An Ontology-Based Architecture for an Adaptable Cloud Storage Broker

Divyaa Manimaran Elango[1], Frank Fowley[1], and Claus Pahl[2(✉)]

[1] IC4, Dublin City University, Dublin, Ireland
[2] Software and Systems Engineering Research Centre, Free University of Bozen-Bolzano, Bolzano, Italy
Claus.Pahl@unibz.it

Abstract. Interoperability and easier migration between offered serviced are aims that can be supported by cloud service brokerage in the cloud service ecosystem. We present here a multi-cloud storage broker, implemented as an API. This API allows objects and collections of objects to be stored and retrieved uniformly across a range of cloud-based storage providers. This in turn realizes improved portability and easy migration of software systems between providers and services.

Our multi-cloud storage abstraction is implemented as a Java-based multi-cloud storage API and supports a range of storage providers including GoogleDrive, DropBox, Microsoft Azure and Amazon Web Services as sample service providers. We focus on the architectural aspects of the broker in this paper. The abstraction provided by the broker is based on a layered ontological framework. While many multi-cloud applications exist, we investigate in more detail the mapping of the layered ontology onto a design pattern-based organisation of the architecture. This software architecture perspective allows us to show how this satisfies important maintainability and extensibility properties for any software system.

Keywords: Cloud Service Brokerage · Cloud storage
Data migration · Ontology · API performance

1 Introduction

Interoperability is a key concern in the cloud service ecosystem. Cloud service brokerage (CSB) aims for more interoperability to enable more portability and easier migration between different service providers [26,33,34]. CSBs can support portability and migration through mechanisms such as integration and adaptation of different provided services into a uniform representation [15].

We present here a multi-cloud storage broker that implements an API to allow objects to be stored and retrieved uniformly across a range of storage providers. Two features characterise the broker. Firstly, the abstraction is based on a layered ontological framework that allows mapping of common concepts of object storage to implementation layers. Secondly, the architecture is organised

Z. Á. Mann and V. Stolz (Eds.): ESOCC 2017, CCIS 824, pp. 86–101, 2018.
https://doi.org/10.1007/978-3-319-79090-9_6

around common software design patterns to ensure maintainability and extensibility. This is important in order to extend the broker to new providers [21].

The central software architecture concepts for the design of the broker and methodology behind the abstraction library will be our core focus. The multi-cloud storage abstraction is realized by a Java-based multi-cloud storage API. Technically, the library is provided as a jar file that supports the selected four service providers, namely GoogleDrive, DropBox, Microsoft Azure and Amazon Web Services [37–40]. The library offers three service categories that reflect the different storagetypes, i.e., a file service, a blob service and a table service.

We focus on the ontology framework for the central storage concepts and functions and show how this is mapped onto a layered, design pattern-based library architecture for the API we developed [30,31]. This is an aspect that has not been sufficiently address in other investigations of multi-cloud brokers. An application of the library can also be used to compare storage operations across different providers, which we an others [16,32] have explored elsewhere.

This document is organized as follows. In Sect. 2, we give an outline of cloud service brokerage. Section 3 describes background and related work. In Sect. 4, the ontology-based interoperability framework is explained, and Sect. 5 looks at other architectural design aspects. Section 6 discusses the implementation effort and the learning outcome and presents some conclusions.

2 Principles of Cloud Service Brokerage and Use Cases

2.1 Cloud Brokerage

A cloud broker [11,17] is an intermediary software application between a client and cloud provider service. Brokerage reduces the time spent by a client in analyzing different types of services provided by different service providers.

In our case, brokerage enables a single platform to offer the client a common cloud storage service. This results in cost optimization and reduced level of back-end data management requirements, but also enables easy migration of data and files through the joint interface [8].

A multi-cloud storage abstraction API can act as the cloud broker library which facilitates the integration of different types of cloud services [18]. The abstraction library allows the broker to adapt to a rapidly changing marketplace [4]. Changeability and extensibility are consequently central requirements for our broker library [9,10]. Figure 1 illustrates the architecture.

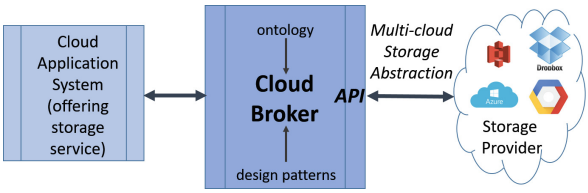

Fig. 1. Service brokerage architecture for cloud storage.

2.2 A Brokerage Use Case

Cloud brokerage shall be illustrated by a use case. Disaster recovery (DR) is a sample specific storage use case, used where there is an interruption of an action or an event in an unpredictable time that causes the services to be unavailable to the end user. Cloud back-up storage is a way of protecting the online resources to make them available in the event of a disaster, such as loss of data.

