Levering object-oriented knowledge for service-oriented proficiency

STUBBINGS, Gary and POLOVINA, Simon <http://orcid.org/0000-0003-2961-6207>

DOI 10.1007/s00607-013-0304-6
The final publication is available at http://link.springer.com

Levering Object-Oriented Knowledge for Service-Oriented Proficiency

Enhancing Service Capability in Developers

Gary Stubbings* · Simon Polovina

Archived: 22 January 2013

Abstract As more and more enterprise systems endeavour to interconnect seamlessly by using a Service-Oriented Architecture (SOA) a number of challenges are beginning to surface as a result of the differences in understanding between Object-Orientated Programming (OOP) and service-orientation in technical development teams. These differences are thus explored to determine the potential of subsidising gaps in knowledge through relational learning in order to better prepare development environments for service migration. It emerges that the principles of Service-Oriented Programming (SOP) can be used within OOP by selectively identifying the existing knowledge found within object-orientation and traditional programming methodologies. The benefit of this approach proposes to lever the expertise of object-oriented developers so as to build service-ready computer software and encourage the seamlessness of SOA.

Keywords Object-Orientation · Service-Orientation · Developer Capability · Expertise · Programming Approach · Knowledge Transfer · Service-Ready

PACS PACS 89.20.Ff

Mathematics Subject Classification (2000) MSC 68M11 · MSC 68U35

Gary Stubbings MSc BSc
t-mac Technologies Ltd,
Sheffield Road, Chesterfield,
Derbyshire, S41 8JT
Tel.: +44 (0)844 287 0007
E-mail: garystubbings@bcs.org.uk

Dr. Simon Polovina
Conceptual Structures Research Group,
Communication and Computing Research Centre,
Sheffield Hallam University, Cantor Building,
153 Arundel St, Sheffield, UK, S1 2NU
Tel.: +44 (0)114 225 6825
E-mail: s.polovina@shu.ac.uk

1 Motivation and Object of Research

Modern software development techniques evolve around the use of object-orientation. However, there is an increasing drive to develop applications using distinct services that facilitate the reuse of business processes through functions rather than objects. As such a number of frameworks and standards have emerged from the evolution of traditional software practices [27]. In contrast, some believe that much of what seems to be modern innovation is actually the rediscovery of existing approaches [41], and that it is important to consider software development on different levels by focusing upon the varying layers of abstraction and reusability. This claim is well justified by the frequency in which traditional object-orientated systems are later extended with service-oriented features such as that of Web services. However, such applications are commonly bundled with a number of constraints that obstruct the simplest approach and thereby demand needlessly complex service integration. In order to resolve these issues we agree with Yang [40] who argues that the consideration of these two paradigms together, rather than separately, could lead to a simpler integration process and thus eliminate the need for complex SOA approaches.

Similarly, a number of practitioners [12, 20, 27] consider the transfer from Object-Oriented Programming (OOP) to Service-Oriented Programming (SOP) an evolutionary process and as a result believe that, due to the enterprise-centric nature in which services are used, the ensuing SOP methodologies only contain a subset of the original object-oriented principles. However, despite their similarities (such as association and granularity) they still have their own unique aspects and methods of handling remote invocation. Fundamentally, in OOP the objects are aware of each other's existence and actively work together to complete business tasks [8], whereas a Service-Oriented Architecture (SOA) typically includes a combination of loosely coupled services with straightforward interactions [36, p. 64]. From a high-level analysis of the two paradigms (Tab. 1) it is clear that whereas OOP is more focused on building the internals of applications through reusability, SOP encourages a combination of techniques focused around achieving a high level of exposure.

This chain of thought draws attention to a critical quandary in software development, in that it is difficult to choose the best approach when both business requirements and technology are continuously changing. Dori [10] discusses how software engineering techniques traditionally used for object-orientation could be adapted for use in service development and thus suggests that the exploration of existing and standardised approaches, by considering the two paradigms in tandem, could lead to simpler service integration. Marks [28], on the other hand, argues that the primary purpose of SOA within a business environment should be as a means of connectivity rather than a replacement of technology and that there is nothing wrong with using conventional OOP methods for building the internal workings of a system. More accuratly, Erl

Table 1 A contrast between OOP and SOP

Aspect	Object-Orientation	Service-Orientation
Concepts	Modelling, Architectural Design, Programming	Modelling, Architectural Design
Exposure	Methods	Services
Focus	Component-level	Business-level
Communication	Primarily internal	Internal and external (Interoperable)
Standards	Extensive standards and techniques with proposed solutions and high maturity	Specific standards in terms of the goals but no precise solution due to system diversity
Complexity	Medium to high with a more controlled environment	High, specifically where there is little control over technology

[12] underpins an existing connection between the two domains, advising that the integration of services which encapsulate object-oriented logic will have an effect on the underlying system design, from which many of the service complexities arise. Accordingly, we recognise that an ideal union of the two would conclude with an approach for developing services that inherit all of the object-oriented fundamentals whilst also increasing extensibility and reusability.

Although these two paradigms can be used alongside each other to develop semantic models it can be argued that most of the existing approaches and frameworks show a clear separation whereby focus is placed on either one or the other and rarely the two together. However, as both paradigms include the same data and underlying processes we believe that the transition to a SOA is only subject to a systems design. When a company decides they want to make an existing system process-driven a number of factors need to be considered; ideally the system would be componentised with an abstracted business logic layer and would require minimal change, however, if the development team were unfamiliar with service-orientation when the initial design took place then it is reasonable to assume that the resulting system would not accommodate service changes so easily. Arsanjani [3] considers the layers of SOA adoption, the first level considers implementing individual Web services from tasks contained within the existing application, however, more often than not the functionality is embedded and not readily accessible. This results in a large quantity of refactoring in order to pull the functionality out and expose it as a service. Understandably, this often leaves development teams learning new approaches and choosing appropriate integration technologies which results in system branching, splitting of resources and an increased risk of bugs. However, more often than not, even when software systems are componentised, the underlying functionality is still limited to that of the original hierarchy, undermining the purpose of loose-coupling in SOA.

