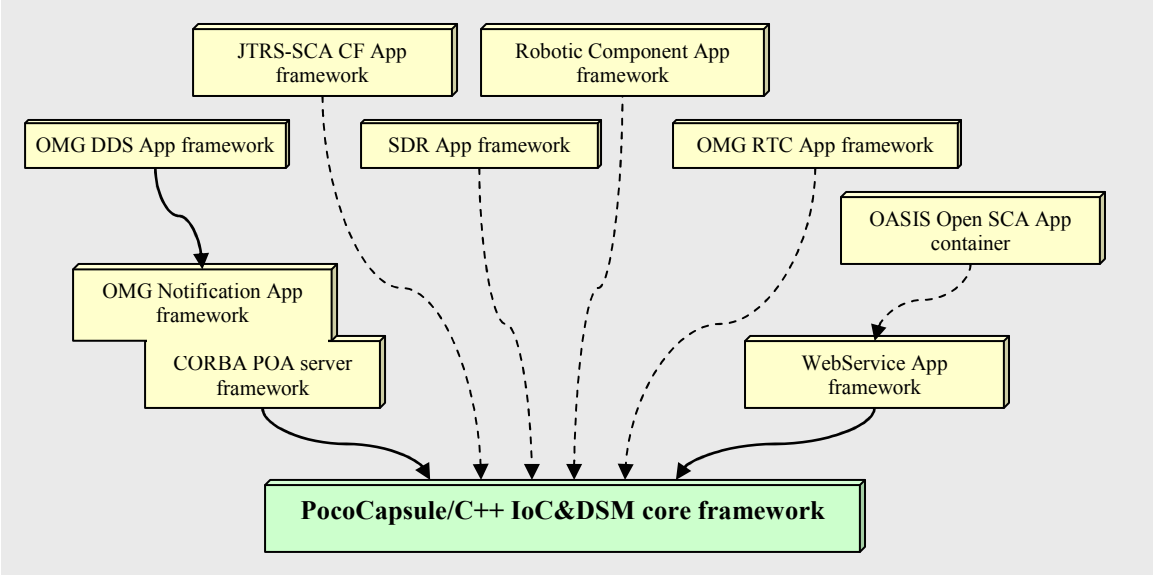
To application developers, the assembly, configuration, and deployment containers produced in this IoC + DSM approach are much simpler and superior to their traditional handcrafted counterparts by largely eliminating contrived aspects of the component models and restrictions that sacrifice the usability for application developers in favor of the convenience for vendors[16].

---

[14] For instance, the reference implementation of JTRS-SCA assembly container (Java) from CRC and the prototype implementation of Open-CSA (C++) from the Apache Tuscany project all used 6,000 to 10,000 lines of code. In PocoCapsule, however, they can easily be built in less than 500 lines of code.

[15] http://www.pocomatic.com/docs/cpp-examples

[16] See appendix A, B, C, D and PocoCapsule examples for comparisons and analysis.



1.3 About this document

This document serves as a developer guide and manual of PocoCapsule/C++ IoC&DSM component framework. The following topics are going to be covered:

- In chapter 2, basic IoC concepts are introduced though two examples. Main focus is to demonstrate how to integrate and manipulate C/C++ application setups through XML descriptions.
- In chapter 3, the PocoCapsule/C++ IoC core schema is explored thoroughly.
- In chapter 4, the support of user extended/defined schema for domain specific modeling (DSM) is detailed.
- In chapter 5, the API of PocoCapsule/C++ container is specified. This API is used to embed PocoCapsule into user applications.
- In chapter 6, PocoCapsule/C++ framework utilities, libraries, and schemas are summarized.
- In Appendix A, Web Services (including OASIS Open CSA) with PocoCapsule is discussed.
- In Appendix B, robotic component (including OMG RTC) application with PocoCapsule is discussed.
- In Appendix C, software defined radio (include JTRS-SCA) application with PocoCapsule is discussed.
- In Appendix D, CORBA (including POA, event/notification, and DDS) application with PocoCapsule is discussed.

Chapter 2 Getting Started

2.1 A Hello Example

In this tutorial chapter, two simple examples are used to illustrate the basic use case of deploying C/C++ applications in PocoCapsule/C++ IoC framework. Source code, makefile(s), and README html pages of these two examples, as well as nearly 25 other examples, can all be found in the examples directory of a PocoCapsule installation or from Pocomatic web site[17].

2.1.1 A simple POCO bean class

In the first example, several simplest forms IoC invocation scenarios are presented, starting from the following three:

- Bean instantiation: using the C++ constructor of the declared *class* type,
- Post-instantiation invocation: on a declared member function of the just instantiated bean,
- Lifecycle control: using a declared *destroy-method*. In this example, this method is declared as the *delete* operator, hence, the destructor of the bean.

The bean is an instance of a plain old C++ class *Foo*, declared in the header file foo.h as follows:

```
class Foo {
  public:
      Foo(const char* msg);
      ~Foo();
      void hello(const char* msg);
};
```

This class has a constructor, a destructor and a non-static member function *hello()*. On Windows, if the bean class implementation is built into a DLL library, to be externally accessible, it should be exported from the library and should be imported wherever it is referenced outside the library. This detail can be found in the example source code and is omitted here for simplicity.

This Foo class is implemented in the source file foo.C as follows:

[17] http://www.pocomatic.com/docs/cpp-examples/basic-ioc/hello

```
Foo::Foo(const char* msg)
{
      printf("Hello %s\n", msg);
}

Foo::~Foo()
{
      printf("That's all folks! Goodbye!\n");
}

void Foo::hello(const char* msg)
{
      printf("Hello %s\n", msg);
}
```

This source file is compiled and linked into a UNIX shared library or a Windows DLL named as foo.so or foo.dll in this example. The implementation of this class and its binary after built are all completely independent of the PocoCapsule/C++ IoC container.

2.1.2 Creating the deployment description

Then, to construct an application out of this *Foo* class, a deployment description is created that describes the following arrangement:

- A bean of class *Foo* is to be instantiated using its constructor, with a string "Foo constructor!" as input argument. This bean is an *eager singleton*, with its *singleton* attribute absent (or equals to "true") and its *lazy-init* attribute equals to "false". The semantic meaning of this will be discussed soon next.
- The instance of above bean is under life-cycle control, and to be destroyed using the "delete" operator, hence the destructor, by life cycle manager.
- A post-instantiation IoC invocation for this bean is declared, with "hello" as the method name and a string "Foo hello()!" as input argument.

Also, this example assumes the binaries of Foo class implementation and its dynamic invocation proxies and reflections are not linked in the runtime environment but to be dynamically loaded as follows:

- The container should first load the foo.so or foo.dll library. The $dll symbol is ensure the deployment description to be OS platform independent. It will be mapped to different suffixes, such as "so", "dll", on different platforms by PocoCapsule container.
- Then, the container should load the reflx.so or reflx.dll library. This is the library containing dynamic invocation proxies of Foo class. These proxies are built from PocoCapsule/C++ framework development tools. It will be discussed soon in next section.

The XML document of this deployment description looks like follows:

```
<?xml version="1.0"?>
<!DOCTYPE poco-application-context SYSTEM
      "http://www.pocomatic.com/poco-application-context.dtd">

<poco-application-context>

  <!-- binaries to be loaded -->
  <load library="./foo.$dll"/>
  <load library="./reflx.$dll"/>

  <!-- bean instantiation and lifecycle control methods -->
  <bean
    class="Foo"
    destroy-method="delete"
    lazy-init="false">
    <method-arg type="cstring" value="Foo constructor!"/>

    <!-- a post instantiation ioc invocation -->
    <ioc method="hello">
      <method-arg type="cstring" value="Foo hello()!"/>
    </ioc>
  </bean>

</poco-application-context>
```

2.1.3 Generating dynamic invocation proxies

The dynamic invocation proxies required by the container reflection engine are generated using the PocoCapsule/C++ instrument utility ***pxgenproxy*** from the deployment description file setup.xml as follows:

```
% pxgenproxy -h=foo.h setup.xml
```

The *pxgenproxy* generates dynamic invocation proxies from function signatures used by given deployment descriptions. This scenario is a reverse of "reflection" and therefore is referred to as "*projection*". The generated source code of proxies will be written out to a file that is named as setup_reflx.cc in this example (the "reflx" stands for REFLection-proXies). This file should be compiled and then linked into a shared/dynamic library (e.g. reflx.so or reflx.dll). If there is any error in the deployment description against the bean class function prototypes, it will be caught and reported during the compiling and linking stages instead of at runtime deployment.

As this shared/dynamic library bridges the PocoCapsule container to the bean, depending on OS platforms, it may need to be linked to both of them, namely the libpococapsule.so or pococapsule.dll and the foo.so or foo.dll. After a successful compiling and linking, this library is ready to be used by the container.

2.1.4 A mini container

A PocoCapsule/C++ IoC container is a runtime library that is expected to be embedded in user application executables. This example uses such a mini container application executable as follows (error handling code has been left-out for simplicity):

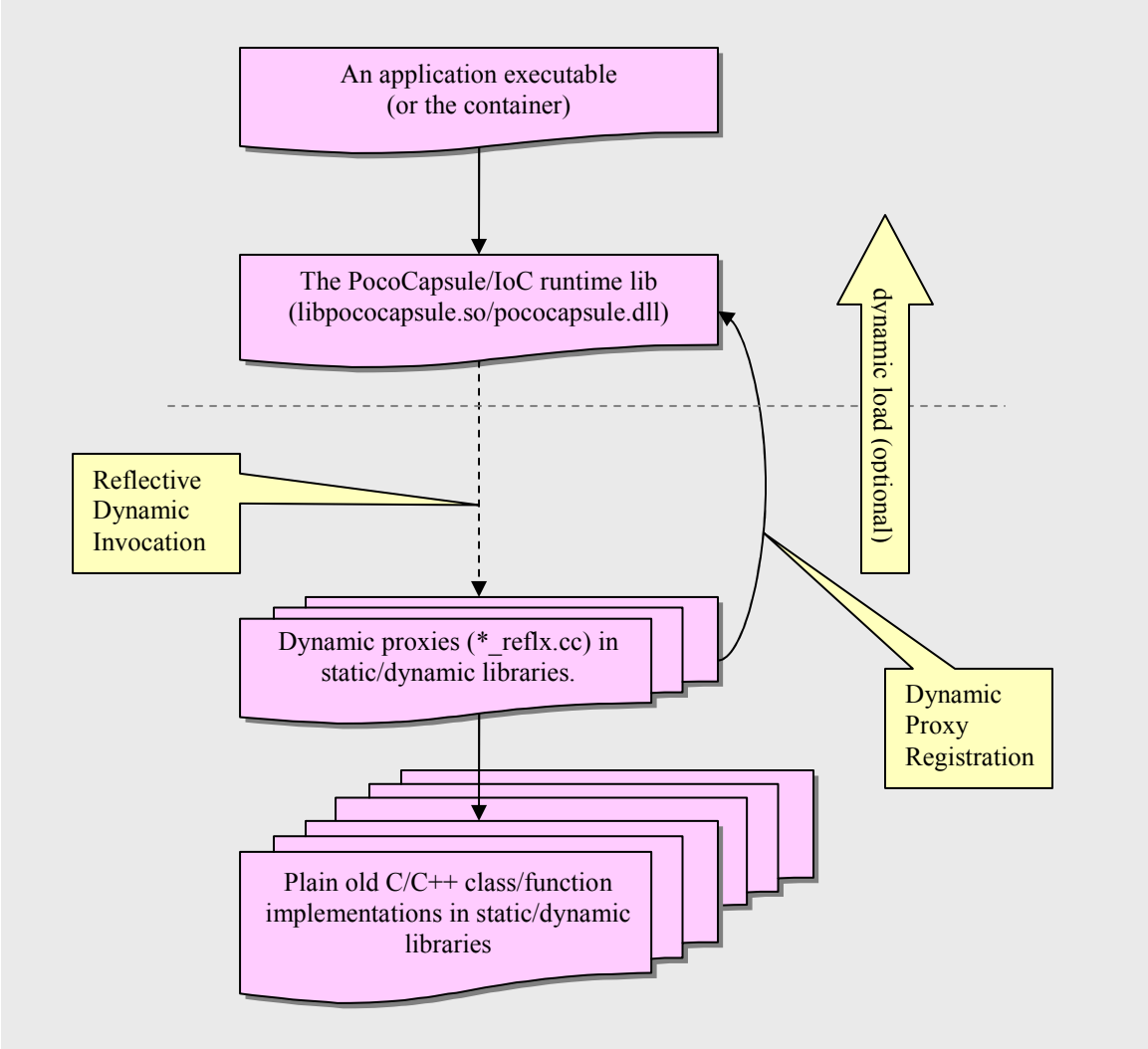
```
#include <pocoapp.h>

int main(int argc, char** argv)
{
      // parsing the descriptor
      POCO_AppContext* ctxt
      = POCO_AppContext::create("setup.xml", "file");

      // all eager singletons will be instantiated. Their
      // post-instantiation ioc methods will be invoked.
      ctxt->initSingletons();

      // all instantiated singletons will be destroyed
      ctxt->terminate();
      ctxt->destroy();
}
```

This code is compiled and linked with PocoCapsule/C++ runtime (the libpococapsule.so or pococapsule.dll) into an executable file (named as *main* on Unix/Linux or *main.exe* on Windows in this example). In executing this application, what will happen are:

- The *POCO_AppContext::create()* method in the code reads/parses the input description file (the setup.xml in this case). This will cause all <load> to be performed.
- The *POCO_AppContext::initSingletons()* method next after it instantiates all *eager singleton* beans, and performs their post-instantiation ioc invocations.
- The *POCO_AppContext::terminate()* method will release all singleton bean instances that have been put under the container's life-cycle control, namely have their *destroy-method* attribute specified.
- The *POCO_AppContext::destroy()* method releases resources of the application context.

Note: *This mini container is completely independent of application business logic components. It is neither compiled against any bean code/header, nor linked with their binaries. In fact, this mini container can be and is used by almost all examples shipped in PocoCapsule/C++. These examples use completely different application components*.

The dependencies of the mini-container, the PocoCapsule/C++ IoC core runtime, the dynamic invocation proxies, and the user implemented component libraries are illustrated in the following diagram:



Notably, the container and IoC runtime on the upper portion of this diagram do not depend on either the dynamic invocation proxies or application business logic POCO implementations in the lower portion. When proxies are loaded up, they register themselves to the IoC container runtime and will be used in a scenario similar to Java reflection.

2.1.5 Deploying the application

As hard coded in the above source code, on starting the mini container executable, the application described in setup.xml will be deployed. To be specifically, on calling the initSingletons() of the application context, the declared eager singleton bean of class Foo will be instantiated. Then, the post-instantiation IoC method hello() will be invoked afterwards. This method prints out the text “Hello Foo hello()!”. On the terminate() of the context, the singleton’s destroy method will be invoked which calls the destructor of the Foo class, that in turn prints out the text “That’s all folks! Goodbye!” These results are shown as follows:

```
% main
Hello Foo constructor!
Hello Foo hello()!
That's all folks! Goodbye!
```

2.1.6 Calling stdio printf() and << operator on std::cout

The other two IoC invocation scenarios illustrated in this example declare invocations on the C stdio printf() global function and then, the << operator on std::cout. To do this, a new bean is declared with two post-instantiation <ioc> elements.

```
<!xml version="1.1"?>
...

<poco-application-context>

  <load library="./foo.$dll"/>
  <load library="./reflx.$dll"/>

  <bean
    class="Foo"
    destroy-method="delete"
    depends-on="do-this-first"
    lazy-init="false">
    <method-arg type="cstring" value="Foo constructor!"/>

    <ioc method="hello">
      <method-arg type="cstring" value="Foo hello()!"/>
    </ioc>
  </bean>

  <bean id="do-this-first" class="void">
    <ioc method="printf">
      <method-arg type="cstring" value="Hello %s\n"/>
      <method-arg type="cstring" value="C stdio printf()!"/>
    </ioc>

    <ioc method="std::cout {{">
      <method-arg type="cstring"
                  value="Hello C++ std::cout!\n"/>
    </ioc>
  </bean>
<poco-application-context>
```

The following three details in the description above are notable:

- The *class* attribute of this bean is "*void*". This implies all post-instantiation <*ioc*> methods on this bean are global or static C or C++ functions.
- This bean is not declared as an eager bean, therefore, it should not be instantiated explicitly by *initSingletons()* on the application context.
- However, the previous declared eager bean of Foo class is now declared to *depend on* this "void" bean. This means, before *initSingletons()* instantiate that Foo class bean, it should first instantiate this "*void*" bean.

The two post-instantiation <*ioc*> elements describe the following operations:

- Calling the global stdio C function printf() with the strings "Hello %s\n" and "C stdio printf()!" as the first and the second arguments.
- Calling the << operator on std::cout, with the string "Hello C++ std::cout!\n" as the argument.

The *method* attribute of the second <*ioc*> element is "*std::cout {{*". This is a user friendly equivalence of the "*std::cout <<*". PocoCapsule/C++ engine will convert it to "*std::cout <<*".

Because the deployment description in setup.xml introduces new kinds of IoC invocations, the proxy library should be regenerated to include proxies for these new signatures. The ***pxgenproxy*** instrument utility is used again as follows:

```
% pxgenproxy -h=foo.h -H=stdio.h -H=stream setup.xml
```

The *–h* option indicates a user-defined header file, and *–H* option indicates a standard library or compiler environment defined header.

Like before, the source code of IoC proxies will be written to setup_reflx.cc. Compiling and linking them also into the reflx.so or reflx.dll, then, it is ready to be used by the container.

The mini container built and used previously does not need to be modified and rebuilt, as it does not depend on bean implementation or application deployment description content change.

Running the same mini container executable again, with the new deployment description in setup.xml, the deployment process will print out the following result:

```
% main
Hello C stdio printf()!
Hello C++ std::cout!
Hello Foo constructor!
Hello Foo hello()!
That's all folks! Goodbye!
```

2.1.7 Tracing IoC invocations

This example can be found in the examples/basic-ioc/hello directory of a PocoCapsule installation, with more features enabled. For instance, if the main executable started with the command line argument "*-Dpococapsule.trace.enable=true*", it will print out the process of IoC invocations to stderr as follows (high lighted. The outputs to stdout from the application itself are low lighted):

```
% main -Dpococapsule.trace.enable=true
printf(...) Hello C stdio printf()!

std::cout <<(...) Hello C++ std::cout!

new Foo(...) Hello Foo constructor!
= (Foo*)0x81ff878
((Foo*)0x81ff878)->hello(...) Hello Foo hello()!

delete((Foo*)0x81ff878) That's all folks! Goodbye!
```

2.2 GPS Example – Dependency Injection

The example presented in previous section is trivial and does not involve dependency among beans. The example to be presented in this section has three components that have dependencies, especially circular dependencies.

This example was originated from an OMG CCM tutorial. Here, it is retrofitted into a POCO IoC example. Source code, Makefile and README.html of this example are located in the examples/basic-ioc/gps directory of a PocoCapsule installation, or frome Pocomatic website[18].

2.2.1 The GPS instrument and its component devices

This example models a GPS device that comprises of the following three components (as they are illustrated by the diagram in the next page):

[18] http://www.pocomatic.com/docs/cpp-examples/basic-ioc/gps

- A tick pulse generator implemented as the TickGenImpl class that generates clock pulses for the instrument,
- A GPS location component implemented as the GPSLocatorImpl class that determines the instrument's current location and notifies subscribed display device on any position change.
- A display component implemented as the NavDisplayImpl class that retrieves location data from GPS device on receiving position change notification, and displays the new position.

The detail interface/operation signatures and implementation of these POCO components are non-critical in comprehending the IoC deployment technique, and therefore are left out, and can be found in ${POCOCAPSULE_DIR}/examples/basic-ioc/gps directory. Besides, a CORBA variant of this example, that have component interfaces defined by plain old OMG IDL, can be found in ${POCOCAPSULE_DIR}/examples/corba/gps directory.

These three components are built into three shared libraries, TickGenImpl.so/dll, GPSlocatorImpl.so/dll and NavDisplayImpl.so/dll, and are all independent of the PocoCapsule/C++ in terms of source code, binaries and runtime.

Although these three components have dependencies, they do not have coupling of one implementation to others or to their factories. The implementation class of any of these components can be changed without affect others, as long as the new class still supports the required interface (e.g. inherits the required super class). The IoC container is now functioning as the object factories. Change implementation class is merely an easy change of the application description.

