Survey on Concern Separation in Service Integration

Tomas Cerny[1(✉)] and Michael J. Donahoo[2]

[1] Department of Computer Science, Czech Technical University, Charles square 13, Prague, Czech Republic
tomas.cerny@fel.cvut.cz
[2] Department of Computer Science, Baylor University, Waco, TX, USA
jeff_donahoo@baylor.edu

Abstract. Ever-changing business processes in large software systems, integration of heterogeneous data sources as well as the desire for legacy service integration drive software design towards reusable, platform-independent, web-accessible microservices. Such independently deployable services provide an interface for retrieval and data manipulation in machine-readable formats. While this approach brings many advantages from the perspective of service integration aiming to separate data manipulation from business processing, the standard approaches provide only limited structural semantics and constraints provided through the interface. This leads to considerable information restatement and repeated decisions in integrating components, which considerably impacts development and maintenance efforts. Integration component operability becomes highly sensitive to interaction with underlying services, which are possibly composed of other services. The sensitivity is especially apparent in the structural semantics of produced and consumed information that must correlate on both sides of the interaction. This paper surveys service integration from the perspective of separation of concerns. In order to reduce the coupling and information restatement on the integration component side, it suggests introducing multiple communication channels with additional information that apply in the service interaction, extending the integration component's ability to derive service expected information structural semantics, constraints or business rules. Finally, we consider the impact of this new approach from the development and maintenance perspectives.

Keywords: Web services · Service integration · Aspect-Oriented programming

1 Introduction

Software design of large systems, which integrate functionality from different heterogeneous data sources and provide decentralize governance, utilize reusable, independently-replaceable, scalable and deployable microservices [13]. Such

R.M. Freivalds et al. (Eds.): SOFSEM 2016, LNCS 9587, pp. 518–531, 2016.
DOI: 10.1007/978-3-662-49192-8_42

services provide composable functionality addressing the disadvantages of monolithic design. [13]. They provide platform independence, support interoperable [1] interaction among different components using standard machine-readable formats. These services emphasize well-defined interfaces and availability on network, with location transparency.

The emphasis on service interface enables easy exchange of service providers and thus reduces coupling between the integration components (or generally peers) and services. An independent self-deployable component that integrates services is in the text referred as an Integration Component (IC). The IC-service interaction can be further mediated to multiple providers to support service availability, scalability, and performance. Services can recursively integrate other services, making such service composition transparent to ICs. Market changes, innovation, and/or evolution in business requirements make it easy for the ICs to use novel services or apply a new business processes on top of an existing service infrastructure. At the same time, such design, especially in the early development stage, may demand higher investments than code-centric monolithic design [1], while opening the services for broader future reuse. Thus from the long-term perspective, the overall costs are expected to reduce. ICs are fragile with respect to failure, flaws, or performance bottlenecks in any of its underlying services. Furthermore, the standard format of communication brings additional performance demands related to serialization and deserialization of information between the machine-readable format and the internal platform-specific format.

Services built on existing technologies provide information formatted in a specific structure, which the ICs must strictly follow. Current standards-based approaches provide data values and some semantics of the internal structure, although it is insufficient to automatically derive the internal structure at the IC side. For example, property data types are limited and their constraints or validation rules are missing. Furthermore, it is not possible to automatically determine the expected structural semantics of information a service consumes. From the design perspective, ICs must implement appropriate internal structural representation for service-provided information. These are usually Data Transfer Objects (DTO) [10], or map structures. These components define structure for the native platform, although this is a restatement since the structure exists and is co-defined at the web service side. This gives a commitment to the IC. Any time service information defining structure changes, its ICs must reflect the change, creating a difficulty in maintenance, as there is no mechanism preventing such inconsistencies.

