ISSN 1512-1232

UDC 004.4

# Runtime Composition of Domain-specific Services

Nikoloz Pachuashvili
St. Andrew the First called Georgian University of the Patriarchy of Georgia, Tbilisi, Georgia

***Abstract***

*Architectural specialization makes easy to create reusable software components which can be effectively reused in future compositions. In this paper we discussed the domain-specific architecture, where general constraints are the nature of components and state (the data). Components are representing as services with sole operation and all service have same interface, i.e. all service accepts the same data, and result of service actions are stored in the same state. We discussed the possibility of runtime service compositions using the domain specific functional concepts, which are organized and represented as independent software components, calling them services. On the other hand, functional concepts are declared using domain-specific semantic functional descriptors.*

***Keywords:*** *composition, reuse, functional concepts, domain-specific modeling, hypermedia*

## Introduction

Reusable components are building blocks for software systems. When we are designing new software application, we have two general issue: satisfy all requirements of given application and create components which can be reused in future to create new or extend existing software systems. Reusable parts of existing software system is the effective way to compose new composite components. However, creating reusable parts of software was and still remains the big challenge. In 1995 David Garlan, Robert Allen and John Ockerbloom published the article [1], where was discussed the problems of reuse which was attributed to architectural mismatch. After the decade and half since that publication, same authors issued the second version of that paper [2], where was described the modern challenges of reuse and the contemporary state of architectural mismatch. Garlan, Allen and Ockerbloom examined four general categories of assumptions that can lead architectural mismatch: The nature of the components (including the control model), the nature of connectors (protocols and data), the global architectural structure, and the construction process (development environment and build) [1, 2]. The Architectural specialization was proposed as one way to help prevent architectural mismatch [2]. Our general purpose is to find effective way for runtime component composition, which is tightly depended on highly reusable parts deployed software. Architectural specialization gives ability to restrict the nature of components, connectors and data, using constraints which are legitimate in the scope of the particular domain. Thus, by narrowing the design context, our purposes becomes more prospective. In “Formal Method of Service Oriented Functional Decomposition” [3] we described technique which was used to extract and identify functional concepts of particular unknown domain. This technique helps us to distinguish functional attributes of service candidates during the modeling stage. In this paper we will describe how can be services composed in runtime.

## Reuse and composition techniques.

The process of composition is the creational process, which uses pre-existing components to create (compose) new software artifacts. Every composite system, contains least one reusable component as the member of composition. This means that we cannot create composite component without pre-existing reusable component, but not vice versa: not every case of reuse can be considered as a composition. For more clarity we will overview the different techniques of reuse and composition. Single inheritance is widely accepted, but not always efficient way of reuse in

ISSN 1512-1232
object oriented programming. This manner of reuse of pre-existing logic cannot be considered as the composition. However, in object oriented design there are more flexible ways of reusing pre-existing components in several composition. For example, composite and decorator design patterns [4] are good examples of flexible object-oriented composition. Mixin-based inheritance is formulated as composition of mixins. A mixin is an abstract subclass that may be used to specialize the behavior of a variety of parent classes, [5] Mixins are the basis for a compositional inheritance mechanism. Another flexible instrument of composition is traits [6]. Because, traits are concerned solely with the reuse of behavior and not with the reuse of state, they avoid the implementation difficulties that characterize multiple inheritance and mixins. In Scala [7] modular mixin composition [8] provides the flexible way to compose components and component types. On the other side, composition pattern can be static and dynamic. In static composition we mean the cases when component explicitly delegates the control to another components. The decorator and composite design patterns allow us to compose runtime components dynamically depending of context. In such cases the factory components are taking responsibility of actual component binding. Our work is attempt to create framework for flexible runtime compositions within the specific domains.

**Runtime service composition**

Before defining our model, let us overview the object-oriented techniques for runtime composition. For mote clarity, let us discuss the real world situation based on motor vehicle service. The car service center provides the variety of the services to motor vehicle owner. Such as:

- Change the engine oil
- Replace the oil filter
- Replace the air filter
- Replace the oil filter
- Tune the engine

