BFH (Bern University of Applied Sciences), CH-2501 Biel, Switzerland

UniBoard Architecture Specification

Version 0.4

Eric Dubuis, Severin Hauser

April 24, 2014

Revision History

Revision	Date	Author(s)	Description
0.1	August 13, 2013	Eric Dubuis	Initial draft.
0.2	September 10, 2013	Eric Dubuis	Introduced chain of components.
0.3	February 17, 2014	Eric Dubuis	..., removed typos.
0.4	April 24, 2014	Severin Hauser	Made changes so that it corresponds to the formal spec.

Contents

1 Introduction

This document presents the architectural specification of UniBoard, a (rather) universal public bulletin board. Specific to this architecture specification is that it defines the *standalone*, monolithic as well as the *distributed* variant of the public bulletin board.

To achieve this, the following architectural goals have been in mind:

- The same specification for the architecture of the *standalone* and the *distributed* variant of UniBoard.
- Establishment of a *component composition* in order to achieve the desired functionality.
- Plug-in facility to switch from the standalone variant to the distributed one, and vice versa.

Note that the supported properties of UniBoard depend of the combination of components used.

2 Component Composition

In (Ref. to be provided), several properties and the basic operations of a public bulletin board have been identified. We present in this document the specification of a software architecture allowing to implement these properties by *composition* of respective building blocks, the so-called *components*. For every operation a *chain of components* is used as the pattern for the composition.

UniBoard consist of one node (for the standalone variant) or a set of nodes (for the distributed variant). For the distributed variant, a specific component is inserted into the chain of components. That component implements the respective agreement protocol by communicating with peer nodes.

Each component implements at least one of the generic service interfaces. This allows the desired chains of components. (Of course, not every arbitrary combination of components results in a useful configuration.) The final component in this chain is a persistence service allowing to store information, and to retrieve it later. In fact, our architecture follows the *chain of responsibility pattern* [**?**].

2.1 Chain of Components

In order to achieve upmost flexibility, the two basic operations of UniBoard are represented with *generic* service interfaces, simply called *GetService* and *PostService* (namespaces in text omitted for brevity). That is, each component implements either one or both interfaces.

The chains of components (Get-Chain and Post-Chain) are achieved by linking a component to the next one, which in turn is linked to another component. Only the last component in the chain, the *PersistenceService*, is not linked to another component. Figure 2.1 shows a possible structure of the chains.

The service interfaces consists each of one general-purpose method:

```
1 package ch.bfh.uniboard.service;
2
3 public interface GetService {
4     public ResultContainer get(Query query);
5 }
```

```
1 package ch.bfh.uniboard.service;
2
3 public interface PostService {
4     public Attributes post(byte[] message, Attributes alpha,
          Attributes beta);
5 }
```

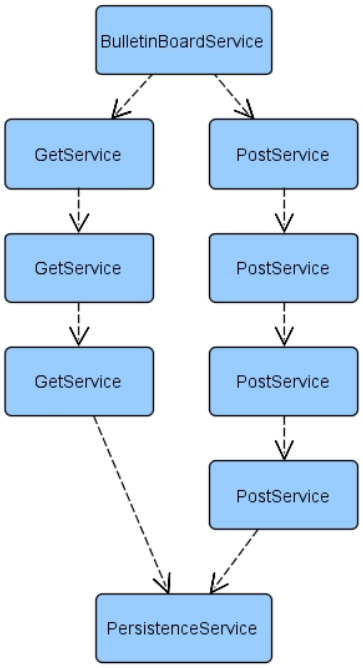

Figure 2.1: Simple setup of the the chains in UniBoard

The provided and returned data containers *Attributes*, *Query*, and *ResultContainer* are kept as generic as possible, so that to ensure a good flexibility for the components. The introduction of generic data containers also minimizes the coupling: Components depend on the core classes only, but not on specific classes of other components.