Note: *Although in this particular example, components are implemented in separated source files and built into separate libraries named after their class names, PocoCapsule/C++ does not mandate how component implementations are partitioned into source code and libraries, and how libraries are named. With PocoCapsule,*

- *Multiple components can be declared in one header file,*
- *Multiple components can be implemented in one source code file and,*
- *Multiple components can be built into the one dynamic or static library of arbitrary name.*

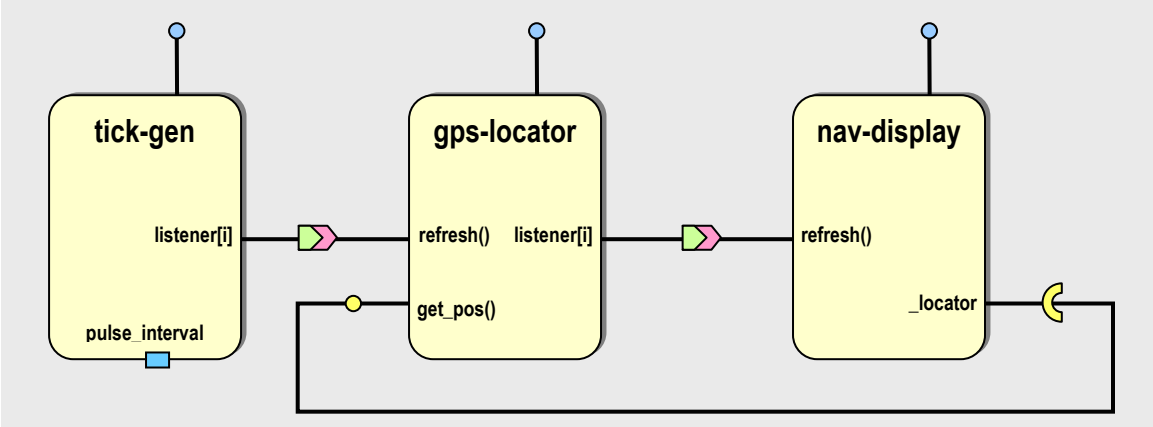
2.2.2 The deployment description

The deployment description for this example declares the following configuration of the modeled GPS device:

- Loading all shared/dynamic libraries of the above three components, as well as the library of reflection proxies. Library loading has higher deployment precedence than others in this example, and will be performed initially regardless where they were declared in this description.

- An eager singleton bean of class TickGemImpl is declared and is assigned with the id “tick-gen”. The declared instantiation method is the class’s container with two short integers as input arguments.
- A post-instantiation IoC invocation of method subscribe() is declared for the bean above. This method takes the reference of another bean of id “gps-locator” as input argument. By default, that bean will be passed as a pointer of its declared class type (namely, value of its *class* attribute). For instance, in this case, it will be passed as a GPSLocatorImpl*. This so-called “dependency injection” wires the “callee” bean gps-locator to this “caller” bean tick-gen.
- A lazy singleton bean of class GPSLocatorImpl is declared and is assigned with the id “gps-locator”. As a lazy bean, this bean will be instantiated on demand, such as when it is referred as an input argument somewhere else. This bean is declared to be instantiated using its constructor with no argument.
- A post-instantiation IoC invocation of method subscribe() is declared for the bean above. This method takes one input argument which is the reference of another bean with id “nav-display”. Similarly, in this case, the “callee” bean will be injected as a NavDisplayImpl* argument.
- A lazy singleton bean of class NavDisplayImpl is declared and is assigned with the id "nav-display”. Similar, this bean will also be instantiated on demand. The declared instantiation method is the class’s constructor with the reference of the bean of id “gps-locator” as the input argument. In this case, the argument type is GPSLocatorImpl*.

The declaration above not only declared three bean instances, but also wired them in circular dependency as illustrated in the following diagram:



The XML document (named as setup.xml in this example) of the deployment description above looks like follows:

```
<?xml version="1.0"?>
<!DOCTYPE poco-application-context SYSTEM
      "http://www.pocomatic.com/poco-application-context.dtd">

<poco-application-context>

  <load library="./TickGenImpl.$dll"/>
  <bean id="tick-gen"
    class="TickGemImpl"
    lazy-init="false">
    <method-arg type="short" value="10"/>
    <method-arg type="short" value="1"/>
    <ioc method="subscribe">
      <method-arg ref="gps-locator"/>
    </ioc>
  </bean>

  <load library="./GPSLocatorImpl.$dll"/>
  <bean id="gps-locator"
    class="GPSLocatorImpl"
    <ioc method="subscribe">
      <method-arg ref="nav-display"/>
    </ioc>
  </bean>

  <load library="./NavDisplayImpl.$dll"/>
  <bean id="nav-display"
    class="NavDisplayImpl">
    <method-arg ref="gps-locator"/>
  </bean>

  <load library="./reflx.$dll"/>

</poco-application-context>
```

2.2.3 Generating dynamic invocation proxies

Similarly, the dynamic invocation proxies required by the container reflection engine are generated using the PocoCapsule/C++ instrument utility ***pxgenproxy*** from the deployment description file setup.xml as follows:

```
% pxgenproxy setup.xml -h=GPSLocatorImpl.h \
   -h=NavDisplayImpl.h -h=TickGemImpl.h
```

Similar to the previous example, the source code of IoC proxies will be written to the setup_reflx.cc. Compiling and linking them into the reflx.so or reflx.dll file again, then, it is ready to be used by the container.

2.2.4 Running the example

The mini container built and used previously in the Hello example can be reused here without any source code change or rebuild, as it is independent of bean implementations or application deployment description content.

Running the mini container executable main (or main.exe) again, the application will print out the following results:

```
% main
Generate 10 pulses in 10 seconds
current location is {1, 199}
current location is {2, 198}
...
current location is {10, 190}
```

2.3 Use encoded and/or in memory descriptions

2.3.1 Encoding the description

In two examples presented so far, deployment descriptions are provided in the form of external XML document files. This is convenient for learning and testing purpose. However, it has the side effect of forcing the PocoCapsule/C++ IoC continer runtime to load and start its XML parsing/encoding engine dynamically. Although this is performed on demand and transparently to application developers, this XML parsing/encoding engine consumes considerable runtime memory when loaded (~3.5Mbytes) and may be undesirable in many cases.

An alternative form of deployment descriptions is the pre-encoded format. Pre-encoded descriptions are explicitly understandable by the PocoCapsule/C++ IoC runtime without any XML parser.

A XML description can be encoded using the PocoCapsule utility pxencode as follows:

```
% pxencode <xml-file-names>
```

By default, encoded descriptions are written to files with "*_poco.ctx*" suffix and extend name. In this example, the encoded description from input file *setup.xml* will be written out to a new file *setup_poco.ctx*. This encoded description can be used to create application context in the same way of using an XML description. For instance, the mini-container used by this example can be changed to read from *setup_poco.ctx*, instead of

setup.xml:

```
#include <pocoapp.h>

int main(int argc, char** argv)
{
      // create application context
      POCO_AppContext* ctxt
      = POCO_AppContext::create("setup_poco.ctx", "file");

      ...
}
```

Detail usage and feature information about the *pxencode* utility can be found in the last chapter of this document.

Deployment description encoding or equivalently packaging is a common practice in component frameworks. For instance, CCM requires descriptions to be "encoded" into a ZIP file with other components instead of stand alone as a XML text file. EJB and WebServices frameworks, usually package descriptions into a .ear or .war file.

2.3.2 Descriptions as buffer or literal strings

In the examples so far, deployment descriptions, in either XML or encoded format, are all appearing as input files. Practical applications may need other alternatives, such as retrieve descriptions from database, remote connection sockets, dynamically generate them in memory, or even as literal strings embedded in the application executables at compile time.

PocoCapsule/C++ allows all of scenarios above by accepting descriptions in buffers or literal strings regardless. To create an application context from such a in memory buffer or string description, the type id (the 2nd parameter) passed to the *POCO_AppContext::create()* should be "string" as illustrated in the following example:

```
#include <pocoapp.h>

int main(int argc, char** argv)
{
      // a XML descriptor as literal string
      const char* desc = "<?xml ...> ...";

      // create a context from buffer or literal string
      POCO_AppContext* ctxt
      = POCO_AppContext::create(desc, "string");
      ...
}
```

<This is a blank page>

Chapter 3 The Core Schema

Typically, PocoCapsule/C++ deployment descriptions are created as XML documents (or strings) that follow various predefined schemas. The PocoCapsule/C++ framework built-in schema is referred to as the "core" schema. This chapter discusses this built-in core schema, and next chapter will focus on extend schemas and domain specific languages.

In this release, the core schema is defined using DTD rather than the XML Schema. This is because that the core schema is very simple. It is sufficient and intuitive to be comprehended by developers through DTD without depending on additional tools. The complexity of XML Schema would far out weight its benefit.

The URI of this DTD is http://www.pocomatic.com/poco-application-context.dtd.

3.1 The <*poco-application-context*> element

The root element of the core schema is <*poco-application-context*> and is defined as follows:

```
<!ELEMENT poco-application-context (bean|import|load)*>

<!ELEMENT bean (method-arg*,ioc*)>
<!ELEMENT ioc  (method-arg*)>
```

There are only handful XML elements in the core schema. Among them, the following three are most important and constantly used:

- <*bean*> : declares a bean instance, including void bean.
- <*ioc*> : declares a post-instantiation IoC invocation.
- <*method-arg*> : declares an input argument of a constructor, of a factory method or of a post-instantiation IoC invocation.

It is critical to be aware of that a deployment description declares the deployment configuration of an application rather than denotes the step by step execution script on constructing the application out of components. Although each individual XML declaration elements in a deployment description will be translated to some C++ function calls (namely IoC invocations) by the PocoCapsule/C++ container eventually, it should not be assumed that these functions would be invoked in the declared order, declared replicate, or at the deployment time.

Therefore, in the following sections, although each element is described with their implied IoC invocation C++ functions, unless specified explicitly, this XML to C++ implication should be understood as "what" C++ function signatures the container will

invoke on demand, instead of as a script interpretation and assuming "how", when" and "where" these functions would be called.

The circumstances of these functions actually being invoked are going to be discussed throughout this chapter, and will be summarized in the "*bean instantiation management*" section and the "*ref-count and lifecycle control*" section at the end of this chapter.

3.2 The <*bean*> element

A <bean> element declares a bean instance in the application context. This bean instance could be in any of the following categories:

- An instance of a C++ class, namely a POCO object. This is the most typical usage of this element.
- A function pointer (not a class pointer).
- A "void" bean, namely, the bean factory method's return value, if any, is discarded. This is typically used to declare a global or static member function to be called when container instantiates this declared bean.

In general, a <*bean*> element declares the following attributes of the bean:

- The bean's C++ class type (or target type), via the *class* attribute,
- The bean's instantiation method (via the *factory-method* and *factory-bean* attributes), arguments (via <*method-arg*> child elements),
- Instantiation policy and scope, via the *lazy-init* and *singleton* attributes.
- The bean's pre-instantiation dependency, via the *depends-on* attribute,
- The bean's post-instantiation IoC method(s), via <*ioc*> child elements.
- The bean's lifecycle control and reference counting methods, via the *dup-method* and *destroy-method* etc. attributes.

The schema of <*bean*> element is defined as follows:

```
<!ELEMENT bean ((method-arg|ioc|property)*)>
<!ATTLIST bean class                CDATA        #REQUIRED>
<!ATTLIST bean singleton            (true|false) 'true'>
<!ATTLIST bean lazy-init            (true|false) 'true'>
<!ATTLIST bean factory-method       CDATA        IMPLIED>
<!ATTLIST bean factory-bean         CDATA        IMPLIED>
<!ATTLIST bean dup-method           CDATA        IMPLIED>
<!ATTLIST bean self-dup-method      CDATA        IMPLIED>
<!ATTLIST bean destroy-method       CDATA        IMPLIED>
<!ATTLIST bean self-destroy-method CDATA         IMPLIED>
<!ATTLIST bean destroy-bean         CDATA        IMPLIED>
<!ATTLIST bean id                   ID           IMPLIED>
<!ATTLIST bean depends-on           IDREF        IMPLIED>
<!ATTLIST bean abstract             (true|false) 'false'>
<!ATTLIST bean key                  CDATA        IMPLIED>
<!ATTLIST bean label                CDATA        IMPLIED>
```

<*bean*>'s attributes are discussed in the following sections, and <*bean*>'s child elements <*method-arg*>, <*ioc*> will be discussed in later sections.

3.2.1 The *class* attribute

The *class* attribute declares the bean's C++ class type name. This class type is not necessarily the bean's implementation class, if the bean instance is not going to be created from the C++ constructor but from a *factory-method*.

The character '{' and '}' in the *class* attribute value will be converted to '<' and '>' respectively by PocoCapsule/C++ framework. Therefore, for instance, a *class* attribute value *"std::map{int, int}"* implies *"std::map<int, int>"*.

The *class* attribute can take the following values:

- A normal class name as well as its typedef, such as *"std::fstream"*, *"MySessionBeanImpl"*, *"std::map{std::string, std::string}"*. In this case, if the *factory-method* is declared, the factory must return a pointer that can be implicitly type casted to a pointer of the declared class.
- A de-referenced typedef'ed pointer class name, such as: *"*FuncPtr"* (deref of a typedef'ed function pointer), *"*CORBA::Object_ptr"* (deref of a typedef'ed pointer for portable purpose). In this case, this <*bean*> has to declare a *factory-method*. This *factory-method* must return a pointer that can be implicitly typecasted to the declared typedef'ed pointer.
- *"void"*: in this case, the return value from the *factory-method*, if any, will be discarded. A *"void"* bean can not be used as *factory-bean* or *destroy-bean* of another <*bean*>, or *ref* of a <*method-arg*> However, a *"void"* bean itself can have a *depends-on* bean, <*ioc*> methods, and can also be the *depends-on* bean of

other beans. Typically, this can be used to declare multiple pre-instantiation <*ioc*> methods for a given bean.

Some related discussion can be found in the section on the *factory-method* attribute.

Here are some examples of the *class* attribute:

```
<bean class="std::map{std::string, std::string}"/>

<bean class="*MyFuncPtr"
      factory-method="MyReflection::getMethod"/>

<bean class="void"
      factory-method="printf"/>
```

3.2.2 The *singleton* attribute

In this release, PocoCapsule/C++ supports two bean scope policies, namely *singleton* and *auto* (also referred to as *non-singleton* in this document). They are indicated by the boolean value ("true" and "false" respectively) of this attribute.

A declared singleton bean will be instantiated at most once in its application context and will be shared by all references. A declared *auto* or *non-singleton* bean, on the other hand, instantiates a separate instance for each of its reference.

The term "*singleton*" in IoC container is different from its meaning in the notorious *singleton design pattern*. In the singleton design pattern a singleton is defined as the only possible instance of a class. In IoC frameworks, a singleton is defined as the only possible instance of a declared bean rather than of a class. A class in an IoC framework can be used to declared an arbitrary number of singleton beans and hence end up with multiple instances. Conceptually, singleton vs auto (non-singleton) beans in IoC frameworks is similar to global/static/heap vs local/auto/stack object instances in C++ program.

In PocoCapsule/C++, by default, beans are singleton regardless they were declared standalone or as inner beans. This slightly differs from Java IoC containers and makes more sense for C++ applications. To declare a non-singleton bean, its *singleton* attribute has to be set to "false" explicitly.

Singleton and non-singleton beans follow different lifecycle controls. More discussion can be found in the "*bean instantiation management*" section and the "*bean ref-count and lifecycle control*" section at the end of this chapter.

This attribute is ignored for abstract beans (see the *abstract* attribute of <bean> element).

3.2.3 The *lazy-init* attribute

This attribute is optional and is "true" by default. This attribute indicates whether the (singleton) bean is lazy or eager. A lazy singleton bean will only be instantiated on demand, while an eager singleton bean will be instantiated when the initSingletons() is called on the deployed application context.

This attribute is ignored for non-singleton beans (see the *singleton* attribute of <bean> element).

3.2.4 The *factory-method* and *factory-bean* attribute

These attributes specify the factory method and factory bean used to instantiate the bean. If the *factory-method* attribute is not specified, PocoCapsule/C++ container will try to use bean's C++ constructor to create the instance. Namely, the default *factory-method* value is *"new MyBeanClassName"*. Here, the *MyBeanClassName* is the value of the <bean>'s *class* attribute. If the *factory-method* attribute is declared and the <bean>'s *class* attribute is not "void", then the return type of the factory method should be able to be implicitly type casted to the declared pointer type.

The character '{' and '}' in the *factory-method* attribute value will be converted to '<' and '>' respectively by PocoCapsule/C++ framework. Therefore, the *factory-method* value of "*func{const char*, NULL}*" implies "*func<const char*, NULL>*" semantically.

Also, if the *factory-bean* is specified, the *factory-method* will be invoked as (non-static) member function of the specified *factory-bean* instance. Otherwise, the *factory-method* is treated as a global or static member function or operator.

For example, the following stanza declares three beans with and without *factory-method*:

```
<bean class="Worker" id="worker-1"/>

<bean class="Worker" id="worker-2"
      factory-method="Worker::create"/>

<bean class="Worker" id="worker-3"
      factory-bean="worker-pool"
      factory-method="getWorker"/>

<bean class="Worker" id="worker-4">
```

It implies the following IoC bean instantiation methods:

```
// instantiate worker-1: via constructor
Worker* w1 = new Worker;

// instantiate worker-2: via static factory method
Worker* w2 = Worker::create();

// instantiate worker-3: via factory method on factory bean
Worker* w3 = pool->getWorker();
```

3.2.5 The *dup-method* and *self-dup-method* attributes

The *dup-method* can be used to manipulate reference counting. See the section of "*bean ref-count and lifecycle control*" at the end of this chapter for more detail.

Similar to the *factory-method* attribute, the character '{' and '}' in the *dup-method* or the *self-dup-method* attribute value will be converted to '<' and '>' respectively by PocoCapsule/C++ framework.

The difference between the *dup-method* and the *self-dup-method* is on their calling syntax:

- A *dup-method* will be called as a static member or global function with the target bean's pointer as an argument. Then, the returned pointer, which could point to a new bean, will be casted to a pointer of the type same as the original target and used as the duplication result.
- A *self-dup-method*, however, will be called as a member function on the target bean itself without an argument. Then, the target bean itself will be used as the duplication result.

These are illustrated by the following example:

```
<bean class="A" id="my-a" dup-method="A::_duplicate"/>

<bean class="B" id="my-b" self-dup-method="add_ref"/>
...

<bean ...>
  <ioc method="foo" target="none">
     <method-arg pass="dup" ref="my-a"/>
  </ioc>
  <ioc method="bar" target="none">
     <method-arg pass="dup" ref="my-b"/>
  </ioc>
</bean>
```

The above declaration stanza implies the following IoC invocation scenario:

```
A* a = ...;
B* b = ...;

...

// the retuned ptr from the dup-method is passed
foo((A*)A::_duplicate(a));

// self-dup, and then, pass the ref itself.
b->add_ref();
bar(b);
```

3.2.6 The *destroy-method, self-destroy-method* and *destroy-bean* attributes

These attributes specify the bean's destroying methods. These methods can be used for reference count as well as lifecycle control and are summarized in the "*bean ref-count and lifecycle control*" section at the end of this chapter.

Similar to the *factory-method* attribute, the character '{' and '}' in the *destroy-method* or the *self-destroy-method* attribute value will be converted to '<' and '>' respectively by PocoCapsule/C++ framework.

The difference between the *destroy-method* and the *self-destroy-method* is on their calling syntax:

- A *destroy-method without a destroy-bean* will be called as a static member or global function with the target bean's pointer as the argument.
- A *destroy-method with a destroy-bean* will be called as a member function of the destroy-bean with the target bean's pointer as the argument.
- A *self-destroy-method*, however, will be called as a member function on the target bean itself without an argument.

These are illustrated by the following example:

```
<bean class="A" id="a-1" destroy-method="A::_release"/>

<bean class="A" id="a-2" destroy-bean="registry"
                         destroy-method="unbind"/>

<bean class="A" id="a-3" self-destroy-method="stop"/>
```

The above declaration stanza implies the following IoC invocation scenario:

```
// destroy the bean of id "a-1"
A* a1 = ...
A::_release(a1);

// destroy the bean of id "a-2" by destroy-bean
A* a2 = ...
registry->unbind(a2);

// self destroy the bean of id "a-3"
A* a3 = ...
a3->stop();
```

A *destroy-method* is not necessary to be the bean's destructor or reference count reducer. An application can choose whatever method as the *destroy-method* as long as it has the required signature and desired outcome. For instance, a *destroy-method* can serve as the last cleanup procedure to unsubscribe the bean from a publish/subscribe event service or deregister the bean from an external directory service.