Our storage abstraction library is suited to support this DR use case as it provides a multi-cloud broker for easy storage back up. Concrete advantages are good time management in terms of restoring processes, increased scalability, security and compliance, redundancy and end to end recovery for the DR application [2,3]. The storage providers supported by our API (Microsoft Azure, Google, DropBox and Amazon Web Service) offer good bandwidth and low cost services that can be used for backup and recovery tasks.

2.3 Vendor Lock-In

Vendor lock-in is a problem in cloud computing, preventing users from migrating between providers. Clients become dependent on a single cloud provider. The client is not given an option to migrate to other providers. Issues can arise, such as legal constraints or increased costs, that consequently negatively impact on key properties by vendor lock-in and lack of standards [6,7].

A multi-cloud storage API can play a crucial role in such cases, making it easier for the client to switch providers. This can be applied across different cloud type environments, like private or public environments which are more beneficial from a business perspective. Furthermore, the extensibility of the library to support new cloud providers gives the client a wide view of portability to many different new cloud providers.

3 Background and Related Work

3.1 Cloud Service Provider APIs

Many cloud storage provider APIs exist, from which we selected four providers with different individual services [37–40]. Some key properties from a software engineering perspective that have impacted on the implementation are:

– Amazon Web Service S3: is a file storage service built on REST and SOAP. An S3 SDK is available in major development languages. The developer portal includes rich documentation. However, from a software development point-of-view, the services have a hight number of classes. The library is heavy since it has many packages for all services. Understanding the class naming can be seen as challenging – many services are listed in the same SDK documentation. We also experienced the service be inconsistent, as there was an occasional delay in read and write requests.

- Azure Storage: supports blob, file, queue and table services. The API is built on REST, HTTP and XML, and can be easily integrated with Microsoft Visual Studio, Eclipse and GIT. Azure is relatively user friendly. The standard portal interface is used for storage account set-up and document DB account parameters. The Azure SDK is available for major development languages. The Azure SDK provides a separate API package for each service and has the same code flow across different service APIs.
- DropBox: is a file hosting service. It uses SSL transfer for synchronization and AES 256 encryption as the security mechanisms. It also enables synchronised backup and web sharing. The DropBox API is lightweight and easy for a new user to go through quickly. Code samples and method explanations are given in the developer's portal.
- GoogleDrive: offers a cloud file storage service. The API is built on OAuth2 authentication. It is generally easy to understand. The structure is clearly documented and the use of method calls is well explained. The GoogleDrive service includes access to a Google API client library. Failure to include http and OAuth client libraries will disable the authentication. The Google developer portal simplifies the way of implementing the API in a workspace and provides details for configuring the authentication.

A survey of the main features of the providers that we carried out has resulted in a grouping of the cloud providers and their services as shown in Table 1.

Table 1. Storage services and their providers.

Service	Azure	AWS	Google	DropBox
File	Azure storage file	-	GoogleDrive	DropBox
Blob	Azure storage blob	AWS S3	-	-
Table	Azure storage table, Azure DocumentDB	Amazon DynamoDB, Amazon SimpleDB	-	-

3.2 Multi-cloud Libraries

For the design of our multi-cloud broker, we looked at existing multi-cloud libraries for inspiration. Cloud providers publish specifications of their services, which are different style and which makes it hard to use them as a common joint interface. We looked at several existing multi-cloud libraries, including Apache jclouds, DeltaCloud, Kloudless, SecureBlackBox, Temboo and SimpleCloud.

A key requirement was flexibility, which would allow our library to be adapted to changing services or completely different services. We decided to construct our

broker based a combination of proven design patterns, adapted to the context here. Patterns and principles from different libraries were adopted:

- In this vein, we decided to adopt an approach that was also followed in the Apache jclouds library to provide abstraction. Apache jclouds provides cloud-agnostic abstraction [27]. The principle is to use a single instance context for the mapping of a user request.
- The concept of a class for each provider across different levels of services was adopted from a similar design that we found in the SecureBlackBox library.
- The structural pattern building around a manager interface layer at each component level was adopted from the Apache LibCloud architecture.

4 The Ontological Framework for Cloud Storage

We discuss the guiding problems and principles, before introducing our storage abstraction ontology that organises the API architecture and showing how provider functionality is mapped onto this.