The meaning of the data abstractions can be summarized as follows:

- *Attributes:* List of attributes, where each attribute is computed by the user or a component. These attributes belong either to a message or result.
- *Query:* Data that originates from an user for a *get* operation. The *Query* defines the set of posts that are returned as the result.
- *ResultContainer:* Consists of two parts. First there is a set of posts that satisfies the *Query*, and there are *Attributes* for this result. Every component may add an attribute. The result is set by the final component, the persistence store.

2.2 Service Request Delegation

At start-up time, a board is configured by a number of concrete components arranged in a chain of component instances. To service a client, there is at first a *BulletinBoardService*. This service provides the interface for the client. It forwards the request to the first component, which passes it to the next component. It may also process the service request before passing it to the next component. Consequently, it may also process the response before returning it to the *BulletinBoardService*.

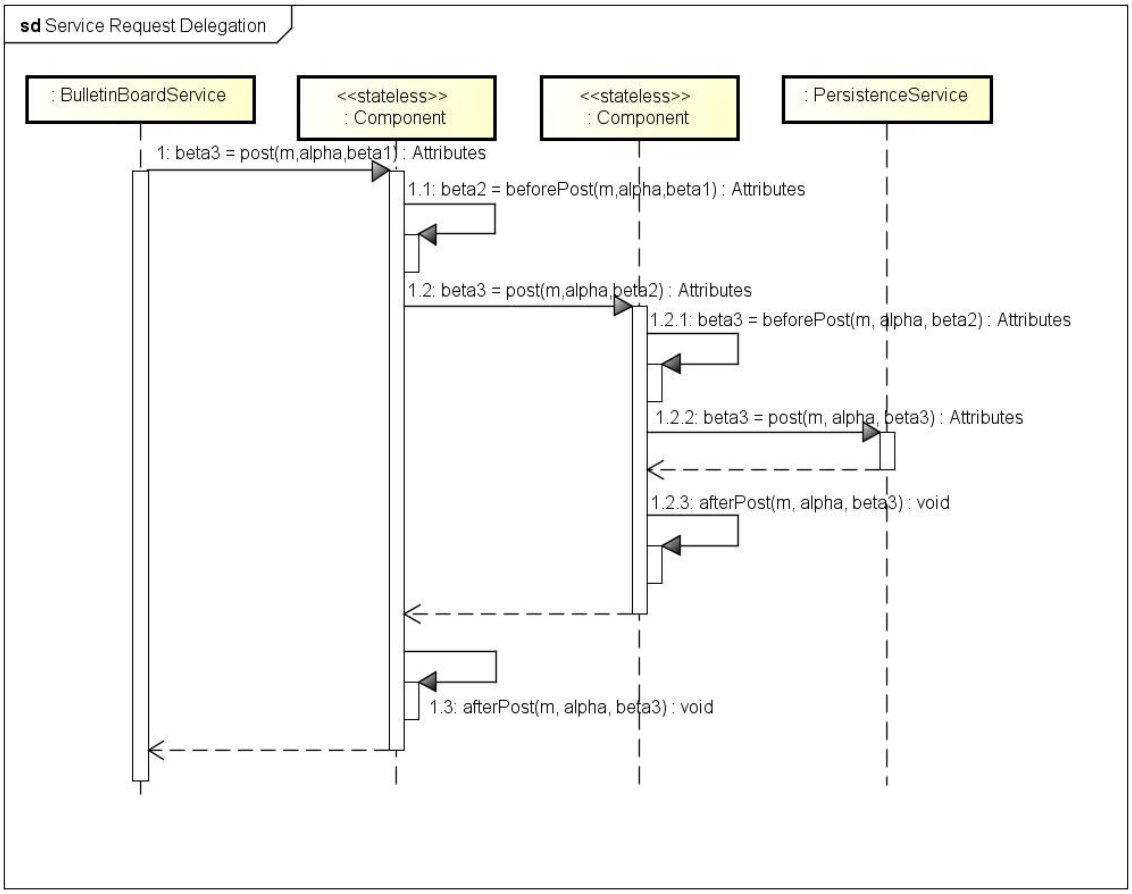

Figure 2.2: UML sequence diagram illustrating the dynamics of the delegation of a service request. Each *post()* request received at a component is processed at first internally by the self-delegated method *beforePost()*, and then passed to the next downstream component. The response of the downstream component is processed at first by the self-delegated method *afterPost()*, and then returned to the calling component (or the application).

