Concept/Topic: *Reference Architectures and Containers*

Context and Motivation

Reference architectures compile architectural patterns so that full quality attribute-driven design processes suggested by methods such as ADD can be streamlined by knowledge and design reuse: The frameworks and containers that implement the reference architectures and patterns have already made many decisions for the framework user (including architects and developers), for instance how to secure application components, how to configure system transaction boundaries (ACID-style), and how to log user activities. Such design challenges commonly arise in large and distributed (information) systems, among others:

- Component lookup and initialization is required when building and testing, when initializing production runtimes.
- Similar, but not identical (F)URPS requirements have to be balanced, for instance, regarding access control (authentication and authorization) and logging needs.
- Dependencies and variability have to be managed when handling change requests and when servicing multiple channels (mobile, Web, rich clients).
- Data and resource consistency management across data sources and backend systems has to be ensured (across systems in system context, within same system).

Reference architectures help you address such challenges. Each reference architecture promotes a particular architectural style. For instance, a dependency injection container decouples components using other ones from the implementation of these other components, which increases *flexibility* (as one aspect of supportability, the S in FURPS) and shifts the responsibility for component lookup from the application programmer to the framework offering the container abstraction (which has to be configured somehow).

Definition(s)

Lectures lessons 4 and 5 introduced the following concepts (among other things, see sibling fact sheet on component modeling):

- Architectural Style
- Reference Architecture
- Container and Managed Container, implementing patterns such as Inversion of Control (IoC) and Dependency Injection (DI)

Architectural styles are sets of principles and patterns aligned with each other to shape an application and make designs recognizable and design activities repeatable. The principles express architectural design intent; the patterns adhere to the principles and are commonly occurring (proven) in practice: "An architectural style determines the vocabulary of components and connectors that can be used in instances of that style, together with a set of constraints on how they can be combined. These can include topological constraints on architectural descriptions (e.g., no cycles) [...] or constraints related to execution semantics." (Shaw and Garlan (1996)). An architectural style improves separation of concerns and promotes design reuse by providing solutions to frequently recurring problems. Other benefits of architectural styles include a common language, allowing discussions of architecture aspects in a technology-agnostic way. Two examples of architectural styles are *Client/Server* and *Layered Architecture* (introduced in VSS, SE 1/2 and AppArch lesson 3). Multiple style collections exist, but no definite, current one. These are a few of these style collections:

- In Wirfs-Brock and McKean (2002), R. Wirfs-Brock talks about centralized control style, dispersed control: no centers, delegated control (OOP)
- Starke (2015) lists these (some of which qualify as architectural pattern but not as styles in our definition): Data Flow (Batch-Sequential, Pipes and Filter), Data Centric (Repository, Blackboard), Hierarchic (Mater-Slave, Layers, Ports-and-Adapter), Distributed (Client-Server, CQRS, Broker, Peer-to-Peer), Event Systems (Unbuffered Event Communication, Message/Event Queue, Message-Service), Interaction-Oriented (Model-View Controller, Presentation Model), REpresentational State Transfer (REST).

Reference Architectures (RAs) make architectural styles more concrete. They aim at accelerating design work, but do not yet reside on a project level of detail (single architecture and code implementing it); hence, RAs can and have to be adopted and refined by definition. For instance, the Application Architecture Guide from Microsoft defines several so-called application archetypes[1]; Java Enterprise Edition (JEE)[2] is an example of a container-based RA that is implemented in application servers from vendors such as IBM, Oracle, and RedHat. Other reference architectures are Reference Model of Open Distributed Processing (RM-ODP), the Amazon AWS cloud RAs, the IBM CCRA for Cloud Computing (CC) and the The Zachman Framework that looks at entire enterprises (systems of systems).