4.1 Abstraction, Interoperability and Extensibility

The architecture of our API is built on multiple layers of abstraction. Abstraction serves here to reduce complexity. It provides for service-neutral functional logic which also realises extensibility, i.e., allows additional vendors to be supported without changing the underlying core functional logic of the API design. In the future, new storage services can be added to the API without code change [28]. A programmable abstraction layer provides flexibility to connect and configure services [29]. Thus, interoperability and portability can be achieved. Such APIs are used for developing cloud-based applications like content delivery platforms and back-up applications, as our earlier use case demonstrates.

The main objective of the cloud storage abstraction API is to produce an effective multi-cloud delivery model, with a single portable view that supports enhanced business capabilities such as brokerage [36].

The advantage of bringing these functionalities to an interoperable multi-cloud application provides (i) an easy way of importing and exporting data, (ii) choice over price, (iii) enhanced SLA, and (iv) the elimination of vendor lock-in. While there are standardisation frameworks in this context such as the Cloud Infrastructure Management Interface (CIMI) and theOpen Cloud Computing Interface (OCCI) that target interoperability, our integration broker provides interoperability based on an extensible API.

4.2 Storage Abstraction Ontology

An ontology-based layered architecture serves to provide interoperability and extensibility. At the core of the architecture is a stroage abstraction ontology that describes the common service concepts across the abstraction layers. This ontology model consists of four main layers, namely Service, Provider, (Level-2) Composite Object and (Level-1) Core Object.

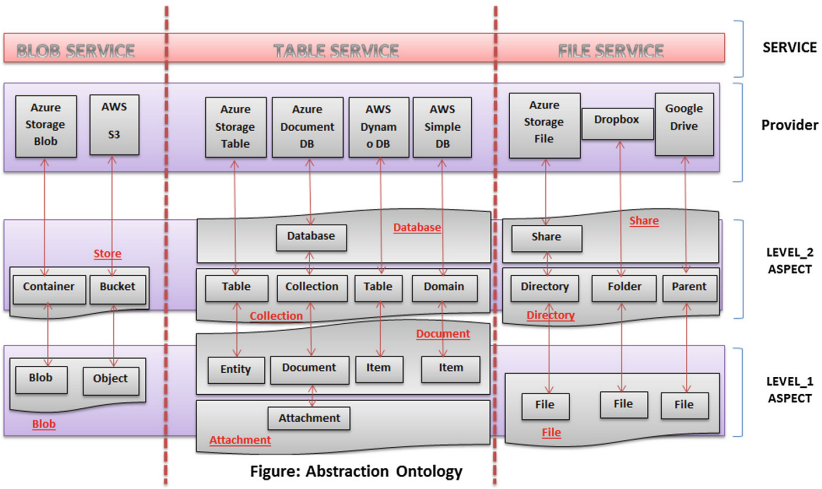

Fig. 2. Storage abstraction ontology based on 4 layers.

- Service: The Service layer is the top layer and is directly integrated into the user interface layer. This layer basically describes the services types supported by the abstraction API, which are blob, table and file service.
- Provider: The Provider layer is the next layer where context object parameters are mapped to the service layer. We supports four main providers – Microsoft Azure, Amazon Web Services, GoogleDrive and DropBox:

Service	Provider
Blob	Azure storage blob; AWS S3
Table	Azure storage Table; Azure DocumentDB; AWS DynamoDB; AWS SimpleDB
File	Azure storage file; DropBox and GoogleDrive

- Composite Objects – Level-2: The Composite Objects (object level-2) represents the first level (or higher level) of abstraction for the object types blob, table and file. It is service-neutral and established a common naming across individual providers and their specific functionalities. Each layer is abstracted based on common operations and of how the main function is applied in that particular service. Common naming allows to easily categorise storage resources and group them to simplify development.
  The Abstraction Ontology diagram (Fig. 2) shows the concept. The blob service has a "Store" concept, which groups for instance 'Container' from Azure Storage Blob and 'Bucket' from AWS S3. The table service has two different sub-layers – where "Database" belongs to Azure DocumentDB Database, and where "Collection" groups 'Table' from Azure Storage Table, 'collection' from Azure DocumentDB Collection, 'Table' from AWS DynamoDB and 'Domain' from 'AWS SimpleDB'. The file service has two sub-layers where "Share" belongs to Azure Storage File and "Directory" groups 'Directory' from Azure Storage, 'Folder' from DropBox and 'Parent' from GoogleDrive.

