Computing and Informatics, Vol. 41, 2022, 357–375, doi: 10.31577/cai_2022_1_357

PORTABILITY OF INTERFACES IN HEALTHCARE EAI ENVIRONMENTS

Severin Linecker

Johannes Kepler University Linz
Austria
&
Vinzenz Gruppe, Linz
Austria
e-mail: severin.linecker@vinzenzgruppe.at

Wolfram Wöss

Johannes Kepler University Linz
Austria
e-mail: wolfram.woess@jku.at

Abstract. Enterprise Application Integration (EAI) and HL7 (Health Level Seven) messaging are well established technologies in healthcare environments. Due to the age and longevity of HL7 standards (especially HL7 V2.x) and their widespread use, many interfaces outlive the middleware on which they run and must be ported to new systems. This often requires the entire code of the interface to be rewritten, which is associated with great effort and costs. This paper shows a generic EAI framework based on configuration and dependency injection for implementing reusable interfaces upfront and the results when applied to a real production EAI environment of an Austrian healthcare provider.

Keywords: Enterprise application integration, EAI, middleware, message-oriented middleware, MOM, HL7, portability, system integration, healthcare, system migration

1 INTRODUCTION

Exchanging clinical data between multiple heterogeneous medical information systems is very common in healthcare environments. The Hospital Information System (HIS) and other special (sub)systems, such as the Radiological Information System (RIS) need to be integrated for digital clinical workflows. Hence, Enterprise Application Integration (EAI) and HL7 (Health Level Seven) messaging have been developed and have become well established technologies in the healthcare sector for a long time.

Due to the message-oriented nature of HL7, Message-oriented Middleware (MOM) is a common paradigm for implementing EAI in healthcare environments [1]. It allows systems to communicate with each other by sending and receiving messages using interfaces directly connected to the middleware, which is then responsible for routing these messages to their correct destinations. This helps to reduce the total number of interfaces needed to connect n systems from $(n * (n - 1))/2$, when using point-to-point interfaces, to n. Especially for complex and big healthcare environments, which often consist of 50 or more connected systems (like in the case of the Vinzenz Gruppe, an association of seven religious-order hospitals and other healthcare facilities in Austria), this is a necessity.

HL7 is a messaging standard specifically developed for exchanging data between information systems in healthcare environments. According to [2], Version 2 of the messaging standard is one of the most widely used standard for healthcare information exchange. The HL7 V2.x standard [3] defines message types and their (real world) trigger events for clinical, financial and administrative data exchange. Messages have a message type and a trigger event, which together define a specific sequence of segments and segment groups. For example, the ADT (Admission, Discharge and Transfer) message type and the trigger event A02 is used for transmitting patient administration information about a patient transfer. Segments are a logical grouping of data fields. They may be mandatory, optional or repetitive within a message type. Two or more segments may be grouped together as a logical unit. Each segment has a unique name called Segment ID with three upper case characters (e.g., MSH, PID, PV1). Fields are character strings within a segment. They have an ordinal position within the segment for reference (e.g., PID.3), have a data type, can be required or optional and may be repetitive. Depending on its data type, a field may consist of components, which in turn may contain subcomponents. HL7 V2.x messages are typically plain text messages with certain special characters used as delimiter. Figure 1 shows the hierarchical structure of HL7 V2.x messages.

In hospital environments the HIS is a central point for data exchange. The connected systems send their data to the HIS, but also require its data. This data is mostly distributed as HL7 messages to the subsystems using middleware technology. The first version of HL7 V2.x was released in 1987, which also means that interfaces in the healthcare sector were newly created with HL7 V2.x at that time. It is therefore quite common that special medical systems have been used in a hospital

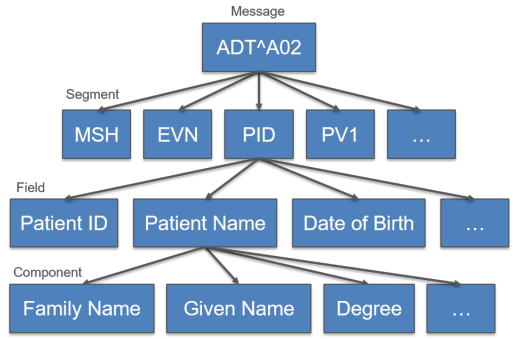

Figure 1. Hierarchical structure of HL7 V2.x messages.