In order to address these concerns we argue that if the original system and its team were better prepared then there would have been mechanisms in place to facilitate the change without impairing functionality. Although we are mindful that there are many distinct solutions for integrating service-like behaviour in software systems, we are also aware of both their quantity and complexity. We are thus led to the hypothesis that knowledge found in existing OOP methods coupled together with a clear understanding of object-oriented competency in developers could lead to new approaches for both teaching the underlying principles of SOP and developing more adaptable object-oriented systems, thus eliminating the intricacy of service integration. By assuming basic object-oriented competency we argue that future research will be able to focus less on robustness and more on good object-oriented design practices, statelessness and scalability. We accordingly examine the research gap with respect to these goals. In the next section (Sect. 2) we explore programming competency in software developers, followed by Sect. 3 which discusses adaptable development processes for knowledge transfer, and finally Sect. 4; where we consider the potential of enhancing service proficiency.

2 Programming Competency in Contrast

There is a general perception that developers have a greater understanding of OOP than SOP, however, the extent of this difference in knowledge is relatively unknown. Competency is an important factor for narrowing down potential candidates for both knowledge transfer and for building extensible object-oriented systems. We theorise that most developers in industry are now actively using or have used object-orientation. Accordingly, similar to Karsten & Roth's [26] approach into student computer competence and confidence, which used self-efficiency to offer insight into how individuals perceive their capability in performing specific tasks, we conducted a short experiment asking 30 computer programmers from around the United Kingdom to assess their knowledge and confidence in the two domains.

As expected, all 30 of the participants had used object-orientation whereas only 20 had experience with SOP. We broke this down further; identifying that over half had more than 3 years of educational attainment with at least some industry experience and that it was this control group that accounted for the majority of the 20 who had service-oriented experience. The participants were then asked to assess their knowledge of the two programming paradigms. From the initial results, a rating of 2.77 to 1.67 (Tab. 2) shows that the majority of participants perceived their knowledge of OOP to be higher than that of SOP, and that 96.7% rated their understanding of object-orientation as knowledgeable or above.

Further analysis showed that respondents who had selected SOP also had a higher level of ability in OOP (Tab. 3), accounting for the majority of experienced and expert results.

Table 2 Self-rating of knowledge

	None	Beginner	Conversant	Experienced	Expert	Rating
OOP	0.0% (0)	3.3% (1)	36.7% (11)	40.0% (12)	20.0% (6)	2.77
SOP	13.3% (4)	26.7% (8)	40.0% (12)	20.0% (6)	0.0% (0)	1.67

Table 3 Self-rating of knowledge in only those who admitted SOP experience

	None	Beginner	Conversant	Experienced	Expert	Rating
OOP	0.0% (0)	10.0% (2)	20.0% (4)	50.0% (10)	20.0% (4)	2.8
SOP	0.0% (0)	25.0% (5)	45.0% (9)	30.0% (6)	0.0% (0)	2.05

The participants were asked how confident they were when implementing the two paradigms (Tab. 4). Inline with self-rating, the results showed a higher level of confidence in implementing object-oriented systems than that of service components.

Table 4 Perceived confidence

	None	Some	Moderately	Very	Completely	Rating
OOP	3.3% (1)	10.0% (3)	16.7% (5)	40.0% (12)	30.0% (9)	2.83
SOP	23.3% (7)	36.7% (11)	10.0% (3)	26.7% (8)	3.3% (1)	1.50

Unsurprisingly, the most confident participants consisted of those who had selected SOP (Tab. 5), with high-end ratings in both self-assessment and confidence.

Table 5 Perceived confidence in only those who admitted SOP experience

	None	Some	Moderately	Very	Completely	Rating
OOP	5.0% (1)	10.0% (2)	15.0% (3)	35.0% (7)	35.0% (7)	2.85
SOP	10.0% (2)	35.0% (7)	10.0% (2)	40.0% (8)	5.0% (1)	1.95

While there is no simple way to validate the claims of the participants, these results underpin the problem domain by presenting a greater level of perceived knowledge in OOP. Although object-orientation is clearly more mature we are also aware that SOP is no new concept and has proven itself as an essential part of software development. However, this is clearly not reflected in the participants knowledge; thus opening further research gaps with regards to why the results present such a difference in understanding. Although we identify that these statistics alone only offer a basic understanding of the varying levels

of ability we also recognise that in future investigations a precise difference in competency would be required for analysing methods and techniques for knowledge transfer.

Maintaining an understanding of current approaches in an area as diverse as SOA has many challenges, Baarda [4, p. 9] suggests when a business has gone for a prolonged period of time without a need for services then it is likely it will suffer from a slip in paradigm focus when service needs arise, thus making service integration more difficult. Presumably, these difficulties also present themselves within businesses that have never previously thought about service implementation and as such, in Sect. 3, we discuss how a promising solution could be to construct service-ready software systems using existing developer skillsets.

3 A Service-Ready Development Process

The use of object-orientation has eased the design of core information systems in businesses for the past 20 years [18]. However, much in the same way that the early 90's saw a change in focus towards object-orientation; many software development teams are facing a new dilemma as service-orientation promises another paradigm shift. This new driving need for services originates from a demand to expose core system functions, however, unlike its predecessors shift from procedural programming, service-orientation is often considered to be an extension of object-oriented principles. Nevertheless, the problem with any paradigm shift is that years of accumulated knowledge, skills and experience cannot always be directly applied in a new context. Accordingly, we argue that before moving toward service-orientation software developers need to think in fundamentally new ways, and thus explore strategies to support the transition.