The above described behavioral pattern is repeated with every intermediate component. Notice that *synchronous* calls are used between the components. However, asynchronous processing may occur within a component, which will likely be the case for the distributed variant of the board.

2.3 Handling Service Requests

A *BulletinBoardService* or a component requests a service provided by the downstream component. Basically, the following three processing actions will occur in the downstream component:

1. The received request is processed.
2. The processed request is passed to the downstream component.

3. The result of the downstream component is processed and returned to the calling component (or the application).

The comments of Figure **??** sketch the above described processing actions. The template [**?**] of this behavior is predefined in the abstract class Component:

```
package ch.bfh.uniboard.service;

public abstract class Component implements Service {
    private Service successor;

  public final Attributes post(byte[] message, Attributes alpha,
      Attributes beta) {
    Attributes beforePost = this.beforePost(message, alpha, beta);
    ...
    Attributes betaReceived = this.getSuccessor().post(message,
       alpha, beforePost);
    this.afterPost(message, alpha, betaReceived);
    return betaReceived;
  }
    public finalResultContainer get(Query query) {
        // analogous to method post() above
    }
    protected Attributes beforePost(byte[] message, Attributes
       alpha, Attributes beta) {
        return beta;
    }
    protected void afterPost(byte[] message, Attributes alpha,
       Attributes beta) {
    }
    // analogous for methods beforeGet() and afterGet()
}
```

Subclasses needing to process a message before passing it to the downstream component merely overwrite method *beforePost()*:

```
package lu.unilu.uniboard.component_x;

import ch.bfh.uniboard.service.*;

public class Component_X extends Component {
    ...
    protected Attributes beforePost(byte[] message, Attributes
       alpha, Attributes beta) {
        // process given message and attributes
        // (or pass back modified attributes object, not shown)
        return beta;
    }
}
```

Similarly, subclasses overwrite methods *afterPost()*, *beforeGet()*, and *afterGet()* in order to implement the desired behavior.

2.4 Generic Data Container

The generic data containers are not only used to transmit data from component to component, but also for the communication with the users of UniBoard.

2.4.1 Attributes

Attributes is basically

2.4.2 Query

2.4.3 ResultContainer

3 Specific Services

In order to implement the desired properties of the public bulletin board, a number of concrete services are required:

- *bfh-webservice* Provides an interface to UniBoard over WebServices.
- *Property Service* This service provides properties required by the bulletin board. Its realization is composed of a series of property implementation, all adhering to the very same *Service* interface, exploiting the pluggable layered architecture.
- *Persistence Service* This service provides the view to the persistence storage of the postings sent to UniBoard.

Some details of the *fault tolerance service* will be described in the next chapter. Course views for a *property service* and the *persistence service* are given in the next sections.

TODO: Provide details on accessing the bulletin board.

3.1 bfh-webservice

The module bfh-webservice offers the basic operations introduced in (do Ref) as a webservice.

- $\text{Post}(m, \alpha) : \beta$
- $\text{Get}(Q) : R, \gamma$

3.2 The Persistence Service

The persistence service provides operations to store and retrieve data. It is a component in the sense that it implements the *Service* interface. However, it does not have any downstream component and, thus, cannot be derived from the *Component* abstraction.

Figure 3.1 shows the structure of the persistence service component.