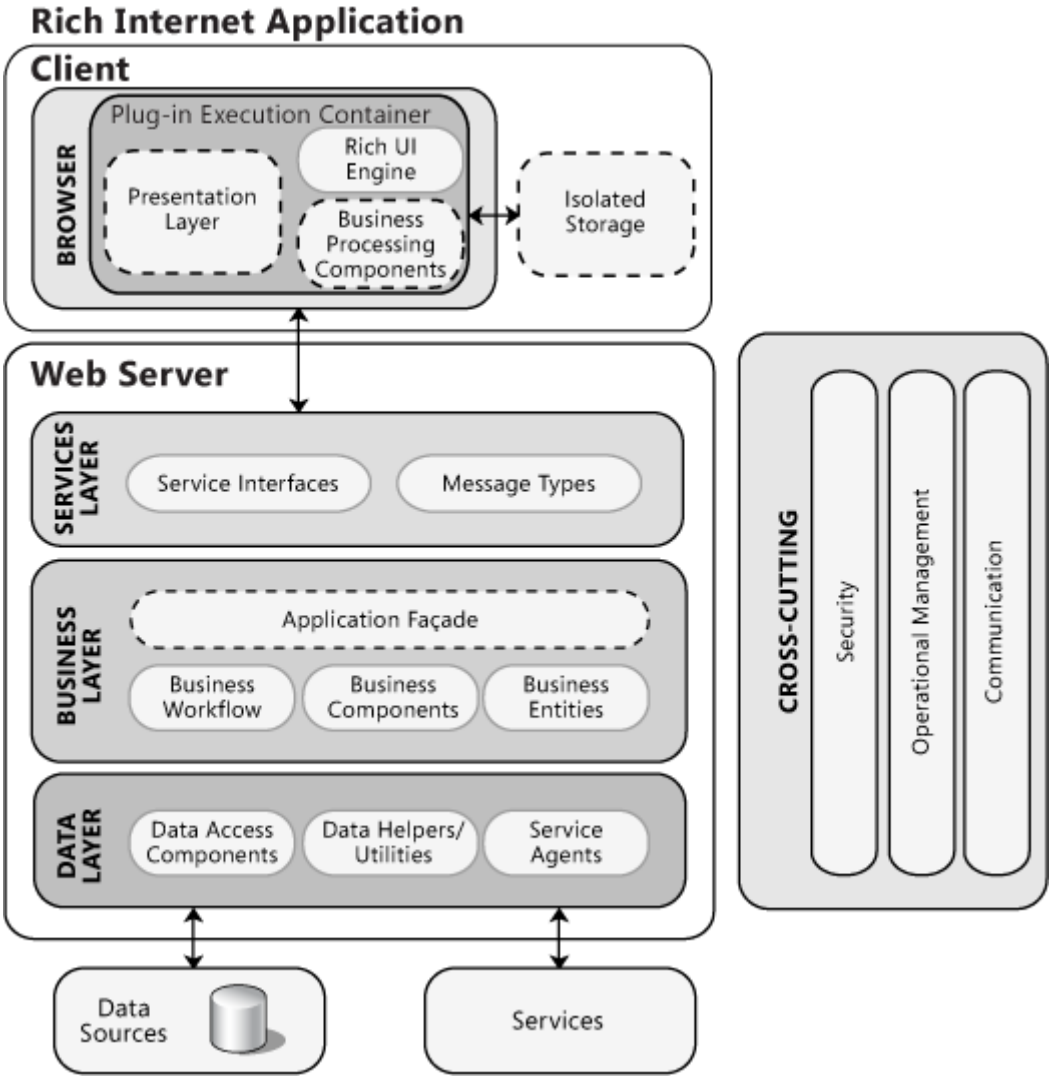


Figure 1: *Logical View on a Reference Architecture for Rich Internet Applications (RIAs)*

[1] https://msdn.microsoft.com/en-us/library/ee658107.aspx
[2] The *Enterprise* in JEE is any organization with advanced NFRs/QASs regarding mission-critical application software (e.g. performance, availability).