3.2.7 The *id* attribute

The *id* attribute is optional. It assigns a name to the given singleton bean. This name scopes within the local description and can be used to refer this bean elsewhere within the description (namely, the same XML document). For instance:

- To be referred as value of the *depends-on* attribute of another bean element (see the *depends-on* attribute).
- To be referred as value of the *ref* attribute of <*method-arg*>.
- To be referred as value of the *factory-bean* attribute of another bean element
- To be referred as value of the *destroy-bean* attribute of another bean element

In the following description stanza, the bean's id "foo" is referred in four of the listed cases:

```
<bean class="A" depends-on="foo"/>

<bean class="B" id="foo"/>

<bean class="C">
  <method-arg ref="foo"/>
</bean>

<bean class="D"
  factory-bean="foo-bean"
  factory-method="create"
  destroy-bean="foo-bean"
  destroy-method="destroy"/>
```

The stanza above implies the following IoC invocation scenarios:

```
B* f = new B;          // foo must be initiated before b

A* a = new A;          // This must be made after the foo-bean

C* c = new C(f);       // foo is ref'ed as input argument

D* d = f->create();    // foo is ref'ed as factory-bean

...

f->destroy(d);         // foo is ref'ed as destroy-bean
```

Also, a bean's *id* can be used to retrieve the bean instance from the application context using getBean(const char* id) method, for instance:

```
POCO_AppContext* ctxt
      = POCO_AppContext::create(...);

B* b = (B*)(ctxt->getBean("foo"));
```

3.2.8 The *depends-on* attribute

This attribute is optional. If specified, its value should refer to the id of another bean declared in the local deployment description. This declares that the being referred bean should be instantiated before this bean. See the example for the *id* attribute.

3.2.9 The *abstract* attribute

The default value of this attribute is "false". If set to "true", the bean will never be instantiated.

3.2.10 The *label* attribute

The *label* attribute is optional. It is only used by utility tools or the container to pin-point the location of an error.

3.3 The <*ioc*> and <*property*> element

An <*ioc*> element should be used as child elements of a <*bean*> element to declare a *post-instantiation* IoC invocation on the specified bean. This invocation is to be made right after the bean's instantiation. The schema of <*ioc*> element is defined as:

```
<!ELEMENT ioc (method-arg*)>
<!ATTLIST ioc method CDATA                  #REQUIRED>
<!ATTLIST ioc target (this-bean|none)       #IMPLIED>
```

A <*bean*> element many have an arbitrary number of <*ioc*> child elements. The order of these post-instantiation IoC invocations is the same as the order of their declaration. For instance:

```
<bean class="A">
  <ioc method="m_pulse_rate=">
    <method-arg type="short" value="10"/>
  </ioc>
  <ioc method="start"/>
</bean>
```

In the description above, the bean is declared with two <ioc> elements. It means the specified two IoC invocations should be made subsequent to the instantiation of the bean. Conceptually as:

```
short rate = 10;
A* a = new A;
// immediately after a's instantiation
a->m_pulse_rate = rate; // 1st method is "m_pulse_rate="
a->start();             // 2nd method is "start"
```

The *method* attribute specifies the method name of the invocation. Similar to the *factory-method* attribute of <*bean*> element, the character '{' and '}' in <*ioc*> element's *method* attribute value will be converted to '<' and '>' respectively by PocoCapsule/C++ framework.

The *target* attribute specifies the target of this IoC invocation. For post instantiation IoC of a void bean (namely, a bean with its *class* attribute equals to "*void*"), the setting of the *target* attribute is ignored and the IoC will always be treated as as global or static function (i.e. *target= "none"*). For post instantiation IoC of a non-void bean, the implied target value is "*this-bean*", namely, the invocation will be treated as an non-static member function on the bean itself. For instance:

```
<bean class="void">
  <ioc method="foo1"/>
</bean>

<bean class="A">
  <ioc method="foo2"/>
  <ioc method="foo3" target="none"/>
</bean>
```

This declaration implies the following post instantiation IoC invocation scenarios:

```
foo1();     // void bean, no target,
            // as global/static function

A* a = new A;   // bean instantiation
a->foo2();  // as member function on this bean target
foo3();     // no target, as global/static function
```

The <*method-arg*> elements, specifies the invocation arguments. See the section about the <*method-arg*> element.

In additional to the <*ioc*> and <*method-arg*>, the <*bean*> element has another child element, namely the <*property*>. In most Java IoC frameworks, <*property*> elements declare post-instantiation bean's configuration through JavaBean style setters. In C++, however, a setter method has nothing special compared to other post-instantiation member functions of the bean. Furthermore, various C++ frameworks have their own and different setter (or property) conventions, such as with or without the "set" prefix. Therefore, to avoid tying PocoCapsule/C++ to a particular convention, the <*property*> is up to users to provide their own favorite definition in their own extended grammar. This can easily be done through the DSL mechanism built in PocoCapsule/C++ framework. See the chapter about DSL as well as the ${POCOCAPSULE_DIR}/examples/basic-ioc/ext-schema example.

3.4 The <*method-arg*> element

The <*method-arg*> elements declare input arguments of constructor, factory and/or post-instantiation ioc methods.

The schema of the <*method-arg*> element is defined as:

```
<!ENTITY % poco-basic-types
            'char|uchar|short|ushort|long|ulong|
             float|double|string|cstring|bean'>

<!ELEMENT method-arg (bean|array|
                      this-bean|ref|map|set|list)?>
<!ATTLIST method-arg type  (%poco-basic-types;|array)
                                              'bean'>
<!ATTLIST method-arg value CDATA              #IMPLIED>
<!ATTLIST method-arg ref   IDREF              #IMPLIED>
<!ATTLIST method-arg pass  (ptr|dup|deref)    'ptr'>
<!ATTLIST method-arg class CDATA              #IMPLIED>
<!ATTLIST method-arg env-var CDATA            #IMPLIED>
```

All <*method-arg*> elements directly under a <*bean*> or <*ioc*> element declare the input arguments of the bena constructor/factory method or of the post-instantiation ioc method respectively.

For instance, in the following example, the bean's constructor A() has two arguments, and the post-instantiation ioc method start() also has two arguments. And the second argument of the start() method invocation happens to be the reference of the just instantiated bean itself:

```
<bean id="foo" class="A">
  <method-arg type="short" value="123"/>
  <method-arg type="string" value="abc"/>
  <ioc method="start">
    <method-arg type="long" value="789"/>
    <method-arg ref="foo"/>
  </ioc>
</bean>
```

This description implies the following scenario:

```
// constructor with two arguments
A* foo = new A((short)123, (const char*)"abc");
// post-instantiation method also has two argument.
foo->start ((long)789, (A*)foo);
```

An alternative and anonymous way to indicate the just instantiated bean as an argument of a post-instantiation <*ioc*> invocation is using <***this-bean***> as child element of <*method-arg*> of the <*ioc*>. For instance, in the following example, two post-instantiation invocations are declared as:

```
<bean class="std::map{std::string, std::string}">
  <ioc method="add_to_map" target="none">
    <method-arg><this-bean/></method-arg>
    <method-arg type="string" value="John W."/>
    <method-arg type="string" value="1 (408) 222-3333"/>
  </ioc>
  <ioc method="add_to_map" target="none">
    <method-arg><this-bean/></method-arg>
    <method-arg type="string" value="Jerry H."/>
    <method-arg type="string" value="1 (650) 888-1234"/>
  </ioc>
</bean>
```

This implies the following IoC invocation scenario:

```
std::map<std::string, std::string>* yp;
yp = new std::map<std::string, std::string>;

::add_to_map(yp, "John W.", "1 (408) 222-3333");
::add_to_map(yp, "Jerry H.", "1 (650) 888-1234");
```

3.4.1 The *type* attribute

The *type* attribute of <*method-arg*> declares the type of the input argument. The default value of this attribute is *"bean"*, indicating that the input argument is a pointer or reference (depends on the *pass* attribute) of a class. The valid values of this attribute and corresponding argument types are listed in the following table:

Value of type attribute	*Argument C++ type*	*Default Value*	*Note*
"char"	char	'\0'	
"uchar"	unsigned char	'\0'	
"short"	short	0	
"ushort"	unsigned short	0U	
"long"	long	0L	
"ulong"	unsigned long	0UL	
"float"	float	0.0	
"double"	double	0.0	
"string"	const char*	NULL	
"cstring"	const char*	NULL	The literal text of '\n', '\t' etc. in the XML will be converted to their C++ characters.
"bean" (default)	the_bean_class*	NULL	pass="ptr" (default), or pass="dup"
	the_typedef_ptr		
	the_bean_class&	N/A	pass="deref"

"array"	int length, array_type array[]	0, NULL	The first argument is the array length, the second argument is the array. Its type is declared by the *type* attribute of the <array> element. It can be all types listed above, as well as the namevalue string pair.

3.4.2 The *value* attribute

The *value* attribute specifies the value of an argument with a type other than "*bean*" and "*array*". If this attribute is not specified, the default value of the given type listed in above table will be applied.

3.4.3 The *ref* attribute and <*ref*> element

The value of the *ref* attribute of a <*method-arg*> element should equal to the id of another <*bean*> element declared in the local description (namely, the same XML document). This declares that the bean being referenced is to be used as the input argument. The *type* attribute of this <*method-arg*> element should either be left-out or explicitly set to *"bean"*.

The <*ref*> element as a child element of <method-arg> is a verbose equivalent of the *ref* attribute. The element itself is defined as:

```
<!ELEMENT ref EMPTY>
<!ATTLIST ref bean   IDREF          #IMPLIED>
```

The *bean* attribute of the <*ref*> element is an id of another <*bean*> element declared in the local description.

3.4.4 The *pass* attribute

The *pass* attribute of a <*method-arg*> specifies the argument passing syntax when its *type* attribute is specified as "*bean*" or is left-out (implies *"bean"*). The default passing syntax is "*ptr*", implying passing by-pointer. If the *pass* equals to "*dup*", the duplicated bean will be passed by-pointer. If the *pass* equals to "*deref*", it implies the de-referenced bean is by-reference or by-value. This attribute is ignored if the *type* attribute of the <*method-arg*> is specified other than "*bean*".

```
<bean id="a" class="A" dup-method="A::_duplicate"/>

<bean class="void" factory-method="foo">
  <method-arg ref="a"/>
</bean>

<bean class="void" factory-method="foo">
  <method-arg ref="a" pass="dup"/>
</bean>

<bean class="void" factory-method="foo">
  <method-arg ref="a" pass="deref"/>
</bean>
```

The declaration stanza above implies the following IoC invocation scenario:

```
A* a = new A;

foo(a);                  // passing by-pointer

foo(A::_duplicate(a));   // dup and then passing by-pointer

foo(*a);                 // passing the deref.
```

More discussions on the semantics of the *dup* attribute can be found in the *bean's ref-count and lifecycle control* section.

3.4.5 The *class* attribute

The *class* attribute of a <*method-arg*> specifies the class of a bean type argument. This is especially useful when the intention is pass a NULL pointer of that class. For instance:

```
<bean class="void" factory-method="foo">
  <method-arg class="B"/>
</bean>
```

The declaration stanza above implies the following IoC invocation scenarios:

```
foo((B*)NULL);
```

3.4.6 The *env-var* attribute

This attribute is ignored when the *type* attribute of the <*method-arg*> is "bean" or "array". The *env-var* attribute of a <*method-arg*> specifies that the container first looks

for the argument literal value from the context's creation application environment. For instance, assuming the application context is created from a description (e.g. setup.xml document) as follows:

```
POCO_AppEnv* env = ...;
env->setValue("server-name", "the-bank-account-server");
env->setValue("max-thread", "32");
...
ctxt = POCO_AppContext::create(
                  "setup.xml", "file", env);
...
```

A bean is declared in the description (i.e. the setup.xml file in this example) as:

```
<bean class="MyServant" ...>
      <method-arg type="string" env-var="server-name"/>
      <method-arg type="short" env-var="max-thread"/>
</bean>
```

Then, effectively, the bean will be instantiated in the following constructor calling syntax:

```
bean = new MyServant("the-bank-account-server", 32);
```

3.5 The <*array*> and <*item*> elements

An <*array*> element declares an array input argument of a constructor, factory method or post-instantiation ioc method. Items in a given array all have the same type or will be casted to the same class pointer type. This type is referred to as the array's type and is declared by the *type* and/or *class* attributes of the <*array*> element similar to what were used for <*method-arg*>. All basic types and bean class pointers can be used as item types. Arrays also support a built-in string pair, namely the "*namevalue*", as item type. This *namevalue* type provides an alternative solution of the <*props*> argument type.

```
<!ENTITY % poco-basic-types
            'char|uchar|short|ushort|long|ulong|
             float|double|string|cstring|bean'>

<!ELEMENT array (item)*>
<!ATTLIST array type  (%poco-basic-types;|namevalue)
                                    #REQUIRED>
<!ATTLIST array class CDATA         #IMPLIED>
<!ATTLIST array env-var CDATA       #IMPLIED>

<!ELEMENT item  (bean)?>
<!ATTLIST item  name  CDATA         #IMPLIED>
<!ATTLIST item  ref   IDREF         #IMPLIED>
<!ATTLIST item  value CDATA         #IMPLIED>
<!ATTLIST item  env-var CDATA       #IMPLIED>
```

3.5.1 Array argument passing convention

A declared <*array*> argument will actually be passed to an IoC invocation as two arguments. The first argument is a C/C++ integer (int) that indicates the item count of the array. The second argument is the array itself.

For instance, the following stanza declares an array of strings of 4 items:

```
<bean class="void" factory-method="A::add_cities">
  <method-arg type="array">
    <array type="string">
      <item value="San Francisco"/>
      <item value="Palo Alto"/>
      <item value="Cupertino"/>
      <item value="San Jose"/>
    </array>
  </method-arg>
</bean>
```

The declaration stanza above implies the following IoC invocation scenarios:

```
const char* arr[] = {
      "San Francisco",
      "Palo Alto",
      "Cupertino",
      "San Jose"};

A::add_cities(4, arr);
```

3.5.2 The *env-var* attribute of <*array*>

The *env-var* attribute is ignored when the *type* attribute of the <*array*> is "bean" or "namevalue". The *env-var* attribute of a <array> specifies that the container first looks for the array literal value from the context's creation application environment. For instance, assuming the application context is created from a description (e.g. the setup.xml file) as follows:

```
int main(int argc, char** argv)
{
      ...
      POCO_AppEnv* env = ...;
      env->setArray("main-argv", argc, argv);
      ...
      ctxt = POCO_AppContext::create(
                  "setup.xml", "file", env);
      ...
```

A bean is declared in the description (i.e. the setup.xml file in this example) as:

```
<bean class="*CORBA::ORB_ptr"
      factory-method="CORBA::ORB_init">
      <method-arg type="array">
            <array type="string" env-var="main-argv"/>
      </method>
</bean>
```

Then, effectively, the bean will be instantiated in the following calling syntax:

```
bean = CORBA::ORB_init(argc, argv);
```

3.5.3 The *name*, *value*, *ref*, and *env-var* attributes of the <*item*>

The *name* attribute specifies the name of a name-value pair item. This attribute will be ignored if the *type* attribute of the <*item*>'s <*array*> container element is not "*namevalue*".

The *value* attribute specifies the literal value of the item, if the *type* attribute of the containing <*array*> is not a "*bean*".

The *ref* attribute refers a local bean's *id* if the *type* attribute of the containing <*array*> is a "*bean*".

The *env-var* attribute of a <*item*> specifies that the literal value of the item should first be looked from the context's creation application environment. This attribute is ignored if the *type* attribute of the containing <*array*> is "bean".

3.6 The <*map*>, <*set*> and <*list*> elements

In the PocoCapsule/C++ core schema, <map>, <set> and <list> are referred to as child elements of <method-arg>, however, without being actually defined based on the following considerations:

- Unlike Java or other OO languages, C++ classes do not necessary root from a single class hierarchy. Defining a generic <*map*> (or <*set*>, <*list*>) in PocoCapsule/C++ core schema would force its key and value types to use (void*), as this is the only generic type in C++. This would undermine type-safety and is considered to be a poor practice in general.
- To support <*map*> in the core schema, the container would either undermine the POCO premise (by defining a proprietary map class) or would tie itself to STL and C++ template (by using std::map template as the class).
- Although map, set, and list are highly useful aggregate types in imperative programming, they are actually much less critical argument types in declarative deployment descriptions. Therefore, it is not worthwhile to sacrifice the type-safety, the POCO premise, and decoupling from C++ STL/template etc. merely for this syntactic sugar.

Finally, similar to the <*property*> element, the elements above are not concretely defined in the core schema but preserved for users to provide customized definitions in extended schemas through the PocoCapsule/C++ framework's DSL mechanism. See the chapter about DSL as well as the ${POCOCAPSULE_DIR}/examples/basic-ioc/ext-schema example.

3.7 Bean instantiation management

In PocoCapsule/C++, bean instantiations are mostly made on demand except *eager singletons* that can be instantiated when *initSingletons()* is invoked on the application context (see application context API). Details of on-demand scenarios vary depending on beans' scope (singleton or non-singleton) and laziness (lazy or eager). The situations that trigger a bean's instantiation are summarized as follows:

- On invoking the *initSingletons()* method on an application context, the container walks though the list of eager singletons in their declared order and instantiates those eager singletons that are not instantiated yet.
- Before instantiating another bean that *depends-on* this *singleton* bean.
- The *getBean()* method is invoked on the application context, with this bean's *id* as the argument.
- Before instantiating another bean that uses this bean as its *factory-bean*, *destroy-bean*, or an input argument of its constructor or factory method.

- Before calling a post-instantiation ioc method that refers this bean as an input argument.

If a destroy bean itself is a singleton, it will be instantiated before instantiating the to be destroyed bean, instead of before performing the destroy task. This is because that the singleton beans' destroying order on *terminate()* is reverse to their instantiation order, namely in the first-in-last-out order. Therefore, to ensure the destroy-bean is still alive on calling for its duty, it is instantiated early.

3.8 Bean ref-count and lifecycle control

A bean's ref-count and lifecycle control methods are declared by its *dup-method*/*self-dup-method* and *destroy-method/self-destroy-method/destroy-bean* attributes.

Typically, the *dup-method* or *self-dup-method* attribute is declared in application deployment descriptions to ensure the involved beans' reference counts will be increased on certain circumstances listed in the table below.

The *destroy-method* or *self-destroy-method*, on the other hand, could be declared in order to have the bean's reference count reduced, or the bean instance released (especially for non-singleton) after use. However, as discussed previously, a *destroy-method* or *self-destroy-method* is not necessarily to be the bean's destructor or reference count reducer. An application deployment can choose whatever method, which has the required signature and meet its specific semantic requirement, as the *destroy* or *self-destroy* method. For instance, a destroy method can serve as the last cleanup procedure to stop a bean's internal threads, unsubscribe it from a publish/subscribe event service, and/or unregister it from an external directory service, but not necessarily to delete the instance.