– Core Objects – Level-1: The Core Object (object level-1) aspect represents the lower level of storage object abstraction. This layer contains the core functionalities of a particular service across different providers.
The classes at this level are extended from an abstract class called AbstractConnector, implementing the abstractor pattern. The class implements the abstract methods defined in an AbstractConnector class. The mapping from Level-2 to Level-1 is performed by an interface class called Manager. This Manager identifies the provider class by its key. Basic CRUD operations on the storage resources arecore methods. In order to implement these functions, each operation "request" should "pass through" the Level-2 mappings and is then mapped across the service and providers.
The blob service has "Blob", which groups 'Blob' from Azure Storage Blob and 'Object' from AWS S3. The table service has two sublayers. It has an "Item" to group 'Entity' from Azure Storage Table, 'Document' from Azure, 'Item' from AWS DynamoDB and 'Item' from AWS SimpleDB. Furthermore, the second sublayer "Attachment" belongs to the Azure DocumentDB Attachment. The file service has "File" grouping 'File' from Azure Storage File, 'File' from DropBox and 'File' from GoogleDrive.

4.3 Storage Service Provider Functionality

An important concern was having common naming for mapping the user's requests onto the different services. The broker acts as an adapter for accessing different providers' services through a common interface. We found a high degree of commonality between different cloud provider functions and their names. However, some operations exists in one provider, but not in others. Furthermore, the parameters in some of the methods also differ between providers.

In the example below, the Level-2 aspect "Store" in the Blob service supports providers with different specific names, namely Azure's storage blob container and AWS's S3 bucket. The table also shows the common createStore() method and the corresponding Azure-specific and AWS-specific underlying method calls. The approach is based on identifying synonyms for common object names, such as container (Azure) and bucket (S3) for 'store':

Common name (Level-2)	Name in azure storage blob	Name in AWS S3
Store	Container	Bucket

The same then applies to function names:

Common method name	Name in azure storage blob	Name in AWS S3
createStore()	create() create container	createBucket() create bucket

The abstraction ontology maps similar service groupings together across different cloud providers. Selected services have similar or the same core functional logic – grouped into levels in the ontology. Based on this, the framework design includes the "service", "provider", "composite object" and "core object" for its implementation.

In the example below, the common level-2 composite "Collection" is already defined for two providers. There are four corresponding service names: Azure

storage table 'table', AWS document DB 'collection', AWS dynamo DB 'table' and AWS simple DB 'domain'. The table below shows the common getCollectionMetadata() method and its corresponding provider API method calls:

Common ontology name	Azure storage table	Azure Document DB	AWS Dynamo DB	AWS Simple DB
Composite: Collection	Table	Collection	Table	Domain
Operation: getCollectionMetadata()	-	-	describeTable() returns information about the table.	domainMetadata() returns information about the domain.
Operation: listCollection()	listTables() Lists the table names in the account	readCollection() Reads a document collection by the collection link	listTables() Simplified method for invoking ListTables	listDomains() Lists all domains associated with the Access Key ID

The method name to retrieve the metadata of a collection in the Table service is not supported by Azure, but AWS does. So, a common operation can not be realised across the level-2 composite Collection. A similar problem exists with the blob and file services. This lack of consistency in the available provider API operations has led to the omission of valuable API method calls. We also introduce another example, which is the common listCollection() method and its corresponding provider API method calls. The method name to retrieve the list of collections in the table service is supported by both Azure and AWS. However, the method names are different, although description and logic are the same.

5 Design of the Cloud Storage API

Our design involves a mapping of an ontology-based conceptual framework onto a layered architecture, which in turn was structured by suitable design patterns. We look at security management of the different service providers.

5.1 An Ontology-Driven Architecture

We applied a "model-driven" software engineering technique to simplify the process of design from concept modeling to implementation. This was done at each level in the abstraction ontology by breaking entities into single components. Adopting this kind of best-practice in software design was important in order to reduce the development overhead and produce a quality library that can be extended easily. Two types of modeling approaches could have been used here.

- Firstly, a provider-specific model, in which the provisioning and deployment of the abstraction library is defined for each cloud provider.
- Secondly, a cloud provider-independent model, which defines the provisioning and deployment of the abstraction library in a cloud-agnostic way.

We adopted the approach taken in the CloudML EU-funded research project. There, a domain-specific modelling language is used to reduce the complexity of cloud system design. CloudML enables to provision and deploy an abstraction library. Its design includes what we call level-1 core objects, which are assembled

based on the CloudML internal component design. These are mapped to level-2 components by using a model-driven approach. A client using the service does not necessarily know about the internal deployment, and there is no limitation on the design and evolution of the multi-cloud abstraction library.

5.2 Application of Design Pattern

Design patterns play a central role in organising the layered ontology-based architecture in order to achieve the required maintainability and extensibility, but also in general the quality of the software.

Mapping Based on an Object Context for Maintainability. For any multi-cloud library design patterns can reduce the need to have an object instantiation for each provider's class using the constructor. This was a problem noted for the jclouds library. A lack of code clarity and high level of complexity in the framework pattern was observed.