Containers are application-level frameworks for tier 2 of a 2-tier or 3-tier application (note the overloaded usage of the word container: application-levelc containers are not to be confused with container-based virtualization on the operating system level, for instance a la Docker). They implement patterns such as *Plugin*, *Inversion of Control* and *Dependency Injection*. A managed container has its own main routine, which is started as an operating system process (e.g., a demon process in Linux,. service in Windows); an unmanaged container is created and started by the application programmer, but takes over control after that. Spring is an example of an unmanaged container, JEE application servers such as JBoss/Wildfly contain one or more managed containers. Managed containers come with many additional design time and runtime tools for configuration, monitoring and other system management disciplines; unmanaged containers may offer such tool support via APIs optionally too.

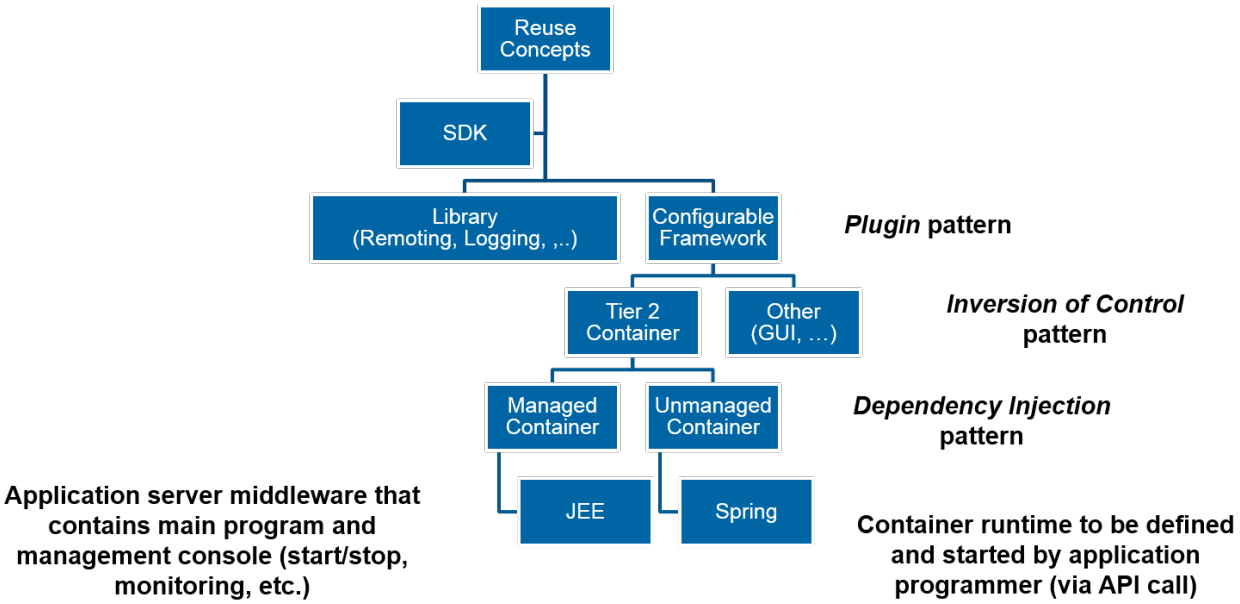


Figure 2: *Reuse Taxonomy: Libraries vs. Frameworks, which include Containers*

RAs such as JEE and containers such as Spring realize the vision of component-based development supported by middleware (here: application servers, managed and unmanaged containers): The basic idea is to let an application server (i.e., managed containers) do the hard work. For instance, key JEE tenets (dt. Grundsätze) are:

- Separation of concerns and information hiding, e.g. role-based application security and transaction boundaries taken care of by application server and its container(s), programmer can focus on business logic
- Shared services, resource pooling (e.g. database connections)
- Inversion of Control and Dependency Injection patterns
- Portability (to avoid "vendor lockin" and make it easy to switch from one JEE vendor to another for instance, from IBM to Oracle and vice versa)
- Declarative configuration (instead of imperative programming), e.g. transaction boundaries, cache sizes, resource pooling (in response to NFRs/QASs) via annotations

Spring Boot adds the principle "convention over configuration", inspired by frameworks for programming languages other than Java (for instance, Ruby on Rails). Whether this vision is sound and implemented well remains to be seen and/or is a matter of opinion; what looks simple at appears to save time when getting started often gets out of hand during debugging and maintenance (a lesson learned from model-driven software engineering generating source code e.g. from UML models). See figure "IoC in Spring embedded into JEE (DDD Sample Application)".

