Restful Contexts

Cevat Serkan Balek, Prof. Dr. Nadia Erdoğan
İstanbul Technical University

Abstract
Too much emphasis on nouns or data leads to architectural styles which almost always neglect the importance of verbs or behavior which are needed to complete the sentence. Inheritance is neither sufficient nor suitable for capturing the form of function, we need more suitable concepts like contexts. This study presents an extension to Restful Objects Specification with all necessary constructs to support DCI execution model, not only to bring back the importance of form of function but also to enable querying contexts and executing interactions; all of which can only be achieved by specifying the communication of network of objects in collaboration as the third dimension in computation besides storage and transformation as properties and methods. As Restful Contexts increases readability and enhances behavioral specifications with new constructs like roles, attacking problems like distributed transactions within micro services will be easier.

Index Terms—readability, specification, MVC, DCI, REST, ORM, OIM, JSON, object orientation, role, context, contract, lateral, systems thinking, naked objects, restful objects

I.BEYOND OBJECT ORIENTATION (*AS WE KNOW IT*)

Object orientation is still being misunderstood and there are strong arguments that it is going on wrong direction [1]. For more than two decades, object orientation is under heavy critics [2][3]. Apart from suggesting to use a completely different paradigm such as functional programming, there are various attempts to extend it by mainly traits, mixins and role constructs to a level which will at last enable it to evolve towards the original vision of Alan Kay who coined the term:

"In computer terms, Smalltalk is a recursion on the notion of computer itself. Instead of dividing "computer stuff" into things each less strong than the whole -like data structures, procedures and functions which are the usual paraphernalia of programming languages- each Smalltalk object is a recursion on the entire possibilities of the computer. Thus its semantics are a bit like having thousands and thousands of computers all hooked together by a very fast network." [4] Alan Kay's original vision of object orientation which is *to hook millions of computers together by a very fast network* has never been so close to be realized than today. Popularity of micro services architecture is a bold sign that his vision to decentralize the data and capabilities of systems as a web of small, independent and self-contained units of computation in strong coordination is gaining wide-spread acceptance.

The main problem with object orientation as we know it, is that, it is not really object but class oriented at its core. The only core way to extend the objects (as instances of classes) is to inherit from a base class which is quite usable for most but not all occasions -even- in the data perspective. However, extension by inheritance approach is really a big mismatch with behavior perspective. Since we have only this core mechanism to differentiate both data and behavior of the objects (which are designated as instances of a class), the inevitable result is a massive dispersion of behavior code in various subclasses. This is against readability principle [5] which leads to other systemic problems, at least testability.

There have been many improvements to fix this problem: interfaces, aspect orientation, traits [6] based language extensions for roles and contexts, JAWIRO [7], OT/J [8]. Most of these solve some part of the problem very well, and they provide rich set of features, but none of them include the constraints and provide a complete conceptual framework for modeling behavioral code to give function its natural form which is depicted as data communication in DCI execution model with all relevant concepts like contexts, roles, role methods, role object contracts as defined in DCI glossary [9].

Concepts of *context* and *role* which have been studied for over two decades is the key to the solution, and *lateral or systems thinking* [10] is the key principle behind it. If we can capture the *form of function* [11] and specify that form to support well formed safe coordination of so called mini-computing blocks, we will have *roles*, *contexts*, and *contracts* as the new mechanism instead of relying only on inheritance to safely hook these mini-computing blocks. Behavior is now a first-class citizen as data, none of them is sub to the other.

II. EMBRACING FULL OBJECT ORIENTATION

In today's computing environment *no code stands alone*. Plethora of systems and sub-systems communicate with each other and with end users to accomplish tasks of various sizes and complexities. They are developed in different languages, frameworks, operating systems, devices, so on.