In our API, in order to avoid this problem and to provide a stable, maintainable code base, the context builder class is added to the architecture. This builder class includes a key and a value parameter pair. This pair is called an item, which adds the service, the provider, the aspect key, the operation key and the input parameters to the context. Then, this context object is passed on to execute the API method call. This mapping is applied for all the services supported by our API. Below, we outline the mapping of parameters into a single context instance.

```
Context context = new Context();
context = addServiceContext(context);
context = addServiceProviderContext(context);
context = BlobService.addParameters(
              IConstants.ASPECT\_KEY, IConstants LEVEL-2\_STORE, context);
context.addItem(new Item(
              IConstants.OPERATION\_KEY, IConstants.OPERATION\_CREATE));
context = BlobService.addParameters(
              IConstants.STORE\_NAME, storeName, context);
```

Extensibility Through a Plug-in Framework. API design principles state that a developer should not have visibility of the underlying low-level abstraction classes, interfaces and methods. If a future extension can support new features and services, then the framework should not have to be redesigned or its behavior changed. We say that the framework acts like a plug-in for any new features, services or providers. We achieve this in the design by enforcing that an abstract class cannot be instantiated, it can only be inherited, as a strict coding rule.

The level-1 layer, which implements the lower-level API methods, is extended from the AbstractConnector class. This abstract class must implement the interface IConnector and all of its associated methods. The reason for this is because an abstract class, by definition, is required to create subclasses of its instance. The subclasses are required by the compiler to implement any interface methods

that the abstract class has left unimplemented. Less effort is required to extend the API because the framework itself remains unchanged in that case.

Multi-service Support Through a Manager Interface Layer. The level-2 layer is limited because of the separation of the user level request which is meant to distinguish between different API methods provided by the same provider. For example, AWS provides DynamoDB and SimpleDB. Similarly, Azure provides storage table and DocumentDB services. In order to remedy this, an interface component called manager is implemented. The manager is responsible for identifying the corresponding "aspect-key" that is encapsulated within the context parameter discussed earlier on.

We use two types of managers: the store manager and the table manager.

- The composite object Level-2 aspect is the higher level of abstraction. Since Level-2 helps to identify the differences between the services, the store manager interface is added in this layer, which splits the request to either blob or file or table service at core object level-1.
- The table manager is used in a similar way. For example, the table service at Level-1 has two APIs, the DynamoDB and SimpleDB, supported by one provider, AWS. In order to differentiate between the services, an interface component called manager was added to identify the common method name.

The relationship between the abstract class and the core logic of the level-2 aspect is managed using the context parameter.

5.3 Apache jclouds and Design Patterns

Our multi-cloud storage abstraction layer was designed using some of the design concepts and patterns of jclouds. Apache jclouds is an open source library available in Java and Clojure, which supports several major cloud providers. The jclouds library offers both a portable abstraction framework as well as cloud-specific features. The main aim of jclouds is to manage errors, concurrency and cloud complexity better.

The jclouds Architecture. jclouds features of a portable abstraction layer called 'View', responsible for splitting the service type and cloud provider. A 'View' is connected to a provider-specific API or library driven API. The Context Builder class maps the context object along with its parameters. The parameters include provider class object, view, API metadata and provider metadata. This object will be bound as a *singleton object* called Context and it is passed to the context builder. The API Metadata class populates friendly names for the key, which has two values – the type and the view information. The Service Registry acts like a manager, which is responsible for holding the key to connect to a provider's class. The framework implements a *builder pattern* for request and response, which connects to a backend API, along with authentication.

In the context of our broker, the jclouds library caters for blob and compute services. The following code block outlines the jclouds library code for calling a context for an Azure blob. It uses the context builder class. The basic concept of abstraction used in the jclouds library is based on the builder design pattern known from software engineering. A context with service provider Azure that offers the portable BlobStore API would look like as follows:

```
BlobStoreContext context =
    ContextBuilder.newBuilder("azureblob")
   .credentials(storageAccountName, storageAccountKey)
   .buildView(BlobStoreContext.class);
```

5.4 Security Analysis – Authentication Mechanism

Security is another concern that needs to be unified across the providers in addition to the mapping of concepts for core and composite storage objects used by the different providers into a common ontology. Authentication, however, differs across the providers selected.