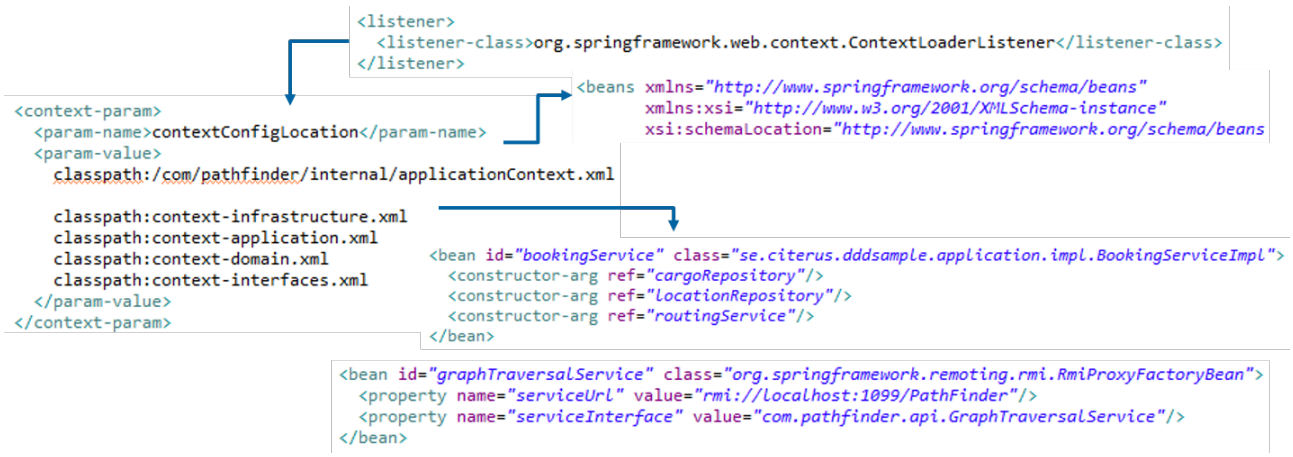


Figure 3: *IoC in Spring embedded into JEE (DDD Sample Application)*

Patterns

Start exploring the container patterns in the article "Inversion of Control Containers and the Dependency Injection pattern"[3] by M. Fowler. It discusses different dependency management options with their pros and cons. Simplified explanations are also available (see links in lecture and exercise).

Plugin pattern The pattern sketches a centralized runtime configuration via files (XML, JSON, YAML, Java properties) or annotations a.k.a. aspects in Aspect-Oriented Programming (AOP). See figure "Plugin pattern (from PoEAA)". Known uses of the pattern are Eclipse plugins, Web browsers, Web container and EJB container in JEE, and the Spring Framework.

Inversion of Control (IoC) pattern See figure "IoC pattern synopsis". The pattern is also known as the "Hollywood principle".

Dependency Injection (DI) pattern See figure "DI pattern synopsis". Three types of Dependency Injection exist: constructor, setter, and interface injection. The Spring community used to prefer setter injection; in the PetClinic sample application, constructor injection is used.

Service Locator See figure "Service Locator pattern synopsis". A Service Locator is a simpler alternative to DI.

See websites Patterns in the Composite Application Library[4] from Microsoft for detailed explanations of all patterns. The content os these websites is marked as outdated but still available.