In this study, instead of providing yet another extension for a language in isolation, I took a different approach to *expose functions as resources* by extending Restful Objects Specification [12] with context, role and relevant concepts which are required to define and orchestrate the interaction of units in

collaboration. DCI (Data-Context-Interaction) [13] is chosen as the supporting conceptual framework and the execution model for the extending Restful Objects because:

- it decouples inheritance mechanism from specifying the behavior in code and restricts the use of classes to only where they are needed and effective most
- it is a natural extension to object orientation to include use cases and the like directly in the code which were thought so far only as analysis artifacts
- it can be applied to a heterogeneous environments with multiple languages and operating systems

A. HTTP/REST

Although there are emerging alternatives like GraphQL, HTTP/REST [14] gradually becomes the norm to hook small units of computation as in micro services implementations. Despite the fact that decentralized computation is gradually becoming the new mainstream, specification of coordination of so called mini-computing blocks is still a huge problem which is not addressed in its entirety so far by any protocol. Design by Contract, OCL, and their derivates like JML [15] which focus on specification of object conditions say nothing about specification of coordination as in UML collaboration.

Restful Objects which follows Naked Objects [16] is a novel attempt to define exposable domain objects with their properties, collections and methods in RESTful style. The study encouraged OIMs (Object to Interface Mapping) which is similar to ORMs (Object to Relational Mapping).

Original vision of MVC, direct object manipulation [17] was demonstrated by the Naked Objects implementations. Although Restful Objects specifies necessary constructs of traditional object orientation in generic sense, it lacks what traditional object orientation (which it is designed to support) also lacks: the specification of behavior in coordination

B. CRUD only APIs

In today's highly dynamic environment, interactions are no longer as simple as they were decades ago; at the times when displaying data to the user or sending it to the system and getting back amendments or new commands from the user or the system was sufficient for most use cases [18].

However, the primary way we cope with today's increasing complexity in interactions is still almost the same as it was decades ago. Modern API's still mainly relies on CRUD. API Gateways [19], for example, as one of the most modern approach today provides a well designed facade to shield the internal API design at server from variety of client request structures and relevant variety of response structures by transformation and redirection of requests and responses to adapt each other. They also provide functions like security, performance and balancing, but specification of functions in a bounding context is never a part of them. As an another example, apart from some promising research on ontology of behavior [20], main interest of object web ontologies is data; individuals and properties which relate individuals [21].

III. PROPOSED EXTENSION

Let's examine the Restful Objects Specification first.

A. Restful Object Specification

The following diagram shows how the RESTful objects as resources are defined, along with the representations that they generate:

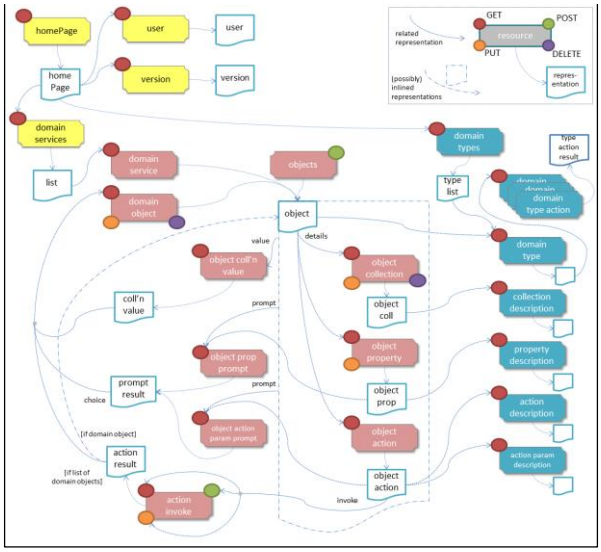

Fig. 1. Resources and Representations [12]

Restful Object Specification is specified in detail in the reference document. Each resource returns a representation.

The Restful Objects Specification defines that these representations are in JSON (Java Script Object Notation). For resources called with a PUT or POST method, a representation must also be provided as a body to the request, describing how the resource is to be modified.

The specification defines a few primary representations:

- *object* (represents any domain object or service)
- *list* (of links to other objects)
- *property*

- *collection*
- *action*
- *action result* (typically containing either an *object* or a *list*, or just feedback messages)

Restful Objects can be thought as a way to expose objects as HTTP resources with all of their properties and methods, not only for getting or setting properties and invoking actions but also providing information about them when queried.