Erl [12] presents a number of guidelines for designing service-oriented classes whereby object-oriented methods are used in contrast. Although this is a good approach in principle the object-oriented counterparts used in the examples are not as they would appear within a typical system. For example, they all return raw types and can be easily translated to single classes, as such; the differences in the examples are relatively easy to understand because the classes are already in a state that we consider being service-ready. We believe that having the system objects in this service-ready state simplifies later transition and should be considered prior to developing object-oriented systems.

3.1 Alternative technologies and approaches

Although we have discussed some of the problems with incorporating services in complex systems, there are already a number of technologies that pertain to the same ideals of a service-ready approach. However, whilst it is possible to

implement middleware technologies to deliver such an environment, we argue that the difficulty and thus level of competency required is dependent upon technology in question. We are henceforth led to consider the claims of languages like Erlang [2] and it's derivative Scala, which boast *"programmer skills are fully re-usable"* [37]. These languages attempt to enable the development of massively scalable real-time systems by following the Communicating Sequential Process (CSP) model, which is different from a typical object as it allows for data to be encapsulated within a process, offering a form of decoupling which keeps the core objects the same.

These languages enable software developers to be more productive by providing common patterns which standardise the way software is written. However, McBeth [29], offers a balanced view of Scala, and proposes a number of arguments that oppose the language's claims. He outlines how *"Scala incorporates features and concepts that are not familiar to many programmers"* and outlines how there is still a learning curve, which is made difficult with the limited commercial support, documentation and tool maturity. He goes on to suggest that organisations are generally opposed to Scala because of the limited developer pool and the additional cost of training which, from our experience, is made more difficult by developers in the professional community being unwilling to even try another approach until mandated at the corporate level. We are thus led to conclude that, although Scala fits in with many of the ideals of service-readiness; what comes as second nature to experienced software developers is often overlooked by novices and as such an approach rather than technology would allow for the gradual introduction of small changes in the Object-Orientated Analysis and Design (OOAD) process, increasing the likelihood of developer commitment.

Udell [38] suggests that SOA is an intellectual style that supports the combination of these approaches in an attempt to simplify a complex solution and that as a result, a number of frameworks and standards for services have emerged from the evolution of software practices. Decision modelling is an approach adopted from software engineering concepts which has led to the development of SOAD (SOA Decision Modelling), a set of concepts and RADM (Reusable Architectural Decision Model), a set of recurring decisions for SOA [42]. We argue that, although these frameworks help with the identification of business requirements, only a small proportion of the ensuring methods lend assistance to the development process itself, making service development considerations difficult for novices to grasp.

3.2 What it means is to become service-ready

In order to ease the difficulties associated with service-orientation, we propose a need for increasing service-awareness in developers who are traditionally focused towards programming using object-orientation.

A lot of work has been put into good software design; however, studies such as Voigt's [39] method of identifying flaws in systems and the conceptual approach to improve existing software through coding standards by Fowler et al. [13] are typically motivated by conflicting advice and focus upon post-development solutions. In contrast to these approaches we propose to enhance the developer's ability to identify potential integration problems by increasing awareness of the service principles. One way in which this can be achieved is by using common, well-founded and appropriate object-oriented principles as mechanisms for knowledge transfer. Rather than teaching SOP directly, the underlying principles would instead be introduced during the development stages by using the same delivery methods of the common object-oriented principles, but adapted where appropriate to address the new context.

To this end we consider two attitudes towards teaching object-orientation. Firstly, that of Ragonis & Ben-Ari's [34] who propose an object-first approach which demonstrates object-orientation prior to basic programming concepts, and secondly, that of Buck & Stucki [7] who suggest that students are exposed to complex object-orientation too early and advise that the basic building blocks should be taught first. Seeing the value in both approaches we perceive how a service-first approach could use a combination of the two, offering a solution which introduces the service principles prior to intricate object-oriented implementation techniques. This service-first approach would educate developers in both the characteristics of services and how a system's design can cater for their integration.

Although there are research gaps with regards to exploring a shift in complexity from services to objects through the addition of software design constructs we argue that, as developers are far more competent in OOP, the relocation of difficulties to the more well-known domain would produce higher quality software systems. Moreover, we foresee the introduction of architectural decisions from the service domain could have many advantages over teaching SOP outright, and contribute towards exposing business processes whilst providing continuity and enabling the development of well-structured object-oriented systems that are more accommodating to modern service requirements.

4 Exploring a Service-Ready Solution

In our experiment (Sect. 2) we found that a surprising number of developers were unfamiliar with service-orientation, suggesting that SOP is yet to position itself as a core aspect of software development. In order to address this lack of expertise we discuss how a service-ready approach could target the complexity issues behind service-orientation (Sect. 3). However, this approach presents a number of challenges; the foremost, which we will discuss in this initial investigation, is how to represent the principles of service design [11].

Portraying the service principles and their relevance to an object-oriented system is clearly more difficult than simply offering the non-contextual information, and as such we consider methods and techniques outlined in software development literature by pursuing those that are suitable for presenting the service principles in an object-oriented environment. We hypothesise that any object-oriented methodology that can be adapted to demonstrate the logic behind a service principle would make a suitable delivery method for knowledge transfer and thus lever the existing expertise of object-oriented developers. To this end, we begin our investigation with Erl's [12] comparison of principles for two reasons. Firstly, to discover potential ways in which an object-oriented system could be designed to accommodate service components, and secondly to dissect his guidelines for designing service-oriented classes so as to identify their feasibility in a service-ready context.