Obviously, this is not the complete list of services which are provided by the motor vehicle service. But for our purposes it is enough. Let us imagine that particular customer (the owner of motor vehicle) appeared with specific requests: “Change the engine oil”, “Replace the oil filter” and “Check the condition of tire”. The motor vehicle can be considered as the sole input parameter for these services. The interfaces all of these services are same, they are accepting same state (motor vehicle) and are returning it back after the processing is done. The customer of motor vehicle service center has no information how to access (invoke) these services, owner of vehicle does not knows and does not care how these services are separated. From service consumer’s view this is one multi-functional service. Thus, customer requests the sole service, which one the other side is the composition of other services. The car service employee is responsible to help customer to locate desired service, then motor vehicle owner can consume this service, by accepting the vehicle as the input parameter.

```
/**
 * The general abstraction of car service.
 */
public interface MotorVehicleService {

    /**
     * Provides the service to vehicle
     *
     * @param vehicle (The state)
     */
    public void serve(Vehicle vehicle);
}
```

ISSN 1512-1232

The “MotorVehicleService” represents the Java interface, which is the abstraction of all types of motor vehicle services. “ChangeEngineOil” and “ReplaceOilFilter”, are concrete implementations of abstract “MotorVehicleService” interface:

```
public final class ChangeEngineOilService implements MotorVehicleService {

    @Override
    public void serve(Vehicle vehicle) {
        System.out.println("Changing the engine oil");
    }
}

public final class ReplaceOilFilterService implements MotorVehicleService {

    @Override
    public void serve(Vehicle vehicle) {
        System.out.println("Replacing the oil filter");
    }
}
```

Appling the composite design pattern, allows us to create runtime component which conforms the abstract “MotorVehicleService” interface, and provides to consumer the composition of simple services. “CompositeService” represents the implementation of composite motor vehicle service, using the classic composite design pattern.

```
import java.util.ArrayList;

public final class CompositeService implements MotorVehicleService {
    private ArrayList<MotorVehicleService> mix = new
ArrayList<MotorVehicleService>();

    @Override
    public void serve(Vehicle vehicle) {
        for (MotorVehicleService service: this.mix) {
            service.serve(vehicle);
        }
    }

    public void addService(MotorVehicleService service) {
        this.mix.add(service);
    }

    public void removeService(MotorVehicleService service) {
        this.mix.remove(service);
    }
}
```

Now, if we have appropriate factory class (Implementation of Abstract Factory design pattern [4]), it is easy to create runtime composite component which add the factory component. For example, if customer request the “Change engine oil” and “Replace oil filters” services, service factory will create the instance of composite service:

```
MotorVehicleService compositeService = new CompositeService();
compositeService.add(new ChangeEngineOilService());
compositeService.add(new ReplaceOilFilterService());
```

ISSN 1512-1232

We aim to create runtime service composition framework which will offer to customers, dynamic service composition. General technique which we are going to use is the domain specific functional descriptors, which will be used by service consumers to originate service request query.

**Domain-specific vocabulary**

In domain-specific vocabulary, we mean the collection of domain-specific semantic descriptors (keywords) with distinguished meanings, which are used to describe the functional characteristics of particular object. Using these descriptors, humans can exchange knowledge about several concepts from domain. The set of domain-specific descriptors, can be represented as a list of key value pairs, where key is the descriptor name and the value is the meaning of this descriptor. We assume that each element of this vocabulary has strongly distinguished meaning in particular domain, and any overlapping case is excluded. Based on this constraint we can assume that any concept in domain can be described using with keywords from domain-specific descriptor vocabulary. This gives us possibility to construct service query, which can be issued from service requestor to extract service. Before we will describe the service query syntax, let us assume that domain-specific vocabulary is accessible as the hypermedia resource.

**Domain-specific functional concepts**

In our previous work [3], we defined the functional concept as the general basis of service components. Using formal method of service-oriented functional decomposition, we showed to how can be extracted functional concepts of unknown, or partially unknown domain. General factor is service request which is formulated using domain specific functional descriptors. Our general purpose is to organize functional concepts such way, that service requestors be able to easily locate appropriate service if it exists. To formalize service request query we will use domain specific functional descriptors and concepts.

**Service Request Query**

Now, let us assume that all domain-specific functional descriptors which can be used to describe functional concepts are populated in domain-specific vocabulary. Using these descriptors service requestor can form the service request query. Service requestor can also use already defined functional concepts in query to identify the particular requirements. We, mentioned that domain specific vocabulary is accessible as the shared resource. Now, we will add another hypermedia resource, we will call them domain-specific functional concept. This resource will be serve service requestor queries to locate corresponding service. Thus, service request query can be formulated as the HTTP request. We are not going to represent final specification of this query syntax, but we will show the prototype. Let us discuss the same example of motor vehicle service.
Functional descriptors of all services, which are provided by motor vehicle service is populated in domain-specific vocabulary. To browse all of these descriptors we can send HTTP query to vocabulary resource server:

```
HTTP/1.1 GET /vocabulary
------------------------
HTTP/1.1 200 OK
Content-Type: application/json
[
      {
            “key”: “change_engine_oil”,
            “value”: “Change the engine oil”
      },
      {
            “key”: “replace_oil_filter”,
            “value: “Replace the oil filter”
```