The situation becomes worse when considering multiple communicating ICs or middlewares that all process the same information representation, considering the same constraints, validation rules, etc. The maintenance becomes complex and correlation fragility grows. Different individuals might manage particular services or ICs at heterogeneous locations and follow different evolution and changes. Service is usually unaware of its ICs, and thus its internal change may lead to catastrophic consequences. Due to such difficulties, it is now common

practice[1] to leave the existing service as it is when structural changes in data take place. Instead of extending the particular service, its copy with changes is made. Consequently, the two services run simultaneously. Such an approach does not naturally scale[2] since multiple such services must be monitored for operation and backups, driving up operation costs.

This paper considers service integration from the perspective of separation of concerns. The motivation is to decrease maintenance effort and mitigate the impact on ICs related to changes in service structural representation, constraints, and validation or business rules. The ICs become adaptable to changes in the above service concerns.

The conventional approaches use a single channel for communicating the information to ICs. As mentioned earlier, only a limited amount of structural information can be derived. For instance, consider a web service producing information in XML or JSON. These formats carry data values and partially describe the data structure with property names, leaving a gap for types, constraints, etc. Thus internal IC representation must exist to provide the missing pieces of information, introducing restatements.

A concern-separating approach suggests a different form of communication. It provides novel meta information that can be used to derive the internal IC structural representations at runtime. This meta information relate to different concerns. Besides the complete data structure information, the channels may consider input validation rules, user context, and business rules. Providing these concerns in a single communication channel, would lead to inefficiencies, as some concerns tend to change more often than others. Instead, we propose the use of multiple communication channels to avoid repetition and support separation of concerns at the communication-level. This extends concern reuse at the IC side and improves caching capabilities [6]. All the novel concerns are provided in a machine-readable format. The ICs become capable to derive internal structural representation, constraints, validation and business rules at runtime based on the service provided information. Later service changes are adopted by ICs and all their communication successors, which help to avoid consistency errors.

The rest of the paper is organized as follows. Section 2 provides background. Details on concern separation analysis in service integration are provided in Sect. 3. Section 4 details the concern separating design. Related work is mentioned throughout Sects. 2 till 4. Section 5 concludes the paper.

2 Background

Web services produce and consume data. A web service is the only component with access to the data source, and the only component that can persist or provide data. Usually, such a service persists data to a relational database, although

[1] Based on experience, while technically consulting with software architects of Czech banks.

[2] The highest number of simultaneously running service copied versions was reported[3] 22.

its design most likely uses object-oriented programming (OOP). Even legacy services designed in non-conventional style can be extended to provide a web service interface [1]. Such services communicate in machine-readable formats, such as XML or JSON.

Contemporary trends in OOP design are apparent from the Java Enterprise Edition (Java EE) platform [9]. The platform has a standard for dealing with Object-Relational Mapping (ORM) for persistence called Java Persistence API (JPA), input validation (Bean Validation), and even for serialization/deserialization of data represented by objects to JSON or XML formats[3] and backwards.

A service in the Java EE platform represents its data model with classes called *entities* that are associated with each other and extended with JPA descriptors for ORM as well as with Bean Validation descriptors to enforce input validation. A service can enforce business rules on the top of the data model by referencing particular entities and their properties. As mentioned in [3,10], a standardization or generally accepted approach for defining business rules is missing. One possibility is to define such rules using OOP [2]; unfortunately this leads to significant restatements of rules across various system modules or layers [3], extending maintenance efforts. Alternative approaches suggest using frameworks describing the rules in Domain-Specific Languages (DSLs), such as Drools [16] or MPS [17]. These approaches isolate rule definitions and enforce their application throughout the system.

A web-service hides its internals from other ICs, even though the information structure it provides or consumes is influenced by its entities and their data structure. Sometimes services aggregate entities or filter their properties using DTOs indirection [10]. The entities or DTOs then determine the desired format (XML/JSON). When considering the produced format, it consists of information relevant to a particular data instance, as well as to the data structure since each data value is provided together with its property. The product thus contains limited structural information; however, it does not provide the expected property data type, constraints, etc.

The IC must follow the service-expected data structure. The IC internal data representation, similarly to the service, uses either DTOs designed and compiled for this purpose, or map data structure, derived from the service provided information. The map structure may seem more flexible, but it provides no type safety or assurance on correct data structure, property types, constraints, etc. when submitting data to the service. From the perspective of service data consumption the map structure may seem impractical as the internal structure accepts any input, but the service may reject the information due to typological errors.