4.1 Approaches that Unify SOA

Josuttis [25, p. 26] describes SOA as *"an architectural paradigm for dealing with business processes distributed over a large landscape of existing and new heterogeneous systems"*, and summaries the key technical concepts as services, interoperability, and loose coupling. We consider the definition of a service-ready architecture to be similar in definition, but without the service and interoperability concepts. Instead, we propose to analyse the approaches to SOA that promote loose coupling in order to discover appropriate patterns that can be applied to object-oriented systems so that services can easily integrate or even evolve from of existing functions when the need arises.

A typical approach for SOA introduces a layer of abstraction that enables new and existing object-oriented systems to be integrated into the business through an Enterprise Service Bus (ESB). The role of an ESB is to act as a universal connectivity middleware solution which enhances communication and simplifies integration [36]. ESBs are a popular business solution due to their ability to retain existing investment in resources whilst providing additional tools for interaction with other business processes, accordingly, we argue that this approach underpins our study as the quantity of retained resources will always subject to system design.

The Service Development Life Cycle (SDLC) takes an iterative SOA development approach and incorporates eight phases [32, p. 661]. However, in our approach we only concern ourselves with a subset of these, the most significant of which being service Planning, Analysis and Design for our service-ready object managers (or pseudo services). Zimmermann et al. [43] suggests that these early service design stages are supported by approaches like Service-Oriented Modelling and Architecture (SOMA) and complementary techniques such as UML. SOMA [23] defines three service modelling steps comprising of Identification, Specification and Realisation; these steps consist of sub-steps prescribing

several artefacts with appropriate techniques at each level. They identify that it is around these steps that grounds for new processes and guidelines may be determined.

Similarly, IBM's [22] Rational Unified Process (RUP) provides an adaptable framework which can be tailored to meet organisations specific needs as it supports a range of disciplines and best practices. Having a number of variations, such as Agile Unified Process and Enterprise Unified Process; its potential lies within its engineering principles and best practices, which have already been adapted for SOA development [30].

Whilst all these approaches support each other, we consider SOMA to be the most significant to our service-ready approach as it focuses on how business functions can be translated into service-based applications in the design phase. However, before we can accurately analyse these SOA approaches, we first place emphasis on identifying some basic guidelines for service-ready object design and consider the characteristics of our pseudo services.

4.2 Service-Ready Design for Object-Orientation

We begin our investigation by looking at the service principles and comparing their meaning in each of the two programming paradigms. With this we discuss their adapted suitability to the service-ready approach we outlined in Sect. 3.2 through the identification of similar guidelines and appropriate design patterns.

4.2.1 Encapsulation

Although encapsulation means the same thing in both paradigms, Erl [12] suggests that they are used somewhat differently.

> *"Services still encapsulate logic and implementations, just as objects do... However, within service-orientation, the term encapsulation is used more so to refer to what is being enclosed within the container"* - Erl [12]

In a service-ready system we envisage encapsulation taking place in distinct manager classes using the Mediator design pattern [15]; this pattern would expose methods via corresponding interfaces for the given context, or multiple interfaces; one for use in the internals of the application and one (providing less functionality) for external.

> *"Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently."* - Gang of Four [15]

However, the Mediator pattern is not without problems, Freeman et al. [14, p 623] states that a drawback of using the pattern is that it is easy for objects to become too complex without proper design. Subsequently, we consider how our pseudo services should be designed by expending upon existing service guidelines such as Erl's [12] *"Do not define public attributes in interfaces"*. Erl proposes that the removal of attributes from public interface forces all communication through methods and therefore places control of state management within the service. In a service-ready context this guideline is of some significance as, although they are not typically stateless, we argue that object managers alone should adhere to the Single Responsibility Principle (SRP) for handling a specific domain aspect.

We are thus led to consider the alignment of the service-ready approach to SOA through the design of its ensuing manager classes and how the Mediator pattern proposes a kind of pseudo service which we see to be consistent with a basic service [25, p 63]. We consequently argue that, in order to later support the first stage of expansion to SOA, Allen's [1] guidelines for achieving the optimum level of granularity for a low level service are entirely relevant for our own service-ready classes:

1. It should be possible to describe the service in terms of function, information, goals, and rules, but not in terms of groups of other services.
2. The function set of a service should operate as a family unit that offers business capability.
3. A single role should take responsibility for the service (SRP).
4. The service should be as self-contained as possible (autonomous).

In object-orientation the Observer pattern is often used instead of the Mediator as whilst they both provide a separation-of-concerns it also promotes association. Both patterns support decoupling through the use of inversion of control (IoC), whereby the simple objects are considered to be more communicative and managers alone are responsible for handling objects. However, unlike the Observer pattern the Mediator promotes further decoupling by keeping objects from referring to each other explicitly, this enforces the SRP by allowing communication handling in a specific class. Accordingly, in order to decouple our service-ready manager classes and simulate service behaviour, we propose to substantiate such guidelines with design rules that restrict message types to those that are either built-in types to the language or instances of the containing object.

4.2.2 Inheritance

Erl [12] argues that inheritance is discouraged in service-orientation, except through its contract and logic.

> *"Due to the emphasis on individual service autonomy and reduced inter-service coupling, inheritance between services is generally discouraged within service-orientation."* - Erl [12]

Nevertheless, there still needs to be a constant awareness that in the future any coarse-grained service may need to be decomposed into finer grained services. Accordingly, Erl's [12] guideline *"Use inheritance with care"* also relates to how the components of an object-oriented system are bound through inheritance structures whereby complex relationships make it more difficult to physically separate services.