The circumstances where these methods would be called depend on the *singleton* and the *pass* attributes of <method-arg>. These are summarized in the following table:

scenarios	*dup-method or self-dup-method*	*destroy-method or self-destroy-method*
A singleton is used as a method-arg.	ignored (by default), unless pass="dup"	ignored. Note: the destroy-method or the self-destroy-method will only be called when terminate() is called on the application context.
A singleton to be returned from getBean() of the application context.	The dup-method or the self-dup-method will be called and the result or the bean itself is returned.	
A non-singleton is used as an method-arg.	ignored (by default), unless pass="dup"	The destroy-method or the self-destory-method will be called following the invocation.
A non-singleton is returned from getBean() of the application context.	Ignored	

3.9 The <*import*> element

In the core schema, a <*import*> element declares a sub-context of the root application context. This element is defined as:

```
<!ELEMENT import EMPTY>
<!ATTLIST import resource CDATA     #REQUIRED>
```

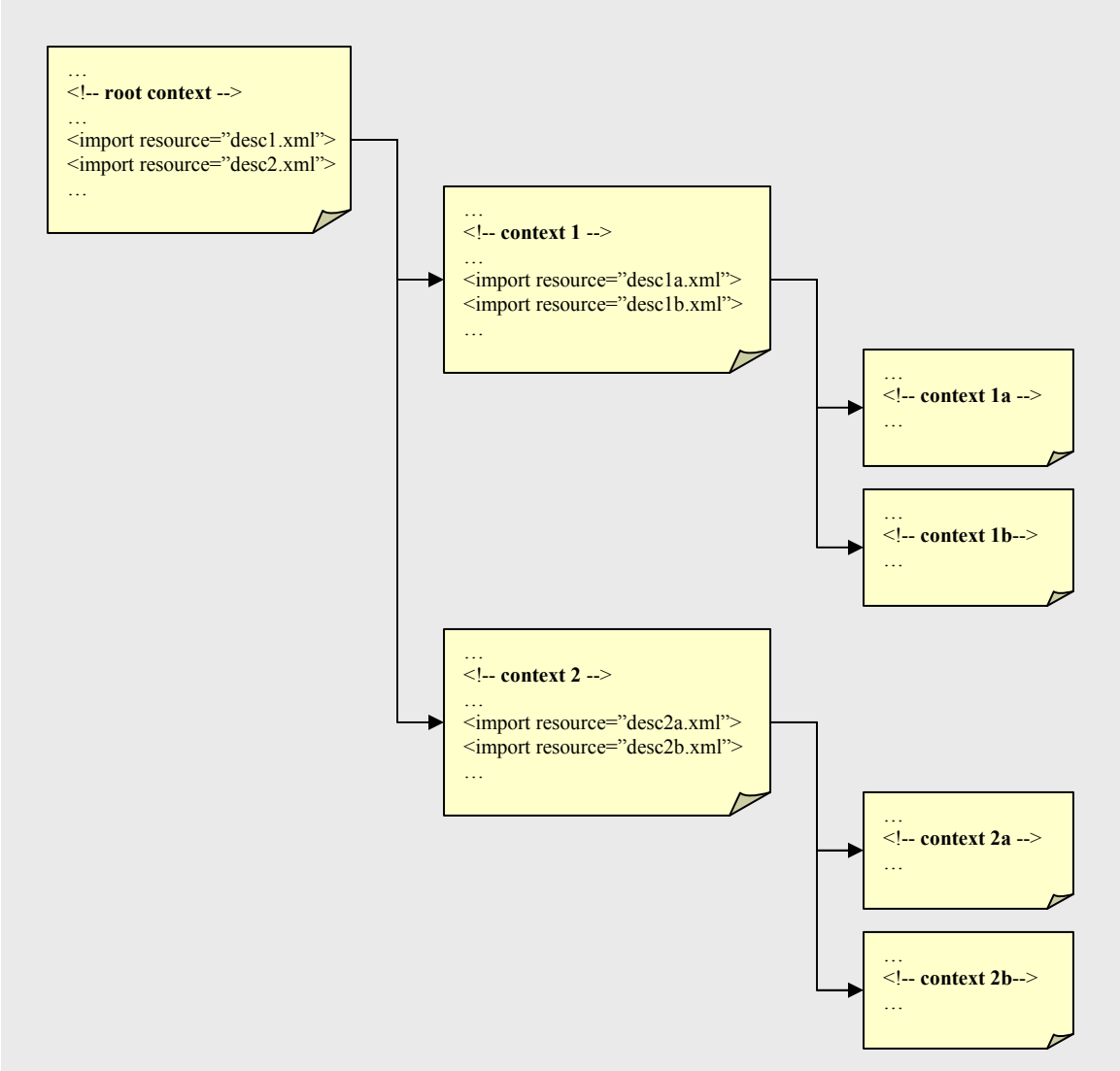
The *resource* attribute of this element specifies the file name of the external application context description (a XML or encoded document).

An application context and its sub-contexts are in separated id scopes. The only semantic rules on <*import*>'ed context are:

- *initSingleton()* on a application context: PocoCapsule first instantiates eager singletons in all imported application contexts by their declared order, before instantiating eager singletons in the current application context.
- *terminate()* on the application context: PocoCapsule will first destroy all singletons in the current context, before recursively destroying singletons in the imported contexts.

For instance, in the context importing hierarchy illustrated in the next diagram, the order of bean instantiation or destroying is as follows:

- Order of *initSingletons()* by contexts : *1a, 1b, 1, 2a, 2b, 2, root.*
- Order of *terminate()* by contexts : *root, 1, 1a, 1b, 2, 2a, 2b.*



As imported eager singletons are instantiated first, therefore, importing contexts can be used as a mechanism to set up application specific system services or facilities, for instance, to initiate transaction or security engine for the main application context.

3.10 The <load> element

<load> elements declare dynamic libraries to be loaded at application creation time. This element is defined as:

```
<!ELEMENT load EMPTY>
<!ATTLIST load library CDATA  #REQUIRED>
```

The *library* attribute of this element specifies the path or file name of the shared library or dll. On Unix/Linux and Windows, if only a file name is specified, the container will

look for the library from the environment specified loading directories (e.g. $LD_LIBRARY_PATH on Linux/Solaris, and %path% on windows) to resolve the file.

Also, the library file name specified by the *library* attribute can use "$dll" as extension for portability. PocoCapsule/C++ container will convert this extension name, or the substring "${dll}" in general, into the conventional shared library or dll extension name of the OS platform, such as "so", "dll".

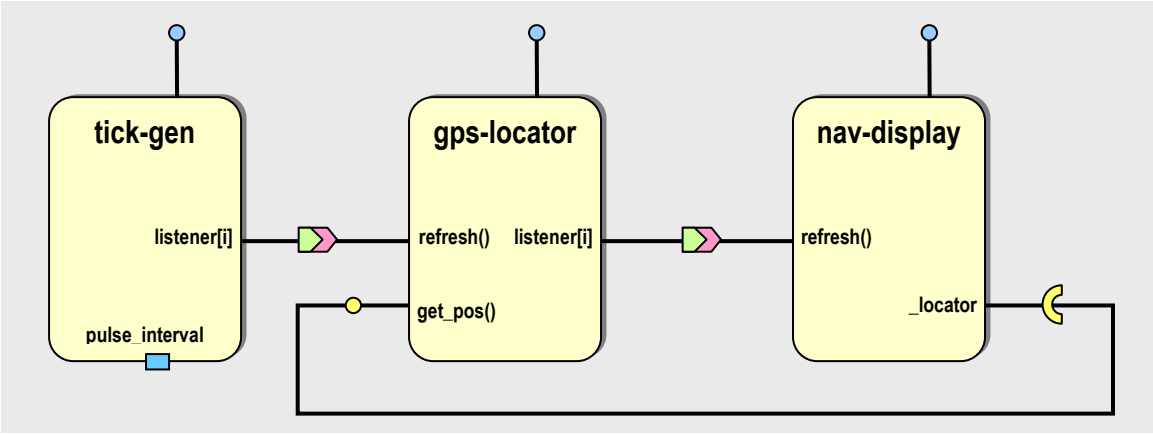
<This is a blank page>

Chapter 4 User Defined Schemas and Domain Specific Modeling

4.1 Why DSM/DSL in IoC frameworks

IoC technology shifts the plumbing complexities of component based softwares into dynamically reconfigurable IoC frameworks and declarative application descriptions. This allows application developers to focus on implementing high level business logic and flexibly assemble up their applications without relying on contrived low level patterns, assuming application descriptions are reasonably expressive.

As illustrated in previous chapters, application descriptions in core schemas of IoC frameworks are more or less plain C/C++ application code expressed in XML. For instance, a simple GPS device consists of three components as illustrated in the following diagram:



As the example of section 2.2, this device can be set up in the following XML stanza[19]:

[19] This XML description is used in the GPS example located in the examples/basic-ioc/gps directory of a PocoCapsule/C++ installation.

```
<poco-application-context>
  <bean id="tick-gen" class="TickGenImpl" lazy-init="false">
    <method-arg type="short" value="10"/>
    <method-arg type="short" value="1"/>
    <ioc method="subscribe">
      <method-arg ref="gps-locator"/>
    </ioc>
  </bean>

  <bean id="gps-locator" class="GPSLocatorImpl"
    <ioc method="subscribe">
      <method-arg ref="nav-display"/>
    </ioc>
  </bean>

  <bean id="nav-display" class="NavDisplayImpl">
    <method-arg ref="gps-locator"/>
  </bean>
  ...
</poco-application-context>
```

This description conceptually implies the following *imperative* C++ code segment:

```
// instantiate the position locator
GPSLocator* locator = new GPSLocatorImpl;

// instantiate a position display,
// with ref of the locator to retrieve position data.
NavDisplay* display = new NavDisplayImpl(locator);

// subscribe the display to locator - to receive position
// change notification.
locator->subscribe(display);

// instantiate the tick generator, with appropriate
// configure setting: the 1st arg is the count of ticks,
// the 2nd arg the is interval (seconds) between ticks.
TickGen* tick_gen = new TickGenImpl(10, 1);

// subscribe the locator to the tick generator, to
// receive the position pulling interval tick.
tick_gen->subscribe(locator);

// start the gps device
tick_gen->start();
```

These IoC core schemas have the advantage of being very straightforward and generic to virtually all problem domains. However, this generic simplicity has the weakness of being less expressive, over verbose, and error prone for large scale applications[20].

To improve the expressiveness and conceal low level programming models, the core IoC framework could raise its abstraction level, for instance, by adding problem specific vocabularies and higher level syntax structures into the core IoC schema. With such an enhancement, the same GPS device setup could be declared in the following XML stanza[21]:

```
<gps-device>
  <tick-generator use="TickGenImpl" count="10" interval="1"/>
  <gps-locator use="GPSLocatorImpl"/>
  <navigation-display use="NavDisplayImpl"/>
</gps-device>
```

For other applications, such as Web Services and SOA, software define radios (SDR), robot vehicles, video games, high performance computing (HPC), telecomm network management (TMN), intelligent networks (IN/AIN), CORBA POA servers, OMG Notification/DDS participants, we could keep adding vocabularies and syntax structures into the core IoC schema. Eventually, the accumulated schema could be expressive for pretty generic applications, however awfully bloated and complex. This is the dilemma of *expressiveness and diversity vs. intuitiveness and simplicity*.

As a solution to this dilemma, instead of enforcing a bloated one-size-fits-all generic schema predefined by the framework, PocoCapsule allows and encourages users or third parties to extend or even redefine the core schema into different customized variations for their corresponding problem domains. These user defined schema variations are referred to as *domain specific modeling* or *domain specific language* (DSM or DSL). As each of these schemas are only designed for one specific problem domain, they are able to be simple, intuitive while still highly expressive with a relative small set of problem specific vocabularies and high level syntax structures.

Modeling application high level setup patterns with DSM/DSL is similar to modeling recurring business entities (objects) with classes. A class in OO programming languages is effectively a schema that captures common naturals of a category of objects. Instead of defining a single one-size-fits-all bloated class or predefining a huge number of classes to model vast diverse high level objects, OO languages allow and encourage developers to define and use user defined classes as abstractions of their specific business objects.

IoC technology was thought to be no more than a design pattern and yet another way of wiring components through dependency injection (DI). This is reflected from the attempt of using Java *annotation* based DI to eliminate the XML based declarative description,

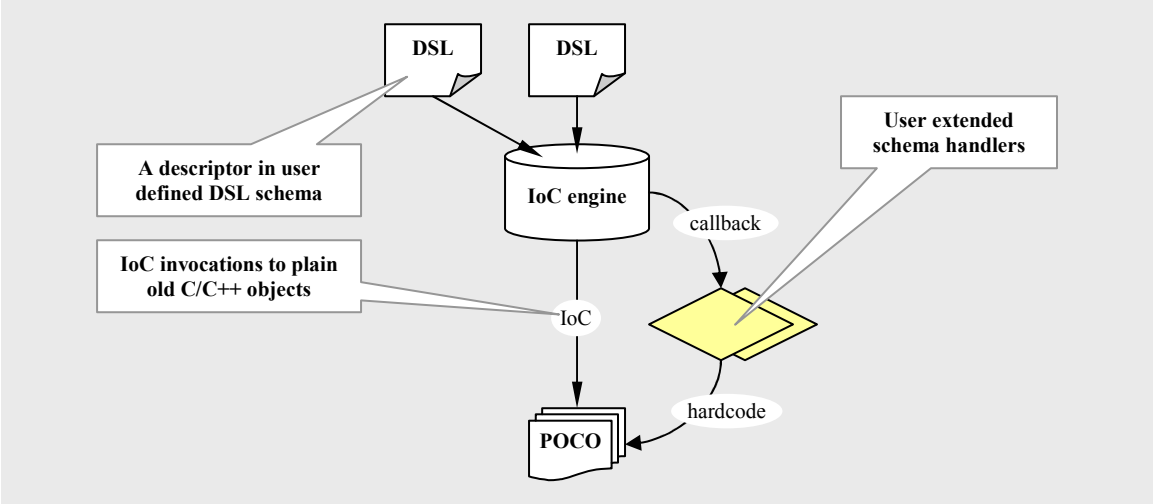
[20] This is one example of the so-called "XML Hell".
[21] This XML description is used in the DSM GPS example located in the examples/basic-ioc/dsm-gps directory of PocoCapsule/C++ installation.

the constant debate of constructor vs setter injection, and the slogan of "*I was expecting a paradigm shift, and all I got was a lousy constructor*".

Nevertheless, IoC is far more than the DI pattern/principle and is the most effective and seamless bridge between the legacy imperative programming paradigm and the declarative DSM realm where tooling and high level abstraction/customization become much more convenient.

4.2 Facilitating DSM/DSL in IoC frameworks

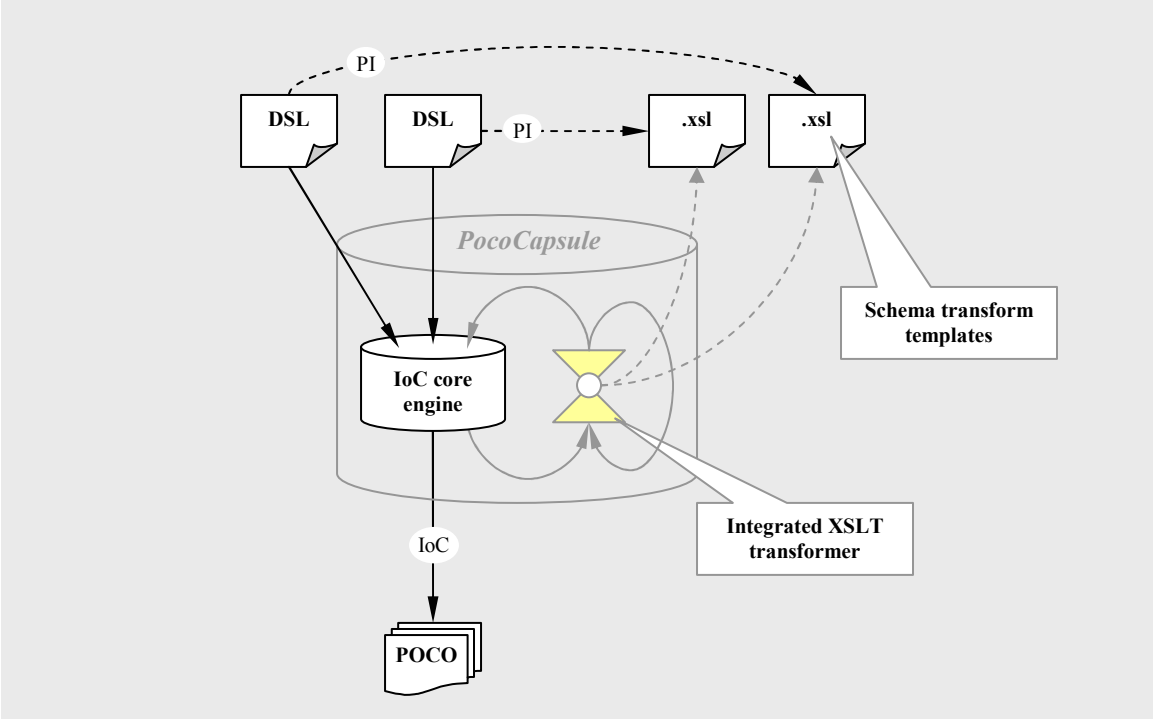
A conventional approach to support user defined/extend schemas in IoC frameworks is using XML DOM element plug-in handlers as illustrated in the following diagram[22]. The programming model of these callback handlers is imperative, proprietary, and relies on low level functions such as XML DOM API. Therefore, this approach and its native callback handlers are non-portable, costly to develop/maintain, not able to be automated/tooled, and tightly locked to the particular IoC container.



In contrast, instead of depending on *imperative* and hard coded DOM parsers or DOM element handlers to process DOM elements of DSM/DSL schemas, PocoCapsule leverages the declarative, component model natural, and container agnostic IoC infrastructure itself to facility DSM/DSL. Based on the standardized, portable, and ubiquitous W3C Extended Stylesheet Language Transformations (XSLT) technology, this solution is completely free from any proprietary plug-in API or any low level XML DOM or SAX programming. Comparing to the proprietary plug-in handler approach, this IoC based approach is much straightforward for domain experts, easily transformable across decent IoC containers, and open for third party tool vendors[23].

[22] Such as the Spring 2.0 extensible XML configuration.
[23] In fact, there are many third party XSLT tools available today.

In this IoC based approach, a DSM/DSL schema is still defined by its XML DTD or XML Schema[24]. This schema can be defined from scratch or extended from an existing schema (such as from the core IoC schema). To facility DSM/DSL, PocoCapsule transparently transforms an input description in DSM/DSL schema into an output description in the core or another target schema specified by the *process instruction* (PI) of the input. This transformation is recursively processed until the final output is in the core IoC schema. Almost all practical useful DSM/DSL semantic designs can be supported in this approach because the underlying PocoCapsule core IoC schema is able to support pretty generic POCO API invocation scenarios[25].



The transformation *process instruction* (PI) specifies the name of transformation templates file[26] used by a given description. These schema transformation templates files are also XML documents in the schema of W3C XSL by default. As XML documents, they can be transformed/generated from other XML documents in higher level problem specific schemas. This brings another innovative and powerful feature of PocoCapsule DSM/DSL, namely, it not only transparently facilities application descriptions in DSM/DSL schemas but also seamlessly supports DSM/DSL transformation descriptions themselves in DSM/DSL schemas.

[24] In the current PocoCapsule release, only the DTD is supported. XML Schema will be supported in next release.
[25] This does not mandate all DSM/DSL logics have to be written into transformation templates. One can write portion of these logics as container agnostic C/C++ object classes or even global functions, and then have transformed descriptions to make IoC invocations on these DSM/DSL supporting POCOs.
[26] Inline templates are not supported in this release.

4.3 GPS in a user defined DSM schema

In this section, the GPS example discussed previously is used again to illustrate the development and usage of DSM/DSL in PocoCapsule. The full version of this example, including all source code, XML/XSL documents, Makefile(s), and README document, can be found in the examples/basic-ioc/dsm-gps directory of a PocoCapsule installation.

The following table summarizes the steps of DSM/DSL development and usage in PocoCapsule, and compares them with the corresponding steps in conventional approach of *Spring 2.0 extensible XML configuration*.

	Open, declarative approach (PocoCapsule DSM)	*Proprietary, imperative approach (Spring 2.0)*
DSM/DSL designers	• Define the schema (DTD or XML Schema) (section 4.3.1) • ***Declare schema element transform templates in a XML file*** (section 4.3.2)	• Define the schema (DTD or XML Schema) • ***Implement schema element parsing in a proprietary plug-in handler.***
DSM/DSL users	• Refer the schema (DTD or XML Schema) in the description. (section 4.3.3) • ***Specify the templates file in the "xml-transform" or "xml-stylesheet" PI.*** (also section 4.3.3)	• Refer the schema (DTD or XML Schema) in the description. • ***Enlist the plug-in handler to the container.***

These DSM/DSL development and usage steps are discussed in the following sections indicated in the table.

4.3.1 Define the DSM schema

The first step in designing a DSM/DSL is to define its schema in DTD[27]. The DTD of the example DSM/DSL for the GPS devices is illustrated as follows:

```
<!ELEMENT gps-device
            (tick-generator, gps-locator, navigation-display*)>

<!ELEMENT tick-generator EMPTY>
<!ATTLIST tick-generator use      CDATA #REQUIRED>
<!ATTLIST tick-generator count    CDATA #REQUIRED>
<!ATTLIST tick-generator interval CDATA #REQUIRED>

<!ELEMENT gps-locator EMPTY>
<!ATTLIST gps-locator use         CDATA #REQUIRED>

<!ELEMENT navigation-display EMPTY>
<!ATTLIST navigation-display use  CDATA #REQUIRED>
```

[27] XML Schema is not supported in this release.

This schema is declarative and significantly simpler and much more expressive than the core IoC schema. For instance, the GPS device presented previously in core IoC example can now be set up in the following DSM/DSL description:

```
<gps-device>
  <tick-generator use="TickGenImpl" count="10" interval="1"/>
  <gps-locator use="GPSLocatorImpl"/>
  <navigation-display use="NavDisplayImpl"/>
</gps-device>
```

In this DSM/DSL, a GPS device can be declared as a composite consists of a tick-generator, a gps-locator, and arbitrary number of navigation-display components. The "*use*" attribute of these elements specifies the POCO implementation class to be used to instantiate the given component. The "*count*" and "*interval*" attributes of the <tick-generator> element specifies configuration parameters of the tick generator. All low level API signatures and component wirings are milted away in this DSM/DSL schema.

