

# Dynamically Adaptable Applications with iPOJO Service Components

Clement Escoffier and Richard S. Hall

LSR, 220 Rue de la Chimie, BP 53  
38041 Grenoble Cedex 9, France

{clement.escoffier, richard.hall}@imag.fr

**Abstract.** Traditional component models and frameworks simplified software system development by enabling developers to break software systems into independent pieces with well-defined interfaces. This approach led to looser coupling among the system pieces and enhanced possibilities for reuse. While the component-based approach led to advancements in the software development process, it still has its limitations. In particular, after a component-based application is developed and deployed it typically is a monolithic and static configuration of components. The advent of service-oriented component (SOC), the rise in popularity of consumer devices, and the ubiquity of connectivity have fostered a growing interest in applications that react dynamically to changes in the availability of various services. To simplify the creation of such dynamic software systems, it is possible to borrow concepts from SOC and include them into a component model, resulting in a service-oriented component model that specifically supports dynamically adaptable applications. This paper presents iPOJO, a service-oriented component framework to help developers implement dynamically adaptable software systems.

**Keywords:** Service Orientation, Component Orientation, Dynamic Adaptable Software, Software Composition

## 1 Introduction

Traditional component models and frameworks simplified software system development by enabling developers to break software systems into independent pieces with well-defined interfaces. This approach led to looser coupling among the system pieces and enhanced possibilities for reuse. Component-based applications are constructed by developing, selecting, and assembling the individual system components. The components in the application are typically bound together with standard connectors and/or custom “glue” code. While the component-based approach led to advancements in the software development process, it still has its limitations.

In particular, after a component-based application is developed and deployed it typically is a monolithic and static configuration of components. The characteristics of the component-based approach (i.e., loose coupling, third-party component selection, and reuse) only extend to the development portion of the software life cycle, not to the run-time portion. While there are some exceptions to this characterization, such as the use of plugins, these exceptions tend to be low scale and coarse grained. The ad-

vent of service-oriented computing (SOC), the rise in popularity of devices such as cell phones, PDAs, and MP3 players, and the ubiquity of both wired and wireless connectivity (e.g., WiFi, Bluetooth) have fostered a growing interest in applications that react dynamically to changes in the availability of various services. For example, software systems may dynamically react to the presence of a Bluetooth-enabled mobile phone or printer in order to offer related services to the end user.

While such dynamic capabilities are feasible in most component models and frameworks, there is no direct support for dynamic component composition. As a result, the developer must manage this non-functional aspect by hand. To simplify the creation of software systems that react dynamically to the changing availability of services, it is possible to borrow concepts from SOC and include them into a component model, resulting in a service-oriented component model that specifically supports dynamically adaptable applications.

This paper presents iPOJO, a service-oriented component framework to help developers implement dynamically adaptable software systems. The rest of the paper is organized as follows. The next section presents SOC concepts and how they are useful in supporting dynamic composition. Section 3 describes the general principles of a service-oriented component model. Section 4 describes the approach of the iPOJO service-oriented component framework. Section 5 describes the iPOJO framework and how to design and implement dynamically adaptable applications, while section 6 describes iPOJO implementation details and experimentation. Section 7 presents related work, followed by a discussion of current and future work in section 8. Lastly, section 9 presents the conclusions.

## 2 Service-Oriented Computing Concepts

Service-oriented computing (SOC) [9][17] is a paradigm that utilizes services as fundamental elements for application design. The central objective of the service-oriented approach is to reduce dependencies among “software islands,” where an island is typically some remote piece of functionality accessed by clients. By reducing such dependencies, each element can evolve separately, so the resulting application is more flexible than monolithic applications.

SOC is based on three actors:

- A *service provider* offers a service.
- A *service consumer* uses a service.
- A *service broker* contains references to available services.

Another central concept to SOC is the *service specification*, which is a description of the functionality provided by a service. Service providers implement a specific service specification and service consumers know how to interact with services implementing the specifications they require. Among these three actors are three kinds of interactions: *service publication* between the provider and the broker to offer services for use, *service discovery* between the consumer and broker to find desired services, and *service invocation* between the consumer and provider to actually use the service.

From these concepts, SOC applications can exhibit interesting characteristics, such as:

- **Loose coupling:** a consumer does not need to know anything about the service implementation.