Josuttis [25, p. 46] underpins this argument stating that *"loose coupling has drawbacks. For this reason, loose coupling should never be an end in itself"*. In order to address these challenges whilst still retaining the advantage of using inheritance in a service-ready intermediary state, we propose using separate class definitions for the internals and externals of a system through an expansion of Ragonis' [34] approach which uses simple and composed classes. The concept of simple refers to a class whose attributes are of built-in types to the language, and composed refers to a class that has attributes of different user-defined types. Similar in design to the Bridge design pattern [15, p. 23], this approach suggests two implementations per object, simple to support use with other services by exhibiting abstracted behaviour, and extended to enable associations with other objects. As a complex inheritance structure is more difficult to decompose we thus propose a constraint whereby inheritance should only be used on an extended object domain model for a specialised hierarchy, placing emphasis on the chosen levels of abstractions for its success.

Almost all design patterns use inheritance to some extent [15, p. 23], the level of inheritance restricts what patterns can be used. At its most basic level we expect a service-ready system to reuse functionality through the use of *"mixin classes"* [15, p. 25] for the two contexts. Whilst this takes away some of the advantages of object-orientation, the Adapter design pattern can be used to retain some benefits as it allows developers to convert the interface of a class into another so that unrelated classes can work together [15].

4.2.3 Association, Composition and Aggregation

Services have little to no ownership structure and as such are free to invoke capabilities within each other. Furthermore, when we talk about composition in service-orientation we refer to a collection of services that should be able to act independently and are not limited to an ownership hierarchy.

> *"As with composition, aggregation does not apply to service-orientation because it is still based on a 'has-a' ownership structure... In service-orientation, the term 'composition' refers to an assembly or aggregate of services with no predefined ownership structure. Therefore, the rules associated with OOAD composition do not apply to service-orientation..."*
> - Erl [12]

Nevertheless, despite these constraints basic association, aggregation and composition can still be achieved in the underlying logic through the use of design patterns. However, as service interaction most closely resembles a *"uses-a"* relationship we argue that other forms of communication should be avoided in a service-ready system. This style of design is often considered best practice in object-oriented systems and as such we believe that as with Erl's [12] guideline *"Avoid cross-service relationships"*, a service-ready systems design should use design patterns that allow for a clear separation and discourage cross-communication between methods in order to avoid later conflicts.

Decomposition is the simplest approach to extending object-oriented systems whereby existing functionality is exposed as a set of services for reuse in other parts of the business [27]. Through a combination of approaches SOA and OOP can enable the orchestration of requirements without needing to worry about platform so as to represent distinct, single purpose functional contexts. To this end, the SRP will be considered in a service-ready approach in order to offer separation in terms of the two basic manager types. Entity (or data) managers for business objects with CRUD-like behaviour (such as get order) and logic managers for specific processes (such as process order). These pseudo services would be decomposed in the same way a well-design object-oriented system would expect by following the Don't Repeat Yourself (DRY) principle so as to reduce redundancy by extracting reusable logic into separate specified functions.

When designing service-based systems, the desired functionality of the software is divided in the same way as component-based systems. However, SOA differs in integration as the services are loosely coupled through their message exchange [32]. Although other distributed programming approaches exist; such as the Common Object Request Broker Architecture (CORBA) and Microsoft's Distributed Component Object Model (DCOM), we argue that a service approach offers the most flexibility and that such restrictions on the external part of the object domain model would enable simpler interactions for future consideration.

Erl's [12] guideline is similar to that of one of the core object-oriented design principles outlined by the Gang of Four [15]. The principle *"Favor object composition over class inheritance"* argues that object composition and delegation in particular provide flexible alternatives to inheritance for combining behaviour. However, they also argue that heavy use of object composition can make designs harder to understand, which undermines the purpose of our approach.

The motivation behind delegation is what drives the need for service composition as it is arguably one way of making composition as powerful for reuse as inheritance [15]. The use of delegation differs in essence as the requester does not receive a result, only a promise that the work will be carried out [6,

p. 466]. Whilst this level of decoupling is appropriate for separating various responsibilities across large-scale systems, we consider delegation to only be appropriate when it simplifies rather than complicates [15, p. 31], and that the delegation pattern is more of an approach for the encompassing service methods that may come later rather than the opening service-ready design.

4.2.4 Generalisation, Specialisation and Abstraction

Erl [12] discusses how both generalisation and specialisation are closely related to abstraction in object-orientation as no formal inheritance relationships are defined.

> *"Within the context of service design, generalization and specialization relate directly to granularity. The more specialized a service, the greater its degree of service-level granularity."* - Erl [12]

Similarly, abstract classes are not used directly within service-orientation, however, as finding the correct functional context of each service is an important part of the design process Erl [12] believes that they are of some use when identifying granularity. Accordingly, his guideline *"Use abstract classes for modelling, not design"* proposes advantages when present in an object-oriented context as they can still be used informally to ensure consistency and (although they are of no use in the service context) can be helpful for considering service candidates whilst modelling.

The effectiveness of decomposition is often dependent upon the existing levels of abstraction. Fortunately for novices, a well-defined architecture is relatively simple to achieve through the implementation of class interfaces [12]. The core difference between the two paradigms lies within the complexity of the returned values whereby only a minimal public interface, which uses base-types (or simple types in WSDL 2.0) would play a key role in designing a service-ready interface. The simple and extended contexts could be supported by the Facade pattern [15, p 199], whereby a unified interface can be used in each of the two subsystems.