Both DTO and map structure properties must correlate with the runtime service representation to avoid inconsistency errors and rejected service submissions. The issue is that service and IC are independent components with different evolution time spans. Thus any time the service-side data representation changes,

[3] Java Architecture for XML Binding, Java API for {RESTful/XML} Web Services.

the IC representation must change accordingly. The ICs' DTO property restatement suffers from tight coupling and the inability to adapt to service changes, its structure is determined at compile time. A mechanism is missing to indicate or prevent inconsistency errors due to changed service internal structure.

Restatement problems are not limited to data representation only; a similar issue arises with input validation. For performance and usability reasons, we apply input validation on the IC before incurring the cost of communication and service-side processing, although to do so, we need to manually apply the input validation at the IC. This negatively impacts development and maintenance efforts since the same validation rules are restated on all tiers. Furthermore, when the IC integrates multiple services together, it might be intended to apply service business rules already at the IC level. This again leads to their replication across tiers.

The situation is exacerbated by context-awareness [4]. For instance, consider various user roles that are authorized to access different data properties. From the service autonomy perspective [1], this should apply at the service, but it cannot be omitted at the IC side, due to usability perspective [14]. The context-awareness is more complex [4] than just security. User may come from different geo locations, at different times, with various devices that all may impact the provided data, data representation, its structure, validation rules or business rules. A considerable number of decisions might be repeated at different tiers, tangling through other application concerns [6].

Current web service system design allows service reuse, composition, distribution, replication, and cross-platform compatibility. On the other hand, it does not effectively handle integration component development or service evolution. As we have shown, many concerns are considered at different tiers, although a mechanism that shares concerns across tiers is missing. This paper proposes a concern separation approach applied to the communication among ICs. The advantage is that concerns considered at the service level can be reused by other ICs, which simplifies service evolution and brings ICs' better adaptability to service changes. The context-awareness causes concern tangling in conventional approaches, deteriorating the complexity, development and maintenance efforts. The proposed approach provides concern distribution through multiple channels and handles context-awareness more effectively than a single channel communication. The multi-channel concern distribution extends reuse [6] and supports caching abilities.

3 Analysis and Discussion on Concern Separation in Services

We identify several problems with conventional service integration design. No matter the internal service design, the interaction with other ICs only provides limited information about service concerns. It focuses primarily on data value interaction. Data representation must structurally correlate among ICs and services. The validation rules and constraints must be replicated on the IC side.

When a service does not expose its source code, IC design can only consider service documentation to apply service business rules in its design to improve usability. Even when code is provided, rule derivation might be very difficult, since one service can capture business rules tangled in the OOP design [3], another may use Drools and another MPS. All later changes must again correlate with ICs, which makes global business rule maintenance hard. Runtime context, such as security, time, IC location, etc., may influence the produced or consumed data. From the concern perspective, we consider the following elements:

① Data values
② Data structure representation
③ Data input validation and constraint
④ Business rules
⑤ Context-awareness (geo location, time, IP address, security, etc.)

Each IC that uses a service and wants to process its data values ①, restates the data structure representation ②and most likely its validation rules ③. The IC may need to restate business rules ④ or even integrate context ⑤. Besides the data values ①, this presents significant responsibility and burden for development, service evolution, and maintenance. Even a small change to the above service elements ②–⑤ may cause inconsistency in multiple ICs that integrate the service, thus requiring manual change propagation. The issue with change propagation is that a service rarely knows its consumers or has only limited capability to control them.

Naturally, the question is whether there exists a way to loosen the coupling between ICs and service with respect to data structure representation ② or even other concerns ③–⑤. In order to do so, the data structure representation ② at the IC side cannot be determined statically at compile/deploy time. Instead the structural representation should be provided in a separate channel of communication and determined at runtime.