Example(s)

Application servers such as JEE compliant ones, .NET, and Spring (Spring Boot/Spring framework) qualify as examples (among others):

- The application Archetype "Web Application" in the Microsoft Application Architecture guide[5]

[3] https://martinfowler.com/articles/injection.html
[4] https://msdn.microsoft.com/en-us/library/ff921146.aspx
[5] https://msdn.microsoft.com/en-us/library/ee658099.aspx

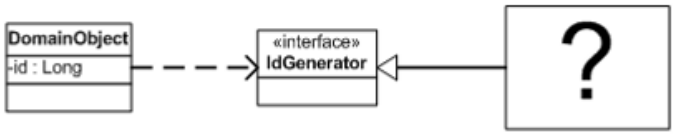
| P of EAA Catalog |

Plugin

by David Rice and Matt Foemmel

Links classes during configuration rather than compilation.

For a full description see P of EAA page **499**



Separated Interface (476) is often used when application code runs in multiple runtime environments, each requiring different implementations of particular behavior. Most developers supply the correct implementation by writing a fac-tory method. Suppose you define your primary key generator with a Separated Interface (476) so that you can use a simple in-memory counter for unit testing but a database-managed sequence for production. Your factory method will most likely contain a conditional statement that looks at a local environment variable, determines if the system is in test mode, and returns the correct key generator. Once you have a few factories you have a mess on your hands. Establishing a new deployment configuration - say "execute unit tests against in-memory database without transaction control" or "execute in production mode against DB2 database with full transaction control" - requires editing conditional statements in a number of factories, rebuilding, and redeploying. Configuration shouldn't be scattered throughout your application, nor should it require a rebuild or redeployment. Plugin solves both problems by providing centralized, runtime configuration.

Figure 4: *Plugin pattern (from PoEAA)*

Context

Many solutions in same problem domain need to be build; middleware will be used to implement common parts with guaranteed service levels (QoS).

Problem

How can the framework control the server-side execution, but still be extensible with application features (without recompilation)?

> *One important characteristic of a framework is that the methods defined by the user to tailor the framework will often be called from within the framework itself, rather than from the user's application code. The framework often plays the role of the main program in coordinating and sequencing application activity. This inversion of control gives frameworks the power to serve as extensible skeletons. The methods supplied by the user tailor the generic algorithms defined in the framework for a particular application.*
>
> *-- Ralph Johnson and Brian Foote*

Solution

Hand over flow management responsibility from a main program to a QoS-aware framework; configure it with Lambdas (functional programming), events or framework-defined interface implemented by application components (beans).

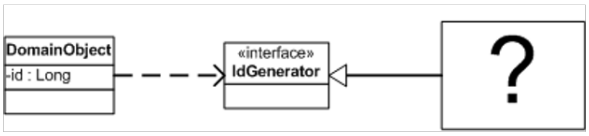
Figure 5: *IoC pattern synopsis*

Context

A plugin links classes during configuration rather than compilation.

Problem

How do we assemble these plugins into an application?



Solution

Extend the Inversion of Control pattern commonly found in user interface frameworks and application servers to support *constructor injection*, *setter injection* and *interface injection* (via files or annotations).

Figure 6: *DI pattern synopsis*

```
class MovieLister...
  MovieFinder finder = ServiceLocator.movieFinder();

class ServiceLocator...
  public static MovieFinder movieFinder() {
      return soleInstance.movieFinder;
  }
  private static ServiceLocator soleInstance;
  private MovieFinder movieFinder;
```

Figure 7: *Service Locator pattern synopsis*

- The Spring sample application `PetClinic`[6]
- JEE tutorial[7]

J(2)EE patterns are collected in Alur, Malks, and Crupi (2013), refining and complementing the ones in Fowler (2002).

Other DI containers are Dagger for Android (note: the latter is not an official asset), PicoNet (a often cited but seemingly stalled project) and Unity for .NET.

Application of the Concept in Products and Projects

- Microsoft products that implement the application archetypes
- JEE application servers such as JBoss/Wildfly, IBM WebSphere, and Oracle WLS (formerly known as BEA Web Logic Server)
- Spring family of projects (Spring Boot, Spring Framework, etc.)