for 10 or 20 years and more, and that the necessary interfaces have often existed for the same period of time. Even if the systems have been further developed and newer versions are now in use, the HL7 interfaces are usually virtually unaffected by such updates, since the basic functionality of a special medical system does not change. A RIS is still a RIS even if the functionality is expanded. The middleware solutions used are often not as long-lasting as the interfaces running on them. There are solutions that already exist for a very long time, but it is also not unusual that some interfaces have to be ported to new systems several times during their lifetime. This is often associated with considerable effort, especially in larger EAI environments with hundreds of interfaces. In many cases, porting or migration means a complete redevelopment of the interfaces, which is aggravated by the fact that the original developers are often no longer present.

But not only (legacy) HL7 V2.x interfaces are potentially affected by a migration to a new middleware solution. Interfaces that are now implemented with modern standards, such as HL7 FHIR [4], may also have to be ported in their lifetime, which is very likely, given the longevity of medical subsystems and the longevity and maturity of all HL7 standards.

In order to save time and resources when switching to a new middleware solution, it is important to have portable interfaces available that are capable of running on top of the new solution without changing the source code of the interface, where the business logic is located. To meet the objective above, this work introduces a generic EAI framework based on configuration and dependency injection used for creating portable interfaces, by introducing an abstraction layer between the middleware and the interface implementation.

The remainder of the paper is organized as follows: Section 2 contains related work. In Section 3 the EAI framework for portable interfaces is described in detail. In Section 4 the evaluation results of applying the proposed solution to a real production healthcare EAI environment are shown. The paper finishes with Section 5 with conclusion and future work.

2 RELATED WORK

The authors of [5] show a middleware architecture consisting of a cloud service and local clients. They seperated the business logic of their hCloud Middleware and their hCloud Client from the persistence layer and used interfaces to mediate the communication between these two layers. This approach, even not directly mentioned in their study, may facilitate portability of their business rules, depending on how the rest of the code is structured.

In [6] the authors ported the proprietary middleware Egate to the Open Source middleware Apache Camel. They show best practices for migrating to Open Source middleware solutions. However, their use-case contains only 13 interfaces and therefore portability was not an issue, because in case of a future migration to another system the effort for the new implementation remains manageable.

In the work of [7] an Apache Camel based implementation of an industrial middleware solution is shown. If Apache Camel should be replaced with another system, the interfaces would also have to be reimplemented.

In our previous work [8] we showed a generic architecture for implementing reusable interfaces in the healthcare domain. We described components acting as building blocks for interfaces. The work is focussed on reusability, but the architecture emphasizes portability as well.

Even though there are some more studies about HL7, healthcare information exchange, EAI and middleware architecture in the healthcare domain like [9, 10, 11, 12, 13], their main objective is not portability. Their concern is the integration problem itself and not porting their interfaces or their business logic to another middleware solution without changing the source code.

The contribution of this work is a generic EAI framework based on configuration and dependency injection, used for implementing portable interfaces in (existing) healthcare EAI environments.

3 EAI FRAMEWORK

In this chapter, a generic EAI framework used for implementing portable interfaces is introduced. The main goal is to provide a custom API for interface development, which is perfectly adapted to the conditions of the corresponding domain. The currently used middleware does not play a role here and should only serve as a possible execution layer that can be replaced at will. To accomplish this flexibility, an abstraction layer is inserted between the middleware and the interface implementation, abstracting the details of the middleware and thus making it interchangeable. In subsequent migrations to new middleware solutions, the interface code remains the same and does not need to be rewritten. Only the actual implementation of the EAI framework itself must be adapted to the middleware, the API for the interface development remains unchanged. Figure 2 shows the layer architecture of the EAI framework.

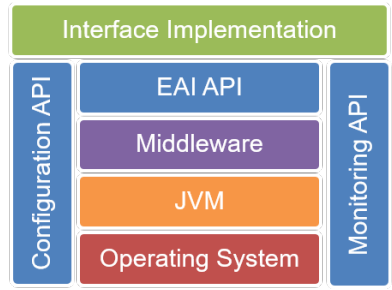

Figure 2. Layer architecture of the EAI framework

The EAI framework is essentially based on two pillars:

1. configuration and
2. dependency injection.

These two basic principles sustain the entire system and all components make use of them.