Let us assume a form of communication where an IC requests the data structure representation ② and then maps the data values ① to it. What is the consequence? *First* the IC must compose the representation at runtime, which either requires metaprogramming [7] or the use of map data structure. Since the structure is not determined at compile time, field references may loose type safety [4] within the scope of IC. This may negatively impact the programming style. On the other hand, since a service may change at any time after the IC deploys, the type safety only helps the initial IC design. *Second*, the runtime, on-demand representation derivation enforces consistency with the service. Thus the IC reflects later changes to the service structures, which improves maintenance and evolution. *Third*, there might be performance degradation due to the communication overhead and metaprogramming. On the other hand, the data structure representation ② request can be issued concurrently with the data value request ① [6]. At the same time, the data structure representation derivation could consider caching scheme similar to HTTP [6], where IC requests a

particular representation version and reuses the derived structure until it changes on the service side.

With the proposed design, the data structure representation ② is provided by service in a separate channel of communication. The same approach can apply to data input validation ③. The properties of this concern has although lot of similarities with the previous ② and when we consider the Beans Validation standard from Java EE, it is even part of the data model (its extension) [6]. This suggests the possibility to integrate the concern within the channel for the data structure representation ②.

The service business rules ④ might be unknown to the ICs throughout their execution. Such rules can be documented, although ICs cannot use the business rules separately from the service, unless restating the rules. If the rules were known or there was a way to provide them to IC in machine-readable format, the IC could take advantage of such knowledge for usability or performance improvements. For instance, consider a situation restricting an airplane selection for particular flight based on the current passenger occupancy. If IC has the knowledge of such restriction, it may avoid additional requests to the service or rejected submission attempts. Alternatively, consider a service business rule being interpreted at the client-side, resulting in a web browser JavaScript execution that verifies constraints before the submission takes place. In order to provide ICs the business rules in a special channel of communication, the service must capture them in a format that allows not only their evaluation but also their transformation onto format suitable for transmission. For instance, a DSL solution that exposes separated parser, internal representation and execution would fit such purpose. From the ICs' perspective, a business rule definition usually references data structure representations ② and their attributes by name. This introduces a coupling and limits the versatility of IC adaptation to service changes. The proposed IC runtime derivation of data structure representations may hinder the definitions of business rules at the IC side.

Context-awareness might be the next evolutionary step in software system abilities [5]. Nowadays production systems only rarely deal with context-aware features [4] and if so, then only in limited scope [6], such as interactive consoles, due to the increased costs of development and maintenance efforts [4]. For instance, Human-Computer Interaction shows existing context-aware prototypes [14] and sleek features; however these prototypes are missing production experience either due to performance requirements [5] or large development efforts [4]. Survey in [4] suggests that the complexity behind context-awareness is related to poor separation of concerns. Concerns that cannot be cleanly decomposed from the rest of the system are called cross-cutting concerns [12]. Conventional programming languages cannot effectively address cross-cutting concerns, and cause code tangling. The state of the art suggests addressing these concerns through Generative Programming (GP) [8] or Aspect-Oriented Programming [12]. Unfortunately, the program structure and the over all design must change.

GP suggests designing applications from conventional components and integrate models, DSLs descriptions, or alternative problem description formats. It takes all the above as input and then, based on a configuration script and templates, produces various combinations from the inputs. The result may produce a large amount of combinations that are later compiled. The difficulty comes when certain inputs present exponential dependency on its composition [4]. In such case, the produced result becomes impractical. Another deficiency is that the approach targets compile time product derivation. Furthermore, the execution uses generated code, which complicates debugging.