TABLE I: EXAMPLE RESTFUL OBJECTS CALLS

Method	URL	RESULT
GET	http://~/	home page
GET	http://~/services	list (of links) to services
GET	http://~/services/ProductRepository/	product repository service
GET	http://~/services/ProductRepository/actions/FindByName	action (to be rendered in UI as a dialog)
GET	http://~/services/ProductRepository/_actions/FindByName/_invoke/?Name=cycle	list of links to matching product objects
GET	http://~/objects/product/8071	object (represents a product)
POST	http://~/objects/product/1234/_actions/AddToBasket/invoke	
GET	http://~/services/BasketService/_actions/ViewBasketForCurrentUser/invoke	list of links to Item objects
POST	http://~/objects/orderitem/1234/_properties/Quantity	
DELETE	http://~/objects/orderitem/517023	

B. DCI Execution Model

As with other role based language extensions, DCI also decouples the algorithm or procedure (interaction) from the code that represents the entity or class (data) as in Figure 1. By doing so, DCI enables and encourages lean architecture which decouples what the system is from what the system does, both of which naturally evolves at very different rates and therefore should never be on the same basket [22]. DCI radically differs from the other role based techniques by restricting the use of polymorphism only

for describing entities (data) where it is needed and effective most. For behavioral code, polymorphism is strictly forbidden.

As Liskov argues, attaining simplicity is hard because people tend to complicate things rather than simplify them as simplification is much harder process [23]. Without a constraining framework like DCI, simplicity is even harder to attain because there is no tool or language support but only blind guesses to distinguish between the levels of rate of change in a complicated code where behavior is dispersed. If software is an art, true art is not possible without technique.

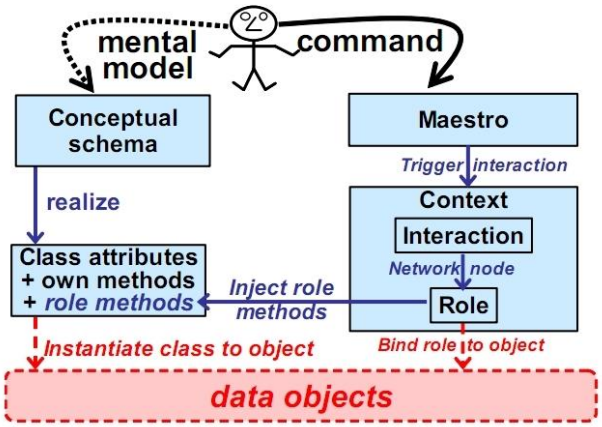

Fig. 2: DCI Conceptual Model [9]

Context may be thought as a container to host interaction code and to bind the data objects to perform their roles.

It is apparent that both data (left) and interaction (right) are first-class citizens, behavior is no longer sub as a method of some object (data). Methods still exist inside data objects, their function is to transform -as always-, not to coordinate.

These are also apparent from the diagram:

- Role methods are only needed as long as the context (use case) is alive, therefore the class is free of satisfying any interfaces or including behavior code for any context that it may play a role within. If a class has all properties and methods of its own which is required by role contract, that's enough. Interaction itself is coded in role methods.
- Interaction code is inside the context and is decoupled. That means, they can be implemented in a very different language and served as a service on a server elsewhere.
- Therefore, it is possible to devise protocols like Restful Contexts to serve interaction.
- When the interaction finishes, the context is disposed and none of the objects remembers none of their roles or role methods to carry out that interaction.

DCI Execution Model and Glossary [9] are the reference documents. There is a reference DCI implementation [24].

There are a couple of things to consider at this point:

1. To design a protocol for heterogenous environments supporting many languages, injection-less DCI should be used. This is convenient for HTTP/REST implementation.
2. Since we are not designing a language extension but a protocol extension, we need to focus on API.

3. We can consider contexts as use case realizations (as in UML collaborations) directly in the interaction code.

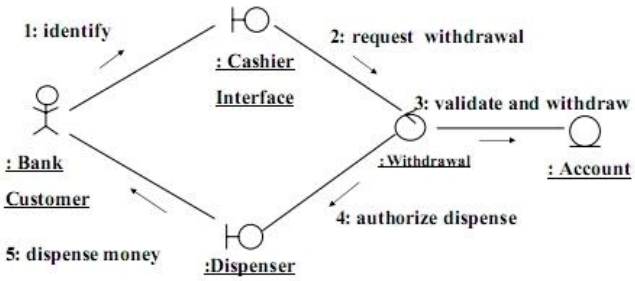

Fig. 3: An example UML collaboration diagram

Also, since we are on HTTP/REST land, we can add even more useful specifications such as use case requirements (pre-conditions), guarantees (post-conditions), and free text.