3.1 API Components

The API components are the basis for portable interfaces and thus form the abstraction layer shown in Figure 2. This layer defines the entire functionality necessary for interface development. In order to define this layer specifically, the necessary functions must first be analyzed and specified. The basic function groups are as follows:

- Connectivity,
- Message Structures/Parsing,
- Data Access,
- Monitoring,
- Utilities.

In the next step, the required functionalities must be implemented as Java interfaces. The focus should be solely on the API and not on the possible implementation. The goal is to obtain an API for interface development that is perfectly suited to the environment. During our implementation it also became apparent that the components should be as lightweight as possible and only those methods should really be included in the Java interface for which there is actually a concrete use case. It is better to define an additional component instead of defining a single one that can handle all possible scenarios, because porting small components is easier and less error-prone. The following listing shows an example component for sending string messages via a JMS (Java Message Service) queue:

```
public interface JmsWriter {

  void sendTextMessage(String msg);

}
```

Finally, it is necessary to define how to implement interfaces with this API. This is basically about defining a method that should be executed when the interface gets triggered (e.g., a message in a JMS queue). There are two variants:

1. using a Java interface which defines a method (e.g.: `execute()`) and this Java interface must then be implemented by all interfaces, or
2. an annotation-based approach in which the method to be executed is marked.

During the development of the EAI framework we implemented both variants. Using a Java interface as in the variant 1 is the most obvious solution. It is easy to implement and we used this approach in early versions of our framework. However, to reduce the footprint of our framework in the interface code and to gain flexibility, we implemented the variant 2, which is used since then in all recent versions of our framework. The following listing shows an example interface developed with our EAI framework, which reads a string message from a JMS queue, performs a string replacement, where search string and replacement string are provided by a configuration class, and sends the new string message to a JMS queue.

```
public class SearchAndReplaceCollab {

  private final Logger logCat;
  private final JmsReader input;
  private final JmsWriter output;
  private final SearchAndReplaceConfig config;

  @Inject
  SearchAndReplaceCollab(
      Logger logCat,
      JmsReader input,
      JmsWriter output,
      SearchAndReplaceConfig config) {
    this.logCat = logCat;
    this.input = input;
    this.output = output;
    this.config = config;
  }

  @OnTrigger
  public void onMessage() {
    String msg = input.getTextMessage();
    // perform string replacement
```

```
    String newMsg = msg.replace(
      config.getSearchString(),
      config.getReplaceString());
    logCat.info("new msg: {}", newMsg);
    output.sendTextMessage(newMsg);
  }
}
```

3.2 Configuration

The configuration of an interface is done by using an initialization file (INI) and each interface must have one. An INI consist of sections and the associated properties. The configuration options defined in this way are bound to fields of a Java class using dependency injection. A section is delimited by square brackets and can contain any number of properties, but empty sections are allowed too. The following listing shows a sample INI:

```
[Properties]
  myIntProperty=23
  myStringProperty=abc123
  myBooleanProperty=true
  list.myList=value1, value2, value3
  list.myListFromFile=${Resources/listFile}
  map.myMapFromFile=${Resources/propertiesFile}
  filter.inbound=!myFilter && myFilter2

[MySection]
  enum.charset=vg.sissi.core.util.Charset.CP1252
  map.myMap=[key1=>value1, ?=>defaultValue]

[EmptySection]
```

An INI can have an arbitrary number of sections and it is also allowed to have the same section multiple times. In such case the section needs an `id` property for later reference. There are four special sections:

System (mandatory): Required for executing the framework itself.

Bindings (optional): User defined bindings of Java interfaces to an implementing Java class.

Resources (optional): Defines file resources. The paths can be relative to the file location of the INI and get resolved automatically.

Properties (optional): User defined properties of the interface implementation. It is the default namespace of all properties which do not have their own namespace (= section).

A property is always assigned to the section above it and the name of the section is its namespace. Primitive data types are supported as values out-of-the-box. For object data types, the property name must start with a prefix. It is also possible to use variables with the `${SECTION_NAME/PROPERTY_NAME}` syntax. On the Java side, the values of the properties are mapped from the INI to class variables. Table 1 shows the available prefixes and their corresponding Java type.