- Late binding: a consumer uses a broker to find desired services at run time.
- Dynamic resilience: a service consumer cannot rely on the same service implementation being returned by the broker between uses.
- Location transparent: providers and consumers are oblivious to the underlying communication infrastructure (e.g., local versus remote, specific protocols, etc.).

To design complex service-oriented applications, it is necessary to compose services to provide higher-level services, which means that providers may require other services to provide their own service. Current approaches to SOC, specifically in web services, offer process-oriented solutions to this issue; however, service-oriented applications can be difficult to develop since the mechanisms tend to require a lot of developer effort. Indeed, developers need to manage service publication/revocation, required service discovery, and run-time tracking of services. The SOC paradigm facilitates the implementation of dynamically adaptable software systems by supporting loose coupling and late binding, but is not sufficient in itself. The next section describes how SOC concepts can be merged into a component model to provide a more complete solution for dynamically adaptable software systems.

### 3 Service-Oriented Component Model

In [6], the general principles of a service-oriented component model were introduced, which are:

- A service is provided functionality.
- A service is characterized by a service specification, which describes some combination of a service's syntax, behavior, and semantics as well as dependencies on other services.
- Components implement service specifications, which may exhibit implementation-specific dependencies on services.
- The service-oriented interaction pattern is used to resolve service dependencies at run time.
- Compositions are described in terms of service specifications.
- Service specifications provide a basis for substitutability.

The model that results from these principles is rather flexible and powerful. It promotes service substitutability since compositions are defined in terms of specifications, not specific implementations. This notion of service substitutability is strengthened by recognizing two levels of dependencies, both specification and implementation levels. Traditional component orientation only recognizes implementation dependencies, hindering substitutability. Service orientation does not typically describe service dependencies as part of the service specification, which eliminates forms of behavior parameterization and structural service composition. Dependency description also simplifies composition because explicit wiring of constituent service specifications is not necessary since it can be inferred. Lastly, the service-oriented architecture and interaction pattern make it possible to defer service implementation selection until run time, which enables the creation of sophisticated adaptable applications.

When using a service-oriented component approach to build an application, the application is decomposed into a collection of interacting services. The semantics and

behavior of these services are carefully described independently of any implementation. By doing so, it is possible to develop the constituent services independently of each other as well as to have variant implementations that are easily interchangeable. Variant implementations can be used, for example, to support different platforms or different non-functional requirements.

Once the application's services have been defined, it is possible to define a mapping to a set of components for implementation. A component may implement zero or more services and there is no requirement on how the mapping from service specification to component implementation occurs. For example, if certain services are related, then it might make sense from a cohesion or performance point of view to implement them using a single component; however, it is not required. Additionally, whether a service represents local or remote functionality is largely irrelevant. If a service does represent remote functionality, then the resulting component implementation is merely a proxy stub for the remote service, which can be treated like any other component implementing a local service.

As pointed out in [7], one of the set of components that comprise a service-oriented component application is typically a “core” component that contains the main service composition or process that guides the application's execution. Other component instances provide the services used by the core component and these instances can themselves require services provided by other instances. This approach is similar to the exogenous connector approach promoted in [13].

In traditional component-oriented composition, the component selection process for a composition occurs at design time. The selection process for a service-oriented composition occurs at run time as component instances are created inside the execution environment. The execution of the application starts the moment the main component instance's dependencies are satisfied. The application composition is thus an abstract descriptor that could be used, for example, by a deployment system to deploy components that satisfy the service specifications required by the composition. The resulting application configuration depends on the specific set of deployed components, which may vary per platform or even dynamically at run time.

## 4 iPOJO Approach

iPOJO is a service-oriented component framework supporting the service-oriented component model concepts of section 3. One of the main goals of iPOJO is to keep service-oriented component development as simple as possible, which means keeping the component as close to a “plain old Java object” (POJO) as possible. The code of a component should focus on business logic, not on SOC mechanisms or non-functional requirements. To achieve this goal, iPOJO provides a component container that manages all SOC aspects, such as service publication, service object creation, and required service discovery and selection. The component developer only focuses on two tasks:

- Implementing the business logic.
- Configuring the component container.

The business logic is domain specific, but implementing it is simplified since it need not contain code that is component model specific. The POJO component is connected to iPOJO by configuring the component container, which consists of declaring

component metadata that will be used by the container for run-time management. Component metadata declares information such as provided services, required services, and configurable properties.