4.3.2 Declare schema transform templates

The next step for the designer is to declare transformation templates from this DSM/DSL to an intermediate schema or directly to the final core IoC schema. Each individual template specifies the macro expanding[28] of a DSM/DSL node (element or attribute) in the target schema. For instance, the template expanding the <tick-generator> element into the IoC schema can be defined as follows:

```
<!--   DSL transform template for element
       <tick-generator
          use="@use" count="@count" interval="@interval">
-->
<dsl-template match="tick-generator">
  <bean class="@use" destroy-method="delte" lazy-init="false">
    <method-arg type="short" value="@count"/>
    <method-arg type="short" value="@interval"/>

    <ioc method="subscribe">
      <method-arg ref="gps-locator"/>
    </ioc>

    <ioc method="start"/>
  </bean>
</dsl-template>
```

For simplicity, this transformation template itself is also declared in a more expressive and less verbose DSM/DSL schema. The corresponding W3C XSL template looks like the following <*xsl:template*> declaration:

[28] A DSM/DSL transformation template in PocoCapsule is similar to a C/C++ preprocessor macro or template, and is a XML variation of Lisp defmacro s-expression.

```
<!--   XSL transform template for element
       <tick-generator
          use="@use" count="@count" interval="@interval">
-->
<xsl:template match="tick-generator">
  <bean destroy-method="delte" lazy-init="false">
    <xsl:attribute name="class">
      <xsl:value-of select="@use"/>
    </xsl:attribute>
    <method-arg type="short">
      <xsl:attribute name="value">
        <xsl:value-of select="@count"/>
      </xsl:attribute>
    </method-arg>
    <method-arg type="short">
      <xsl:attribute name="value">
        <xsl:value-of select="@interval"/>
      </xsl:attribute>
    </method-arg>

    <ioc method="subscribe">
      <method-arg ref="gps-locator"/>
    </ioc>

    <ioc method="start"/>
  </bean>
</xsl:template>
```

This transformation template defined either in DSL or in the equivalent XSL form is fairly intuitive. It implies that the <tick-generator> element in this example is to be transformed (expanded) into the following core IoC declaration:

```
<!-- transformed from the DSL expression
     <tick-generator use="TickGenImpl" count="10" interval="1"/>
-->
<bean class="TickGenImpl" destroy-method="delte" lazy-init="false">
  <method-arg type="short" value="10"/>
  <method-arg type="short" value="1"/>

  <ioc method="subscribe">
    <method-arg ref="gps-locator"/>
  </ioc>

  <ioc method="start"/>
</bean>
```

4.3.3 Using the user defined DSM/DSL schema

For a user or an application developer who is going to set up a GPS device application, it is fairly straightforward and transparent to use a DSM/DSL schema. What the user should do is specifying the following schema properties in the application setup description:

- *Specifying the DTD file (or URL) in the DOCTYPE declaration, and,*
- *Specifying the schema transformation templates file (or URL) in the **href** attribute of the xml-transform process instruction (PI).*

Assuming the DTD and XSL file names of this example are gps-devicce.dtd and gps-device2poco.xsl, the application setup description using the gps-device DSM/DSL schema will look like the follows:

```
<!-- DTD of this descriptor -->
<!DOCTYPE gps-devicce SYSTEM "file:gps-device.dtd">

<!-- Process instruction (PI) specifies the transformation -->
<?xml-transform type="text/xsl" href="file:gps-device2poco.xsl"?>

<gps-device>
  <tick-generator use="TickGenImpl" count="10" interval="1"/>
  <gps-locator use="GPSLocatorImpl"/>
  <navigation-display use="NavDisplayImpl"/>
</gps-device>
```

With correctly declared DOCTYPE and xml-transform process instruction as above, this application description can be handled by the PocoCapsule/C++ core container and all its utilities (such as pxgenproxy, pxencode) seamlessly as if this DSM/DSL schema was a natively built-in feature of the framework.

4.3 A framework of frameworks

Within the IoC+DSM/DSL paradigm, PocoCapsule is not only able to improve the expressiveness of component assembly/configure/deployment models for diverse high level domain problems, but also qualified as a framework for easily and quickly building other standardized or user defined high level component frameworks. Typically, an assembly, configure, and deployment container of such a framework would imply thousands or even tens of thousands lines of code and weeks or even months of efforts of a skillful framework developer, if it was built from scratch. In a significant contrast, using an IoC+DSM/DSL framework, such as PocoCapsule, an average developer or even a domain expert without framework building experience is able to comfortably construct the same high level container in just a couple of days or even hours with less than a few hundreds lines of highly descriptive and self-documenting code.

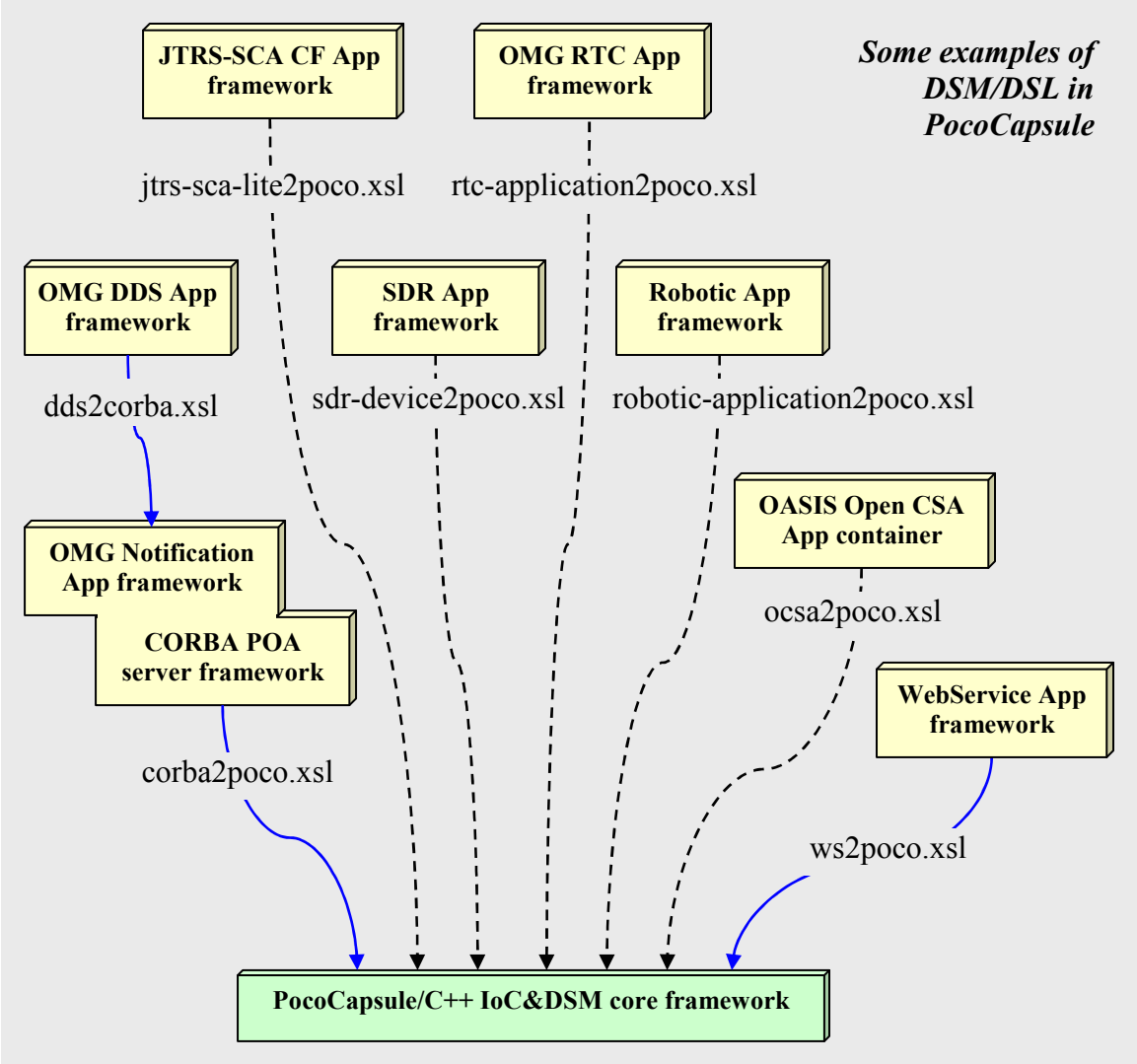
For instance, the reference implementation of JTRS-SCA assembly container (Java) from CRC and the prototype implementation of Open-CSA (C++) from the Apache Tuscany project all used 6,000 to 10,000 lines of code. In PocoCapsule, however, they can easily be built in less than 500 lines of code.

The following standardized or user defined component frameworks, built from IoC and DSM technologies, are available out-of-the-box in PocoCapsule.

- The OASIS Open Composite Service Architecture (***Open-CSA***) for web service and service oriented applications.
- The US Navy JTRS Software Communication Architecture (***JTRS-SCA***) core framework (CF) for software defined radio (SDR) applications.
- The OMG Robotic Technology Component (***OMG-RTC***) framework for robotic applications.
- A CORBA server application assembly, configuration and deployment framework supporting POA server, OMG Event/Notification, OMG DDS, and OMG RTC applications.
- Other user defined deployment frameworks for WS, SOA, SDR, and robotic component applications.

These frameworks and their relations to the PocoCapsule core framework are illustrated in the following diagram. Their usages are described in appendixes A, B, C, D, and demonstrated in PocoCapsule user examples[29].

[29] http://www.pocomatic.com/docs/cpp-examples



To application developers, the assembly, configuration, and deployment containers produced in this IoC + DSM approach are much simpler and superior to their traditional handcrafted counterparts by largely eliminating contrived aspects of the component models and restrictions that sacrifice the usability for application developers in favor of the convenience for vendors[30].

[30] See appendix A, B, C, D and PocoCapsule examples for comparisons and analysis.

<This is a blank page>

Chapter 5 PocoCapsule/C++ API

5.1 The POCO_ApplicationContext class

The PocoCapsule/C++ runtime engine, namely the container, is implemented and provided as a runtime library to be embedded in user applications. The application programming interface of this runtime engine is the POCO_AppContext class as follows, defined in the ***pocoapp.h*** header file:

```
class POCO_AppContext {
      ...
  public:
      static POCO_AppContext* create(
                  const char*       res,
                  const char*       type,
                  POCO_AppEnv*      env = NULL);

      static POCO_AppEnv*     getDefaultAppEnv();
      static POCO_AppEnv*     setDefaultAppEnv(
                  POCO_AppEnv*      env);
      static POCO_AppEnv*     initDefaultAppEnv(
                  int               argc,
                  char*             argv);

      virtual void* getBean(
                  const char*       id,
                  POCO_AppEnv*      env = NULL);

      virtual void* getBean(
                  const char*       id,
                  const char*       class_name,
                  POCO_AppEnv*      env = NULL);

      virtual void* getBeanPtrTypeId(
                  const char*       id,
                  POCO_AppEnv*      env = NULL);

      virtual int initSingletons(
                  POCO_AppEnv*      env = NULL);

      virtual int terminate(
                  POCO_AppEnv*      env = NULL);

      virtual void destroy(
                  POCO_AppEnv*      env = NULL);
};
```

5.1.1 The *create ()* method

The *create()* reads a description from a file (if type equals "file") or embedded string buffer (if type equals "string") and creates the specified application context. During this, the following two procedures will be executed:

- All shared/dll libraries specified by <*load*> elements are load.
- Then, all sub-contexts specified by <*import*> are created recursively.

The content format of the given input description file can be either XML or PocoCapsule encoded text. Other formats can be supported through plug-in description readers.

On failures of this method, the container will invoke the *fatal_error()* method on the *env* argument (if it isn't null) or on the default environment with the error message as the input argument. See the description about the *POCO_AppEnv* class in the next section.

5.1.2 The *initSingletons()* and *terminate()* methods

The *initSingletons()* method called on a given application context will first recursively calls the *initSingletons()* on all of its imported sub-contexts by order, and then instantiates all eager singletons declared in the context. For a given application context, the container will iterate through all eager singletons in their declared order, and instantiate those that are not instantiated yet.

The *terminate()* methods will destroy (by calling specified destroy methods) all singletons in this context, and then recursively invokes the *terminate()* on all imported sub-contexts by order.

On failures of these methods, the container will invoke the *fatal_error()* method on the *env* argument (if it isn't null) or on the default environment with the error message as the input argument. See the description about the *POCO_AppEnv* class in the next section.

5.1.3 The *getBean()* and *getBeanPtrTypeId()* methods

There are two flavors of *getBean()*. The first *getBean()* simply returns the bean by casting it from the pointer pointing to the declared class type to a void* pointer. If the bean object is to be accessed, it should be casted back to a pointer pointing to exactly the declared class type. For instance, assuming class A, B, and an object factory create_A() are defined as follows:

```
class B : public A {
   ...
};

A* create_A() { return new B; }
```

A bean of class A is declared in a context as follows:

```
...
<bean id="my-bean" class="A" factory-method="create_A"/>
...
```

Then, the returned (void*) should be casted to (A*):

```
...
A* a = (A*)(ctxt->getBean("my-bean")); // this is correct
...
B* b = (B*)(ctxt->getBean("my-bean")); // this is wrong
...
```

The second *getBean()* adds some type safety by comparing the expected class name against what was declared in the description. If this comparison fails to match, the getBean() will return a NULL pointer, as follows:

```
...
A* a = (A*)(ctxt->getBean("my-bean", "A"); // this will success
...
B* b = (B*)(ctxt->getBean("my-bean", "B"); // this will return NULL
...
```

To ensure type correctness, the returned pointer of *getBean()* should always first be casted to a pointer pointing to bean's declared class, before further casting or dynamic casting to another pointer type. For instance, in above example, to get the my-bean instance from the application context and cast it into a pointer of class B, the application should first retrieve the bean instance and cast to its declared type (i.e. A*), and then, type cast or dynamic_cast (if A is a polymorphic class) to B* as follows:

```
...
A* a = (A*)(ctxt->getBean("my-bean", "A"));
...
B* b2 = dynamic_cast<B*>(a);
...
```

Note:

- *getBean() will instantiate the bean if it is a non-singleton, or it is a singleton but has not been instantiated yet. For non-singleton, it is the caller's responsibility to release the returned bean.*

- *getBean() can also be called with an id assigned to a void bean. This is useful to instantiate the bean from the container. The return value on getBean() for a void bean is always a NULL pointer. Therefore, a NULL return value in this case does not indicate a failure.*

The *getBeanPtrTypeId()* returns the type name of the specified bean pointer. For instance, in the example above, the *getBeanPtrTypeId("my-bean")* will return "A*".

On failures of these methods, the container will invoke the *fatal_error()* method on the *env* argument (if it isn't null) or on the default environment with the error message as the input argument. See the description about the *POCO_AppEnv* class in the next section.

5.1.4 The *setDefaultAppEnv()*, *getDefaultAppEnv()*, *initDefaultAppEnv()* methods

The *setDefaultAppEnv()* replaces the current default application environment object with an user specified environment object and returns the original default application environment object.

The *getDefaultAppEnv()* retrieves the current default application environment. See the description of the *POCO_AppEnv* class in the next section.

The *initDefaultAppEnv()* is just a combination of calling the *getDefaultAppEnv()* and then invoking the *setArray("pococapsule.init.argv", argc, argv)* on the returned result (see the description of the *setArray()* of the *POCO_AppEnv* in next the section). Namely, the following code:

```
int main(int argc, char** argv)
{
      POCO_AppEnv env =
            POCO_AppContext::initDefaultAppEvent();
      ...
```

is equivalent to:

```
int main(int argc, char** argv)
{
      POCO_AppEnv env =
            POCO_AppContext::getDefaultAppEvent();
      env->setArray("pococapsule.init.argv", argc, argv);

      ...
```

5.1.5 The *destroy()* method

The destroy() method effectively calls the destructor of the underlying application context implementation and releases allocated memory. This method does not terminate the context, namely it will leave already instantiated beans intact.

5.2 The *POCO_AppEnv* class

PocoCapsule/C++ does not use exception by default. The container runtime engine neither catches nor throws exceptions. Exceptions raised from a bean's IoC instantiation or invocations (namely constructor, factory-method, destroy-method, dup-method, post-instantiation methods, etc) are all forwarded to the callers of *POCO_AppContext* functions in the container.

All methods of the *POCO_AppContext* class have a *POCO_AppEnv* object pointer as the last argument. If the PocoCapsule runtime detects an error on performing a *POCO_AppContext* method, it will make a callback on the *fatal_error()* method of that environment object and returns (int)0 or NULL. If this argument is NULL, the callback will be made on the default application environment.

The application environment object is defined as the following class, defined in the ***pocoapp.h*** header file:

```
class POCO_AppEnv {
      ...

  public:
      POCO_AppEnv();
      POCO_AppEnv(int argc, const char** argv);
      virtual void reset();
      virtual const char* get_message();
      virtual void warning(const char*);
      virtual int  has_error();

      virtual void fatal_error(const char* msg);

      virtual void setValue(const char* name,
                            const char* value);
      virtual const char* getValue(const char* name);

      virtual void setArray(const char* name,
                            int         argc,
                            const char* argv);
      virtual const char** getArray(const char* name);
};
```

A default instance of this class is allocated by the PocoCapsule/C++ container, and is used as the default application environment by default. This default application environment can be retrieved using the *getDefaultAppEnv()* method. After a

POCO_AppContext invocation, application can call the *has_error()* on the environment and then, if it indicates an error, call the *get_message()* to retrieve the error message.

Users can extend this class and provide their own implementation by overriding the *fatal_error()* method. Usually, this can be used to force the container to throw an exception right on the spot of the error occurrence. For instance, in the following example, the default environment is replaced by a user provided alternative, the MyErrorHandler. This alternative throws exception within the *fatal_error()* function. This changes the error handling behavior of all *POCO_AppContext* methods from return (int)0 or NULL to throw the specified exception(s).

```
#include <pocoapp.h>
#include <string>

class MyErrorHandler : public POCO_AppEnv {
  public:
      virtual void fatal_error(const char* msg) {
            throw std::string(msg);
      }
};

int main(int argc, char** argv)
{
      MyErrorHandler my_error_handler;
      POCO_AppContext::setDefaultAppEnv(&my_error_handler);

      try {
            POCO_AppContext* ctxt =
            POCO_AppContext::create("setup.xml", "file");

            ctxt->initSingletons();
            ...
      }
      catch(const std::string& ex) {
            ...
      }
      ...
}
```

5.2.1 The *reset()* method

The *reset()* method clear the method status of the environment. After this method, the *has_error()* and *get_message()* on the environment object will return 0 and NULL respectively.

5.2.2 The *get_message()* method

By the default implementation, this method returns the error message passed to the environment by the *fatal_error()* or *warning()* method.

5.2.3 The *warning()* method

By the default implementation, the *warning()* method will save a message to the environment that can be retrieved using the get_message() method. However, the *warning()* method does not change the error status. Namely, if the environment *has_error()* is 0, it should still return 0 after the *warning()*.

5.2.4 The *has_error()* method

By the default implementation, the *has_error()* always returns 1 after a *fatal_error()* call, until the error status is cleared via *reset()*.

5.2.5 The *get_message()* method

By the default implementation, *fatal_error()* sets an error status on the environment and saves the error message argument to the environment, until they are cleared by the *reset()* method.

Application can override this method to force the container to throw exception on error.

5.2.6 The *setValue()* and *getValue()* methods

The *setValue()* method specifies a name-value mapping that will be used to resolve the literal value referred by the *env-var* attribute of <*method-arg*> and <*item*> elements of the assembly/deployment description. This value resolving is performed at context creation time.

For instance, assuming an application context is created from a description file setup.xml as follows:

```
POCO_AppEnv* env = ...;
env->setValue("my-server-name", "the-bank-account-server");
env->setValue("max-thread", "32");
...
ctxt = POCO_AppContext::create("setup.xml", file, env); ...
```

Here, assuming a bean is declared in the description file (i.e. in the setup.xml) as:

```
<bean class="MyServant" ...>
      <method-arg type="string" env-var="my-server-name"/>
      <method-arg type="short" env-var="max-thread"/>
</bean>
```

Then, effectively, the bean will be instantiated in the following constructor calling syntax:

```
bean = new MyServant("the-bank-account-server", 32);
```

The *getValue()* is called by PocoCapsule/C++ to retrieve the named literal value setting on the application environment. It should return NULL if the required value isn't set.

5.2.7 The *setArray()* and *getArray()* methods

The *setArray()* method specifies a name-array mapping that will be used to resolve the literal array value referred by the *env-var* attribute of <*array*> elements of the assembly/deployment description. This value resolving is performed at the context creation time.