In addition, as service abstraction also aims to limit access to service implementation through contacts (interfaces), we consider Erl's [12] guideline *"Limit class access to interfaces"* to be appropriate in a service-ready system, which would make use of specific interfaces for public access and limit interaction to the relevant domain segments. However, a service-ready context would also consider this principle on different levels and present extended classes that support a basic ownership structure for use in the internals of the application so that they can be abstract in design to support basic association, aggregation and composition for polymorphic behaviour and code reuse. The difference being that, only the simple context would strictly adhere to the aforementioned Mediator pattern (Sect. 4.2.1). In addition, as sub-classing is not an option

in a service context, there are a couple of patterns which could be used to support the design of such objects. These include the Decorator [15], which allows the addition of responsibilities in objects, and the Proxy which provides a substitute when it's inconvenient or undesirable to access a subject directly [15]. Both approaches would allow one service to manage multiple states.

In order to carry out OOAD with service readiness in mind it can be argued that modelling is an important aspect as it grants novices a visual representation of what they are aiming to achieve. A cross-section of the modelling techniques [10], suggests that decoupling need not be specific to SOA, and that a similar affect could be achieved within OOP to help prepare systems for service integration. In theory, this alone would make the later transition simpler. However, retaining the object-oriented advantages whilst suggesting this guideline is a different matter entirely, and would inevitably shift the focus during the design phase to modelling without complex relationships whereby new informative and communicational objects would be introduced in their place.

4.2.5 Polymorphism

Inheritance between objects and services is different because there is no formal concept of subtyping in service-orientation. Since polymorphism is generally reliant on this specific form of inheritance, which is commonly known as *"true inheritance"* [9], it also is not used in the same way within service-orientation. The closest thing to it is derived from the support of service contracts or interfaces (through interface inheritance) in the underlying implementation, whereby CRUD-style operations can be exposed for consistency.

Glen [16] discusses how XML extensions can be used to support polymorphic web services by providing basic inheritance capabilities, thus enabling the use of base objects in services. However, the paper continues by discussing that, whilst this provides some degree of polymorphic behaviour, it is still no substitute for object-orientation, as web services generally adopt a stateless invocation pattern.

> *"This typically results in similar or identically named capabilities across numerous services."* - Erl [12]

This links to the Open-Closed Principle (OCP) which is central to the success of service-integration as it suggests classes should support extension but disallow modification. This common best practice refers to interface consistency and suggests that the system structure should support override and extension, thus keeping the same name for a consistent service contract.

4.3 Findings and Feasibility

We argue that with the right guidance a service-ready approach could be formed to empower developers, enabling them to produce object-oriented systems that can both easily accommodate services and support the transition to SOA. In his article Erl [12] compares the service principles to object-orientation in order to help readers understand the problem domain. We recognise this method of knowledge representation as an easy way to demonstrate our service-ready ideas and as such have considered our own principles by looking at those found within existing literature [1,12,15,25].

An alignment of basic service design principles and the goals of our service-ready manager classes (pseudo services) are made apparent in Sect. 4.2.1, we thus consider the same guidelines for achieving the optimum level of granularity but also conclude that the following would be valuable when designing a service-ready object-oriented system:

1. Always use interfaces, but encourage the separation of internal and external logic, whereby the core logic is kept within the internals of the application (Encapsulation).
2. Use inheritance on common functionality within the internals of the application (Polymorphism).
3. Interfaces should use a consistent naming convention, so as to avoid overhead when switching over to use service contracts at a later date (Polymorphism).
4. Avoid functional overlap by ensuring individual interfaces are used for public access (Inheritance).
5. Avoid forms of communication between external parts of the application (services) other than a *uses-a* relationship (Inheritance).
6. Avoiding complex relationships and plan for multiple implementations for internal and external aspects (Delegation).
7. Separate into groups, such as managers for handling CRUD-like behaviour for entities and services for handling business processes (Association, Composition and Aggregation).
8. Only use managers for handling simple objects, thus enabling a clear separation of concerns (Abstraction).

We discussed each of the service principles and their associated guidelines to come up with our own basic guidelines for novices (above). In addition, we considered design patterns to provide direction during implementation, as they often come bundled with extensive literary support making them relatively easy for novices to learn and understand. However, from our experience novices seldom use patterns (or at least the correct one) because it is difficult to select the most appropriate for a given solution [15]. As such, whilst any patterns which facilitate scalability and adhere to one or more of the service principles are theoretically applicable, we have initially supplemented each area with appropriate behavioural patterns that simplify object composition.

While behavioural patterns are targeted towards code separation and avoiding functional overlap, creational patterns play an important part as systems evolve to depend more on object composition than class inheritance [15, p. 96]. In contrast to creational patters, which are relatively easy to implement (as all creational patterns have the same goal), many structural patterns such as the object system layer [17] have evolved from the need to resolve complexities found in the integration process. Nonetheless, Yang et al. [40] argues that current approaches are somewhat lacking for service composition and in later works [31] unifies these principles and concepts to promote an approach for extending conventional SOA. The extension of these early approaches are of some significance as their adaption to a new context may open the way for further service-ready design aspects.

In this section we have discussed how *objects may be designed to act like services* so that novices with little to no understanding of SOA can design a system that supports its adoption. The overall goal of this approach is to be able to later expose these existing functions as services with relative ease due to a more appropriate structure. By designing object-oriented systems that support services we expect to increase the service-oriented expertise in novice developers, however, the delivery of such ideas will be core to their understanding.

Riel [35] suggests that an experienced system analyst can interrogate a design to identify its strengths and weaknesses and that consequently, much work has been put into place to capture this skill through the development of heuristics. In addition, we consider that due to their extensive use in industry; heuristics not only offer the simplest implementation, but would also support us in identifying existing concepts that closely adhere to Erl's [11] service-oriented principles. There are many varying types of heuristics that have been explored by a number of authors [5,24,35], however, in their study into automatic problem detection Bär & Ciupke [5] found that heuristics from different sources contained both a mixture of similarities and contradictions. Accordingly, we foresee that such a delivery method would need to consider how the introduction of patterns into an unknown environment could conflict with any already in place so as to not compromise their underlying purpose.