Figure 1 illustrates how the container automatically manages SOC activities, such as providing or requiring services. The container of C2 publishes its provided service. C1 also provides a service, but it has a dependency on the service that C2 provides. At run time the container of C1 tries to select a service implementation to resolve the dependency. If the dependency is not resolvable, the container will not publish C1's provided service. On other hand, if the dependency is resolvable, then the container will publish the service and manage component instance creation, not creating the component instance until its service is actually used.

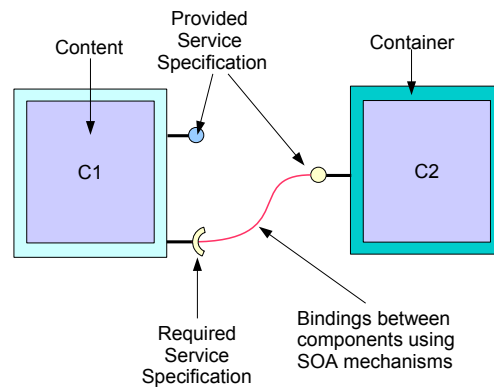


Fig. 1: Service component binding.

The container for each component must be configured by declaring the metadata to provide and require the appropriate services. It should also be noted that the container manages the ongoing dynamic availability of services, which means that it automatically deals with publishing and revoking services as dependencies are resolved and broken at run time if new component instances are introduced or existing ones are removed.

## 5 iPOJO Framework and Component Composition

iPOJO's approach to composition is very similar to traditional component composition, except that iPOJO compositions are in terms of service specifications rather than component instances, which allows for late selection and binding of component instances and substitutability. An iPOJO component declares its required services in its metadata and the iPOJO component container uses this information at runtime to automatically manage any necessary bindings between the component instance and any required services. The following subsections discuss how to implement iPOJO service components and the run-time support available to them.

## 5.1 Service Component Implementation and Description

To design an application with service components, developers need to describe which services a component requires (service dependency) and provides (provided service). With this information, iPOJO can create a composition at runtime. iPOJO tracks required services at runtime and “injects” required services into the component when they become available.

The component in figure 2 illustrates a service component. This component exposes a multimedia message service (MMS) that allows clients to send an MMS message containing pictures from any number of attached cameras. To achieve this, the component requires two services: the first required service is *MmsService*, which allows a client to send an MMS message, and the second service is *Camera*, which allows a client to take a digital photograph. The metadata for the component's camera service dependency indicates that it is an aggregate dependency, meaning that it can be bound to one or more camera services.

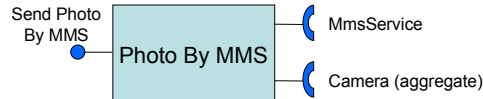


Fig. 2: Architecture of the Photo By MMS component.

### Service Component Implementation

The iPOJO approach is based on POJOs. Developers should only manage the logic of the component, forgetting service discovery and service publication. The code snippet in figure 3 shows a possible implementation of the above example component.

```
package fr.imag.adele.escoffier;
...
public class PhotoByMMS implement PhotoSender {
    private MmsSender sender;
    private Camera[] cameras;

    public void send() {
        MMS mms = new MMS();
        synchronized (cameras) {
            for (int i = 0; i < cameras.length; i++) {
                mms.add(cameras[i].getPhoto());
            }
        }
        sender.send("1234567890", mms);
    }
}
```

Fig. 3: Photo By MMS implementation

The component implementation is very simple. Service dependencies are simply coded with the assumption that the services are available as member fields (e.g., `sender` and `cameras`); the one caveat is for aggregate dependencies, which require a synchronized block to avoid run-time list modification during the loop. The component provides the `PhotoSender` service by simply implementing the service in-

terface.

### Service Component Description

When a component is implemented, it needs metadata to describe which services are required and provided. Figure 4 shows the description of the previous component. The metadata declares two service dependencies and a provided service. At a minimum, a service dependency must specify the needed service specification name (the fully qualified Java type), but may also contain optionality, cardinality, and filter information.

```
<iPOJO>
<component className="fr.imag.adele.escoffier.PhotoByMMS">
  <dependency field="sender"/>
  <dependency field="cameras"/>
  <provides/>
</component>
<instance component="fr.imag.adele.escoffier.PhotoByMMS"
  name="Sender"/>
</iPOJO>
```

Fig. 4: iPOJO Metadata for the PhotoByMMS component

In the example, the first service dependency is for *MmsService*. The declared dependency contains only the component's member field name to which the dependency will be associated; it is not necessary to specify the service specification type since it can be identified from the member field type using reflection. By default, service dependencies are assumed to be mandatory. If a dependency is declared as optional, then the *nullable object pattern* is used to avoid null-case testing by the component developer.