C. DCI Based HTTP/REST Execution Model

Along with services and objects which are already defined in Restful Objects Specification, we define context as the new kind of resource.

- *Context* specifies a networks of communicating objects. A *context* coherently maps all the *role*s onto the objects, keeping the result in the *role map* of the context *instance*. Both the *context* itself and its instances can be queried.

http://~/contexts/MoneyTransfer

With GET, this *context* resource gives all the necessary information which conforms to DCI as below [9]:

- *Requirement* is a condition which is written in terms of *role object contracts* and *role methods* to satisfy in order to start the interaction (MAESTRO block in the figure).
- *Role* is an alias for an object in a network of interacting objects within a c*ontext* instance.
  - *Role Object Contract* is an assurance that only objects reifying a required set of messages (i.e. properties and methods) can play a certain *role.*
  - *Role Map* maintains the current mapping between the *roles* and *role players* as data objects.
  - *Role Method* is a stateless and non-polymorphic method that is shared among all *role player* objects.
  - *Role Player* is an object that fills the position of a r*ole* in a network of communicating objects.
- *Responsibility* is a condition which is written in terms of *role object contracts* and *role methods* to satisfy to prove the validity of the interaction.

Again, since we are in hypermedia land, we are able to add any kind of human readable description to any of these information. This removes the impedance mismatch between coders and analysts. Just ask the necessary information about the *context* directly to the server which is responsible of carrying out the interactions in that *context*.

So, where is the algorithm (interaction)?

Algorithm (interaction) is a self-contained code written in the language that serves the *context* in terms of *roles* and *role methods*. There is no polymorphism here, we don't need it. *Polymorphism is never the essence of form of function.* End users don't categorize the flow of steps in their mind and calculate their new path to follow based on those categories as they exactly do with data. So we do the same in the code! We think about cases not hierarchies. No hidden surprises.

Notice that, *recursive contexts* suddenly becomes feasible which paves the way to apply systems thinking to design systems of systems of systems, ... Each level will know only and completely the details needed at that level, nothing less or nothing more. *Requirements* and *responsibilities* as in use case pre-conditions and success guarantees will bind these complex web of interactions without sacrificing readability.

Being able to decouple and serve the interaction code in its own is a neat solution for distributed transaction problem within micro services architecture too. It is possible to code the complete transaction (orchestration) with all the use cases including what to do or not to do in exceptions, timeouts, etc. No surprises and complete readability of the transaction.

So, what about PUT, POST and DELETE a *context*?

Semantics follows REST style for them too.

With PUT, we create the *context* by bringing the collaborating parties together with their corresponding Restful Objects link, check their eligibility to play all the *roles* required by the *context* and check the requirements to start the interaction. If everything is okey, interaction starts. If not, *context* waits for either POST or DELETE. Of course, after a certain period the *context* is automatically deleted.

With POST, we can update the *context* before it is started. There is a special keyword MAESTRO to initiate checking requirements, etc. and restart the *interaction* when the client of the *context* is sure about all the *role players*. Otherwise, *context* assumes that other POST requests may follow to further modify (mutate) the collaborators in the *context*.

As the *context* has hypermedia links to all collaborating objects with their properties, methods and collections, etc., interaction code progresses by calling collaborating parties as necessary as their turn comes in the script. That's it!

You can imagine how this simple but intelligent system enables a very alive, dynamic and also a very reliable web of connections to carry out variety of tasks done. Collaborators won't remember how to play their role as their part is written in script and saved in context, nowhere else.

“Simplicity is the ultimate sophistication”

Leonardo da Vinci

D. HTTP/REST Execution Protocol of Restful Contexts

We add contexts to Restful Objects as

http://~/contexts/<ContextName>

In this general form, system returns general specification of the context as lists of requirements, roles, responsibilities.