Tips and Tricks

The Business Logic Layer (BLL) is both "brain" and "soul" of a two- or three-tiered application:

- Avoid a thin BLL for the sake of JEE standards compliance (pass through).
- Remind components in other layers (and their owners) about their responsibility for data integrity etc. but do not rely on them (for instance, check input at layer boundary).
- Distinguish logical and physical usage of the Layers pattern.
- Not each business object needs to run on its own server and offer a remote interface (not even a virtual one)
- Component identification and specification should be explicit steps in the architecture design process; not all use cases/user stories or analysis-level domain model entities justify an implementation-level representation as components (or classes).
- Avoid "architectural astronauting"[8] when applying/crafting reference architectures: alway map the abstract components to something concrete and tangible (by the way this is one of the reasons why the ZIO version of CRC cards has the candidate implementation technologies/known uses cell).

Some (subjective) advice about dependency injection and annotation usage in frameworks is:

- Be parsimonious with annotation usage and introduction of custom ones (unless you have fully bought into the concept).

[6] https://github.com/spring-projects/spring-petclinic
[7] http://www.turngeek.press/javaeeinaday/front-matter/introduction/
[8] https://www.joelonsoftware.com/2001/04/21/dont-let-architecture-astronauts-scare-you/

- Never lose control over development and deployment artifacts: what is installed where, and what does it do?
- Have a "plan B" in case the annotation auto magic gets out of hand (explicit configurations, even POJO usage). In other words, fall back to more conservative approaches to dependency management if in doubt, for instance by switching of auto wiring and component scanning in critical environments.
- Carefully observe whether the frameworks are documented sufficiently, and manage to keep documentation current when upgrading to new versions, refactoring, and introducing new features (which might not be backward compatible).
- Never loose sight of the original project goals, keep track of effort vs. benefit of framework usage (in the short run and in he long run).

Generally speaking, be aware of the "The Sorcerer's Apprentice" effect, (dt. "Zauberlehrling"), with the framework and container technology taking the role of the broom in the metaphor.[9]

To summarize the information in this fact sheet in the context of other lecture topics:

- When/before transitioning from component specification to component realization, a buy vs. build decision is made (leveraging CRC cards)
- The components that will be realized in code are implemented in one or more classes (assuming Object-Oriented Programming, OOP) and deployed into a server-side framework (tier 2 container)
- The containers in JEE application servers and Spring realize the Inversion of Control pattern (also known as Hollywood principle). THere is a tradeoff between convenience and control. The complexity of the middleware is an element of risk, particularly for small organizations.
- The configuration management in the container is simplified/achieved by the patterns Plugin, Service Locator and Dependency Injection
- Full JEE application server (e.g. JBoss) with application management vs. "lightweight", unmanaged container (e.g. Spring Framework/Boot. Both offer services such as security, transaction management, data access

More Information

Related Topics and Concepts

Client-Server Cuts (CSCs), Component Modeling, Service-Oriented Architecture and Microservices

Miscellaneous Links

- See links in slides of lectures 4 and 5, as well as those in the FAQ sections of exercise 4 and 5.

References

Alur, Deepak, Dan Malks, and John Crupi. 2013. *Core J2ee Patterns: Best Practices and Design Strategies*. 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall Press.

Fowler, Martin. 2002. *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Shaw, Mary, and David Garlan. 1996. *Software Architecture: Perspectives on an Emerging Discipline*. Upper

[9] Note that metaphors some times get out of hand too; hopefully the framework/container does not turn into the apprentice, with you turning into the broom.

Saddle River, NJ, USA: Prentice-Hall, Inc.

Starke, Gernot. 2015. *Effektive Software-Architekturen*. 7. Auflage. Hanser-Verlag.

Wirfs-Brock, Rebecca, and Alan McKean. 2002. *Object Design: Roles, Responsibilities, and Collaborations*. Pearson Education.