5 Conclusion

5.1 Implications

While OOP ordinarily has limitations exposing processes, SOAs present core business logic as services. However, the ability to manage large-scale change and bridge the gap between functions and system operations requires developers to have the right skillsets.

The service-ready design methodology offers a different perspective to traditional software engineering techniques whereby focus is placed on reducing complexity rather than offering a solution to a specific set of requirements. By providing a collection of guidelines that uses object-orientated syntax to prepare systems and their developers for knowledge transfer we believe competency would be increased to permit even the most intricate service adoption techniques.

Possible implications for such work could bring about frameworks for facilitating skill transfer whilst acting as an enabler for change within organisations. This change, to consider service-oriented principles, has advantages from a business perspective through the modernisation of methods and approaches that would allow for the continued use of existing assets.

5.2 Future

Throughout this paper we have outlined the advantages of a service enriched environment. Using this mind-set we anticipate that future work would see the extension of traditional classes in order to prepare object-oriented systems for service integration [33], however, in contrast to existing approaches we foresee that the service-ready ideals would not offer any advantage until much later in the product life-cycle. Although we looked at approaches that unify SOA (Sect. 4.1), we foresee a need to analyse RUP and SOMA to identify specific approaches that align with the service-ready guidelines and pseudo service characteristics outlined in this paper (Sect. 4.3), along with investigations into studies that look into the early introduction of service components through additional constructs, such as that of Guizzardi et al. [19, p. 5] who create stereotypes to support object modelling.

In his work Dori [10] recognises the differences in the top-level design processes of the two paradigms and proposes an underlying model for SOA called the Object-Processes Methodology (OPM) which enables the graphical representations of objects, states and processes.

> *"While OO puts objects and their encapsulated behavior at the center stage, emphasizing primarily rigid structure, SOA hails services as the prime players to cater primarily to behavior."* - Dori [10]

Similar to our own objective, this approach sets out to enable developers to begin thinking in fundamentally different ways during software design. However, as we have primarily focused upon the levels of granularity, our goal to ultimately simplify service integration for novices still has a number of shortcomings, accordingly, in future works; modelling would be an excellent place to continue our research as it helps visually portray our ideas and place focus on all aspects of service design. Moreover, while we have identified design patterns that help achieve our goal, we foresee a need to conduct an in-depth analysis

of existing techniques from industry for identifying services, including other structural patterns which may be altered to align to one of the service-oriented principles.

Finally, we recognise that heuristics are another good first delivery element for service-ready design as they often contain object-oriented strategies that adhere to the service principles. In future work we will consider qualitative associated studies such as that of Ragonis & Ben-Ari [34] who evaluate student marks in order to identify 58 conceptions and difficulties for analysis as carried out by novices and that of Holland et al. [21] who identify equivalent concerns in teaching OOP with a list of students' misconceptions and a probable source for each. The identification of such potential problem areas will ultimately be used to expose which object-oriented methods effectively interpret the service principles.

5.3 Summary

From our initial findings we foresee little need to re-educate developers with expertise in OOP, forcing them to learn a different paradigm needlessly. Rather, the benefits of SOA can be exploited and integrated with OOP. Through this service-ready approach, OOP expertise can resolve the technical dichotomies between it and SOA, bridging the gap between two programming perspectives that are often considered individually and aligning the latest trends in both paradigms to provide an avenue from which key topics like inheritance and hierarchy can be compared and discussed. We are thus persuaded that the service-ready approach is a positive way forward, and seek its adoption whilst taking on board such considerations as it progresses.

In this paper we reflect upon a present lack of understanding when it comes to identifying when and where to use OOP and SOP, and that as a result mistakes in process are only realised after it is too late. Subsequent investigations would identify the point at which novices are made aware of such difficulties and the implications it has on service-oriented software. This information can then be analysed alongside common challenges, existing methodologies, approaches that unify SOA (Sect. 4.1) and modelling aspects for knowledge transfer. This analysis would then discover a number of appropriate research methods and techniques that fit the service principles, ultimately producing a seamless conceptual framework for a service-ready object-orientated approach to software development.

References

1. Allen, P.: Service Orientation: Winning Stratergies and Best Practices. Cambridge University Press, New York, NY, USA (2006)