- *Requirements* (for the context to start the transaction) as names and short explanations of each business rule. Actual requirements check code is isolated and is written in terms of only *role object contracts* and role methods.

  http://~/contexts/<ContextName>/Requirements

- *Roles* as names and short explanations of each *role* required by the *context*. Each role has a list of *role object contracts* to specify the properties and methods required to exist in the object to play that *role* with all their types and parameters information as specified in Restful Objects.

  http://~/contexts/<ContextName>/Roles

- *Responsibilities* (of the context to consider it as valid) as names and short explanations of each *responsibility* which are to be fulfilled to consider the interaction as valid. Actual *responsibility* check code is isolated and is written in terms of only *role object contracts* and *role methods*.

  http://~/contexts/<ContextName>/Responsibilities

Notice that, we don't need to expose *role methods* at all, since conceptually they are needed only inside the *context*. In injection-less DCI, they are not even needed to be included in the specification as there is no need to inject *role methods.*

http://~/contexts/<ContextName>/id

gives additional information about the *context* instance.

To demonstrate the idea, *role map* is implemented as the only required additional information which returns the current mapping between the *roles* and *role player* objects.

http://~/contexts/<ContextName>/id/RoleMap

E. Role Object Contracts

Role object contracts are very similar to interfaces, i.e. an object need to fulfill all role object contracts in order to play that role. There is one structural advantage of decoupling the behavior here. Definition of an object property or method which is semantically the same as the required role object contract may be syntactically different than the role object contract. Although they mean the same thing in the context, they will be considered as different by rigid interfaces. We would have room for adapting both sides in Restful Contexts.

Neglecting design issues, suppose a class named *Client* has *CheckEligibility* method which takes *ForCountry* as the only parameter. Also suppose that we have *BookHotelRoom* context which requires - among other roles- a role which is played by an object of type *Client* in this case to fulfill its *Proceed* role method. *Proceed* does not take any parameters as it checks the eligibility of client only for Czech Republic.

- *Client* has *CheckEligibility(string ForCountry)* method
- *BookHotelRoom* needs *Proceed()* role object contract

By reading specifications of both the object's method and role object contract, analyst decides that *CheckEligibility* and *Proceed* means the same thing within their context as they plan to book hotel rooms only in Czech Republic. Therefore, whenever *Proceed* role method is called inside interaction, calling *CheckEligibility('CZ')* method instead is sufficient. Good news is that, this is just a simple HTTP redirection in Restful Contexts from *Proceed* to *CheckEligibility('CZ')* in the gathering stage of the

BookHotelRoom context. Although not included in this reference implementation, it is easy to extend this idea to include some transformations with simple calculations such as converting the units of measurement via the use of simple scripts or other simple mechanisms.

With Restful Contexts, the need for transformation of properties or methods in object code to denote the essentially same thing with similar but slightly different signatures in variety of contexts is completely eliminated by dependency inversion. It is context's responsibility to check the eligibility of the object for a role, not the other way around. Object is just what it is designed to be, no additional clutters for behavioral code. If an object is essentially suitable for a role, all the necessary transformations are done at context level.

IV. FURTHER STUDY

To the best of my knowledge, this study is one of the first studies to apply RESTful architectural style to behavioral code where the interaction is considered as a direct resource.

In this study, focus is on program comprehensibility and demonstration of the idea, not the performance. Restful Contexts Specification under caching systems, under load or under other circumstances are good candidates for further study. Application of the idea to GraphQL would be great too as GraphQL is more flexible and it requires less bandwidth overall for API communications in dynamic environments. Also, the protocol may be enhanced to give more information and even history about *context* instances, such as the results of requirement and responsibilities check with actual values.

V. CONCLUSION

The original vision of Alan Kay about hooking thousands of computers together by a very fast network now became a reality. IoT (Internet of Things), APIs, web applications, mobile phones, etc. everything is connected to internet and doing zillions of fast and reliable transactions every second.

Architectural styles such as micro services requires such protocols as Restful Contexts, and enterprises that already use or plan to use micro services should already start to care about such protocols to specify the coordination of webs of interacting services.

In this study, a reference HTTP/REST implementation is done to extend Restful Objects Specification with DCI to demonstrate how code comprehension and readability will dramatically increase along with systemic improvements when interaction as first-class citizen is decoupled from data.