A post is always constituted of three elements: the *message*, the *alpha attributes* that are post attributes defined by the poster and *beta attributes*, which are attributes defined by the board. The persistence service has to store these three elements.

3.2.1 Storing the message

The persistence service uses MongoDB as database. MongoDB is a JSON document based database. This means that a post must be transformed in a JSON document previous to saving it. This allows to store a JSON structured message in a post without having the board knowing the structure of the message in advance. A message must be given in the form of a byte array, this allowing the poster to store everything in the message. If the poster wants to store a JSON structure as message of the post, the poster has to convert the JSON

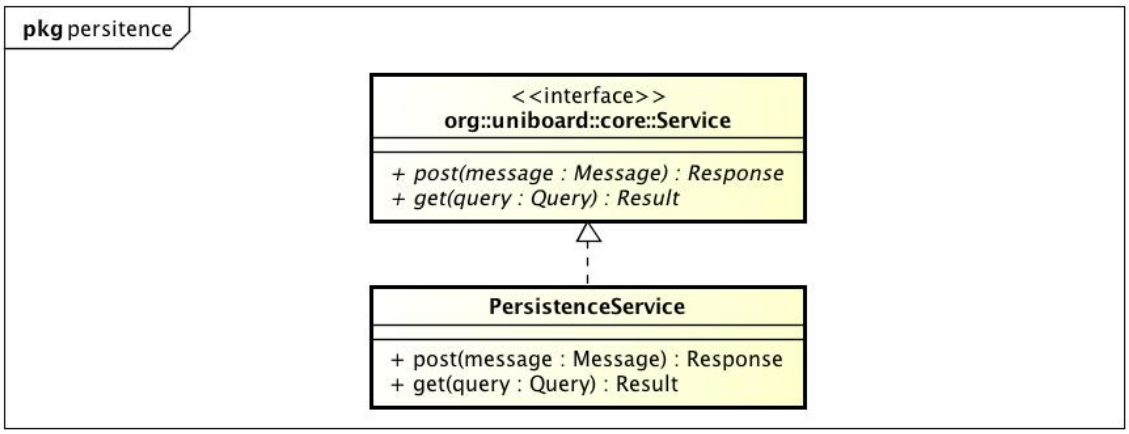

Figure 3.1: UML class diagram for the illustration of the persistence service component. Notice that it merely implements the *GetService*, *PostService* and *InternalGet* interfaces, and that it is not derived from the *Component* class.

string to a byte array using UTF-8 encoding, before passing it to the persistence service. When the persistence service receives a post to store, it tries to interpret the message as a JSON string. If it successes, it stores the message in that format, otherwise, it stores it as byte array.
MongoDB then allows to retrieve a post giving a JSON key and the value it should have. So, a user can retrieve a post by indicating a key used in the message of a post. Therefore, the user must know the structure of the JSON message, but the board doesn't.

3.2.2 Storing the attributes

The attributes *alpha* and *beta* are of type *Attributes* which is a simple class containing a map storing the attributes as key-value pair. Using MondoDB for storing these attributes offers another advantage: the database can accept an undefined number of attributes since they are also stored as JSON elements.

The key of an attributes must be a string. The value however can have different type. An interface called *Value* defines the abstract type of the value, its concrete implementations define the concrete type allowed for an attributes. They are: *StringValue*, *IntegerValue*, *DoubleValue*, *DateValue*, *ByteArrayValue* as shown in Figure **??**. All these types are natively supported by MongoDB for storing and querying.