ISSN 1512-1232

```
      },
      {
            “key”: “tune_engine”,
            “value: “Tune the engine”
      }
      . . .
]
```

This collection contains all functional descriptors which can be used to form service request query. Now, let us discuss the hypermedia resource of domain-specific functional concepts. To browse all functional concepts we have to send HTTP request to concept server:

```
HTTP/1.1 GET /concepts
-----------------------
HTTP/1.1 200 OK
Content-Type: application/json
[
      {
            “name”: “change_engine_oil”,
            “descriptors”: [“change_engine_oil”],
            “sub_concepts”: [],
            “service_ref”: “/services/change_engine_oil”,
      },
      {
            “name”: “replace_oil_filter”,
            “descriptors”: [“replace_oil_filter”],
            “sub_concepts”: [],
            “service_ref”: “/services/change_oil_filter”,
      },
      {
            “name”: “change_oil_and_filter”,
            “descriptors”: [],
            “sub_concepts”: [“change_engine_oil”, “replace_oil_filter”]
            “service_ref”: “/services/change_oil_and_filter”
      }
      . . .
]
```

In this example, we have three static functional concepts, which have corresponding service, specified in “service_ref” element. Now let us discuss the case when motor vehicle owner request to change motor oil, replace oil filter and tune the engine. We have to send corresponding HTTP request to server:

```
HTTP/1.1 GET /concepts?tags=[change_engine_oil,replace_oil_filter,tune_engine]
-------------
HTTP/1.1 200 OK
{
      “service_ref”: “/runtime-services/change_oil_and_filter_tune_engine”
}
```

This response contains the service reference which was composed in runtime using two existing service: “change_oil_and_filter” and “tune_engine”.

**Future Work**

The service query syntax is the very experimental illustration of our future work. We are going to define both domain-specific vocabulary and concept resource as the true hypermedia

ISSN 1512-1232
resource. Finally we are planning to create simple framework for flexible composition. The consumers of this framework will be humans and applications. Humans can use this framework during the designing phase.

## Conclusion

We examined that the architectural specialization, makes it easy to estimate the possibilities of reuse and c`omposition in particular situations. Main constraints in our method is that we have the few states (the data model) and many components (functions) which are modifying the state. Because the nature of such elements and constrains, it becomes easy to achieve architectural flexibility.

## References

1. Garlan D., Allen R., Ockerbloom J., Architectural Mismatch: Why Reuse Is So Hard, 1995, *IEEE Softw.* 12, 6 (November 1995), 17-26. DOI=10.1109/52.469757 http://dx.doi.org/10.1109/52.469757
2. Garlan D., Allen R., Ockerbloom J., Architectural Mismatch: Why Reuse Is Still So Hard. 2009, *IEEE Softw.* 26, 4 (July 2009), 66-69. DOI=10.1109/MS.2009.86 http://dx.doi.org/10.1109/MS.2009.86
3. Pachuashvili N., Kiviladze T., Formal Method of Service Oriented Functional Decomposition, Journal of Mathematics and System Science, 2013, 3, 4, 195-200.
4. Gamma E., Helm R., Johnson R., Vlissides J., Design Patterns: Elements of Reusable Object-Oriented Software, New York, Addison-Wesley, 1995. - P. 11-13.
5. Bracha G., Cook W., Mixin-based Inheritance, In *Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications* (OOPSLA/ECOOP '90). ACM, New York, NY, USA, 303-311. DOI=10.1145/97945.97982 http://doi.acm.org/10.1145/97945.97982
6. Shärli N., Ducasse S., Nierstrasz O., and Black A., 2003. Traits: Composable units of behavior. In Proceedings ECOOP 2003 (European Conference on Object-Oriented Programming). LNCS, vol. 2743. Springer Verlag, 248–274.
7. Odersky M., The Scala Language Specification Version 2.9, 2014, Programming Methods Laboratory, EPFL, Switzerland
8. Odersky M., Zenger M., Scalable Component Abstractions, In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (OOPSLA '05). ACM, New York, NY, USA, 41-57. DOI=10.1145/1094811.1094815 http://doi.acm.org/10.1145/1094811.1094815

**Article received: 2014-03-25**