Describing Dynamism in Service Dependencies: Industrial Experience and Feedbacks

Clément Escoffier, Pierre Bourret, Philippe Lalanda

Describing dynamism in service dependencies

Industrial experience and feedbacks

Clement Escoffier
Dynamis-technologies and Grenoble University
Grenoble, France
clement.escoffier@dynamis-technologies.com

Pierre Bourret
Grenoble University
Grenoble, France
pierre.bourret@imag.fr

Philippe Lalanda
Grenoble University
Grenoble, France
philippe.lalanda@imag.fr

***Abstract*—The rise of dynamic applications is coming with new development challenges. Indeed, dynamism is a complex concern, difficult to perceive and manage by developers. In the context of a large industrial project dealing with fleet management, we had to deal with important environmental and evolutionary dynamism. To make it easier for the development team, we have used and extended the iPOJO service component model. This paper presents how the dynamism is described in component metadata and how it is managed at runtime. The extensions have been integrated into the Apache Felix iPOJO source code.**

Keywords-service; dynamism; service dependencies; service component model.

I. INTRODUCTION

Dealing with dynamism has been a long-standing challenge in software engineering [1][2]. However, it was rarely a stringent requirement in industrial projects. This situation has dramatically evolved in the past years. More and more, applications have to handle dynamism. Pervasive applications need to manage the availability of devices, web applications must handle failures and disruptions of remote services on which they rely, and continuously delivered systems have to handle the transient situation where a part of their system is updated.

Applications are now commonly developed using software components [3]. By using such approach, software systems are divided into independent *islands*. This modularization has improved the management of software project development, throughout software lifecycle. First, at development time, teams can develop components in parallel with reduced risks when it comes to the integration step. At runtime, components can evolve separately, which makes monitoring, failures detection and replacements much easier. Finally, during maintenance, each component can be updated rather independently. Several well-known component frameworks are used today such as .NET, Spring and EJBs. However those frameworks lack flexibility to integrate dynamism easily.

To handle dynamism, traditional component frameworks need to be extended with a more flexible component binding mechanism. Service-orientation defines a loose-coupled way to integrate components [4]. The binding process can be delayed until run time, paving the road to dynamism management. Based on a publication-discovery-binding approach, service-orientation provides the required flexibility to develop dynamic systems. Several component models have adopted service-orientation to handle dynamism such as OSGi[5], iPOJO[6] or even Android[7].

One of the main differences between traditional component models and service-oriented component models lies in the dependency description. In traditional component models, bindings between components are statically defined at development time [8]. In service-oriented components, by contrast, developers merely specify service dependencies, which are resolved at runtime through service bindings. There are several implementations of service-oriented component models such as Spring Dynamic Modules and OSGi Blueprint. However, their dynamism management is too limited to cover the whole dynamism spectrum. The uncovered dynamism behavior is delegated to developers rarely used to such paradigm.

This paper analyses service dependencies in service-oriented component models and their reification at runtime. It also proposes new approaches to dependencies specification and reification. We focus more specifically on the way dynamism can be described and on the different policies that can be used to manage dynamism at runtime. This paper also explains how we extended the iPOJO service dependency model to handle sophisticated dynamic applications. Technical details such as proxies, consistency and lessons learned during development are also presented.

Those results come from the development of a large fleet management infrastructure handling 10 000 vehicles that we conducted with akquinet A.G. (http://akquinet.de). This system is deployed and in production since one year and is under intensive use.

II. FROM COMPONENTIZATION TO SERVICE-ORIENTATION

Developing software applications has always been challenging. This complexity has reached unexpected levels in the past years. This is not only due to the complexity of the business logic, but essentially to the need to integrate third-party libraries, components and services [9]. Today's software is more heterogeneous and more dynamic than a few years earlier [10].

A major evolution in the software industry has been the emergence of component-based frameworks. Since the 90's, most of the applications are developed using component models and frameworks. COM, Corba, and more recently JavaEE and .NET promote the division of applications into software components. Using components not only turns the code into a more modular and understandable abstraction, but it also makes the assembly of more complex applications possible [11].