1. The authentication in *GoogleDrive* is based on a client_secret json file. A project is created in the Google developer's console. The Drive API and OAuth protocol is enabled. The credentials are generated and saved as a client_secret json file. In the coding, the authentication method should have the permission scope and drive scope set to 'GRANT'. When the browser opens for the authentication response, the client is permitted full read and write access.
2. The authentication in *DropBox* uses an access token, which is generated in the console. An application is created under the app console and its permission is set to 'FULL'. Later, the authentication is set by linking the account using the access token when passing the instance of the API client.
3. The authentication in *Microsoft Azure* is based on an account subscription that allows for the accessing of the resources available within an azure account. The Blob service is provided within an Azure storage account. The azure storage account name, also known as namespace, is the first level for processing authentication to the services within the storage account (blob, file and table). It uses token-based authentication. The authentication of the Azure storage blob is done using a connection string which has the parameters of the storage account name and primary key. Similarly, the Table service is supported within an Azure storage account. The authentication of the Azure storage table is done using a connection string which has the parameters of the storage account name and primary key. An Azure Document DB account is required for accessing the Azure Document Db service and requires a master key and URI (end-point).
4. The authentication in *Amazon Web Services* is based on a secret key and an access key, which is common across all services supported by our storage API. An AWS user should have a specified role with the required resource access permissions. Identity Access Management allows the user to set the role and access privileges, and this provides each user with sufficient credentials.

The account can be activated using phone verification and authentication. Credential auditing and usage reports can be used for review purposes.

The different authentication processes were considered for the broker authentication method, i.e., calls from the multi-cloud storage abstraction API. Our solution is based on a credentials object:

- Credentials storage: The credentials are stored in a common config file. So, the user is not shown the authentication part as it happens in the back-end.
- Credentials update: If the credentials need to be changed, they are only changed in the configuration file, which reduces overhead for the user to set up the authentication process.

6 Discussion and Conclusions

We have discussed maintainability and extensibility as centralobjectives that need to be evaluated here throughout the architecture discussion in the previous section. In a summarising discussion, we return here to a few important concerns such as establishing testability and maintainability through suitable design patterns, e.g., the dependency injection pattern, to point out benefits. Other aspects have already been discussed throughout Sect. 5 above.

Design Patterns and Software Quality. From the discussion above, specifically the code, it can be clearly seen that jclouds gets a separate instance for each provider's class and, in some cases, it makes direct REST calls to the underlying provider API. Thus, the programming style in jclouds follows the *dependency injection software design pattern*. It uses two *programming frameworks*: firstly, Google Guice, which is a Google library alternative to Spring, and, secondly, Guava, which supports transformation, concatenation and aggregation for storage services.

Another quality concern shall be addressed: recompilation overhead. Dependency injection avoids code duplication, is unit-testable and modular. It thus allows injecting of the service class instead of calling the API service method, achieved by writing custom code and connecting it at run time, which avoids recompiling. Custom code instantiates an object for each service and provider.

Quality Factors – Testability and Extensibility. Testing is a further concern. The jclouds library uses dependency injection, as discussed above, which makes reference to an object before it will proceed for execution. Implementing dependencies by constructors, using the 'new' constructor may result in difficulties for unit testing. Performing dependency injection using a *factory method* is a traditional solution to the testing concern.

This is also known as indirect dependency, where the factory method is realised by having an interaction class between the client class and the service class. It was considered that the use of too many interaction classes would make

the code more complex and result in tight coupling between the abstraction layers. This would hide the definition of abstraction and furthermore, it would not facilitate the future extension of the library.

According to the principles of API design, there should be a small number of functionalities shared across the entire cloud provider API. This has been achieved in the abstraction design used.

Final Comments. The aim of cloud service brokerage is customising or integrating existing services or making them interoperable. Following the classification schemes in [12,13], we have developed an integration broker:

- the main purpose is intermediation between cloud consumers and providers to provide advanced capabilities (interoperability and portability),
- it builds up on an intermediary/broker platform to provide a marketplace to bring providers and customers together,
- the broker system type is a multi-cloud API library.

We presented here on a broker solution [1] for cloud storage service providers to implement a joint interface to allow

- portability and migration for the user, i.e., the consumer of the services,
- extensibility for the broker provider to allow changed or new services to be included.

Our broker enables through its joint API also the opportunity for a cloud storage user to easily migrate between services or to use multiple services at the same time, depending on prefered characteristics such as security or performance [5].

Many broker implementations and multi-cloud APIs exist. We provide a novel view by focussing here on the construction of a broker API and looking at software architecture principles behind it. Again, ontologies have been used before, but we demonstrate here how a layered ontology and a corresponding layered architecture together with the use of appropriate design patterns can better help to achieve extensibility and efficiency of the implementation. The selection of design patterns has a significant impact on the testability, maintainability and extensibility of the layered architecture that we have developed.