AOP proposes to design application from two building blocks. The base functionality is captured through conventional components and their extensions that are separable concerns, or even cross-cutting concerns that are captured by another building block called aspect. Aspects can use DSL or the same programming language. The aspect brings a mechanism to separate a particular concern from the base program. The way components and aspects connect together is the main AOP instrument. The base component program is transformed onto a join point representation [12]. A join point might be a name of a method, method call, location in the program or a method extended with annotation. It indicates a location in the program where an aspect may extend the program execution. Such a join point representation is a simplified skeleton of the program or a particular subsystem. An aspect has a condition formed from join points that indicates when and under what context it becomes active. The condition may use any logical or arithmetical operators to generalize the condition. The aspect integration can be compile time or runtime [4], and thus only aspects activated by given context are applied to the program execution. The component and aspect integration performs an aspect weaver, an instrument similar to a compiler [12] or renderer [4]. Since it is possible to apply the approach at runtime, the produced result is not affected by exponential concern dependency, since only context-selected concerns apply for given request, although the complexity related to debugging remains.

Thus extending the service with context-awareness while aiming for efficient design, the service should consider separating out the basic functionality from the contextual extension through aspects. Although different from other service concerns ①–④, the context might not be something we aim to provide to ICs as a separate channel of communication. Instead we might expect that context is something related to or provided by the IC requesting the service (e.g., request parameters, location, access rights) or something derived at runtime at the service side (resource usage, time). Thus context may influence the provided result of ①–④.

To demonstrate, consider that the service is requested by a IC with a low level of authorization. The service should only consume or produce a subset of data values ① or to expose the IC limited scope of data representation ②. Alternatively, consider ICs requesting personal information sensitive to the geolocation. One IC may receive information including *country*, *state* and custom date formatting, while another only receives *country* and general date formatting.

4 Design and References to Concern-Separating Approaches

In order to separate concerns mentioned in previous section and stream them in separate communication channels, we must be able to interpret concern description at the service side. However, we should avoid reinventing existing solutions and not expect the industry to make big changes in conventional development or programmer attitudes. For these reasons, we may tend to avoid Model-Driven Development (MDD) [4]. Next option is to design custom DSL for the service description on higher level of abstraction [15], but this approach is not much different from MDD since developer must learn a new language and change the design abstraction. Instead, a minimal impact on developer should be expected for easy adoption and transition to production development.

One possibility is to use a code-inspection mechanism that uses metaprogramming. These approaches allow reusing existing code for the purpose of transformation, which is in our case the ② data structure representation. For example, [11] uses this approach in MetaWidget framework that derives User Interfaces (UIs) from the data model. Similarly [4] uses metaprogramming to derive join point representation [12] for later use in AOP-based transformation. Applying code-inspection does not considerably affect the service development perspective, as its use is transparent.

Similarly, when considering the existing validation or ORM standards, code-inspection can derive the validation rules and constraints ③ and thus further extend the join point representation. Additional data structure representation extensions can be considered in the same way (access, presentation extension [6], etc.).

At this point, the development impact does not involve significant changes, even though the service provides communication channels for the concerns ①–③. The AspectFaces framework [4] provides an example code-inspection tool. [6] shows its use for the separated concern delivery for UI derivation. In order to apply it at the service level, the frameworks' aspect weaver is pointed to application data model from which it derives the join point representation that can be bidirectionally transformed to XML/JSON formats. An IC derives the service internal, platform-specific data structure representation from the received join point model and feeds it with the provided data values ①. [5,6] show the usage for UI derivation for mobile, standalone and web clients (Google Web Toolkit, AngularJS, HTML5) and demonstrate platform-independence for the concern delivery. Changes to the service data structure representation are adopted by all the derived UIs across various platforms. Furthermore, [6] gives details on caching options and performance, which can be applied. For instance, as mentioned earlier, the requests to various channels (data values and join point representation) can be done concurrently. The join point representation can be cached and reused with invalidation mechanism using versioning similar to HTTP.

Usage at an IC impacts the development perspective. The IC usually aims to integrate multiple services and works with multiple ② data structure representations. The internal IC structure representation is a proxy with defined name. Its properties, such as fields are received at runtime, which deteriorates the type-safety at the development time. This is the trade-off for the ability to adapt to service structural changes. The benefit is that validation and constraint enforcement ③ on data values is part of the proxy and can be performed at the IC side. When the IC applies business rules and processes, explicit references bind to the proxy by property names. This limits IC adaptivity to changes in data structures, since service changes to property name do not update the ICs' named binding. Usage of DSLs for business rule definition and enforcement at the IC side is not impacted by the approach, since such DSL has already limited type safety. The IC may apply business processing and forward the proxy to another ICs (e.g., client providing presentation and UI). A proxy propagation is not different from the above description. The data consumption is equivalent to the conventional approach at the service side, with the difference that IC knows what the service expects, what properties it has, which types, constraints and validations are considered, etc.