The second dependency is for *Camera* services; it is an aggregate dependency. All *Camera* services will be tracked. The metadata does not have to express the cardinality explicitly. Instead, reflection can determine that the field type in the component class is an array, which iPOJO interprets as an aggregate dependency. Aggregate dependencies can also be optional. An empty array is returned if no consistent providers are available.

The component declares that it provides a service by explicitly declaring it in its metadata. However, the service type need not be mentioned explicitly since it can also be inferred using reflection. The metadata can also contain properties to attach to a published service, which can then be used for service filtering; the current example does not include any service properties.

### Composition Binding Description

iPOJO components declare their dependencies on other service specifications. As a result, composition bindings do not need to be explicitly declared since they can be inferred from the individual component metadata. At runtime, iPOJO injects the final bindings into the components and also manages the dynamic availability of the services associated with the bindings and consequently the life cycle of the component instances.

## 5.2 Dynamic Run-Time Composition Management

Managing the dynamic availability of services is difficult to do manually and it results in code that mixes business and non-functional logic. By separating the component class and the component metadata, iPOJO is able to externally manage service dynamics on behalf of the component; figures 5 and 6 depict this process.

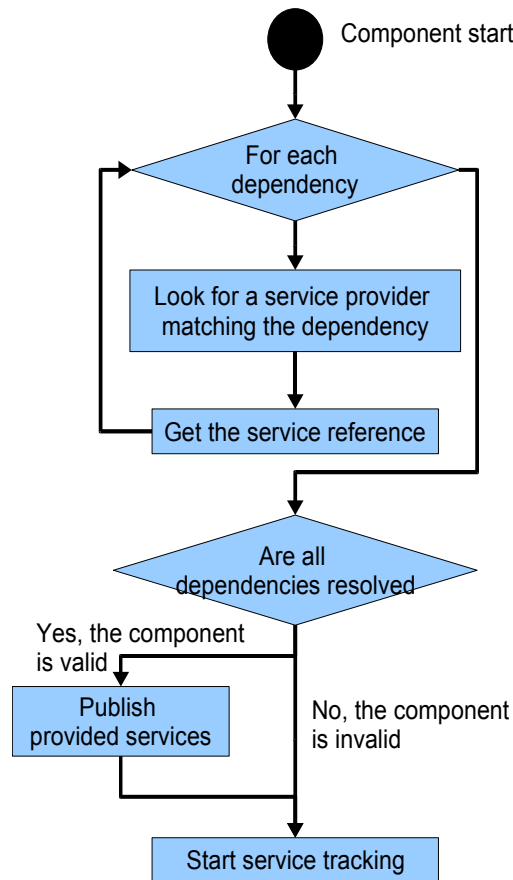


Fig. 5: Component initialization process.

When a component starts, for each service dependency iPOJO discovers all matching providers and injects one or more references to them as necessary into the component instance (see figure 5). If the component provides a service, iPOJO manages service publication according to the state of the component. A service component is *valid* if all service dependencies are resolved (i.e., at least one service provider exists for each mandatory dependency). In all other cases, the service component is *invalid*. An invalid component cannot have its service published, since its requirements are not met. As soon as the component becomes valid, its services can be published. As a component is initialized and enters the valid or invalid state, iPOJO continues to listen for service events indicating service arrival or departure and updates the component's state accordingly (see figure 6). For aggregate dependencies, iPOJO looks for all

matching service providers and injects them as a list of service references. A mandat-