Prefix	Java Type	Value-Syntax
`list.`	`java.util.List<String>`	`val1, val2, val3, ...` or reference to a list file
`map.`	`java.util.Map<String, String>`	`[key1=>val1, key2=>val2]` or reference to a mapping file
`enum.`	any enum class	`f.q.n.EnumClass.ENUM_CONSTANT`
`time.`	`java.time.LocalTime`	`HHmm[ss][SSS]`
`date.`	`java.time.LocalDate`	`YYYYMMDD`
`datetime.`	`java.time.LocalDateTime`	`YYYYMMDDHHmm[ss][SSS]`

Table 1. Property prefixes and their mapped Java type

In order to be able to map the properties from the INI and the class variables of a configuration class, these fields must be provided with an `@Property` annotation. It has two properties: `ns` and `value`. The `ns` property defines the namespace of the configuration option, which is equal to the name of the section from the INI, where the property was declared. The `value` property defines the name of the property from the INI. Properties can be optional or mandatory and may have a default value. The following listing shows the Java equivalent to the previous INI snippet:

```
public class MyConfig {

  @Inject(optional=true)
  @Property(value="myIntProperty")
  private int myIntProperty = 10;

  @Inject
  @Property(value="myStringProperty")
  private String myStringProperty;

  @Inject
  @Property(value="myBooleanProperty")
  private boolean myBooleanProperty;

  @Inject(optional=true)
  @Property(value="list.myList")
  private List<String> myList;
```

```
  @Inject
  @Property(value="list.myListFromFile")
  private List<String> myListFromFile;

  @Inject
  @Property(value="map.myMapFromFile")
  private Map<String, String> myMapFromFile;

  @Inject(optional=true)
  @Property(value="filter.inbound")
  private Filter filter;

  @Inject
  @Property(ns="MySection", value="map.myMap")
  private Map<String, String> myMap;

  @Inject(optional=true)
  @Property(ns="MySection", value="enum.charset")
  private Charset encoding = Charset.UTF_8;

  // getter and setter ...
}
```

During the implementation of the framework, we mainly worked with standalone configuration classes. Those are classes where the sole purpose is to represent all configuration options of the component to be implemented (e.g., a connector). The advantage of this approach is that for unit tests the different variants can easily be provided by simply setting the fields of a configuration class in the testing code. There was also a lot of consideration given to the visibility of these classes. Most configuration classes are package private, since they are to be seen as implementation detail and should not be necessarily publicly accessible.

3.3 Dependency Injection

Dependency injection is an essential part of the EAI framework. The dependency injection framework Guice from Google [14] is used as the basis. Guice is based on modules that provide the bindings. A module can load other modules and can be public or private. The difference is that private modules provide only those bindings that are explicitly exposed, whereas public modules expose all bindings. These components were used as a basis to implement a very flexible and dynamic dependency injection solution. The basic principle here is that only those bindings should be provided and loaded which are actually needed by the interface. Furthermore, it should be possible to easily extend the functionality of the EAI framework by simply adding an additional Java archive (jar) containing the new enhancements to the classpath.

As already mentioned in Section 3.1, each functionality is defined with a Java interface. The binding to a concrete implementation is then done via a module. That means that for each functionality or for each Java interface of the EAI framework there is a corresponding module class, which provides the bindings. Each of these modules needs a name, so that they can be loaded later. This is done by using a custom `@Module` annotation, with a `value` property holding its name. The whole binding process is tightly coupled to the INI. The modules are loaded by applying the sections from the INI. The presence of a section in the INI triggers the loading of a module if the section name is equal to the name from the `@Module` annotation. The following listings show an example of this approach:

```
[MySection]
  myProperty=myValue]
```

The corresponding module that would be loaded if a section `MySection` exists in the INI is specified as follows:

```
@Module("MySection")
public class MySectionModule ... {
```

There is a main module provided by the EAI framework that serves as the entry point. The INI is passed to this module, which is the only module that is responsible for more than one section. It handles the sections listed in Section 3.2 that are needed by the framework itself and initiates the loading of those modules that are needed to satisfy the dependencies of the interface represented by the currently loaded INI. The following algorithm shows the schematic procedure of the main module:

1. Find all module classes in the classpath, which have a `@Module` annotation and store their name and their class in a map `moduleMap`.
2. Iterate over all sections of the INI and check the `moduleMap` if there is a module, with the same name as the current section.
3. If yes, instantiate the module class, pass the INI, install the module and mark the module as loaded, so that it will not be loaded again even if there is another section with the same name.
4. If no, provide named bindings of all properties within this section using the `@Property(ns=SECTION_NAME, value=PROPERTY_NAME)` annotation.