For instance, if the application context is created from a description file setup.xml as follows:

```
int main(int argc, char** argv)
{
      ...
      POCO_AppEnv* env = ...;
      env->setArray("my-argv", argc, argv);
      ...
      ctxt = POCO_AppContext::create(
                  "setup.xml", "file", env);
      ...
```

Here, assuming a bean is declared in this description (i.e. in the setup.xml file) as:

```
<bean class="*CORBA::ORB_ptr"
      factory-method="CORBA::ORB_init">
      <method-arg type="array">
            <array type="string" env-var="my-argv"/>
      </method>
</bean>
```

Then, effectively, the bean will be instantiated in the following calling syntax:

```
bean = CORBA::ORB_init(argc, argv);
```

The *getArray()* is called by PocoCapsule/C++ to retrieve the named literal array setting on the application environment. The returned value is an array of const char* that ends by a NULL pointer to indicate the end of the array.

The constructor *POCO_AppEnv(int argc, const char** argv)* simply allocates a *POCO_AppEnv* instance, and then invokes its *setArray("pococapsule.init.argv", argc, argv)* method.

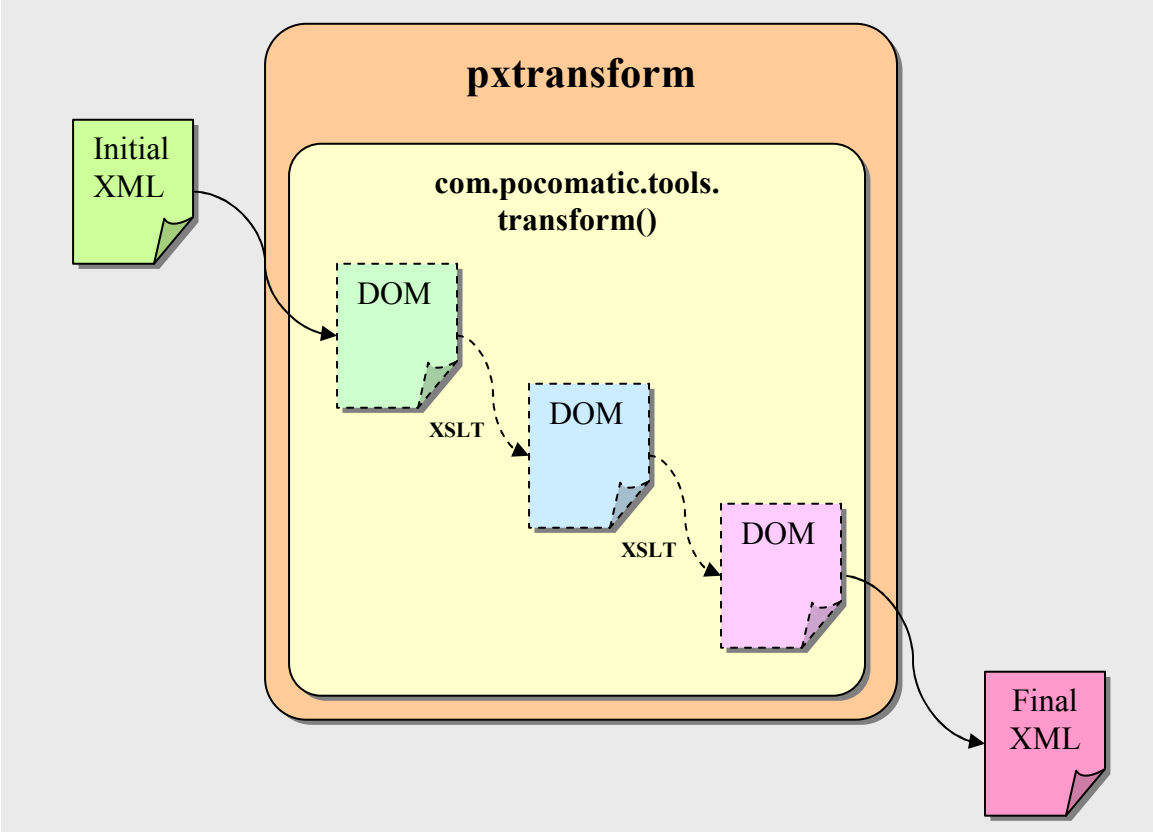
<This is a blank page>

Chapter 6 Utilities, Libraries, Schemas, and Stylesheets

6.1 pxtransform

The command line utility ***pxtransform*** is useful for DSL design and their model transformation XSLT stylesheet development. It transforms an input XML document into another XML document using the XSLT stylesheet specified by the <?xml-transform> process instruction (PI) of the input XML document.

The transformation will automatically be applied on the output XML document as illustrated in the following diagram, until the final resulting XML document, the one that does not have a <?xml-transform> process instruction.



The final resulting XML document will be written to either a new output file or the stdout depends on the command line option.

```
pxtransform [options] <input-xml-file-names>
```

Here, the options are:

- ***-help*** : print out command line usage help information.
- ***-E*** : the output will be written to the stdout instead of a file.
- ***-c*** : disable the schema validation on the input document.[31]
- ***-s=<suffix>*** : specifies the suffix to be used for the output file. The default suffix value is "_poco.xml".

For instance, in the following example:

```
pxtransform -s=_p.xml my_setup.xml
```

The final output XML document will be written to my_setup_p.xml.

Multiple XML files or wildcard XML file names can be specified, for instance as follows:

```
pxtransform my*.xml
```

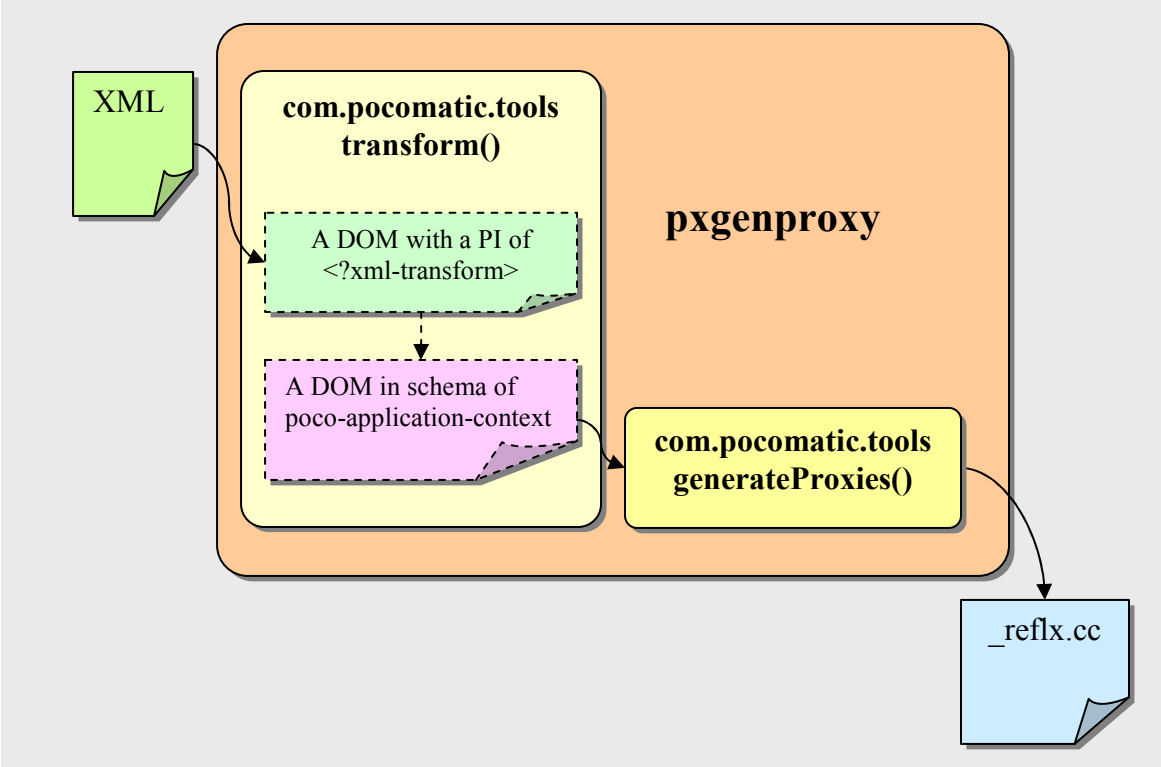
This is equivalent to applying pxtransform to all matched XML files one by one, and therefore, multiple final XML document files will be generated after this transformation.

6.2 pxgenproxy

The command line utility ***pxgenproxy*** generates the dynamic invocation proxies implied by the input XML file, the application assembly and deployment description.

As illustrated in the following diagram, the input XML file is first recursively transformed until the final resulting DOM object that does not have a <?xml-transform> process instruction, and should be in the core schema of PocoCapsule (namely, to be a <poco-application-context> document). Then, this final DOM object will be used by this utility to generate proxies.

[31] This is primarily useful in developing transformation DSM/DSL.



The syntax of this command is:

```
pxgenproxy [options] <input-xml-file-names>
```

Here, the options are:

- ***-help*** : printing out command line usage help information.
- ***-s=<suffix>*** : specifies the suffix and extend name of generated proxy files. The default value is "_reflx.cc".
- ***-h=<user-header-file>*** : specifies a user defined header file to be included by the generated proxy source code. Multiple instances of this operation can be used.
- ***-H=<system-header-file>*** : specifies a standard header file to be included by the generated proxy source code. Multiple instances of this operation can be used.
- ***-r=<gather|scatter>*** : recursively step into imported resources and gather or scatter proxy generation result into one or multiple individual proxy file(s). The default choice is not to do the recursive generation.

For instance, in the following example:

```
pxtransform -r=gather -s=_proxy.cc my_setup.xml
```

The generated proxies, implied by the my_setup.xml itself and all its imported files, will be written to my_setup_proxy.cc.

Multiple XML files or wildcard XML file names can be specified.

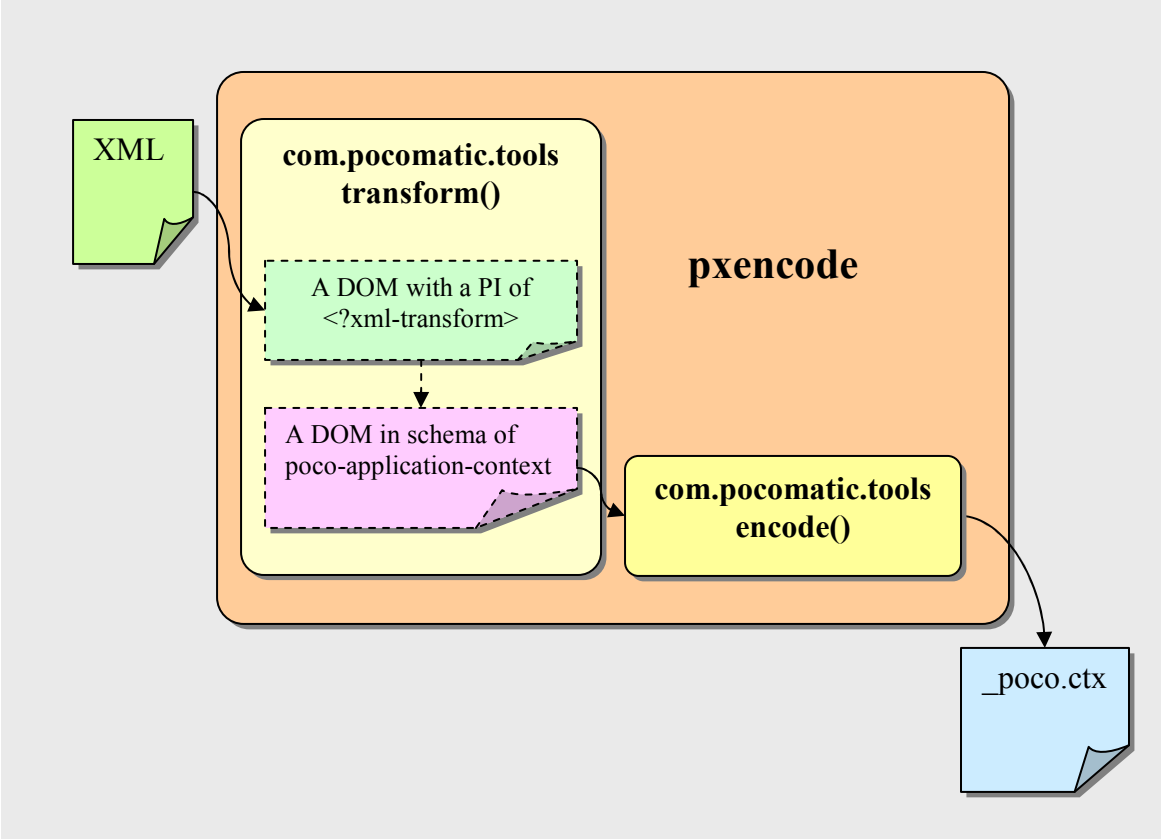
```
pxtransform my*.xml
```

This is equivalent to applying pxgenproxy to all matched XML files one by one, and therefore, multiple proxy source code files will be generated.

6.3 pxencode

The command line utility ***pxencode*** produces the encoded application description from the input XML description file. Encoded application descriptions can be used by PocoCapsule/C++ runtime engine directly without using XML parser and XSLT transformers. This significantly reduces the effective memory consumption of PocoCapsule/C++ runtime from multiple megabytes to less than 100 kilobytes.

As illustrated in the following diagram, the input XML file is first recursively transformed until the final resulting DOM object that does not have a <?xml-transform> process instruction, and should be in the core schema of PocoCapsule (namely, to be a <poco-application-context> document). Then, this final DOM object will be encoded by this utility.



The syntax of this command is:

```
pxencode [options] <input-xml-file-names>
```

Here, the options are:

- ***-help*** : print out command line usage help information.
- ***-s=<suffix>*** : specifies the suffix and extend name of generated encode files. The default value is "_poco.ctx".

For instance, in the following example:

```
pxencode -s=_p.ctx my_setup.xml
```

The final encoded description will be written out to my_setup_p.ctx.

The imported resource names, specified as the **resource** attribute of <import> elements in the input XML file (or its final transformed result), will also be replaced with this specified suffix and extend name in the encoding result. Therefore, in the example above, if the original XML file my_setup.xml imports a resource file my_setup2.xml, then, the encoded my_setup_p.ctx will import from file my_setup2_p.ctx.

Multiple input XML files or wildcard XML file names can be specified, for instance as follows:

```
pxencode my*.xml
```

This is equivalent to applying pxencode to all matched XML files one by one, and therefore, multiple encoded descriptions will be generated after this encoding.

6.4 libraries

6.4.1 libpococapsule.so (pococapsule.dll)

This is the PocoCapsule/C++ IoC container runtime. It contains the implementation of POCO_AppContext and POCO_AppEnv classes.

6.4.2 libpocoxsl.so/libpocoxml.so (pocoxsl.dll/pocoxml.dll)

The *libpocoxsl.so (pocoxsl.dll)* is a JNI proxy to the PocoCapsule/C++ XML encoder implemented in standard JAXP. This encoder, as described in the previous section, parses, transforms and encodes a XML deployment description and returns its stringified final internal DOM. Therefore, this library should be linked or loaded if the application explicitly uses XML deployment descriptions, especially when these XML descriptions use user defined schema(s). This proxy will start the encoder on demand. If all input descriptions are pre-encoded already, this proxy will not load and start the XML encoder.

The *libpocoxml.so (pocoxml.dll)* is similar to libpocoxsl.so (pocoxsl.dll), except:

- It only supports the core schema of PocoCapsule, and does not support XSLT transformation of DSL schemas.
- It is a native C++ implementation using Apache Xerces/C++, therefore, although performance and memory consumption are similar, it can be loaded much faster than loading/staring the JAXP version XML encoder.

6.4.2 libpoco2*.so (poco2*.dll)

These libraries are adapters used to support PocoCapsule/CORBA, POCOCapsule/WS, etc. and are documented by their respective user documents.

6.5 Schemas and transformation stylesheets

The following XML schemas are ready available to applications in PocoCapsule/C++ shipment. The are all embedded in the XML parsing and XSLT transforming engine of PocoCapsule/C++ runtime or utilities. Text copies of them can be found in the ${POCOCAPSULE_DIR}/resources directory, for user reference.

6.5.1 DTD of schemas

- *poco-application-context.dtd*: The document type definition of the core schema.
- *corba-application-context.dtd*: The document type definition of the PocoCapsule/CORBA schema, extended from the core schema.
- *corba-dds-application.dtd*: The document type definition of the PocoCapsule/CORBA+DDS schema, extended from the corba-application-context.dtd.
- *ws-application.dtd*: The document type definition of the PocoCapsule/WebServices schema, extended from the core schema.
- *osca-composite.dtd*: The document type definition of the OASIS Open Composite Service Arcitecture (OpenCSA) schema.
- *jtrs-sca-lite.dtd*: The document type definition of a lite version of JTRS-SCA software assembly model.

6.5.2 Schema transformation stylesheets

- *corba2poco.xsl*: This stylesheet specifies corba-application-context schema to the core context schema transforming templates.
- *dds2corba.xsl*: This stylesheet specifies dds-corba-application schema to the corba-application-context schema transforming templates.
- *ws2poco.xsl*: This stylesheet specifies ws-application schema to the core schema transforming templates.
- *ocsa-composite2poco.xsl*: This stylesheet specifies ocsa-composite schema to the core schema transforming templates.
- *jtrs-sca-lite2poco.xsl*: This stylesheet specifies jtrs-sca-lite schema to the core schema transforming templates.

<This is a blank page>

Appendix A. Web Services and the OASIS OpenCSA

A.1 Web Services containers

Web Services applications are software systems consist of service users (clients) and service providers (servers). Clients use SOAP protocol to access services provided by remote servers. These services are business logic components implemented, for instance, in conventional programming languages (such as C++ and Java).

Service components are deployed to a runtime environment, referred to as *WebServices container* here after, which adopts them as service endpoints to a message dispatch engine atop a SOAP stack. Ideally, such a container should reduce the complexities and cost of WebServices application development and maintenance.

Unfortunately, WebService containers available so far are mostly proprietary and primitive, especially for C++ applications. Not only they rely on vendor specific component programming, configuration, and deployment models, but also they do not have a declarative assembly model. These problems largely prohibit the reusability of third party and legacy service components, rule out the transferability of developer/user skills and experiences, and discourage the interchangeability of development tools across containers from different vendors.

A.2 The SCA (OASIS CSA) model

Several vendors known as the Open SOA group (www.osoa.org) have been attempting to solicit their Web Services component architecture, the Open Service Component Architecture (SCA), as an industrial standard. Lately, the SCA effort has been moved into OASIS and renamed as Open Composite Service Architecture (CSA).

As a design-by-committee design, the SCA (i.e. OASIS-CSA), especially its C++ binding, is more complex than necessary. It incurs steeping learning curves and implies high application development/maintenance costs and risks. The SCA C++ binding committee decided not to use the non-invasive inversion of control on plain old C++ objects (POCO) for component wiring and property setting. Instead, the EJB 2.x style context (directory) lookup was chosen, a technique that has been abandoned by the mainstream industry for years.

For instance, in a plain old C++ object implementation (and IoC framework), a dependent service object reference can be injected to the user object through a setter method (accountDataService()) as follows:

```
// a setter method set the dataService dependency
void AccountServiceImpl::accountDataService(
            AccountDataService* service) {
      ...
      _dataService = service;
}

AccountReport AccountServiceImpl::getAccountReport(
            const std::string& customerId) {
      // calling a method on the service reference.
      ... = _dataService->getCheckingAccount(customerId);
      ...
}
```

In SCA, however, this wiring is only forged in the component context after the deployment. The business logic method (getAccountReport()), before using the service, has to lookup the context to resolve the wired service reference and blindly cast the returned void pointer to the assumed type as follows:

```
AccountReport AccountServiceImpl::getAccountReport(
            const std::string& customerId) {
      ComponentContext* context
            = ComponentContext::getCurrent();

      // lookup the wiring from context
      void* ptr = context->getService(
            "<id-of-account-data-service>");

      // Dangerously cast the void pointer to the
      // assumed class type without any type check.
      AccountDataService* _dataService
            = (AccountDataService*)ptr;

      ... = _dataService->getCheckingAccount(customerId);
      ...
};
```

This design is not only dangerously type unsafe but also forces component implementations to be tightly coupled to the underlying SCA (OASIS-CSA) container. Consequently, SCA C++ components have to be designed and implemented specifically for a SCA container, debugged and tested within the SCA container, and built against the SCA container runtime library.

The C++ binding of SCA also introduces many restrictions in its component and deployment models, such as rules on file names of composite descriptions and component interface C++ header files, rules on interface define C++ classes (and all methods must be pure virtual, etc.). These restrictions help vendor/technology lock-in but

sacrifice the simplicity, usability, legacy compatibility, and flexibility of the framework itself and its application implementations.