//TODO class diagrams

3.2.3 Querying a post

To retrieve a post, a so-called *Query* object must be submitted to the persistence service. A *Query* is composed of list of *Constraint* objects. These constraints allow to specify conditions that a post must fulfill in order to be returned as result for this query. Different types of constraints are supported as show in Figure **??**. Each concrete constraint contains one key to identify which part (for example which attribute) of the post must fulfill the condition, at least one value and a *PostElement* indicator, which specifies whether the key the user is interested in is located in the message, the alpha attributes or the beta attributes. This

last element is important since the query strategy of MongoDB requires to indicate the structure of the key searched. For example, if the value we are interested in is identified by the key *subsubkey3*, it is necessary to indicate the path to access to this key (for example: *key.subkey2.subsubkey3*. Since, when storing the post, a different key is used to separate the message, the alpha attributes and the beta attributes, the persistence service must know in which of this element the indicated key should be found. This information is given through the *PostElement*. The structure of keys can be passed as list of strings, the first element being considered as the root.

//TODO class diagrams

The value searched must be a subtype of *Value*. The constraints shown in Figure **??** are applicable for all these subtypes. *Greater*, *GreaterEquals*, *Less*, *LessEquals* and *Between* consider the alphabetical order on *StringValue* type and the binary value on *ByteArray*.

Multiple constraint can be combined with the *and* operator. This is done by inserting multiple constraints in the list that is passed at creation of the *Query* object.

4 Distributed Bulletin Board

This chapters explains how a bulletin board is built as a distributed system. The distributed system consists of a set of nodes. Each node is assumed to be configured accordingly.

The architectural model also assumes that a message to be published is sent to one (trusted) node. (If the application does not trust a given node then it can choose among the other ones.) The message then is processed by the Byzantine fault tolerant client component *BFTClient*. The BFT client component in turn may use downstream components, and it will have to use peers, the so called Byzantine fault tolerant replica components of sort *BFTReplica*.

4.1 BFT Client Component

In its first view, a *BFTClient* acts as a component. That is, it subclasses the generic component class *Component*. As such, it may use the service of the successor (local) component. However, it will also initiate an agreement protocol with peer replica components of sort *BFTReplica*. Figure 4.1 shows the static structure of the Byzantine fault tolerant client component.

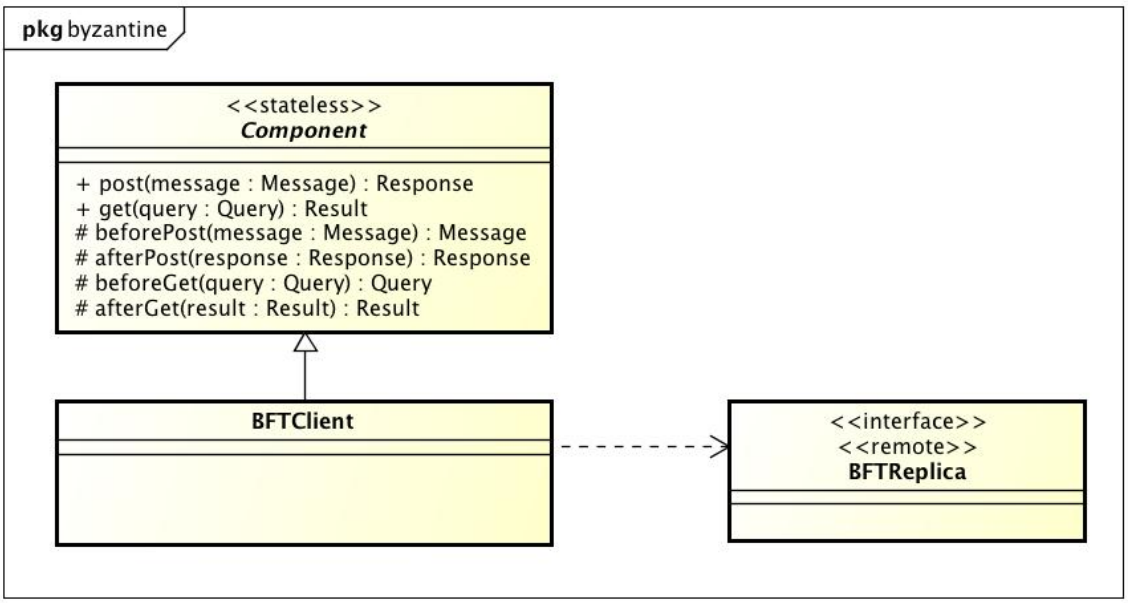

Figure 4.1: UML class diagram illustrating the client side of the replication component. Class *BFTClient* extends the *Component* class. In addition, it uses the remote service accessible via the *BFTReplica* interface.