Components interact using pre-defined interfaces, containing mostly syntactic metadata (typing information). Loose coupling allows the substitution of components by other components, granted that they provide the same interface. The connection between components is either defined during the development phase, or made at runtime using reflexive frameworks for instance [12]. Reconfiguration and substitution processes are difficult to support efficiently. Indeed, a system must be protected against state corruption and the reconfiguration process needs to reduce the service disruption. This trade-off is often hard to reach. Approaches like *quiescence* [13] can protect the system against inconsistencies, but the involved interruption of service is not acceptable by numerous systems. Approaches such as *tranquility* [14] tries to reduce the duration of the service disruption by identifying the updated parts of the system. However, such detection requires lots of data, not necessary available, about the system. In web and enterprise applications, a combination between replication and routing (i.e. load balancing) is often used to reduce disruption, but this technique is not applicable on all systems. Some applications such as those involving physical devices cannot be easily replicated.

More recently, service-oriented computing has emerged. Service-orientation is an architectural style where computing elements interact through a publication-lookup-binding mechanism. Service providers advertise their services on a discovery channel (often a service registry). Consumers look up for the required service in that channel, and use the matching publishers. Such approach relies on two main principles. First, only the service specification is shared between the provider and the consumer. Depending on the technology, the content and the format of the specification differ. However, most of specifications rely on syntactical information. This loose coupling eases service provider substitution. Second, the bindings between providers and consumers are woven at runtime. Service-orientation can be extended with dynamism by supporting the notifications of service provider's arrivals and departures. Consumers can listen to these notifications and react accordingly. For example, a consumer can decide to choose a *better* provider, or to switch to another provider when the one currently used is not available anymore. Dynamic service-orientation offers the features required to build dynamic applications. But such development remains challenging. The service lookup mechanism and dynamism management need to be handled inside the component implementation. Such code is particularly error-prone and complex as it involves concurrency and state management.

To lower complexity, dynamic service-oriented component models have proposed to infuse dynamic service-orientation within component models [15]. Dynamism is thus handled by the component framework, reducing the complexity of the component code. In such approaches, the framework is given the description of the provided services and service dependencies. Most of the frameworks allow describing the service dependencies in term of the specification, the cardinality and optionality and let use a filter to select providers such as service component architecture (SCA), or Spring Dynamic Modules.

However a number of features are required to handle more complex scenarios, such as the reaction to dynamism and the selection of the adequate / better service provider. The impact on the development model is also a stringent requirement as developers are not used to dynamism and are often stumped when dealing with it. Aspects such a concurrency and consistency are primordial to successfully conduct dynamism management at runtime.

III. SERVICE DEPENDENCY DESCRIPTION AND MANAGEMENT AT RUNTIME

We have developed a fleet management infrastructure for an industrial customer. The main goal of the project was the monitoring of the vehicles and their reconfiguration, which includes, for instance, naming the drivers allowed to drive or deciding on maintenance time. In this project, dynamism was a real challenge for the communication platform. This gateway, deployed in warehouses, is responsible of the communication between the fleet and the backend infrastructure (figure 1).

Figure 1. Global overview of the fleet management infrastructure

We had to address different forms of dynamism. The vehicles come and leave the area where the interactions are done. As this communication relies on Bluetooth, the vehicles must be close to the communication platform. Then, the communication platform must be running 24/7. So updating components on the platform without disrupting the service is a must-have feature.

The fleet management system was developed using the iPOJO service-oriented component model. iPOJO provides a simple development model hiding most of the complexity of the dynamism management. Configuring iPOJO is done using Java annotations, avoiding writing external descriptors and the burden of keeping them synchronized. The team was composed of Java developers, familiar with JavaEE development and injection mechanisms, such as CDI [16], but unfamiliar with dynamism. As the project was intended to reach an important size (today, the system includes more than one million of lines of code), the simplicity of the model was important. In addition, the learning curve was an important aspect, as the turnover of the team is unavoidable.