A.3 The PocoCapsule for Web Services

The non-invasive IoC scheme of PocoCapsule inherently supports Web Services component assembly and configuring. In contrast to those proprietary, complex, and restrictive Web Services component solutions and SCA, this IoC based assembly and configuration alternative is open, straightforward, and flexible. It has the following characteristics:

- *Neutral to component programming models*: Hence, it seamlessly supports component programming models imposed by any SOAP protocol stacks or WSDL to C++ mappings, such as what supported by Apache Axis C++, gSOAP C/C++, and HP Systinet WASP Server for C++.
- *Assembly/configuration model is orthogonal to Web Services*: Therefore, development experience and tools for this model are fully transferable from and applicable to other problem domains.
- *Agnostic to container*: Components are completely decoupled from the framework runtime environment. Components can be designed, implemented, built, debugged, tested, and even used without using or even without knowing the container.
- *Free from senseless restrictions*: It does not mandate the names of composite description files, does not even mandate they are described in files. It does not impose any restriction on component C++ class declarations and implementations with regarding to the number of implementation declared in a given header file, header file names, and implementation programming paradigms. For instance, an implementing can made in interface-paradigm (inherited from pure abstract classes), generic-programming-paradigm (from template classes), or non-OO paradigm (explicitly generated from certain tools).

The IoC technique also provides a straightforward and declarative method to deploy service components to arbitrary WebServices runtime. For instance, with explicit imperative program, a Web Services service component could be deployed as a service endpoint in the following pseudo C++ code:

```
AccountServiceImpl my_service; // a service component
...

// deploy it to the WS runtime adaptor env
WSAdaptor::export_service(
      ..., &my_service, "AccountService", ...);
```

From IoC perspective, such an deployment is no more than injecting references of either service component beans or their factory/manager beans as dependencies to the Web

Services runtime environment adapter bean(s). The above imperative code, therefore, can be expressed in the following declarative application description:

```
<bean id="my-service" class="AccountServiceImpl"/>

<bean class="void"
      factory-method="WSAdaptor::export_service">
      ...
      <method-arg ref="my-service"/>
      <method-arg type="string" value="AccountService"/>
      ...
</bean>
```

Further, to conceal low level complexities of this service deployment programming model from domain users, a domain specific modeling schema can be defined that uses an <endpoint> element to express the factory injection based on the factory method *WSAdaptor::export_service()*. With this DSM schema, the application description becomes:

```
<bean id="my-service" class="AccountServiceImpl"/>

<web-server>
      <endpoint ref="my-service" name="AccountService"/>
</web-server>
```

A.4 The bigbank example in PocoCapsule

As concluded in the previous section, PocoCapsule IoC framework is inherently a web services component assembly, configuring, and deployment container. With minimum extension, the DSM schema is able to be used to express web service endpoint deployment in very high level declarative application descriptions. This web-services application model, defined in the *http://www.pocomatic.com/ws-application.dtd*, is supported by PocoCapsule/C++ out-of-the-box.

The bigbank example from SCA[32] is used to illustrate how to assemble, configure, and deploy a component based Web Services application in this IoC model. The full source code and document of this example can be found in the examples/web-services/bigbank-ws directory of a PocoCapsule/C++ IoC&DSM installation. This application is declared in the following description:

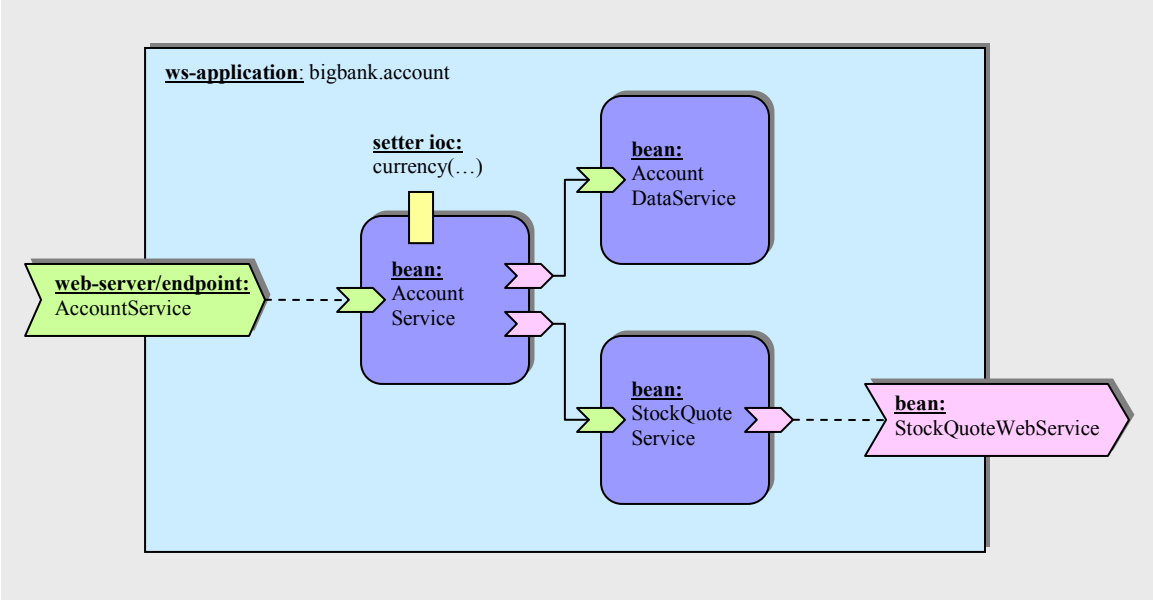
[32] http://www.osoa.org/download/attachments/28/Building_YourFirstApplication_V0.9.pdf

```
<ws-application>
   <bean id="AccountService" class="AccountServiceImpl">
     <ioc method="accountDataService">
        <method-arg ref="AccountDataService"/>
     </ioc>

     <ioc method="stockQuoteService">
        <method-arg ref="StockQuoteService"/>
     </ioc>

     <ioc method="currency">
        <method-arg type="string" value="EURO"/>
     </ioc>
   </bean>

   <bean id="AccountDataService" class="AccountDataService"/>

   <bean id="StockQuoteService" class="StockQuoteService">
     <ioc method="webService">
        <method-arg  ref="StockQuoteWebService"/>
     </ioc>
   </bean>

   <bean id="StockQuoteWebService" class="StockQuoteSoap"/>

   <web-server>
     <endpoint name="AccountService" ref="AccountService"/>
   </web-server>
</ws-application>
```

By the description above, this application consists of the following beans and organization structure:

- Three plain old C++ object beans: AccountService, StockQuoteService, and AccountDataService. They will be instantiated on demand using their C++ constructors.
- A soap stub object StockQuoteWebService, will be instantiated on demand using the StockQuoteSoap class constructor.
- These beans and object are configured and wired together using IoC setters.
- The AccountService bean is deployed as a service endpoint using the DSM <web-server> and <endpoint> elements.

The application structure described above is illustrated in the following diagram:



A.5 SCA via IoC and DSM

PocoCapsule IoC & DSM is not only an inherent and straightforward framework for component-based Web Services applications, but also an open and effective platform to support other user defined or standardized assembly/deployment models, including the SCA.

A full blown SCA assembly/deployment model implemented as a domain specific model (DSM) is available out-of-the-box from PocoCapsule. It not only illustrates that PocoCapsule is neutral to component and assembly models, but also demonstrates a plain old C++ object (POCO) based SCA component model and an non-invasive inversion of control (IoC) based SCA assembly model could eliminate all flaws and restrictions from the current SCA specification, as summarized in the following table:

	SCA reference implementation from an Open SOA member[33]	***SCA implementation as a DSM based on PocoCapsule framework***
Engineering cost	Several thousand lines of code. A team of senior developers, nearly a year of heavy engineering effort.	500 lines of code. Few days or even few hours of casual exercise.
Component-container coupling.	Components have to lookup container served contexts to resolve dependency wiring and property setting.	Components are container agnostic. Dependency and property setting are explicitly injected and set in IoC invocations.
Type safety	Dangerously unsafe (getService() returns void*). The potential type mismatch is very hard to be diagnosed.	Type safe, with both build time and runtime type check. Type mismatch will be detectable at build time as well as runtime.
Life cycle control	No	Yes

[33] http://cwiki.apache.org/confluence/display/TUSCANY/SCA+Native

Wiring and property setting through constructors/factories	No	Yes
Names of SCDL composite description file names	Has to use ".composite" as the extension name.	No restrictions
Component class in header file.	All methods have to be pure-virtual. (SCA C++ Appendix 3)	No restrictions. Any plain old C++ classes, declared and implemented in any conventional style.
C++ preprocess macros, friend classes, templates, non-virtual methods, inline methods, static/global functions, function pointers, etc.	Largely prohibited (SCA C++ Appendix 3)	Fully and seamlessly supported.
The SCDL component descriptions.	Component type files	No longer needed.
Component wiring multiplicity attribute	Crucial for the assembly to work properly in case of multiplied wiring end (subscribe).	No longer needed. Container is agnostic to multiplicity.
Proprietary SCA C++ component annotation	Tying components to the container. Violated the plain old object premise. Implementations have to be specially implemented for SCA, or heavy manual refactoring.	Not needed at all.

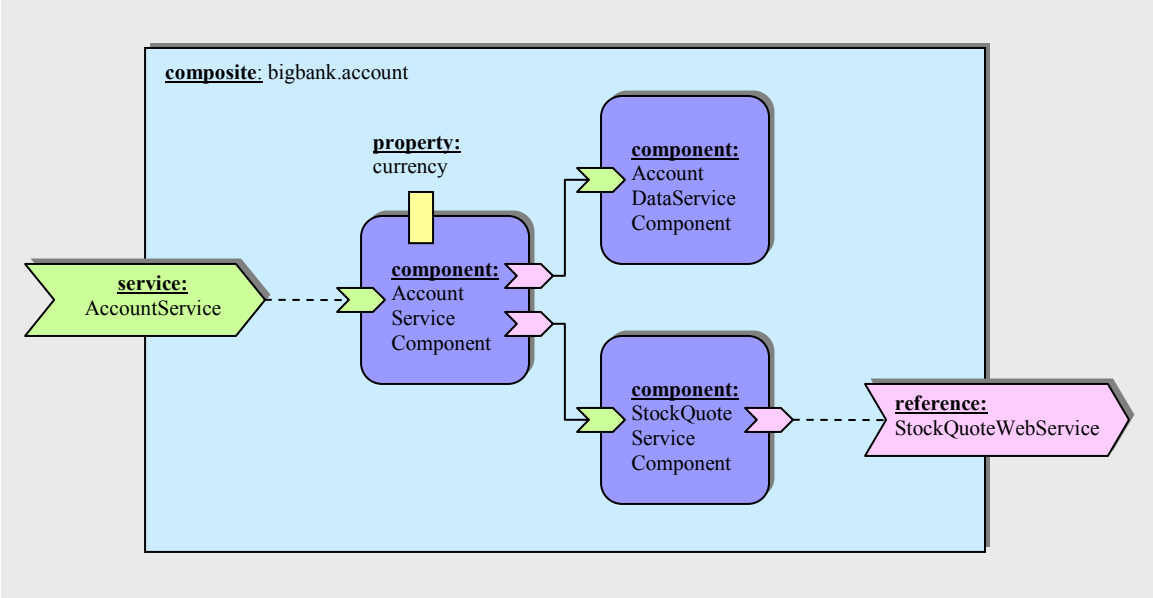
The SCA bigbank example from the reference implementation is used to illustrate the POCO component implementation and IoC wiring/deployment in SCDL schema of SCA. The full source code and document of this example can be found in the examples/web-services/bigbank-ocsa directory of a PocoCapsule/C++ IoC&DSM installation. In SCDL schema of SCA, this application is declared in the following composite description:

```
<composite name="bigbank.account">
  <service name="AccountService">
    <binding.ws port="http://www.bigbank.com/AccountService
                      #wsdl.endpoint(AccountService)"/>
    <reference>AccountServiceComponent</reference>
  </service>

  <component name="AccountServiceComponent">
    <implementation.cpp class="AccountServiceImpl"/>
    <reference
      name="accountDataService">AccountDataServiceComponent</reference>

    <reference
      name="stockQuoteService">StockQuoteServiceComponent</reference>
    <property type="xsd:string" name="currency">EURO</property>
  </component>

  <component name="AccountDataServiceComponent">
    <implementation.cpp class="AccountDataService"/>
  </component>

  <component name="StockQuoteServiceComponent">
    <implementation.cpp class="StockQuoteService"/>
    <reference name="webService">StockQuoteWebService</reference>
  </component>

  <reference name="StockQuoteWebService">
    <interface.wsdl interface="http://www.webserviceX.NET/
                               #wsdl.interface(StockQuoteSoap)"/>
  </reference>
</composite>
```

By this description, this SCA composite consists of the following components and organization structure:

- Three plain old C++ components: AccountServiceComponent, AccountDataServiceComponent, and StockQuoteServiceComponent are declared as <component> elements. They will be instantiated using their C++ constructors.
- A remote reference: the StockQuoteWebService, declared as a top level <reference> element. A stub class StockQuoteSoap will be instantiated to represent this reference.
- Service components and reference stub are configured and wired together using IoC setters declared as <property> and <reference> child elements of <component>s. These components can also be wired using the <wire> element, as illustrated in the calculator example in the examples/web-services/calculator-ocsa directory of PocoCapsule/C++ installation.
- The AccountServiceComponent service component is deployed as a service endpoint using the <service> and <binding.ws> elements.

The application structure described above is illustrated in the following diagram:


composite: bigbank.account
property:
currency
component:
Account
DataService
Component
service:
AccountService
component:
Account
Service
Component
component:
StockQuote
Service
Component
reference:
StockQuoteWebService

<this is an empty page>

Appendix B. Robotic Component

Driven by the ever increasing complexities against the constant requirements of quick development, flexible integration, and low total lifecycle costs, it becomes more and more desirable that robot software systems are going to be built from reusable components. Robotic systems are inherently component-based at the hardware layer. Therefore, there are far less hesitations and far more practices on pursuing "component-based development" at the software layer in the robot community than in others, indicated by the overwhelming number of robotic component collections, toolkits, and frameworks, such as Player/Stage[34], CARMEN[35], OROCOS [36], ORCA[37], MARIE[38], CLARAty[39], MIRO[40], Microsoft Robotics Studio[41], JAUS[42] implementations[43], as well as the OMG robotic technology component (RTC) specification[44] and its experimental implementation the OpenRTM-aist[45].

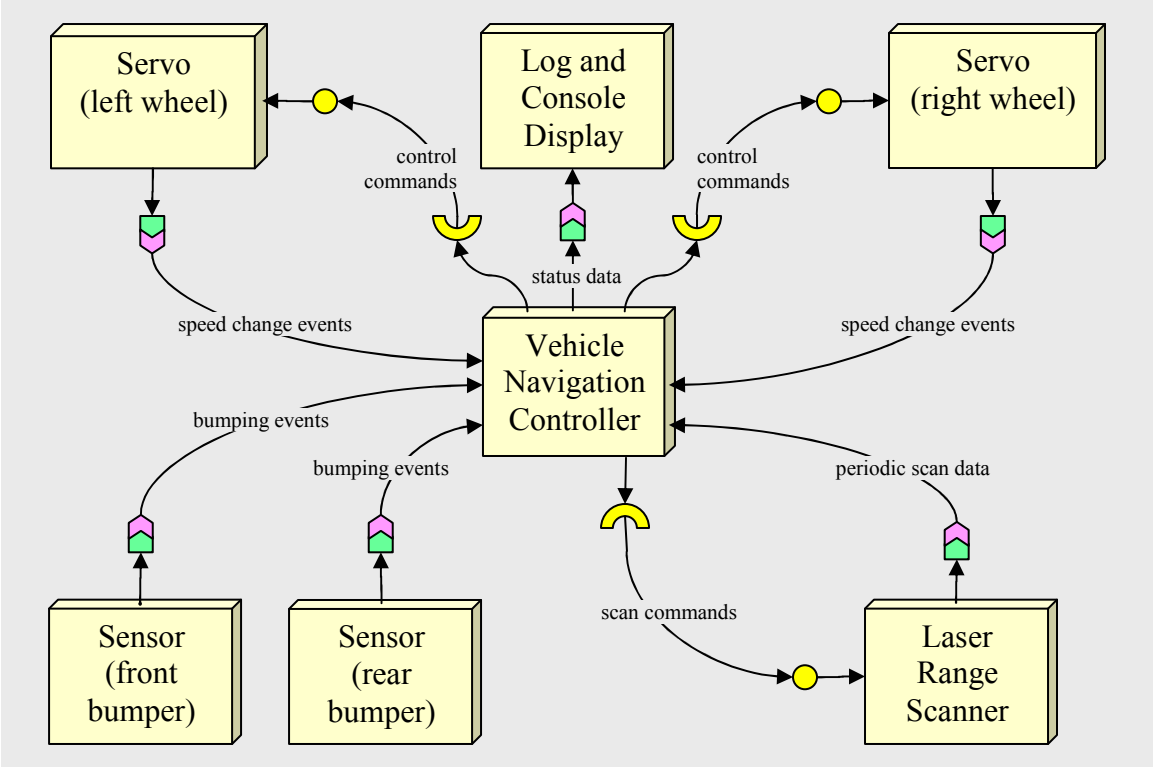
The productivity in developing a given robotic application and the customizability and extendibility in deploying and maintaining it are not solely decided by how well robot monitoring and controlling functions are factored in separate components achieved in component collections or toolkits above, but also heavily rely on how effective and flexible these components are able to be assembled back together into a seamlessly collaborated system and deployed in a service adapting environment. Although some frameworks listed above do offer assembly solutions[46] applicable to their particular component models, robotic applications are mostly assembled, deployed, and configured manually (using APIs and/or proprietary configuration files). Therefore, it is desirable to have a common deployment framework that meets the following criteria:

- ***Neutral to heterogeneous component models***: The diverse needs of heterogeneous robotic systems and the assets of the vast existing robotic component collections, toolkits, and frameworks rule out any attempt of reinventing a single dominated robotic component standard model. Therefore, the new component framework should not only be orthogonal to middleware platforms, but more importantly be neutral to component models without enforcing a particular one.

[34] http://playerstage.sourceforge.net
[35] http://carmen.sourceforge.net
[36] http://www.orocos.org
[37] http://orca-robotics.sourceforge.net
[38] http://marie.sourceforge.net
[39] http://claraty.jpl.nasa.gov
[40] http://www.informatik.uni-ulm.de/neuro/310.html
[41] http://msdn.microsoft.com/robotics
[42] http://www.jauswg.org. Instead of as a component framework, JAUS should be more appropriately categorized as a message framework.
[43] http://openjaus.com and http://www.resquared.com/JAUS-SDK.html
[44] http://www.omg.org/docs/ptc/06-11-07
[45] http://www.is.aist.go.jp/rt/OpenRTM-aist
[46] Such as the XML deployment of OROCOS, and the OMG-D&C for OMG-RTC.

- ***Simple core assembly model/schema***: The new component framework should not incur steep learning curve, long ramp up time, and high development and maintenance cost. The core model/schema itself should be substantially simple, intuitive, and straightforward to application developers and novice, without relying on any UI and/or other massive code generating tools.
- ***Easily extensible and customizable assembly schema***: Instead of mandating a single rigid "one size fits all" model/schema at a predefined abstraction level or scheme, the core assembly model/schema should be able to be extended and customized not only by framework vendors but more importantly by domain application developers. Domain specific assembly models/schemas should be merely domain specific languages (DSLs) realized on top of the core model/schema in a matter of a few hours with a few hundreds lines of declarative code by domain experts instead of weeks or even months of efforts and thousands lines of code by framework vendors.
- ***Lightweight and zero overhead***: The component framework should be able to fit in the most rigorous real-time embedded environments and should not introduce performance overhead or real-time latency and jitter.

PocoCapsule/C++ is able to meet all these criteria. A robotic vehicle example from Microsoft Robotics Studio is used to demonstrate these characteristics and methods. In this example, a robotic vehicle motion control application is wired up from 7 components using 9 connections, as illustrated in the following diagram:



This robotic vehicle control system is implemented in two different component models and assembled in the following two different schemas:

- ***Plain old C++ components and a DSM assembly schema***:
  - In this scheme, controllable ports and event sinks of a given plain old C++ component are exported as pointers of callback C++ objects returned from getter methods of the given provider component.
  - Then, to wire up components, these pointers are injected into controller or event emitter ports of the paired components through their setter methods (namely, the *receptacle*s).
  - These wirings are declared in a quite descriptive DSM schema and set up by the PocoCapsule/C++ container accordingly.

    This DSM schema is defined on top of the PocoCapsule/C++ core IoC schema with only <*component*>, <*controls*>, and <*listens*> elements in the DTD file[47]:
    - A <*component*> element declares a robotic component instance of the specified C++ *class*.
    - A <*controls*> child element of a <*component*> declares a wire from the specified controller port (*receptacle*) of this component to the specified controllable port (*facet)* (of the specified *type*) of a *device* component.
    - A <*listens*> child element of a <*component*> declares a wire from the specified event *sink* port of this component to the specified *receptacle* port of an event *emitter* component.