To provide the bindings, a distinction must be made whether there can/should be multiple instances of the same type or not. If this is the case, the module must ensure that the specific instances are also identifiable using a unique name. This is done by a named binding using the `@javax.inject.Named` annotation. The bindings for the configuration options of the respective functionality are also provided in these modules. The instances of the configuration options (the concrete value e.g., as string) are provided over a named binding too. Instead of the `@javax.inject.Named` annotation the `@Property` annotation already mentioned in Section 3.2 is used.

3.4 Implementation and Middleware Integration

The EAI framework can be used in two ways. Either as a library, i.e., on top of an existing middleware platform, or as a standalone solution. The interface code remains the same, only the underlying platform changes. These two variants will be described in more detail in the next sections.

3.4.1 On-Top of a Middleware

In this variant, the EAI framework is embedded in an existing middleware platform. The basic prerequisite for this approach is, of course, that the middleware solution allows the execution of arbitrary Java code and that the EAI framework can be integrated by using one or more Java archives. The native API of the existing middleware is only needed to create a frame in which the code of the EAI framework can be executed. This can be, for example, a Java class directly, or a component in which Java code can be executed. The existing middleware solution is also responsible for triggering the executing of the interface code, and therefore it is not necessary to be considered by the EAI framework.

Depending on how much functionality is provided by the EAI framework, the connector classes of the existing solution must also be accessible. These connector classes can be used in the implementation of the Java interfaces of the EAI framework to provide the actual functionality, meaning that the components of the EAI framework are just a facade or adapter and in the background exactly the same code or classes are used that would be used in a native implementation based on the existing platform. Sometimes there is a need for special functionality which is not available in the native API, and therefore these components of the EAI framework need to be implemented without classes of the underlying platform. It is often possible to use open source libraries, but their API should not be used directly in the interface code and should be provided via a suitable facade by the EAI framework. For example, the well-known open source library HAPI [15] is a very powerful library for HL7 V2.x parsing and messaging. Even if HAPI itself would be portable, i.e., the portability of the interface code would not be affected, it is better to define custom components in the EAI framework in order to be able to replace the implementation later if required. For example in case of porting an interface to a new middleware platform, which natively supports a previously missing feature implemented using open source libraries, the native implementation can be used instead using the same facade and therefore the interface code itself remains unchanged. This is a key aspect of the solution, as an existing middleware can be extended in such a way, that the new functionality can later be used on another platform, which supports this functionality natively.

Figure 3 shows the class diagram of the Java interfaces, which we have defined for integrating the EAI framework into an existing middleware solution. We defined two Java interfaces `Collaboration` and `CollaborationFactory`. The `CollaborationFactory` is responsible for creating `Collaboration` instances.

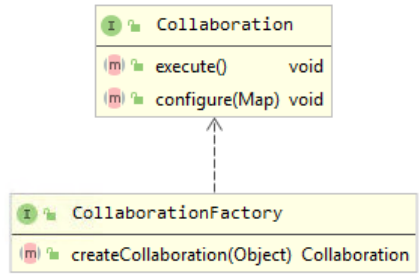

Figure 3. Class diagram for middleware integration

Therefore it takes the instance of the outer Java class, respectively the interface class from the middleware, as an argument. This instance is used as a reference for finding the correct interface implementation from the EAI framework. It searches for the corresponding INI for the requested interface. Therefore the INI must be accessible from the middleware system. The factory returns an instance of the `Collaboration` Java interface which represents an interface from the EAI framework. It defines two methods: `execute` and `configure`. The `configure` method takes a `java.util.Map` as parameter, which should contain all instances of the required classes from the native API of the middleware solution. The key of the map has to be a unique string identifier to later access the required classes. The `configure` method passes the dependencies from the middleware to the EAI framework. Using dependency injection, these objects are then made available to the classes that require them. The last step is calling the `execute` method which should then trigger the execution of the interface code developed with the EAI framework.