4.2 BFT Replica Component

Primarily, a Byzantine fault tolerant replica component of sort *BFTReplica* extends the generic *Component* implementation. In addition, it also implements the *BFTReplica* interface. Figure 4.2 shows the static structure of the Byzantine fault tolerant replica component.

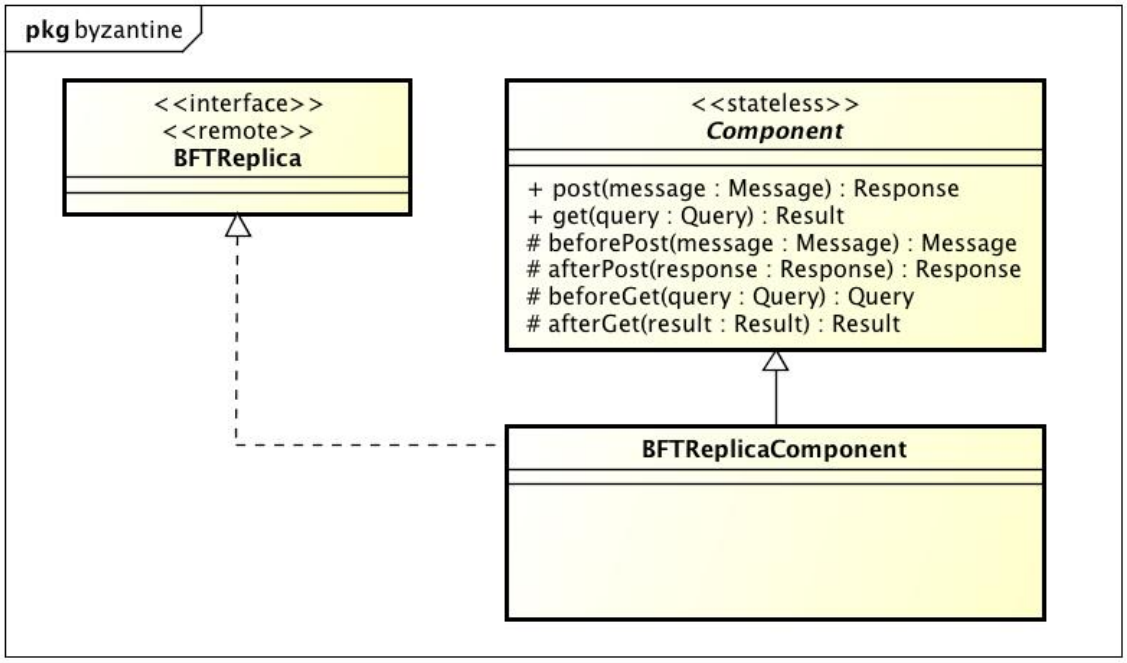

Figure 4.2: UML class diagram illustrating the replication side of the replica component. Class *BFTReplicaComponent* extends the *Component* class (and, thus, is a *Service*). In addition, it implements the *BFTReplica* interface providing specific services to peer replica clients.

The communication between the BFT client and the BFT replicas is an implementation detail and, therefore, not discussed here. (It may involve yet another layered architecture and specialized protocols, and it will use whatever communication technology is appropriate.)

4.3 Deployment View of the Replicated Board

For the replicated configuration, each node is identically configured with a set of required *Component* instances, and with the BFT client and BFT replica component instances.

5 Implementation Hints

Some implementation hints are given here.

5.1 Namespaces, Names

> *Note.* The namespace *org.uniboard* is already used in the domain name system of the Internet. Thus, we will have to discuss its use here.