  An example in this scheme can be found in the examples/basic-ioc/robot-vehicle directory of the PocoCapsule/C++ installation.

- ***OMG-RTC CORBA components model and a DSM assembly schema***:
  - In this scheme the OMG-RTC component model is applied. Components are simply CORBA objects with IDL interfaces extended from *RTC::LightweightRTObject*.
  - By OMG-RTC, services (*facet*s) provided by RTC components are also CORBA objects. A service *facet* reference can be retrieved from the given component using the *provide_<facet_name>()* IDL method.
  - By OMG-RTC, event *sink*s of RTC components are also CORBA objects. A *sink* reference can be retrieved from the given component using the *get_consumer_<sink_name>()* IDL method.
  - By OMG-RTC, a service or event sink object reference can be connected/subscribe to its user/emitter components through the *connect_<receptacle_name>(in Type ref)* IDL method of its user components.
  - These connections are declared in a quite descriptive DSM schema and set up by the PocoCapsule/C++ container accordingly.

[47] The examples/basic-ioc/robot-vehicle/robotic-application.dtd file in the PocoCapsule/C++ directory.

This DSM schema is defined on top of the PocoCapsule/C++ core IoC schema with only <*component*>, <*uses*>, and <*listens*> elements in the DTD file[48]:

- A <*component*> element declares a robotic component instance of the specified C++ *class*.
- A <*uses*> child element of a <*component*> declares a wire from the specified *receptacle* of this component to the specified *facet* port (of the specified *type*) of a service *provider* component.
- A <*listens*> child element of a <*component*> declares a wire from the specified event *sink* port of this component to the specified *receptacle* port of an event *emitter* component.

Either of the two deployment schemes above only requires a DSM schema definition and its transformation XSLT style sheet. This accounts for less than 150 lines of declarative code that can be created in less than an hour by a domain expert, comparing to potentially hundreds or thousands lines of code and days or weeks efforts of a framework experts if the assembly/deployment engine was developed manually from scratch.

Other robotic component framework schemes (namely combinations of component model and assembly schema) can easily be supported in PocoCapsule/C++ as user or third party defined DSMs. These DSMs can be defined directly on top of the PocoCapsule/C++ core IoC scheme, or progressively on top of other high level DSM schemes.

[48] The examples/corba/rtc/rtc-application.dtd file in the PocoCapsule/C++ directory.

Appendix C. Software Defined Radio and the JTRS-SCA

In a simplified and conceptual description, a software defined radio (SDR)[49] is a wireless communication device or equipment that, in order to quickly and easily support multiple and/or different radio bands/modes within a wide spectrum, uses reconfigurable software programs running on generic hardware/OSs[50] to perform certain or all radio signal processing.

To facilitate the quick composable and reconfigurable requirements, SDR waveform applications are commonly architected as composites wired out of software functional components. The de facto standard of such architecture is the Software Communications Architecture (SCA)[51] from US Navy Joint Tactical Radio System (JTRS), or its commercial adaptation the Software Radio Architecture (SRA)[52] from the SDR forum (SDRF). The core framework (CF) of this architecture defines a CORBA based component framework for SDR applications.

By the JTRS-SCA component model, components of a single SDR waveform application are assumed to be distributed in multiple POSIX address spaces across processes or host boundaries. Therefore, SDR components are simply defined as conventional CORBA objects that can be invoked remotely. The OMG Name Service and Event Service are defined as the distributed component registry used by the SCA assembly controller and the SCA distributed publish/subscribe service respectively.

Although modeling a single SDR waveform application as a composite wired up from distributed components is claimed to be necessary, it is one of the major obstacles for mainstream adoption of SDR. For radio wireless devices that have all their components resided in a single address space, a CORBA middleware layer is only an unnecessary burden of footprint, power consumption, learning curves, and excessive engineering complexities, product vulnerabilities, costs, and risks. This observation has prompted some practitioners to seek alternative component architectures for SDR waveforms without CORBA, for instance by directly using conventional non-CORBA C++ classes developed manually or generated automatically from the middleware platform independent model (PIM) UML diagrams of SCA specification[53].

---

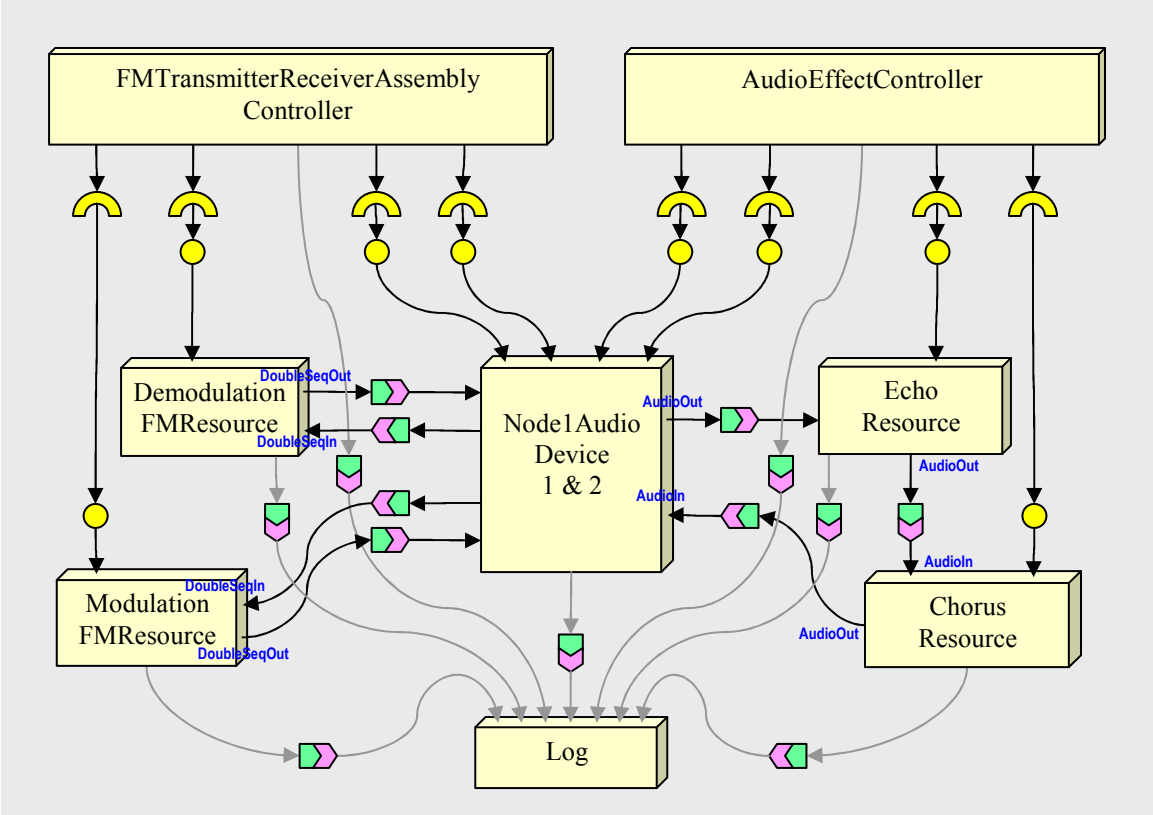
[49] http://en.wikipedia.org/wiki/Software-defined_radio
[50] Such as POSIX platforms.
[51] http://jtrs.spawar.navy.mil/sca
[52] http://www.sdrforum.org, http://www.omg.org/docs/sdo/00-12-05.pdf
[53] This transition also appeared in the Parlay/OSA (http://www.parlay.org/en/specifications/). Early CORBA based Parlay specifications were CORBA based and has been recognized to have unnecessary overhead and complexities. Later Parlay/OSA is changed to UML based, with the CORBA/IDL as one of its mappings.

Therefore, to receive acceptance of mainstream defense and commercial users, the component framework for SDR waveform applications has to take a significantly step forward beyond JTRS-SCA and CORBA. The next generation framework should meet the following four criteria:

- ***Neutral to heterogeneous component models and neutral to middleware platforms***: The new component framework should not only be orthogonal and independent to middleware platforms, but more importantly not enforce a particular component model. For instance, it should seamlessly support plain old user defined C/C++ objects as SDR components as well as JTRS-SCA components (e.g. *CF::Resource* objects) based on CORBA (or other PSM mappings of JTRS-SCA).
- ***Simple core assembly model/schema***: The new component framework should not incur steep learning curve, long ramp up time, and high development and maintenance cost. The core model/schema itself should be substantially simple, intuitive, and straightforward to application developers and novice, without relying on any UI and/or other massive code generating tools.
- ***Easily extensible and customizable assembly schema***: Instead of mandating a single rigid "one size fits all" model/schema at a predefined abstraction level or scheme, the core assembly model/schema should be able to be extended and customized not only by framework vendors but more importantly by domain application developers. Domain specific assembly models/schemas, including the JTRS-SCA assembly schema, should be merely domain specific models (DSMs) realized on top of the core model/schema in a matter of a few hours with a few hundreds lines of declarative code by domain experts instead of weeks or even months of efforts and thousands lines of code by framework vendors.
- ***Lightweight and zero overhead***: The extra power consumption and IO overhead should be significantly reduced to near zero. The memory footprint should be one or two orders of magnitude smaller than the first generation SCA CF implementations available today. Namely, the footprint should be reduced from one or several megabytes to, for instance, less than 100Kbytes.

PocoCapsule/C++ is able to meet all these criteria. A classic SDR example from the SCA reference implementation (SCARI)[54] by CRC is used to demonstrate these characteristics and methods. In this example, a SDR waveform is wired up from 8 components using 22 connections, as illustrated in the following UML 2.0 and CCM component diagram:

[54] http://www.crc.ca/scari



This waveform is implemented in different component models and assembled in various schemas, including the component and assembly models of JTRS-SCA, as illustrated in the following three schemes:

- ***Plain old C++ components, and the PocoCapsule/C++ IoC core assembly schema***:
  - In this non-CORBA waveform application scheme, services of a given plain old C++ component are exported as pointers of callback C++ objects returned from getters of the given provider component.
  - Then, to wire up components, these pointers are injected into their user components through their setter methods.
  - These wirings are declared in the IoC schema and are set up by the PocoCapsule/C++ container accordingly.

    Using the PocoCapsule/C++ core IoC schema:
    - A SDR component instance is declared by a <*bean*> element with the desired C++ class name.
    - Retrieving a service object pointer (of type *SDR::PushPort**) from a provider component is declared as a <*bean*> element with the name of the getter as the value of the *factory-method* and the *id* of the provider component bean as *factory-bean*.
    - Injecting a service object pointer (of type *SDR::PushPort**) into a user component is declared as a <*ioc*> child element of the user component, with the setter name as the value of the *method* attribute.

An example in this scheme can be found in the examples/basic-ioc/sdr directory of the PocoCapsule/C++ installation. This example is originated from a SCARI example, with mocked service implementations.

- ***Plain old C++ components, and a DSM assembly schema***:
  - Similar to previous scheme, the services provided by SDR components are exported (of type *SDR::PushPort**) through getter methods of provider components.
  - Similarly, these services (of type *SDR::PushPort**) are injected to user components through their setter methods.
  - These wirings are declared in a quite descriptive DSM schema and set up by the PocoCapsule/C++ container accordingly.

    This DSM schema is defined on top of the PocoCapsule/C++ core IoC schema with only <*component*> and <*wire*> elements in the see the DTD file[55]:
    - A <*component*> element declares a SDR component instance of the specified C++ class.
    - A <*wire*> child element of a <*component*> declares a wiring between the specified *receptacle* port of this component and the specified *facet* port of a specified service *provider* component.

  An example in this scheme can be found in the examples/basic-ioc/dsl-sdr directory of the PocoCapsule/C++ installation. This example is originated from a SCARI example, with mocked service implementations.

- ***JTRS-SCA CF CORBA components model and JTRS-SCA software assembly schema***:
  - In this scheme the JTRS-SCA core framework component model is applied. Components are simply CORBA objects supporting the *CF::Resource* IDL interface.
  - Services (of type *CF::Port*) provided by these components are also CORBA objects and their references can be retrieved from the JTRS-SCA standard *CF::Resource::getPort(in string service_name)* IDL method on these components.
  - These service object references are connected to their user components through the *CF::Port::connectPort(in Object service, in string connection_id)* IDL method of these user components.
  - These connections are declared in a JTRS-SCA software assembly descriptor (SAD) and are set up by the PocoCapsule/C++ container accordingly. In fact, the SAD schema is handled also as a DSM schema in PocoCapsule/C++ in less than 150 lines of declarative code (XSLT style sheet).

[55] The examples/basic-ioc/dsm-sdr/sdr-device.dtd in the PocoCapsule/C++ installation directory

An example in this scheme can be found in the examples/corba/jtrs-sca directory of the PocoCapsule/C++ installation. This example is originated from a SCARI example, with mocked service implementations.

Similarly, other SDR component framework schemes (namely combinations of component model and assembly schema) can easily be supported in PocoCapsule/C++ as user or third party defined DSMs. These DSMs can be defined directly on top of the PocoCapsule/C++ core IoC scheme, or progressively on top of other high level DSM schemes.

<This is a blank page>

Appendix D. Component-Based CORBA applications

D.1 The need for a component framework

Handcrafting a large scale distributed application from scratch is hard, expensive, risky, and bound for heavy maintenance burdens. The goal of the CORBA architecture was to reduce these pains by factoring out reusable low level functions/services in order for application developers to focus on high level business logic and domain issues.

With a CORBA middleware, an iteration cycle in developing a distributed application can be divided into the following three sequential phases:

- ***System partition and interfaces design*:** In this phase, architects/developers partition the application into business logic units (modules and objects) and define their interfaces using the OMG IDL[56].
- ***Implement business logic*:** In this phase, developers write business logic implementations for the partitioned units. These implementations support all operations of their respective IDL interfaces.
- ***Deployment and configuration (D&C)*:** In this phase, developers assemble individual business logic units together and/or enlist them to the underlying ORB runtime and/or CORBA common services (such as OMG Event/Notification and DDS) with desired policy or QoS configuration settings.

CORBA indeed simplified the first two phases of this development iterate cycle by making distributed applications similar to conventional well understood non-distributed OO applications. However, CORBA also brought in considerable framework level complexities and is far from satisfactory on simplifying the third phase. The programming models (APIs) of ORB server (such as the POA API) and most common services, such as Event/Notification and DDS, are heavily system or framework oriented and are awfully complex to most domain/business application developers and even cumbersome to most system application developers and middleware experts as well as ORB and OMG common services implementers themselves.

This situation[57] calls for a component framework that could substantially simplify CORBA applications for domain experts whose skills and interests are not low level middleware plumbing but solving high level domain issues and building sophisticated and/or intelligent business applications. This framework should even allow domain

[56] In MDA, developers could use UML as modeling language and generate CORBA IDL using tools.

[57] This is a generic situation, also observed in J2EE, WebServices, high performance computing (HPC), robotic motion control applications, or vertically in enterprise (EAI, ERP), healthcare (HL7, EHR/HISA), intelligent traffic and/or vehicle control systems (ITS), datacom and telecom (TMN, IN/AIN, Parlay/OSA, SIP, VoIP, and NMS/OSS/BSS), defense (SDR, ATC, C4I, JBI, and OACE).

experts with little programming skill to quickly, comfortably, and flexibly assemble, deploy, and configure sophisticated CORBA applications.

D.2 CCM: More trouble than worth

The CORBA Component Model (CCM) was supposed to be the solution, however, made the situation much worse. CCM significantly increased the complexity of all three phases of the CORBA application development cycle with much more contrived framework complexities and imposed a steep learning curve even for CORBA experts, not to mention the dreadful entry barrier for business application developers.

CCM also introduced a compliance barrier that not only tightly locks in compliant component implementations, but also locks out almost all existing CORBA business logic implementations (servants) and even certain client applications[58] and important OMG common services. To be reusable in CCM, these CORBA application implementations and common services have to be refactored or even completely redefined and reimplemented. All of these largely defeat the very purpose of having this component framework.

For instance, numerous scholastic publications claimed that CCM greatly reduced lines of code of OMG Event/Notification Service applications. Nevertheless, as a matter of fact, the "*Events as Valuetypes of IdL*" (EVIL) model of CCM not only forces developers to write more rather than less plumbing code[59], but also rules out most existing Event/Notification Service implementations[60] and almost all already deployed, standard based Event/Notification Service applications[61]. These standards/specifications and their service and application implementations had to be significantly rewritten to be compliant to and interoperable with the CCM EVIL. Even if additional plumbing code and such a massive reengineering exercise were acceptable, it would still incur considerable performance overhead and significant network bandwidth penalty due to the EVIL model requires valuetypes representing complex events/alarms to be inserted into and transferred as *CORBA::Any*s.

CCM was largely influenced by techniques developed during later 90's in the Apache Avalon project and EJB 1.0. These techniques have been abandoned in the mainstream industry for years and the EJB 1.x/2.x has become a classic negative example of architecture design. Nevertheless, taking either an ostrich attitude or a "if you can't convince them (users), confuse them!" tactic, CCM is still constantly being hyped with buzzwords such as "advanced", "mission-critical", and "lightweight", even though its techniques are out-of-date, its implementations are vulnerable and heavy. Whenever

[58] For instance, in some CCM implementations, it becomes very difficult to develop OMG asynchronous method invocation (AMI) client applications to access a CCM server.
[59] For instance, event valuetype factories, factory registration, and double dispatch.
[60] Among a handful (more than 6) Event/Notification Service implementation only one (VisiNotify) is able to support interoperable valuetype pass-through without non-portable vendor specific workaround (such as link or load user application specific valuetype factory into the channel executable).
[61] Examples of such applications include, for instance, almost all CORBA/TMN applications based on 3GPP, TMF and ITU-T standards/specifications.

being criticized for its heaviness and complexity, CCM always responses with its favorite equation claiming that advanced component frameworks for distributed mission-critical applications deserve to be heavy, complex, recondite, and hard to use, and their simplifications could only yield less featured inferior editions.

D.3 PocoCapsule: Simple is better

PocoCapsule/C++ IoC&DSM overturns the CCM's equation above definitely in the "David vs Goliath" face-off and reiterates the classic "less is more" aphorism by demonstrating an intuitive, easy to use, and lightweight while substantially powerful and reliable component and D&C framework for CORBA server applications, OMG Event/Notification service applications, OMG-DDS applications, JTRS-SCA SDR applications, and OMG-RTC robotic component applications, etc.

PocoCapsule/CORBA is a DSM framework built on top of PocoCapsule/C++ IoC&DSM. This framework does not force a particular component model and accepts virtually any C++ objects, referred to as *plain old C++ objects* (POCO), as component and is therefore neutral to all component models. Business logic object implementations and common services that were designed, developed, and built with or without knowledge of PocoCapsule/CORBA framework can all be seamlessly used and manipulated as components by PocoCapsule/CORBA and can still be tested and reused in their respective frameworks without PocoCapsule as well. These components include, for instance, existing CORBA 2.x business logic implementations (servants), CORBA 3.x and CCM components, JTRS-SCA core framework (CF) resources and devices, OMG-RTC components, and existing and new OMG common services such as OMG Event/Notification Service and DDS and their application object implementations (consumers, suppliers, data readers and writers). This POCO component model implies:

- There is no compliance barrier. Legacy implementations are seamlessly supported.
- There is no ramp up time and no learning curve[62] for another contrived component model.
- The straightforwardness of design and implementation phases of CORBA 2.x application development is retained.
- Existing and matured asserts (various vertical domain standards and implementations of ORBs, OMG common services, and applications) and investments (trainings) can mostly be preserved and are reusable explicitly with minimum cost, impact, uncertainty, and risk.

For the third development phase, the declarative assembly and deployment model of PocoCapsule/CORBA largely melts away plumbing complexities and eliminates the need for domain/business application developers to deal with or even to learn those low level system details. Many CORBA application scenarios and common services used to be

[62] A CORBA, CCM and COS services (Naming and Event/Notification) training course offered by one CCM advocate toke 15 weeks. In comparison, it needs less than 15 hours for a CORBA novice to be able to master mach more features in PocoCapsule/CORBA.

notoriously obscure to CORBA experts and dreadful obstacles to CCM now become fairly obvious and handy even to CORBA novices.

The complete developer guide and specification of PocoCapsule/CORBA framework is in a separate document (*PocoCapsule/CORBA POA server, Notification, DDS, SCA, and RTC C++ Developer Guide*[63]). The framework also comes with ten user examples, covering POA server, OMG Notification/Event, OMG DDS, OMG RTC, JTRS-SCA, etc. applications. These examples are available in the examples/corba directory of a PocoCapsule/C++ installation.

[63] http://www.pocomatic.com/docs/pococpp-corba-dev-guide.pdf