3.4.2 Standalone Middleware Solution

In the standalone variant, all components of the EAI framework are implemented without being able to fall back on an existing middleware solution. This means that a native implementation of the EAI framework is made (the purple layer of Figure 2). The scope varies depending on the requirements from a single interface to a complete EAI solution. The complexity of the implementation increases, as functionality that is otherwise provided by an existing middleware solution must be taken into account:

- Running an interface and handling its states (start, stop, ...).
- Triggering the interface, i.e., listening for incoming messages or events to trigger the execution of the interface code.
- Transaction handling.
- Thread safety and parallel execution.
- Error handling.

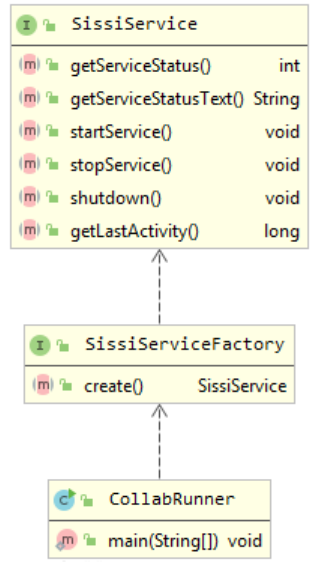

Figure 4. Class diagram of the standalone solution

To run an interface, we implemented the `CollabRunner` class with a `main` method, which takes an initialization file as an argument and uses a `SissiCollaborationFactory` to create an instance of `SissiService`. Figure 4 shows an UML class diagram of the classes and interfaces used for running interfaces in this variant. The `SissiService` represents an interface service that can be started (`startService`), paused (`stopService`) and completely stopped (`shutdown`). Furthermore, it offers methods to read the current state and the timestamp of the last activity. The service is responsible for listening for incoming messages or scheduled events in order to trigger the interface processing. There are two alternatives to implement this:

1. Implementing a generic service, which allows to react on incoming messages or events by using different inbound connectors (JMS, File, Socket, ...). These inbound connectors must then provide suitable methods to inform the parent service that a certain number of messages or events are pending. This can be done, for example, using a custom `@CheckTrigger` annotation that can be used to mark a method of an inbound connector for this purpose.
2. Implementing a service for each type of inbound connector. The service is solely responsible for handing messages or events from a single type of connectivity. For example, implementing a `JmsService`, which only handles incoming JMS messages.

As the code of the interface implementation is the same as for the embedded variant, the service implementation has to invoke the method annotated with `@OnTrigger` in order to initiate message processing. There are interfaces, which need proper transaction handling support. To provide a simple transaction handling solution, there are two annotations, `@OnCommit` and `@OnRollback`, which can be used to annotate methods that should be called for the corresponding event.

4 EVALUATION

The introduced EAI framework was implemented and deployed in the EAI production environment of the Vinzenz Gruppe. The considerations for the implementation of a custom EAI framework started in 2014. At that time, it was clear that the current middleware solution Java CAPS 5.1.3 [16] had reached its end of life and a new system had to be found. This, in turn, meant that when porting to a new system, all interfaces would have to be reimplemented, including those developed in the meantime, i.e., until the new system goes live. Furthermore, this would become a recurring process, since any new system would also have to be replaced at some point. This problem gave rise to the idea of developing an EAI framework adapted to the environment of the Vinzenz Gruppe, which could run on any Java-based middleware. At the end of 2014 the production EAI environment consisted of 342 interfaces deployed to three production servers.

The first production version of the EAI framework V1 was completed in February 2015. From this point on, all new interfaces were implemented on the basis of the new EAI framework and ran on the Java CAPS 5.1.3 middleware. At the end of 2015, it was decided to implement the EAI framework as a standalone variant V2 in order to completely replace Java CAPS. The first interface in this form went live on December 6, 2016. From the beginning of 2017, all new interfaces were implemented using V2 only.

From 2015 to 2017, 16 V1 interface classes were developed, with a total of 64 running instances in the Java CAPS environment. Table 2 shows the interface classes and their number of instances.

All 64 interfaces and/or their 16 interface classes from V1 could be ported directly to V2, meaning that no code change was necessary to run these interfaces with V2. However, minor changes were made to improve code quality and/or readability. The current EAI production environment consists of 565 interfaces, deployed to eight production servers. 87 interfaces are legacy interfaces, which are neither V1 nor native V2 interfaces. From the original 16 V1 interface classes and their 64 instances, 10 classes with 29 instances still exist. One class became obsolete. The other five classes have been further developed, however the new classes evolved from the original ones. Three of these classes are among the most frequently used interface classes in the entire EAI production environment. Table 3 shows the interface classes and their running instances in the current EAI production environment.