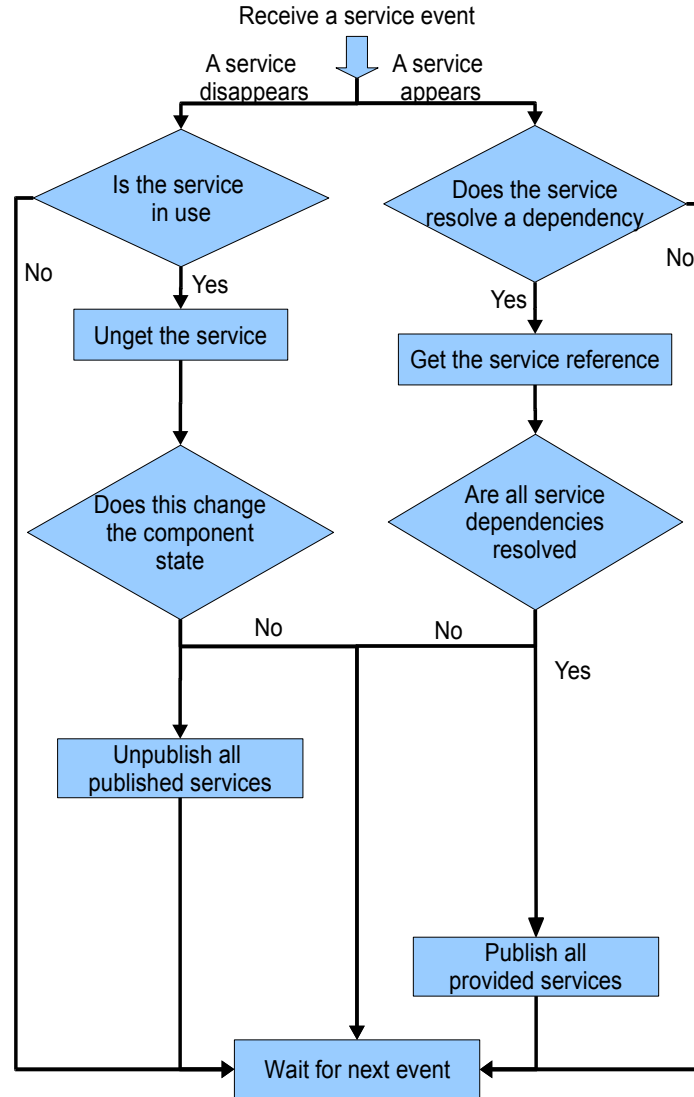


Fig. 6: Service event processing.

ory aggregate service dependency is resolved if it at least one service is available.

Figure 7 illustrates the impact of dynamic service availability on a service component. The component in the figure requires a service and provides another. The environment contains two providers matching with the required service. In (a) the service component is bound to the first service provider. Consequently, its provided service is published because all of its service dependencies are resolved. In (b) the bound service provider goes away. The original component can no longer use the departed service and must find another one. iPOJO looks for and finds another provider and cre-

ates a binding to it (c). During this time the instance is frozen. When the second provider goes away in (d), iPOJO is unable to find a replacement. As a result, the original component becomes invalid and iPOJO revokes its provided service.

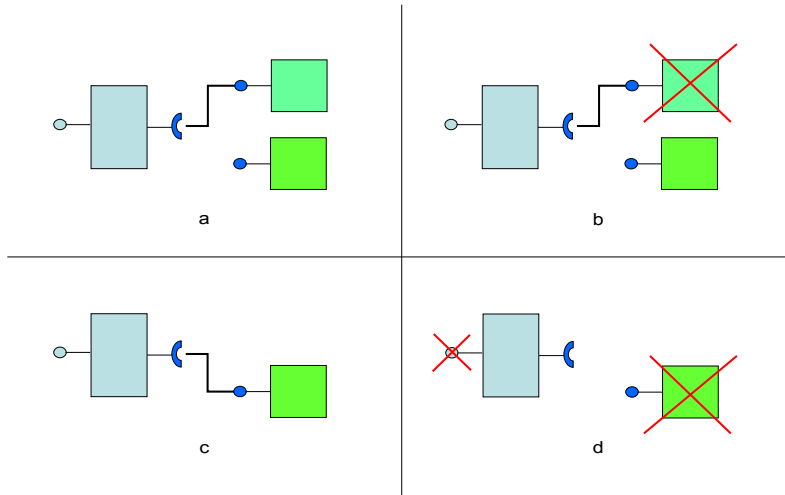


Fig. 7: Run-time composition and dynamics management

For aggregate dependencies, such as the example in the previous section, iPOJO needs to track all matching services and inject this set into the component instance. This set is automatically updated when a new service arrives or an existing one disappears; figure 8 depicts this behavior. An aggregate dependency is unresolved when no service implementation can be found at run time.