Common use cases [4] for service maintenance consider changes in structure naming, property naming, property constraint/validation modification, property removal and mostly addition of a novel property. How does the service using ICs react to them?

The conventional approach using DTOs is impacted by any service structural change that is promoted to the machine-readable format and thus causes inconsistency at the IC side. The constraint/validation change is not known and thus may occur at production environment as an inconsistency.

The proposed concern-separating approach cannot deal with changed naming of the given structure since the IC's proxy is determined by the name, although a key-based indirection would address adaptability. The proxy reflects all service changes of property names, constraints and validation rules. It further reflects property additions or removals. Although it has the ability to deal with the changes, the IC application may explicitly reference given properties to apply business processing or enforce business rules, which limits the adaptivity. [5,6] show that this is rarely the case for UIs, even though local coupling may exist. In UIs, it usually uses generic approach to access all provided fields rather than to make explicit references. In the UI, the structural representation can be seen as a logical unit.

When assuming that local reference to given structure exists, then the adaptivity degrades. The IC application no longer adapts to change of property names or property removals. Although the proxy still adapts, the reference to changed property may fail, similarly to conventional design. On the other hand, the adaptivity to changes in constraints and validation rules promote all changes to ICs. Thus when the service maintenance follows the policy that only allows increments in properties and allows constraint/validation changes, then the integrity

is preserved and reflected by all ICs. This avoids consistency errors and preserves functionality.

From the above it is apparent that, for the maintainability purposes, the most suitable approach would embed business rules to services rather than to ICs, which [13] suggests for microservices. Then the IC adaptivity promotes to most of the structural changes. However, not all situations allow promoting business rules to services. Furthermore, it might be the target design to use business process and rule indirection to promote flexibility in business changes and evolution. Naturally, structural changes must promote to referencing business rules no matter the origin of the business rule, and thus next we consider the ability to reuse single business rule definition across multiple tiers.

The service ability to provide its business rules to other ICs requires design changes in the way the rules are captured. [2,3] show possible approach that captures the rules in DSL and binds rule to data through annotations. The approach brings the ability to perform business rules inspection and transformation. This can use transformation to machine-readable format and thus be used by a separate distribution channel. An IC can interpret the provided rules locally and avoid rejected submission or improve usability. In service composition, this brings the advantage of combining business rules from various services as well as a centralized view on variety of rules from heterogeneous service environment. The ability of sharing business rules across services can be utilized in business process modeling and execution, although this is left for future work.

The most challenging perspective is the service context-awareness. [4] suggests that many contemporary systems follow the "one for all" approach in the UI design due to its development and maintenance demands. Context can influence the production as well as the consumption of data. We may hardly imagine significant changes in data values ① or structure representations ②, beyond property access restriction or conditional rendering. The ③ input validation and constraints or business rules ④ may differ more significantly basing on the context. For instance, company vendor agent may submit orders with a given delivery date only until a certain time before the distribution stage starts. The accepted order time may differ based on the geo-location of the order destination as the shipment time from a central warehouse differs in the delivery time. As a bonus for customers with high turnover, the agent is able to submit the order after the deadline. When agent updates customers profile he/she must provide all personal information and follow the validation rules on provided formats. An administrator can update customers with a subset of information, while skipping all the validation rules.