As future work, we plan to extend the broker by adding further services by other providers to empirically verify the extensibility of the library. While our API-based architecture only supports public cloud providers, this can be extended to include private clouds in future. A more long-term usage beyond some performance testing on the provider services, should also help to better judge the maintainability in addition to the expected positive affect from the pattern application. More work could also go into more uniform specification of cloud services [25,35] aiming at more standardisation of the interfaces.

Acknowledgements. This work was partly supported by IC4 (Irish Centre for Cloud Computing and Commerce), funded by EI and the IDA.

References

1. Ried, S.: Cloud Broker – A New Business Model Paradigm. Forrester, Cambridge (2011)
2. Benslimane, D., Dustdar, S., Sheth, A.: Services mashups – the new generation of web applications. Internet Comput. **12**(5), 13–15 (2008)
3. Bernstein, D., Ludvigson, E., Sankar, K., Diamond, S., Morrow, M.: Blueprint for the inter-cloud: protocols and formats for cloud computing interoperability. In: International Conference on Internet and Web Applications and Services (2009)
4. Buyya, R., Ranjan, R., Calheiros, R.N.: InterCloud: utility-oriented federation of cloud computing environments for scaling of application services. In: Hsu, C.-H., Yang, L.T., Park, J.H., Yeo, S.-S. (eds.) ICA3PP 2010. LNCS, vol. 6081, pp. 13–31. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13119-6_2
5. Elango, D.M., Fowley, F., Pahl, C.: Testing and comparing the performance of cloud service providers using a service broker architecture. In: Mann, Z.Á., Stolz, V. (eds.) ESOCC 2017. CCIS, vol. 824, pp. 117–129. Springer, Cham (2018)
6. Cloud Standards (2017). http://cloud-standards.org/
7. ETSI Cloud Standards (2017). http://www.etsi.org/newsevents/news/734-2013-12-press-release-report-on-cloudcomputing-standards
8. Fehling, C., Mietzner, R.: Composite as a service: cloud application structures, provisioning, and management. Info. Technol. **53**(4), 188–194 (2011)
9. Pahl, C., Jamshidi, P., Weyns, D.: Cloud architecture continuity: change models and change rules for sustainable cloud software architectures. J. Softw. Evol. Process **29**, e1849 (2017). https://doi.org/10.1002/smr.1849
10. Pahl, C., Jamshidi, P., Zimmermann, O.: Architectural principles for cloud software. ACM Trans. Internet Technol. (2018, to appear)
11. Forrester Research: Cloud Brokers Will Reshape The Cloud (2012). http://www.cordys.com/ufc/file2/cordyscmssites/download/09b57cd3eb6474f1fda1cfd62ddf094d/pu/
12. Fowley, F., Pahl, C., Zhang, L.: A comparison framework and review of service brokerage solutions for cloud architectures. In: Lomuscio, A.R., Nepal, S., Patrizi, F., Benatallah, B., Brandić, I. (eds.) ICSOC 2013. LNCS, vol. 8377, pp. 137–149. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06859-6_13
13. Fowley, F., Pahl, C., Jamshidi, P., Fang, D., Liu, X.: A classification and comparison framework for cloud service brokerage architectures. IEEE Trans. Cloud Comput. (2017). https://doi.org/10.1109/TCC.2016.2537333. http://ieeexplore.ieee.org/document/7423741/
14. Javed, M., Abgaz, Y.M., Pahl, C.: Ontology change management and identification of change patterns. J. Data Semant. **2**(2–3), 119–143 (2013)
15. Garcia-Gomez, S., et al.: Challenges for the comprehensive management of cloud services in a PaaS framework. Scalable Comput. Pract. Exp. **13**(3), 201–214 (2012)
16. Elango, D.M., Fowley, F., Pahl, C.: Using a cloud broker API to evaluate cloud service provider performance. Research report 471, Department of Informatics, University of Oslo, pp. 63–74 (2017)
17. Gartner: Cloud Services Brokerage. Gartner Research (2013). http://www.gartner.com/it-glossary/cloud-servicesbrokerage-csb
18. Grozev, N., Buyya, R.: InterCloud architectures and application brokering: taxonomy and survey. Softw. Pract. Exp. **44**, 369–390 (2012)
19. Pahl, C., Jamshidi, P.: Microservices: a systematic mapping study. In: Proceedings CLOSER Conference, pp. 137–146 (2016)