The iPOJO component model did not support all the dynamic scenarios encountered in the project. In this section, we describe the service dependency attribute offered by and introduced into the iPOJO component model. We focus especially on the runtime management regarding dynamism. For the reason given above the simplicity of the development model was also important.

As depicted by figure 2, on the implementation level, components contain metadata expressing service dependencies. At runtime, this information is used by the component framework to generate the service bindings and manage the dynamism. The dynamic behavior is constrained and controlled by the characteristics of the service dependency such as the selection of service providers, the resilience of the dependency to dynamism, and the reaction to service unavailability. This section explores those aspects.

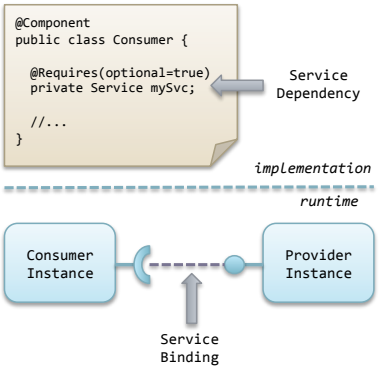

Figure 2. Service dependencies and service bindings

A. *Selection and Filtering*

As promoted by the service-orientation, service dependencies target a service specification in order to enforce the loose coupling and allow substitution. However this may lead to a large set of providers. To reduce the set of services, filters using a set of properties published by the service providers are set up. Languages for filter differ in the various service-oriented technologies. OSGi-based component models, such as iPOJO, use the LDAP syntax to express filters. Web services stacks generally use XPath.

At runtime, the framework must not only track the arrivals and the departures of services, but also track these modifications and translate those events into arrivals, departures or status quo. Updating the filter at runtime is often required during a human-driven dynamic reconfiguration. In such case, the set of matching providers must be recomputed.

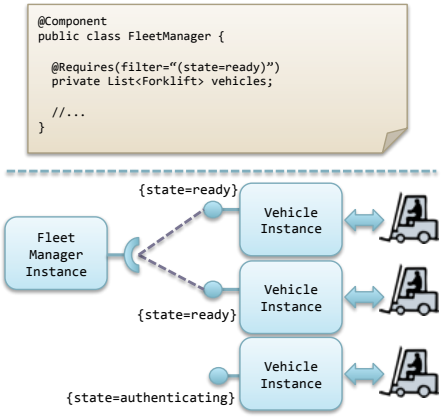

Figure 3. Filtered service dependencies and bindings

Handling filtering correctly and efficiently is a stringent requirement. In the fleet management system we have developed, vehicles are reified as services. However, vehicles may be published as services, but not ready yet. To reflect those states, a service property is published (figure 3). The fleet manager is only bound to 'ready' vehicles. As this state is volatile, tracking the changes must be done carefully. From the fleet manager point of view, only the fluctuation of the set of ready vehicles is important. On the picture, the third vehicles will become 'ready' as soon as the authentication process will be completed. The fleet manager will be bound to this third vehicle as soon as the new state is published. On the opposite, once a vehicle becomes 'busy' (i.e. a new configuration is pushed and processed), the binding between the fleet manager and the vehicle service is removed.

Although service-orientation promotes substitution and implementation abstraction, in some case, components need to be bound to specific, already known, providers. To implement this without violating the interaction pattern of the service-orientation, we use a strong filter targeting the specific provider. This type of binding is close to traditional component bindings, where composition is defined at design time. Even if this may be controversial, and removes the substitution ability, it supports the update of the provider in a transparent way.

In the fleet management context, such feature is used to create strong coupling between the vehicle service (representing the vehicle) and the service used to communicate with the real device (*driver*). As this communication stack is specific to a particular device version (vendor, product, model, series), we must forbid substitutions with any other implementation (figure 4).

Nevertheless, the *driver* can be updated to a higher version dynamically.