Interface Class	Instances Running
A	14
B	11
C	9
D	7
E	7
F	3
G	2
H	2
I	2
J	1
K	1
L	1
M	1
N	1
O	1
P	1
In total	64

Table 2. V1 interface classes and their number of instances.

Interface Class	Instances Running
A’	75
B’	49
C	7
C’	33
D	7
E	7
F	obsolete
G’	7
H	2
I	2
J	1
K	1
L	1
M	1
N	1
O’	1
P’	1
V2 only	283
legacy	87
In total	565

Table 3. Interface classes and their usage in the current production environment

The pie chart in Figure 5 shows that the running instances of the original V1 interfaces together with the running instances of the interface classes, which evolved from the original V1 class, make up 34 % of all instances running in the current production environment. In other words, 34 % of the current interfaces did not need to be reimplemented during migration, which saved time and money during the migration, especially concerning that these classes were all developed at a time when the decision to replace the existing EAI system had already been made, but a specific replacement date was not foreseeable or defined.

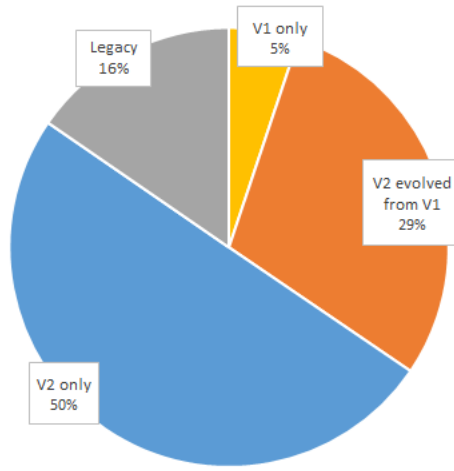

Figure 5. Pie chart of interface class usage

5 CONCLUSION AND FUTURE WORK

In this paper a solution for portable interfaces using a flexible and configurable EAI framework with a custom API for interface development is introduced. Interface code that has been written once using this API could easily be ported from a proprietary middleware solution to a full featured native implementation of the EAI framework. Use of this approach helped to save time and resources during the migration process and should also reduce the effort for future migrations. This is a major advantage, especially with regard to the longevity of HL7 interfaces. The downside of our approach is, that it is not sustainable for small EAI environments with only a few interfaces, because the development of an EAI framework is probably more complex and time consuming than porting a few interfaces in case of a migration to a new middleware solution. Furthermore, the chosen programming language or runtime environment affects the future-proofness of the solution. There

is no guarantee, that the chosen technology stack is also available in ten or more years.

For future work, it is planned to analyse the capabilities of our EAI framework using another proprietary middleware solution. Furthermore, we will present an approach for migrating legacy interfaces to the proposed EAI framework using code generation and emulation. We are also planning to release our EAI framework V2 as open source software.

REFERENCES

[1] Bezerra, C.—Araujo, A.—Sacramento, B.—Pereira, W.—Ferraz, F.: Middleware for Heterogeneous Healthcare Data Exchange: A Survey. Tenth International Conference on Software Engineering Advances (ICSEA 2015), 2015, pp. 409–414.

[2] HL7 International: HL7 Version 2 Product Suite. 2021, `https://www.hl7.org/implement/standards/product_brief.cfm?product_id=185` [accessed April 20, 2021].

[3] ISO: ISO/HL7 27931:2009. Data Exchange Standards – Health Level Seven Version 2.5 – An Application Protocol for Electronic Data Exchange in Healthcare Environments. 2009, `http://www.iso.org/iso/catalogue_detail.htm?csnumber=44428`.

[4] Braunstein, M. L.: Health Informatics on FHIR: How HL7's New Api Is Transforming Healthcare. Springer, 2018, doi: 10.1007/978-3-319-93414-3.

[5] Bezerra, C. A. C.—de Araújo, A. M. C.—Times, V. C.: An HL7-Based Middleware for Exchanging Data and Enabling Interoperability in Healthcare Applications. In: Latifi, S. (Ed.): 17th International Conference on Information Technology – New Generations (ITNG 2020). Springer, Cham, Advances in Intelligent Systems and Computing, Vol. 1134, 2020, pp. 461–467, doi: 10.1007/978-3-030-43020-7_61.