In order to ease the setup of the code base (project structure, Maven artifacts, names to be defined), certain namespaces must be fixed. The following namespace for the core service abstraction shall be used:

```
package org.uniboard.core;

public interface Service ...
public abstract class Component ...
public abstract class Request ...
public abstract class Response ...
public abstract class Query ...
public abstract class Result ...
```

Implementations will use additional namespaces such as *lu.unilu.uniboard.component1*, *lu.unilu.uniboard.application1*, *ch.bfh.uniboard.application2*, or *ch.bfh.uniboard.component2*.

Class names can also be concretized by following respective technology conventions. When using EJB, the convention for session bean classes is to use the suffix *Bean*.

5.2 Access to the Downstream Component

In order to achieve upper-most flexibility, is is mandatory to access the next downstream component by using the *generic* service interface. For example:

```
package org.uniboard.core;

import javax.ejb.EJB;
import javax.ejb.LocalBean;
import javax.ejb.Stateless;

@Stateless
@LocalBean
public abstract class ComponentBean implements Service {

    @EJB
    private Service successor; // injected by the EJB container

    public Response post(Message message) ... {
        Message beforePost = beforePost(message);
        Response response = successor.post(beforePost);
        Response afterPost = afterPost(response);
```

```
        return afterPost;
    }
    ...
    protected Message beforePost(Message message) ... {
        return message;
    }
    protected Response afterPost(Response response) ... {
        return result;
    }
}
```

Notice: By giving the component class the name *ComponentBean* we follow the propose name convention of EJB (TODO: Ref. to be provided).

The injection performed by the container can be configured at deploy-time. If EAR deployment units are used then the original deployment descriptors or annotations can be overridden upon configuring the EAR deployment unit.

5.3 Skeleton for a Concrete Component

The skeleton of a concrete component looks like:

```
package ch.bfh.uniboard.authorization;

import org.uniboard.core.Service;
import org.uniboard.core.ComponentBean;

@Stateless
@LocalBean
public class AuthorizationServiceBean extends ComponentBean {
    // overwrite beforePost(), afterPost() accordingly
    // overwrite beforeGet(), afterGet() accordingly
}
```

Another example:

```
package lu.unilu.uniboard.byzantine;

import org.uniboard.core.Service;
import org.uniboard.core.ComponentBean;

@Stateless
@LocalBean
public class BFTClientBean extends ComponentBean {
    // overwrite beforePost(), afterPost() accordingly
    // overwrite beforeGet(), afterGet() accordingly
}
```

5.4 A Simple Scheme for Data Containers

Application and components use the generic data containers as described above (TODO Ref. to be provided) for the communication of information to components. This is to keep a high degree of flexibility and to reduce the coupling between components.

The following flexible scheme for the implementation of generic data containers shall be used:

```
package org.uniboard.core;

import java.util.*;
import java.io.Serializable;

public class Message implements Serializable {
    private Map<String, Object> entries = new HashMap<>();
    public void put(String key, Object value) {...}
    public Object get(String key) {...}
}
```

Keys are strings to be agreed among components and application programmers. (TODO: Envision a pre-defined set of keys.) The types of the values are also agreed among components and application programmers. In order to reduce the coupling, basic types and Java system types should be used.

Here is a sketch of code illustrating how components access the information in, for example, a *Message* container:

```
// in a component class
@Override
protected Message beforePost(Message message) ... {
    String id = (String) message.get("IDENTITY");
    byte[] signatureHashValue = (byte[]) message.get("SIGHASH");
    ...
    // Create new message, fill new message m accordingly...
    // Alternatively, the given message object may be updated...
    Message m = new Message();
    // Add information... Then:
    return m;
}
```

The above sketch shows how keys (here *IDENTITY* or *SIGHASH*) can be used, and that applications or components must agree on the types of values (here *String* or *byte[]*).