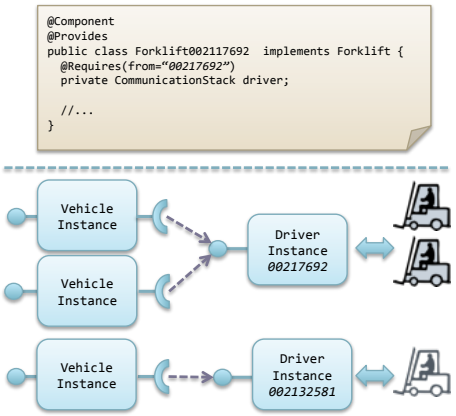

Figure 4. Bindings targeting specific a service provider

B. *Aggregation and Optionality*

The previous attributes describe to whom a component is bound. Optionality and aggregation define the cardinality of the service dependency. Dependencies can be resolved as 0..n service bindings according to those characteristics.

Aggregate dependencies are mapped to multiple bindings: one per matching provider. Aggregate dependencies are used as an extensibility mechanism where the set of *extension* is modified at run time. In such case, the framework has to track all providers and manage the modification of the set. Concerns such as sorting and iterations over the (dynamic) set also have to be handled.

In the fleet management platform, the fleet manager is bound to all ready vehicles. As depicted on figure 4, the dependency between the fleet manager and the vehicles is aggregate[1]. At runtime, this dependency is reified under a set of service bindings.

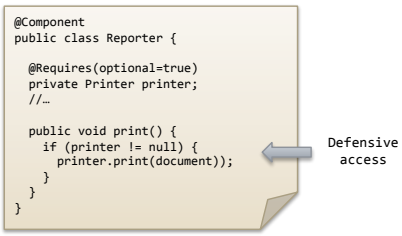

Figure 5. Defensive access involved by optional dependencies

Optional dependencies impact directly the code of the component. Indeed, the component code must be aware that the service may not be available when it uses it. Dealing with such dependencies is a trade-off between reducing the complexity of the code by trying to mock the missing service and giving enough flexibility to the developer to handle this case. Injecting *null* references transfers the responsibility to developers. However, all accesses to the service must be done defensively, and even such defensive accesses can be wrong in dynamic environment (figure 5). The printer service provider could leave between the check and the usage.

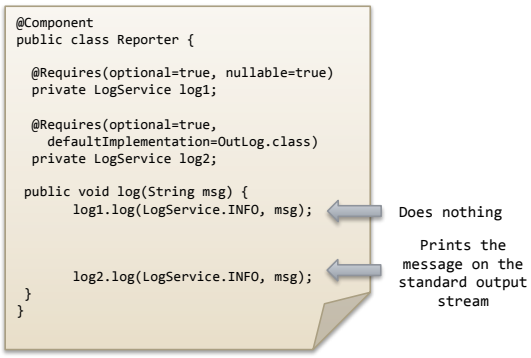

Figure 6. Difference between nullable and default-implementation

To reduce the burden of this approach, *fake* objects can be injected such as a *Nullable* object [17] or a default implementation. Nullable objects are mocks returning default values. Such objects avoid defensive accesses, and so make the code cleaner. But objects returned by the methods are useless. *Default-implementations* are created by developers, letting them define a default behavior. For instance, a *nullable* log service would ignore calls, while a default-implementation can print messages on the standard output stream (figure 6). For aggregate dependencies, injecting an empty collection gives the best efficiency during development.

Those policies are used only when no service providers are available, but would be injected with the right service object as soon as one matching provider is available.

C. *Service binding policies*

The dynamism offered by the dynamic service-orientation handles most of the case where dynamism is required. But the default dynamism management does not address all cases, and other binding policies are proposed exhibiting different degree of dynamism. We have identified three policies determining when substitution occurs: the *dynamic* policy (the most used), the *static* binding policy for stateful communication, and the *dynamic-priority* binding policy for comparable providers. In all policies, the binding is delayed until the first use is detected.

This *dynamic* binding policy tracks service providers, and handles the dynamism lazily. According to this policy, substitution happens only if the used provider is not available anymore. This policy exhibits too much dynamism in some case.

For stateful or conversational interactions, a provider cannot be substituted during the interaction and the consumer state must be reverted if the set of bound services is changed.