20. Taibi, D., Lenarduzzi, V., Pahl, C.: Processes, motivations and issues for migrating to microservices architectures: an empirical investigation. IEEE Cloud Comput. **4**(5), 22–32 (2018). http://ieeexplore.ieee.org/document/8125558/
21. Hofer, C.N., Karagiannis, G.: Cloud computing services: taxonomy and comparison. J. Internet Serv. Appl. **2**(2), 81–94 (2011)
22. Jamshidi, P., Sharifloo, A., Pahl, C., Arabnejad, H., Metzger, A., Estrada, G.: Fuzzy self-learning controllers for elasticity management in dynamic cloud architectures. In: 12th International ACM SIGSOFT Conference on Quality of Software Architectures QoSA (2016)
23. Arabnejad, H., Jamshidi, P., Estrada, G., El Ioini, N., Pahl, C.: An auto-scaling cloud controller using fuzzy Q-learning - implementation in openstack. In: Aiello, M., Johnsen, E.B., Dustdar, S., Georgievski, I. (eds.) ESOCC 2016. LNCS, vol. 9846, pp. 152–167. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-44482-6_10
24. Gacitua-Decar, V., Pahl, C.: Structural process pattern matching based on graph morphism detection. Int. J. Softw. Eng. Knowl. Eng. **27**(2), 153–189 (2017)
25. IEEE Cloud Standards (2015). http://cloudcomputing.ieee.org/standards
26. Jamshidi, P., Ahmad, A., Pahl, C.: Cloud migration research: a systematic review. IEEE Trans. Cloud Comput. **1**, 142–157 (2013)
27. jclouds: jclouds Java and Clojure Cloud API (2015). http://www.jclouds.org/
28. Ferrer, A.J., et al.: OPTIMIS: a holistic approach to cloud service provisioning. Future Gener. Comput. Syst. **28**(1), 66–77 (2012)
29. Konstantinou, A.V., Eilam, T., Kalantar, M., Totok, A.A., Arnold, W., Sniblel, E.: An architecture for virtual solution composition and deployment in infrastructure clouds. In: International Workshop on Virtualization Technologies in Distributed Computing (2009)
30. Pahl, C.: Layered ontological modelling for web service-oriented model-driven architecture. In: Hartman, A., Kreische, D. (eds.) ECMDA-FA 2005. LNCS, vol. 3748, pp. 88–102. Springer, Heidelberg (2005). https://doi.org/10.1007/11581741_8
31. Pahl, C., Giesecke, S., Hasselbring, W.: Ontology-based modelling of architectural styles. Inf. Softw. Technol. **51**(12), 1739–1749 (2009)
32. Mietzner, R., Leymann, F., Papazoglou, M.: Defining composite configurable SaaS application packages using SCA Variability Descriptors and Multi-tenancy Patterns. In: International Conference on Internet and Web Applications and Services (2008)
33. Pahl, C., Xiong, H.: Migration to PaaS clouds - migration process and architectural concerns. In: IEEE 7th International Symposium on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems MESOCA (2013)
34. Pahl, C., Xiong, H., Walshe, R.: A comparison of on-premise to cloud migration approaches. In: Lau, K.-K., Lamersdorf, W., Pimentel, E. (eds.) ESOCC 2013. LNCS, vol. 8135, pp. 212–226. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40651-5_18
35. Papazoglou, M.P., van den Heuvel, W.J.: Blueprinting the cloud. IEEE Internet Comput. **15**, 74–79 (2011)
36. Petcu, D., et al.: Portable cloud applications–from theory to practice. Future Gener. Comput. Syst. **29**(6), 1417–1430 (2013)
37. Amazon Simple Storage Service (S3) Cloud Storage AWS https://aws.amazon.com/s3/
38. Dropbox. https://www.dropbox.com/
39. Azure Storage - Secure cloud storage. https://azure.microsoft.com/en-us/services/storage/

40. Google Drive - Cloud Storage & File Backup. https://www.google.com/drive/
41. Jamshidi, P., Pahl, C., Mendonca, N.C.: Pattern-based multi-cloud architecture migration. Softw. Pract. Exp. **47**(9), 1159–1184 (2017)
42. Pahl, C., Brogi, A., Soldani, J., Jamshidi, P.: Cloud container technologies: a state-of-the-art review. IEEE Trans. Cloud Comput. (2017). https://doi.org/10.1109/TCC.2017.2702586. http://ieeexplore.ieee.org/document/7922500/
43. Aderaldo, C.M., Mendonca, N.C., Pahl, C., Jamshidi, P.: Benchmark requirements for microservices architecture research. In: 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering. IEEE (2017)
44. Heinrich, R., van Hoorn, A., Knoche, H., Li, F., Lwakatare, L.E., Pahl, C., Schulte, S., Wettinger, J.: Performance engineering for microservices: research challenges and directions. In: Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion (2017)