Context can more or less impact any of the other considered concerns ①–④, and this impacts the service design. [4] suggests an AOP-based approach, which is not significantly different from conventional design from the development perspective. The data model together with its extensions referencing validation, business rules, etc. is transformed to the join point representation. This representation is produced (once) and utilized by the aspect weaver that mediates service requests. On each request, the weaver clones the join point representation

and considers separately-defined aspects that may modify the join point representation. Aspects trigger based on supplied context and join points found in the particular processed representation section. As a consequence, this may modify given constraints, hide properties, etc. The result of the weaving, as shown in [5,6,14], is transformed to a machine-readable format. The corresponding data values ① follow the same cycle in order to determine which data properties are authorized for the delivery.

Context-awareness is mostly notable in UIs where the UI adjusts to particular user, browsing device abilities, location, etc. Earlier we mentioned that [5,6] show multiple prototypes for mobiles, standalone and web clients that can process the communication channels for the concerns ①–③. When service considers context-awareness, these prototypes adjust to the provided output and become context-aware. The impact is on caching abilities, which applies more strict invalidation [6]. The usage across different platforms demonstrates the approach versatility. Furthermore, [6] shows that, for web delivery, the approach untangles UI concerns and supports their reuse at the IC side, which positively impacts service performance. [6] provides evaluation that show the impact in production environment where concern separating approach outperforms the conventional single channel delivery regarding to UI responsiveness as well as reduces service side resources.

5 Conclusion

This survey discusses service integration from the perspective of separation of concerns. Conventional approaches for web service design bring many advantages over the code-centric, monolithic approaches. Unfortunately, service integration posses multiple deficiencies. Data structures considered by services must be understood and followed by all ICs that become tightly coupled to the data structures. This disallows service evolution and usually results in new, slightly modified service introduction to avoid correlation errors with legacy. Moreover, ICs are unaware of service internal constraints, validation or even business rules. Service knowledge distribution would improve performance, consistency, and usability as well as provide a centralized view on combined services.

Separation of concerns and concern distribution addresses the above deficiencies. Services can provide additional information to ICs in multiple distribution channels that are utilized, providing data structure semantics to derive the service data structure representation at runtime. This overwhelms the tight coupling regarding of data properties. This has two sides. Runtime derivation allows the IC to adapt service-side structural changes and thus open the ability for the service to evolve. On the contrary, the IC uses structure proxies unaware of its properties at compile time, unless referencing the service. IC reference to particular properties introduce coupling and limit its adaptivity to structural services changes, although as [5,6] show generic referencing can be applied as demonstrates the usage in UIs. The benefit is that the proxy representation comes with all constraints and validation rules that can be applied at the IC

side, avoiding restatement. Using a suitable form of business rule definition at the service side brings the ability to inspect rule definitions and provide them in a separate distribution channel in machine-readable format for the IC. Such rules can be applied earlier in the request processing or even combined with other service rules. Novel business rule definitions at the ICs level might be affected by the weak type safety introduced by the approach. Another solution is to promote business rules to a particular service supporting their reuse, although deteriorating the flexibility of their modification and evolution.

Context-awareness pushes the service design towards the direction of AOP, since it fosters efficient design that avoids tangled code and replication. The context influences both production and consumption of service data values, as well as impacts the resulting data structure representation, validation rules, and business rules.

The main advantages of our approach are its ability to adapt ICs to service changes in data structure, although the extent of the adaptation is influenced by IC references to data structure properties. The approach opens meta information and thus shares its constraints, validation rules, and business rules with other ICs, which provides them with extended abilities that impact performance, composability, and usability. Context-awareness security involvement enforces authorization to all distribution channels.

Future work will address the business rule involvement in business processes definitions in distributed environment. The limited type safety in IC development could use verification mechanism with respect to the service provided meta information. For instance, the IC could use a DSL language, such as MPS [17], as a verification instrument. It could use metaprogramming and suitable constructs to resolve valid references. Alternatively, in case of Java, we can even avoid the impact on the IC design and apply bytecode manipulation framework such as Apache BCEL or ASM [18] to enforce the property correlation at the development/compile time. Our research will also consider the AOP approach to the service backwards compatibility. Each time a service changes, novel aspect is introduced, responsible for backwards compatibility transformation. ICs may use given services and indicate compatibility version. The version triggers a chain of aspects that apply transformation rules to the latest service version and mediate the communication with the IC acting as the older service version.