[1] Even if it is possible to set the dependency as aggregate specifically, the framework can analyze the type of the dependency to infer this aspect. In this case, using a list implies being aggregated.

The behavior of the *static* policy is closed to a transaction. It freezes the provider set once the transaction begins. If this set is modified by a departure or a mismatching modification, the consumer state must be rolled back. This binding policy also ignores new providers once the used set is determined (figure 7).

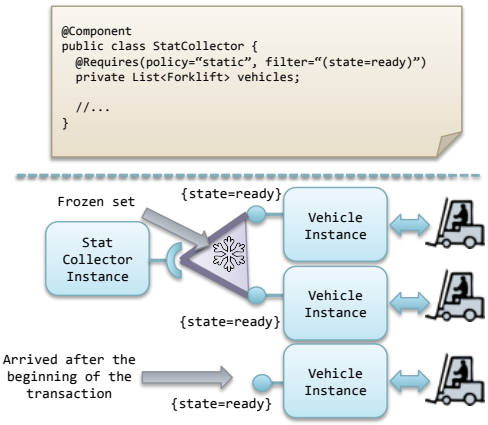

Figure 7. The *static* binding policy involves freezing the provider set during a transaction

In the fleet manager, statistics are computed on a set of vehicles. Such computation is directly impacted by the considered set of vehicles, and would be invalid if this set changes during the computation. In this case, the computation is restarted if the set of available vehicles changes during the process. For such service dependencies, the *static* binding policy is used.

The last policy, called *dynamic-priority*, exhibits a higher degree of dynamism. With these dependencies, the framework substitutes providers if a *better* provider becomes available. On aggregate dependencies, it sorts the set of providers. The difference with the *dynamic* policy comes from those substitutions. In the *dynamic* policy, the provider is not replaced until it becomes unavailable, while the *dynamic-priority* policy triggers the substitution immediately.

Using the *dynamic-priority* policy requires the ability to compare providers. This comparison can be based on service properties or on a monitoring service tracking the quality of service of the different providers.

This binding policy is also used in the fleet manager on the UI layer. This UI is composed by a set of tabs published as services. As the order of the tabs is a relevant aspect of usability, the tab services are bound to the host, ordered using the *dynamic-priority* binding policy.

D. Invalidation and timeout

The binding policies define when providers are substituted, however they do not deal with service unavailability. When no provider matches a dependency anymore, several actions are conceivable:

- Invalidating the component, protecting it from any use until the dependency is satisfied again
- Blocking all accesses to the service until a specific timeout expires

Choosing between the two policies depends on the use of the component and the dynamism of the requested services.

Invalidation is used for component offering services that cannot be satisfied if the service dependencies are not fulfilled. When one of the mandatory dependencies cannot be reified as binding at runtime, the framework removes the service from the service registry, and by this way stops all usages of the component. The service is published again when all dependencies are fulfilled.

If the service is known to be temporally unavailable because of an update of the provider, then a timeout policy can be used. During the unavailability period, all calls are blocked. If no provider matches the dependency before the timeout, then an exception is thrown. In such cases, the unavailability was unexpected and the error must be propagated.

As depicted on figure 8, the dependency between the user manager and the authentication service uses a timeout. The authentication service is a core service that must always be published, expect during its own update. Such update is expected to take less than 10 seconds. If the user manager accesses the authentication service during the update, the call is blocked until the service is published again. If the update takes more than 10 seconds, an exception is thrown, and the application is stopped (to avoid misuse).

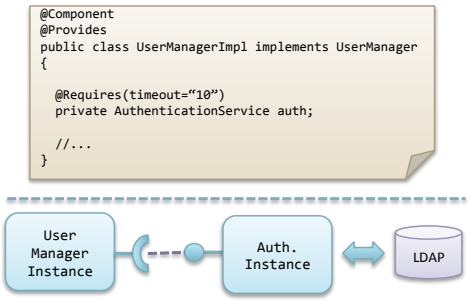

Figure 8. Service dependency with timeout

E. Conclusion

This section has presented a set of characteristics to describe the dynamic behavior of service dependencies. This description is used at runtime by the framework to reify and control the bindings between components. We paid a special attention to the simplicity of use of these characteristics. At the same time, we aimed to cover the whole set of dynamic behavior we encountered in the fleet management project.