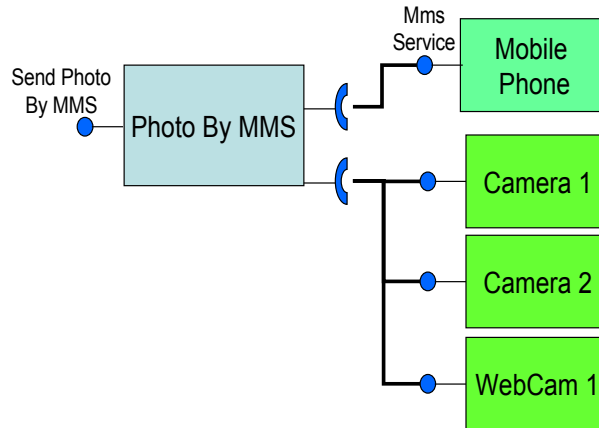


Fig. 8: Bindings for the Photo By MMS component.

## 6 Implementation and Experimentation

iPOJO is implemented for the OSGi Service Platform [15] and is available as a sub-project of the Apache Incubator Felix<sup>1</sup> project. iPOJO has been used in an industrial prototype for an European project named ANSO. This section briefly describes how iPOJO is implemented and how it is used in the ANSO project.

### 6.1 iPOJO Implementation

iPOJO is implemented on top of the OSGi Service Platform, which defines a framework to dynamically deploy services in a centralized (i.e., non-distributed) environment. The core OSGi framework automatically manages aspects of local service deployment, such as Java package dependency resolution, but leaves service dependency management as a manual task for component developers. iPOJO is built on top of the OSGi service platform for three main reasons:

- It is service-oriented platform,
- It is applicable to a large range of use cases (from mobile phones to application servers), and
- It is a dynamic platform, which is particularly interesting from a research perspective.

Although iPOJO is built on top of the OSGi Service Platform, the concepts it embodies are applicable to any service-oriented platform. The name *iPOJO* is derived from the phrase “injected POJO”, since the general approach of iPOJO is to inject POJOs with byte code to perform the management of non-functional behavior. Specifically, at component packaging time iPOJO instruments the component byte code to intercept all member field accesses. With this simple modification, iPOJO is able to inject member field values inside of the component when needed.

At runtime, iPOJO uses the OSGi service registry to discover, track, and publish services. iPOJO also manages component property configuration, dynamic service properties, and publication of component factory services.

### 6.2 Experimentation

The ANSO project, supported by ITEA, is an European project attempting to design a residential gateway. The gateway is intended to manage devices available in the home and provide higher-level services like alarm management and video-on-demand. A prototype of the ANSO gateway [1][2] was developed using iPOJO to manage available services. The motivation for using iPOJO was due to the fact that residential environments are very dynamic: devices (e.g., mobile phones, PDAs) appear and disappear dynamically and device state may change often as well as the end user context. iPOJO is used to aide developers in creating applications in such an environment; developers do not need to manage service events, device discovery, and invocation protocols.

Additionally, iPOJO is used by EDF (French electricity company) to manage multi-modal interactions. In this context, iPOJO is used to manage dynamic availabil-

---

<sup>1</sup> <http://incubator.apache.org/felix/>

ity of interactive devices (e.g., a mouse or joystick) in order to find the best configuration for disabled people [18].

## 7 Related Work

Component models and frameworks are not new. Numerous well-known examples exist, such as Common Object Model (COM) [3], JavaBeans [19], Enterprise JavaBeans [20], Fractal [4], and CORBA Component Model [14]. These component models and frameworks target various application domains. These approaches typically have a significant developer cost associated with them, such as the need to implement specific interfaces, extend specific base classes, and use specific application programming interfaces. A newer trend for component models is appearing that promotes the notion of POJOs, like EJB 3.0 [21] and Spring [12]. In these approaches, like in iPOJO, developers program POJOs and then configure a *container* that contains the POJO. Despite these similarities, these component models do not tackle the issues of dynamic availability and dynamic run-time composition.

Using components to implement services is becoming relatively popular [22][23]. Some service-oriented component models exist, like Jini, Service Component Architecture (SCA) [10], Service Binder [5], Declarative Services [16], and Spring-OSGi [11].

Jini is a Java-based distributed middleware platform that supports multiple service registries. Jini uses the concept of service leasing as a mechanism to limit the time a client can access a service. However, Jini does not support a composition model and is intimately tied to Java remote method invocation (RMI).

SCA provides a service-oriented component model mainly designed for web services. SCA defines an assembly model for loosely coupled web services. SCA components can support several kinds of implementation languages, e.g., Java, C++, and BPEL [8] processes. The SCA programming model uses Java annotations and is close to POJOs. SCA does not address dynamic availability and does not manage dynamic composition.