Acknowledgments. This work was supported by the Grant Agency of the Czech Technical University in Prague, grant No. SGS14/198/OHK3/3T/13.

References

1. Buelow, H., Deb, M., Kasi, J., LHer, D., Palvankar, P.: Getting Started with Oracle SOA Suite 11G R1 a Hands-On Tutorial. Packt Publishing, Birmingham (2009)
2. Cemus, K., Cerny, T.: Aspect-driven design of information systems. In: Geffert, V., Preneel, B., Rovan, B., Štuller, J., Tjoa, A.M. (eds.) SOFSEM 2014. LNCS, vol. 8327, pp. 174–186. Springer, heidelberg (2014)

3. Cemus, K., Cerny, T., Donahoo, M.J.: Automated business rules transformation into a persistence layer. Procedia Comput. Sci. J. **62**, 312–318. Elsevier (2015)
4. Cerny, T., Cemus, K., Donahoo, M.J., Song, E.: Aspect-driven, data-reflective and context-aware user interfaces design. In: Applied Computing Review, vol. 13, no. 4, pp. 53–65. ACM (2013)
5. Cerny, T., Donahoo, M.J.: On separation of platform-independent particles in user interfaces. Cluster Comput. **18**(3), 1215–1228. Springer, USA (2015). http://dx.doi.org/10.1007/s10586-015-0471-7
6. Cerny, T., Macik, M., Donahoo, J., Janousek, J.: On distributed concern delivery in user interface design. Comput. Sci. Inf. Syst. **12**(2), 655–681. ComSIS Consortium (2015)
7. Chiba, S.: Proceedings of the ACM OOPSLA 1998 workshop on reflective programming in C++ and java. In: Javassist - A Reflection-Based Programming Wizard for Java (1998). http://www.csg.is.titech.ac.jp/~chiba/oopsla98/proc/chiba.pdf
8. Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications. ACM Press/Addison-Wesley Publishing Co., New York (2000)
9. DeMichiel, L., Shannon, B.: JSR 342: Java^{TM} Platform, Enterprise Edn. 7 Spec (2013). https://jcp.org/en/jsr/detail?id=342
10. Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley Longman Publishing Co. Inc., Boston (2002)
11. Kennard, R., Leaney, J.: Towards a general purpose architecture for UI generation. J. Syst. Softw. **83**(10), 1896–1906 (2010). http://www.sciencedirect.com/science/article/pii/S0164121210001597
12. Kiczales, G., Irwin, J., Lamping, J., Loingtier, J.-M., Lopes, C.V., Maeda, C., Mendhekar, A.: Aspect-oriented programming. In: Akşit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241. Springer, Heidelberg (1997)
13. Lewis, J., Fowler, M.: Microservices (2014). http://martinfowler.com/articles/microservices.html
14. Macik, M., Cerny, T., Slavik, P.: Context-sensitive, cross-platform user interface generation. J. Multimodal User Interfaces, **8**(2), 217–229. Springer, Heidelberg (2014). http://dx.doi.org/10.1007/s12193-013-0141-0
15. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Surv. **37**(4), 316–344. ACM, New York (2005). http://doi.acm.org/10.1145/1118890.1118892
16. Proctor, M.: Drools: a rule engine for complex event processing. In: Schürr, A., Varró, D., Varró, G. (eds.) AGTIVE 2011. LNCS, vol. 7233, pp. 2–2. Springer, Heidelberg (2012). http://dx.doi.org/10.1007/978-3-642-34176-2_2
17. Voelter, M., Kolb, B., Warmer, J.: Projecting a modular future. IEEE Softw. **99**, 1. IEEE Computer Society, Los Alamitos, CA, USA (2014)
18. Wu, J., Huang, L., Wang, D.: ASM-based model of dynamic service update in OSGi. SIGSOFT Softw. Eng. Notes **33**(2), 8:1–8:8. ACM, New York (2008). http://doi.acm.org/10.1145/1350802.1350815