Naturally, service dependencies can filter and select service providers. Even if the selection of specific providers does not follow the service-orientation philosophy, it covers tight bindings between components without avoiding

individual evolution. This characteristic was added to the iPOJO component model. Optional dependencies were particularly challenging because of the impact on the development model. Several strategies were integrated to the iPOJO framework to give more freedom to the developers. As stated below, those strategies are heavily used in the project.

In numerous cases, dynamism needs to be controlled carefully. For such purpose, dependencies can specify the resilience of the service bindings to dynamic arrivals and departures. Optimizations of the *static* policy and the definition of the *dynamic-priority* policy were contributed to the iPOJO component model.

Finally, the reaction behavior to service unavailability can be configured in the service dependency. We extended the iPOJO dependency model with the timeout management, as this case is often required in the project.

IV. Implementation and Validation

The attributes presented in this paper allow describing the dynamism management to cover the situation we encountered in industrial projects such as the fleet management infrastructure. This section presents technical details about the iPOJO component model and its extensions.

A. Apache Felix iPOJO overview

iPOJO[2] is a Java-based, dynamic component framework, based on the OSGi™ dynamic service platform. One of the main goals of iPOJO is to make developing dynamic applications as simple as possible. To this end, the overall approach is to keep a component as close to a "plain old Java object" (POJO) as possible. The code of a component should focus on business logic, not on mechanisms for dynamism or other non-functional requirements. iPOJO provides an extensible component container that manages all issues regarding dynamism and can be extended to support other non-functional concerns, such as configuration, persistence, and security. iPOJO also defines a composition model to describe an architectural view of dynamic service assemblies.

Component development is greatly simplified since it does not contain component model specific code. The POJO component is connected to iPOJO by configuring the component instance container, which consists of declaring component type meta-data (using Java annotations) that will be used by the container for run-time management. The iPOJO framework relies on a bytecode modification of the POJO class. This instrumentation intercepts constructors, fields and methods accesses and let the container handles the dynamism.

iPOJO containers are not monolithic, but are composed of handlers (figure 6). Each handler manages non-functional concerns. Handlers are plugged into the component instance container at run time. Only the required handlers are plugged into the container. The resulting container manages the interaction between the POJO and its execution context. Custom handlers can be developed for iPOJO, allowing developers to handle other non-functional properties.

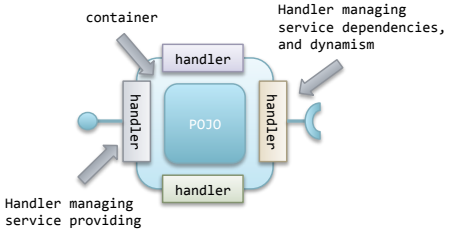

Figure 9. An iPOJO component instance and its container

A number of companies use iPOJO today, in diverse industrial projects. Its simplicity and flexibility makes using iPOJO possible in a lot of contexts from M2M infrastructures, to JavaEE application servers, and mobile applications.

B. Releasing references and performances

Handling injected reference on service objects is an important aspect of OSGi-based systems. OSGi™ defines a centralized service platform supporting the dynamic loading and unloading of modules (i.e. unit of code), and so updates of those modules. Unfortunately, such feature impacts the development of application: code can be unloaded only if all references on its objects are released. In other words, when a service leaves, we must ensure that nobody is keeping references on the leaving component. This would block the updating process. Debugging stale references is a time-consuming task requiring a deep expertise in the OSGi™ platform [18].

To address this issue, using proxies is a common technique. However, using proxies must be done carefully regarding the involved performance cost. At the beginning of the development of the fleet manager, the iPOJO component model supported only direct references or Java dynamic proxies. The direct references were too dangerous to be used in regards of the skills of the team. On the other side, the cost of dynamic proxies is too big and is limited to Java interfaces. For this reason, we have implemented a third proxy strategy using bytecode generation. The performance benchmark (table 1) shows that this strategy has a reasonable cost in term of performance while protecting against stale references. This strategy is now the default strategy of the iPOJO component model. Notice that the smart proxy is generated once on the first demand. This time is not shown on the table but is negligible.