2. Armstrong, J.: The development of erlang. SIGPLAN Not. **32**(8), 196–203 (1997). doi:10.1145/258949.258967
3. Arsanjani, A.: Toward a pattern language for service-oriented architecture and integration, part 1: Build a service eco-system. http://www.ibm.com/developerworks/webservices/library/ws-soa-soi/ (2005). Accessed 30 October 2011
4. Baarda, P.J.: Your SOA needs a business case. http://www.via-nova-architectura.org/files/magazine/baarda.pdf (2008). Accessed 18 February 2012
5. Bär, H., Ciupke, O.: Exploiting design heuristics for automatic problem detection. In: Workshop ion on Object-Oriented Technology, ECOOP '98, pp. 73–74. Springer-Verlag, London, UK (1998)
6. Brown, P.: Implementing SOA: Total Architecture in Practice. TIBCO Press Series. Prentice Hall (2008)
7. Buck, D., Stucki, D.J.: Design early considered harmful: Graduated exposure to complexity and structure based on levels of cognitive development. SIGCSE Bull. **32**(1), 75–79 (2000). doi:10.1145/331795.331817
8. Clark, D.: Designing OOP solutions: Identifying the class structure. In: Beginning Object-Oriented Programming with VB 2005, pp. 11–30. Apress (2006). doi:10.1007/978-1-4302-0095-6_2
9. D'Andrea, V., Aiello, M., Aiello, M.: Services and objects: Open issues. In: European workshop on OO and Web Service, pp. 23–29 (2003)
10. Dov, D.: SOA for services or UML for objects: Reconciliation of the battle of giants with object-process methodology. In: SwSTE, pp. 147–156. IEEE Computer Society, Herzlia, Israel (2007). doi:10.1109/SwSTE.2007.10
11. Erl, T.: SOA Principles of Service Design (The Prentice Hall Service-Oriented Computing Series from Thomas Erl). Prentice Hall PTR, Upper Saddle River, NJ, USA (2007)
12. Erl, T.: Service-orientation and object-orientation part i: A comparison of goals and concepts. http://www.soamag.com/I15/0208-4.php (2008). Accessed 29 January 2011
13. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston, MA, USA (1999)
14. Freeman, E., Freeman, E., Bates, B., Sierra, K.: Head First Design Patterns. O' Reilly & Associates, Inc. (2004)
15. Gamma, E., Helm, R., Johnson, R.E., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA (1994)
16. Glen, S.: Polymorphic web services, part 1: Polymorphic data (2008). URL http://www.ibm.com/developerworks/xml/library/ws-polymorphic/. Accessed 07 December 2012
17. Goedicke, M., Neumann, G., Zdun, U.: Message redirector. In: Proceedings of EuroPloP 2001, Sixth European Conference on Pattern Languages of Programs. Irsee, Germany (2001)
18. Graham, I., O'Callaghan, A., Wills, A.: Object-oriented methods: principles & practice. Addison-Wesley Object Technology Series. Addison-Wesley (2000)
19. Guizzardi, G., Wagner, G., van, M.S.: A formal theory of conceptual modeling universals. In: Workshop on Philosophy and Informatics (WSPI), Cologne, Germany, 2004. Deutsches Forchungszentrum fur Kunstliche Intelligenz (2004)
20. Hoffman, K.: Microsoft Visual C# 2005 Unleashed, chap. Designing for Service-Oriented Architectures (SOA). Pearson Education (2006)
21. Holland, S., Griffiths, R., Woodman, M.: Avoiding object misconceptions. SIGCSE Bull. **29**(1), 131–134 (1997). doi:10.1145/268085.268132
22. IBM: Rational process library. http://www.ibm.com/software/awdtools/rmc/library/ (2003). Accessed 05 December 2012
23. IBM: Service-oriented modeling and architecture. http://www.ibm.com/developerworks/library/ws-soa-design1/ (2004). Accessed 07 December 2012
24. Johnson, R.E., Foote, B.: Designing Reusable Classes. Journal of Object-Oriented Programming **1**(2), 22–35 (1988)
25. Josuttis, N.M.: SOA in Practice: The Art of Distributed System Design. O'Reilly, Beijing (2007)

26. Karsten, R., Roth, R.M.: The relationship of computer experience and computer self-efficacy to performance in introductory computer literacy courses. Journal of Research on Computing in Education **31**(1), 14–24 (1998)
27. Koskela, M., Rahikainen, M., Wan, T.: Software development methods: SOA vs. CBD, OO and AOP. http://www.soberit.hut.fi/t-86/t-86.5165/2007/final_koskela_rahikainen_wan.pdf (2007). Accessed 20 December 2010
28. Marks, E.A.: Service-Oriented Architecture (SOA) Governance for the Services Driven Enterprise. Wiley Publishing (2008)
29. Mcbeath, J.: Scala pros and cons. http://www.scala-lang.org/node/8462 (2010). Accessed 30 October 201
30. Mittal, K.: Service oriented unified process (soup). http://www.kunalmittal.com/html/soup.html (2010). Accessed 24 December 2012
31. Papazoglou, M.P., Heuvel, W.J.: Service oriented architectures: approaches, technologies and research issues. The VLDB Journal **16**(3), 389–415 (2007). doi:10.1007/s00778-007-0044-3
32. Papazoglou, M.P., Traverso, P., Ricerca, I., Tecnologica, S.: Service-oriented computing: State of the art and research challenges. IEEE Computer **40**, 38–45 (2007)
33. Ragonis, N.: Teaching object-oriented programming to novices. Ph.D. thesis, Weizmann Institute of Science (2004)
34. Ragonis, N., Ben-Ari, M.: A long-term investigation of the comprehension of OOP concepts by novices. Computer Science Education **15**, 203–221 (2005). doi:10.1080/08993400500224310
35. Riel, A.J.: Object-Oriented Design Heuristics, 1st edn. Addison-Wesley (1996)
36. Rosen, M., Lublinsky, B., Smith, K.T., Balcer, M.J.: Applied SOA: Service-Oriented Architecture and Design Strategies. Wiley Publishing (2008)
37. Scala: The scala programming language. http://www.scala-lang.org/node/25 (2008). Accessed 30 October 2012
38. Udell, J.: The spiral staircase of SOA. InfoWorld **27**(40), 46–46 (2005)
39. Voigt, J.: Characterising the use of encapsulation in object-oriented systems (2009)
40. Yang, J., Papazoglou, M.P.: Service components for managing the life-cycle of service compositions. Inf. Syst. **29**(2), 97–125 (2004). doi:10.1016/S0306-4379(03)00051-6
41. Zdun, U.: Pattern-based design of a service-oriented middleware for remote object federations. ACM Trans. Internet Technol. **8**(3), 15:1–15:38 (2008). doi:10.1145/1361186.1361191
42. Zimmermann, O.: SOA decision modeling (SOAD). http://soadecisions.org/soad.htm (2009). Accessed 17 December 2012
43. Zimmermann, O., Schlimm, N., Waller, G., Pestel, M.: Analysis and design techniques for service-oriented development and integration. pp. 606–611 (2005)