Declarative Services in the OSGi Service Platform Release 4 specification was inspired by the work on Service Binder. Both address building component-based applications from dynamically available services. The approach taken by these component frameworks is somewhat complex since it involves using stylized programming and specific application programming interfaces. iPOJO is actually a continuation of the Service Binder work and attempts to rectify its shortcomings. iPOJO also provides an extensibility mechanism so that it can address an open ended set of issues.

Spring-OSGi is the integration of the Spring framework with the OSGi framework. As with iPOJO, this new service component model uses POJOs and tries to address dynamic availability. Spring-OSGi uses an aspect framework to inject service dependencies. However, the developer need to explicitly manage exceptions when an unavailable service is used. Spring-OSGi also does not manage dynamic service properties and component factories.

## 8 Current and Future Work

This section explores current and future work that is being investigated as part of the overall research strategy of iPOJO. These issues can be divided into two main categories: service specification description and hierarchical composition.

### 8.1 Service Specification Description

Typically, service description is limited to the service interface definition and a set of properties. In order to automatically compose services, it is necessary to have a richer description of services. One simple example is the need for service-level dependencies. iPOJO's approach is to define compositions in terms of services, but without service-level dependencies then all composition must occur in glue code or within component implementations, which limits reusability and substitutability.

The iPOJO approach allows developers to expose service-level dependencies at design time (see figure 9). The benefit is that these dependencies enable simple structural composition purely in terms of services so that management and verification of these dependencies can be offloaded to the component framework. This is different than web service composition, where services merely provide functionality and do not expose any structural information. Additionally, service specifications become richer since it is now possible to parameterize service behavior in a well-defined way and create service specifications that follow patterns like model-view-controller. The service consumer is not impacted by this increased richness and continues to use the service as a black box.

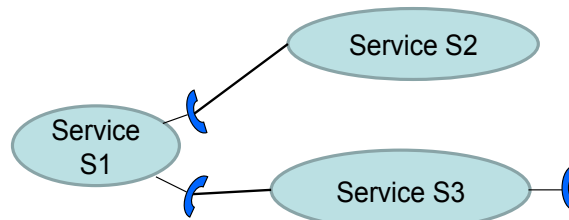


Fig. 9: An example of structural service composition.

It is also necessary to investigate forms of semantic, behavioral, and contextual description of services. Such additional richness will further improve the component framework's ability to determine when a service is appropriate for a given composition or which service to choose when many potential candidates exist.

### 8.2 Hierarchical Service Composition

Generally, service compositions are created using orchestration, like BPEL. In such a case, an orchestration engine manages the service invocation sequence and the information exchange among services.

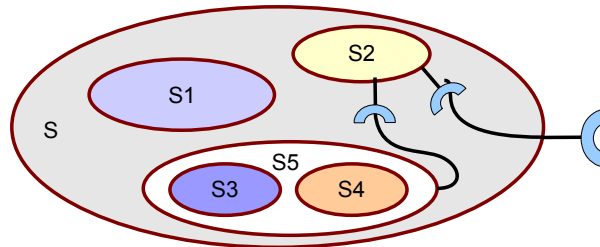


Fig. 10: Composite service.

Another way to create service compositions is following a more component-oriented approach using hierarchical compositions to create composite services. Unlike traditional component-oriented approaches, these composite services are expressed in terms of services instead of component instances to enable late selection and binding, which enables dynamic adaptability through substitutability. Figure 10 depicts an example of hierarchical service composition.

As service-oriented component applications grow in size and complexity, composite services also provide a way to mitigate this complexity by further subdividing the application. A composite service creates a scope that encapsulates the constituent services from external services and vice versa. A composite service can import and export services from/to its parent composite. The service bindings among components inside a composite service are resolved only with services available in the composite or those which are imported.

## 9 Conclusion

Traditional component models and frameworks simplified software system development by enabling developers to break software systems into independent pieces with well-defined interfaces. While the component-based approach led to advancements in the software development process, it still has its limitations. In particular, after a component-based application is developed and deployed it typically is a monolithic and static configuration of components. This contrasts the growing interest in creating applications that react dynamically to changes in the availability of various services.