TABLE I. Proxy strategy invocation overhead

Strategy	Average time for 1 000 000 calls	Ratio against direct invocation
Direct invocation	3 ms	1
Dynamic Proxies	15 ms	5
iPOJO Smart Proxies	5 ms	1,6

[2] http://ipojo.org

C. Dynamism & Consistency

Even with proxies, dynamism impacts the development model. One of the biggest issues is to not being sure of the availability of the service after the first use. Worse, this service could have been substituted between two uses.

To avoid this situation, we have implemented a consistency model in iPOJO. Once a method accesses objects from a service binding, we keep this set consistent for the whole method including all nested method invocations.

```
@Component
public class Reporter {

  @Requires(optional=true, nullable=true)
  private Printer printer;
  //…

  public void print() {
       [...]
       Job job = printer.print(document));
       [...]
       checkJobStatus(printer, job);
       [...]
  }
}
```

Transaction boundaries

Figure 10. Transaction boundaries used to ensure service consistency

By combining the iPOJO interception mechanism and a local Java Thread, we can track the accesses and the boundaries of the *transaction* (figure 10). Within this transaction, the set of service objects is kept consistent. With such mechanism we reduce the need of defensive programming when accessing services and ensure the safety of iterations over a set of service objects. This protection mechanism is now part of the iPOJO framework and is used by default.

D. Statistics about service dependencies usage

In the fleet management project, we have developed 736 components, for a total of 2954 service dependencies. A large majority of the dependencies are using the default behavior: not-filtered, scalar, mandatory, *dynamic* binding policy, and an invalidation strategy. The following table present statistics on the service dependencies.

TABLE II. STATISTICS ABOUT ATTRIBUTE USAGE

Attribute	Number	Percent
Not-filtered	2471	84%
Filtered	468	15%
Specific provider	15	1%
Scalar	1804	61%
Aggregate	1150	39%
Mandatory	1092	37%
Optional	1862	63%
Dynamic	2731/	92%
Static	210	7%
Dynamic-priority	13	1%
Invalidation	2392	81%
Timeout	562	19%

The amount of optional dependencies is due to the options given for its injection. There is an equal repartition of the usage of *null* injection, *nullable* object and default implementation.

The static and dynamic-priority are used only when required. A large majority of the dependencies are using the default binding policy.

From those statistics, we have a heuristic about the usage of service dependencies. The project is under development and maintenance since 2010. We have covered all the dynamic behavior we encountered. Thanks to the usage of iPOJO and our extensions, the robustness regarding vehicle dynamism and updates has drastically improved. Since the deployment in production, the platform has coped with numerous updates without introducing stale references.

V. CONCLUSION & LESSON LEARNED

The development of dynamic systems is far from simple. Service-orientation fits well to represent the dynamism at runtime. However the dynamism impacts the development model, and as a consequence makes it hard to master.

During the development of the fleet management infrastructure, we realized that expressing the dynamism outside of the code is crucial to keep under control the technical debt. In-code dynamism management is particularly error-prone as it involves concurrency, and so a risk of deadlocks and stale references. Only senior or expert developers are able to handle this complexity. In our project, we could not guarantee the availability of those developers and so had to choose a framework managing this aspect.

Choosing a technology that lets you express the whole spectrum of dynamic behavior is important. Even with the large set of features provided by iPOJO, we had to extend it. Such extensions are now part of the framework.

Describing dynamism needs to be carefully done against several dimensions: the provider selection, the number of bound providers, the substitution policy, and the unavailability behavior. The set of attributes we have presented covers all the cases we encountered in the project. We have foreseen a couple of features required in the near future.

First, the service bindings are decided by the component itself. However more global knowledge would help determining the *best* bindings. We are working on letting components collaborating with an autonomic manager to select the optimal set of providers.