[6] Iyer, R.—Balasundaram, C.: Best Practices and Case Study for Open Source Middleware Migration: Egate to Apache Camel Migration. International Conference on Software Engineering and Mobile Application Modelling and Development (ICSEMA 2012), 2012, pp. 1–7, doi: 10.1049/ic.2012.0140.

[7] Gosewehr, F.—Wermann, J.—Borsych, W.—Colombo, A. W.: Apache Camel Based Implementation of an Industrial Middleware Solution. 2018 IEEE Industrial Cyber-Physical Systems (ICPS), 2018, pp. 523–528, doi: 10.1109/ICPHYS.2018.8390760.

[8] Linecker, S.—Wöss, W.: Reusability of Interfaces in Healthcare EAI Environments. Proceedings of the 14th International Joint Conference on Biomedical Engineering Systems and Technologies – HEALTHINF, INSTICC, Vol. 5, 2021, pp. 417–423, doi: 10.5220/0010242004170423.

[9] Alenazi, T. M.—Alhamed, A. A.: A Middleware to Support HL7 Standards for the Integration Between Healthcare Applications. 2015 International Conference on Healthcare Informatics, 2015, pp. 509–512, doi: 10.1109/ICHI.2015.93.

[10] Bortis, G.: Experiences with Mirth: An Open Source Health Care Integration Engine. Proceedings of the 30th International Conference on Software Engineering (ICSE '08), Acm, 2008, pp. 649–652, doi: 10.1145/1368088.1368179.

[11] Liu, L.—Huang, Q.: An Extensible HL7 Middleware for Heterogeneous Healthcare Information Exchange. 2012 5th International Conference on Biomedical Engineering and Informatics, 2012, pp. 1045–1048, doi: 10.1109/BMEI.2012.6513196.

[12] Lu, X.—Gu, Y.—Yang, L.—Jia, W.—Wang, L.: Research and Implementation of Transmitting and Interchanging Medical Information Based on HL7. The 2nd International Conference on Information Science and Engineering, 2010, pp. 457–460, doi: 10.1109/ICISE.2010.5689687.

[13] Wadhwa, R.—Mehra, A.—Singh, P.—Singh, M.: A Pub/Sub Based Architecture to Support Public Healthcare Data Exchange. 2015 7th International Conference on Communication Systems and Networks (COMSNETS), 2015, pp. 1–6, doi: 10.1109/COMSNETS.2015.7098706.

[14] Google Inc.: Guice. 2021, `https://github.com/google/guice` [accessed February 20, 2021].

[15] HAPI: HAPI HL7 V2.x. 2021, `https://hapifhir.github.io/hapi-hl7v2/1` [accessed February 23, 2021].

[16] Czapski, M.—Krueger, S.—Marry, B.—Sahai, S.—Vaneris, P.—Walker, A.: Java CAPS Basics: Implementing Common EAI Patterns. Prentice Hall PTR, 2008.

Severin Linecker works for the Vinzenz Gruppe, an association of religious-order hospitals and other healthcare facilities in Austria. Furthermore, he is a Ph.D. student at the Johannes Kepler University Linz (JKU). His main research interest is Enterprise Application Integration (EAI), especially the migration of legacy interface code.

Wolfram Wöss worked for a manufacturing company between 1990 and 1993. Since 1993, he has been employed at the Johannes Kepler University Linz (JKU), where he received his doctorate (Ph.D.) in 1996. In 2002 he completed his habilitation in applied computer science. He is Deputy Head of the Institute for Application-Oriented Knowledge Processing (FAW) since 2004. His research and teaching activities include topics in the field of intelligent information systems, integrated information systems, semantic information integration, ontologies, knowledge graphs, information engineering, data modeling, data quality, big data, business intelligence, and data mining. He has published on these topics in scientific journals and conference proceedings. He is also a member of numerous program committees and was program chair of the International Conference on Warehousing and Knowledge Discovery (DaWaK 2003 and 2004) and chair of the International Workshop on Web Semantics (WebS) in 2003–2013. He received the Best Paper of the Year 2016 award from Elsevier Data and Knowledge Engineering journal. During his tenure at JKU, he managed both industrial and publicly funded projects.