Half a century ago, both object orientation and MVC are introduced. Decades later, direct object manipulation as one of the original visions of MVC was realized by Naked Objects implementations and Restful Objects born out of it. A few more years later, DCI was born and complemented MVC. I believe this study combines all of these techniques as a coherent whole in a reference HTTP/REST implementation to pave the way for further studies and experiments.

This promising field of research and opportunity to write cleaner code requires a radically different way of looking both at the problems and at the solutions. As computation gets increasingly distributed and complicated, interactions as collaboration of objects should have their own place in code.

To conclude with Tony Hoare:

“There are two ways of constructing a software design: one way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult… It requires a willingness to accept

objectives which are limited by physical, logical, and technological constraints, and to accept a compromise when conflicting objectives cannot be met. No committee will ever do this until it is too late." [25]

REFERENCES

1. T. Reenskaug, "Rethinking the foundations of object orientation and of programming", OreDev, 2009 https://vimeo.com/8235394
2. M. Fowler, "Anemic domain model", 2003 http://www.martinfowler.com/bliki/AnemicDomainModel.html
3. G. Bjørnvig, J. O. Coplien, N. B. Harrison, "A story about user stories and test driven development", Better Software Magazine, October 200
4. A. C. Kay, "The early history of Smalltalk", ACM, 1993
5. T. Reenskaug, "The case for readable code", Department of Informatics, University of Oslo, 2007 http://heim.ifi.uio.no/~trygver/2007/readability.pdf
6. N. Schärli, S. Ducasse, O. Nierstrasz, A. Black,"Traits: composable units of behavior", Proc. of ECOOP'03, LNCS, vol. 2743; Springer Verlag, page 248—274, [DOI] 10.1007/b11832, July 2003
7. Y. E. Selçuk, N. Erdoğan, "Role models—implementation-based approaches to using roles", Software: Practice and Experience, 2011
8. S. Hermann, "Object Teams for Java", https://wiki.eclipse.org/OTJ
9. T. Reenskaug, J.O. Coplien, "http://fulloo.info/Documents/", 2014
10. L. Mononen, "Systems thinking and its contribution to understanding future designer thinking", in Design for Next : Proceedings of the 12th European Academy of Design Conference (pp. S4529-S4538). Design Journal, Vol. 20, 2017
11. J. O. Coplien, "Restoring function and form into Patterns", Gertrud & Cope, 2010
12. D. Haywood, "Restful Objects Specification", July 2015, https://github.com/restfulobjects/restfulobjects-spec
13. T. Reenskaug, J.O. Coplien, "The DCI architecture: a new vision of object-oriented programming", Artima, 2009 http://fulloo.info/Documents/ArtimaDCI.html
14. R. T. Fielding, "Architectural styles and the design of network-based software Architectures", in dissertation to University of California, Irvine, 2000 https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf
15. G. T. Leavens, Y. Cheon, "Design by contract with JML", http://www.eecs.ucf.edu/~leavens/JML//jmldbc.pdf, 2006
16. R. Pawson, "Naked Objects", in thesis to University of Dublin, Trinity College, June 2004
17. T. Reenskaug, P. Wold, O. A. Lehne, "Working with objects" (The OOram Software Engineering Method), Taskon Work Environments, March 1995, https://folk.uio.no/trygver/1996/book/WorkingWithObjects.pdf
18. K. Sandoval, "7 Growing API trends", https://nordicapis.com/7-growing-api-design-trends/
19. C. Richardson, "API gateway pattern", https://microservices.io/patterns/apigateway.html
20. C. Bock, J.Odell, "Ontological behavior modeling", in Journal Of Object Technology, 2011
21. C. Bock, A. Fokoue, P.Haase, R. Hoekstra, I. Horrocks, A. Ruttenberg, U. Sattler, M. Smith, "OWL 2 web ontology language structural specification and functional-style syntax (second edition)", in https://www.w3.org/TR/owl2-syntax/
22. G. Bjørnvig, J. O. Coplien, "Lean architecture", Wiley, ISBN-10: 0470684208, 2010
23. B. Liskov, "The power of abstraction", OOPSLA, 2009 http://www.infoq.com/presentations/liskov-power-of- abstraction
24. J.O. Coplien, https://github.com/jcoplien/trygve, last commit: 2018
25. C. A. R. Hoare, "The emperor's old clothes", ACM Turing Award Lecture, 1980