Dynamism is not only difficult to handle right, it's also complex to debug. Being able to introspect the state of our components and their bindings is crucial. iPOJO comes with a minimal introspection tool. But efficient debugging would require a runtime model of the architecture.

As a consequence of the project specifications, we often had to change the filter of service dependencies at runtime. Even if the iPOJO component model offers this feature, implementing such change is not easy. We would like to extend the filter syntax of iPOJO to allow the use of contextual and configuration variables. In addition, the LDAP-syntax does not offer a good readability on long filters. We believe that a more typed language would help writing and maintaining those filters.

Finally, the iPOJO component model focuses on the service-orientation, but we also needed data-based interactions. Integrating this new type of dependencies would definitely ease the development of the project. We believe that the characteristics presented by this paper could be applied on event-based dependencies.

REFERENCES

[1] R. S. Fabry, "How to design a system in which modules can be changed on the fly," in *2nd International Conference on Software Engineering*, 1976, pp. 470–476.

[2] P. Oreizy, N. Medvidovic, and R. N. Taylor, "Architecture-based runtime software evolution," in *20th International Conference on Software Engineering*, 1998, vol. 1998, pp. 177–186.

[3] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, vol. 1743. Addison-Wesley, 2002.

[4] M. P. Papazoglou, "Service-Oriented Computing : Concepts , Characteristics and Directions," *Information Systems Journal*, vol. 3, no. 10–12 Dec. 2003, pp. 3–12, 2003.

[5] OSGi Alliance, "OSGi Service Platform Core Specification Release 4," IOS Press, Inc., 2007.

[6] C. Escoffier, R. S. Hall, and P. Lalanda, "iPOJO: an Extensible Service-Oriented Component Framework," in *IEEE International Conference on Services Computing*, 2007, vol. 0, no. Scc, pp. 474–481.

[7] M. Butler, "Android Changing the Mobile Landscape," *IEEE Pervasive Computing*, vol. 10, no. 1, pp. 4–7, 2011.

[8] K.-K. Lau and Z. Wang, "Software Component Models," *IEEE Transactions on Software Engineering*, vol. 33, no. 10, pp. 709–724, 2007.

[9] S. Vinoski, "Integration with Web services," *IEEE Internet Computing*, vol. 7, no. 6, pp. 75–77, 2003.

[10] B. Fitzgerald, "Software Crisis 2.0," *Computer*, vol. 45, no. 4, pp. 89–91, Apr. 2012.

[11] I. Crnkovic, J. Stafford, and C. Szyperski, "Software Components beyond Programming: From Routines to Services," *IEEE Software*, vol. 28, no. 3, pp. 22–26, 2011.

[12] J. Dowling and V. Cahill, "The K-Component Architecture Meta-Model for Self-Adaptive Software," in *3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection'2001)*, 2001, vol. 3, pp. 81–88.

[13] J. Kramer and J. Magee, "The evolving philosophers problem: dynamic change management," *IEEE Transactions on Software Engineering*, vol. 16, no. 11, pp. 1293–1306, 1990.

[14] Y. Vandewoude, P. Ebraert, Y. Berbers, and T. D'Hondt, "Tranquility: A Low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates," *IEEE Transactions on Software Engineering*, vol. 33, no. 12, pp. 856–868, 2007.

[15] H. Cervantes and R. S. Hall, "Autonomous adaptation to dynamic availability using a service-oriented component model," in *26th International Conference on Software Engineering*, 2004, vol. 3, pp. 614–623.

[16] Gavin King, "Contexts and Dependency Injection in Java EE 6," *Contexts and Dependency Injection in Java EE 6*, 2009. [Online]. Available: http://jcp.org/en/jsr/detail?id=299.

[17] B. Woolf, "The null object pattern," in *Pattern Languages of Programs*, 1996.

[18] K. Gama and D. Donsez, "A Practical Approach for Finding Stale References in a Dynamic Service Platform," in *Component Based Software Engineering*, 2008, vol. 5282 LNCS, pp. 246–261.