This paper presented iPOJO, a service-oriented component framework that is trying to simplify creating dynamically adaptable software systems. The iPOJO approach is based on a model that combines concepts from service-oriented computing with component orientation. iPOJO implements its component model using byte code instrumentation, enabling the use of simple POJOs as components. From this foundation, iPOJO is able to manage all service-oriented aspects of its model on behalf of the components (e.g., service dependency resolution, service publication, dynamic service properties, and ongoing service availability tracking). The result is applications that are dynamically adaptable to changing service availability. iPOJO has demonstrated its usefulness in various prototypes and is available as part of the Apache Incubator Felix project. Work continues on enriching iPOJO and evaluating its usefulness after its initial successes.

## 10 References

1. J. Bourcier, C. Escoffier, M. Desertot, C. Marin, and A. Chazalet. "A Dynamic SOA Home Control Gateway," International Service Computing Conference, September 2006.
2. J. Bourcier, C. Escoffier, and Philippe Lalanda. "Implementing Home-Control Applications on Service Platform," Proceedings of the IEEE Consumer Communications and Networking Conference, January 2007.
3. D. Box. "Essential COM," Addison-Wesley, January 1998.
4. E. Bruneton, T. Coupaye, and J.B. Stefani. "The Fractal Component Model Specification, Version 2.0-2," <http://fractal.objectweb.org/specification/index.html>, September 2003.
5. H. Cervantes and R.S. Hall. "Automating Service Dependency Management in a Service-Oriented Component Model," Proceedings of the 6th ICSE Workshop on Component-Based Software Engineering (CBSE6), May 2003.
6. H. Cervantes and R.S. Hall. "Autonomous Adaptation to Dynamic Availability Through a Service-Oriented Component Model," Proceedings of the International Conference on Software Engineering, May 2004.
7. H. Cervantes and R.S. Hall. "A Framework for Constructing Adaptive Component-based Applications: Concepts and Experiences," Proceedings of the 7th International Symposium on Component-Based Software Engineering (CBSE7), May 2004.
8. F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, and S. Weerawarana. "Business Process Execution Language (BPEL) for Web Services Version 1.0," <ftp://www6.software.ibm.com/software/developer/library/ws-bpel1.pdf>, July 2002.
9. M.N. Huhns and M.P. Singh. "Service-Oriented Computing: Key Concepts and Principles," IEEE Internet Computing, Volume 9, January 2005.
10. IBM Corp et al. "SCA Service Component Architecture Assembly Model Specification," <http://www-128.ibm.com/developerworks/library/specification/ws-sca/>, November 2005.
11. Interface21. "Spring OSGi Specification (v0.7)," <http://www.springframework.org/osgi/specification>, 2006.
12. R. Johnson, J. Hoeller, A. Arendsen, T. Risberg, and C. Sampaleanu. "Professional Java Development with the Spring Framework," Wiley Publishing, Inc., 2005.
13. K. Lau and V. Ukis. "Encapsulating Data in Component-based Systems," Proceedings of Ninth International Symposium on Component-based Software Engineering, June 2006.
14. Object Management Group. "CORBA Components Specification," Version 3.0, June 2002.
15. OSGi Alliance. "OSGi Service Platform Core Specification Release 4," <http://www.osgi.org>, August 2005.
16. OSGi Alliance. "OSGi Service Platform Service Compendium Release 4," <http://www.osgi.org>, August 2005.
17. M.P. Papazoglou and D. Georgakopoulos. "Service-Oriented Computing," Communications of the ACM, Volume 46, Issue 10, October 2003.
18. S. Renouard, M. Mokhtari, D. Menga, and G. Brisson. "SCYLLA: A Toolkit for Document-Based Human Environment Interaction," Proceedings of the International Conference on Smart Homes and Health Telematics, June 2006.
19. Sun Microsystems. "JavaBeans Specification Version 1.01," <http://java.sun.com/products/javabeans/docs/beans.101.pdf>, August 1997.
20. Sun Microsystems. "Enterprise JavaBeans Specification, Version 2.1," <http://java.sun.com/products/ejb/docs.html>, November 2003.
21. Sun Microsystems. "Enterprise JavaBeans, Version 3.0, Simplified API," <http://java.sun.com/products/ejb/docs.html>, May 2006.
22. J. Yang. "Web Service Componentization," Communications of the ACM, Volume 46, Issue 10, October 2003.
23. J. Yang and M.P. Papazoglou. "Service Components for Managing the Life-Cycle of Service Compositions," Information Systems, Volume 29, Issue 2, April 2004.