Chapter 1

The Adapter Framework

1.1 Introduction

Changing requirements force software systems to include new components to extend their features. Software components are self-contained entities with a well-defined interface and behavior. For two components to function together they must be *integrated* through a compatible interface. Typically such compatibility is not a given and it should be, often manually, established through additional programming. The integration task also entails binding the data values, i.e creating dependencies between the components. Once the data is represented through a compatible interface it is assigned to a target field and kept synchronized with the source data, this operation is called data binding.

The problem of incompatible interfaces is a common software engineering problem and its solution is captured in the *adapter pattern* (cite Gamma). An adapter is a level of indirection between the incompatible interfaces, using one interface as the adaptee and the other as the target. This means the when an object of the target interface is expected but a source object of another interface is offered, the adapter which wraps the source object and implements the target interface is initialized.

There are some issues attached to the traditional adapter pattern. Firstly, its implementation can be problematic with some programming languages. If we look at a class adapter, we see that it should be implemented with a language that allows multiple inheritance. Single inheritance programming languages like Java needs to use workarounds to achieve the same effect. The implementation of object adapter pattern is more flexible, since it only needs to inherit from the target type. However this implementation also does not allow a single object adapter to adapt to multiple target types. A second point is that the adapter pattern adds a level of indirection between the source and the target, which needs to be maintained. Another drawback comes from the additional dependencies introduced by the adapter classes. The integration code has to refer to the specific adapter classes to initialize the desired objects. This has two side effects, first the implementation contains a direct reference to the adapter type instead of just the source and the target types. This introduces an additional maintenance efforts. Secondly the user is required to know exactly which adapter is responsible for a specific interface conversion. Adapters are not a well-documented part of a software. Usually this information is communicated through a particular adapters name or one has to go inside that adapter class to understand the expected source and the target types. This process is error-prone as well as time consuming.

Adaptation is only one part of the integration problem. The second part entails establishing dependencies between the integrated components. i.e binding. In component-based design loose-coupling is an important principle. Dependency injection (DI) (cite fowler) is a lightweight method for keeping modules loosely coupled by delegating the creation of concrete objects to so-called *injectors*. This approach allows a customizable and a decoupled way of creating dependencies, while maintaining loose coupling.

Typically these two parts, adaptation and binding, are handled separately from each other during integration. We think these parts can be unified from the users perspective by providing a concise interface for the whole integration task. In order to achieve this goal we have designed the *zamk* framework, which unites dependency injection with *under-the-hood* adaptation logic. For our framework we have decomposed the traditional adapter pattern and created *converters*, which are user-defined inheritance-free classes. *zamk* comes with its own dependency injection mechanism that is used with a designated domain-specific language called *Gluer*. The dependency injection logic is intertwined with the adaptation logic which uses the converter registry to perform automated adaptation between source and target types. We automate the adaptation process by exploiting the type hierarchies and provide checks and context-relevant messages for correct integration. The details of the framework will explained throughout the chapter.

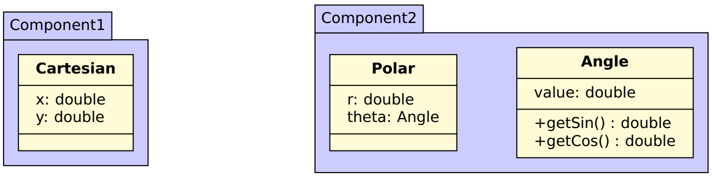


Figure 1.1 – UML diagram for the two components

1.2 Motivation

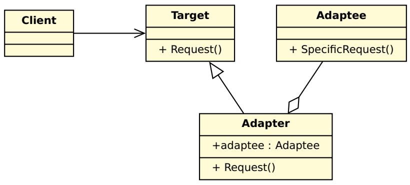
We have mentioned some problems attached to the traditional adapter pattern[1]. Let us give an example which illustrates these problems in a detailed manner. We assume we have a plot drawing software which uses the Cartesian coordinate system to represent the points in the plot. The software includes a data component which contains a class called `Cartesian` which includes two fields `x, y`, that represents the values on the x-axis and y-axis respectively. This class also includes getters and setters for these fields (`Component1` in Figure 1.2). A new requirement is received which states that the software must also support polar coordinates, and the points on a plot should be shown in a selected format (Cartesian or polar).

In order to support polar coordinates, a new component which contains classes to represent such data is introduced (`Component2` in Figure 1.2). The class `Polar` contains two field `r` representing the radius and `theta` of type `Angle`. This component is to be integrated with the `Cartesian` component. It should be possible to obtain the `Polar` representation of any `Cartesian` object by using an *adapter*.

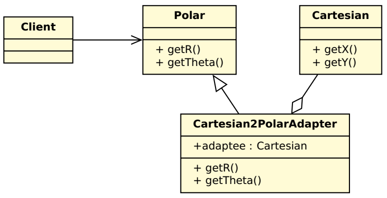
An abstract view of the traditional object adapter pattern is shown in Figure 1.2a. The adapter pattern relies on inheritance and adds a level of indirection between the `Client` and the `Target`. The application of this pattern to the example case requires creating a Cartesian to polar adapter (`Cartesian2PolarAdapter`) which takes a `Cartesian` object as an *adaptee* and extends the `Polar` class to override its methods (Figure 1.2b). An implementation for this adapter is given in listing 1.2c.

In their paper on aspect-oriented implementation of gang-of-four design patters ([1]) Hannemann and Kiczales mention an adapter pattern implementation using inter-type declarations. According to the auxiliary code they provide with this study, they propose the adaptee class should subclass the target class, which results as the

[1]Throughout this chapter, the term adapter pattern will refer to the object adapter pattern. The class adapter pattern will explicitly include the term class.



(a) Abstract view of the object adapter pattern



(b) Concrete view of the object adapter for the example case

```
public class Cartesian2PolarAdapter extends Polar{
 Cartesian adaptee;
 public Cartesian2PolarAdapter(Cartesian c) {
  this.adaptee = c;
 }
 public double getR()
 {
  return Math.sqrt(Math.pow(adaptee.getX(), 2) +
      Math.pow(adaptee.getY(), 2)));
 }
 public Angle getTheta()
 {
  return new
      Angle(Math.atan(adaptee.getY()/adaptee.getX())));
 }
}
```

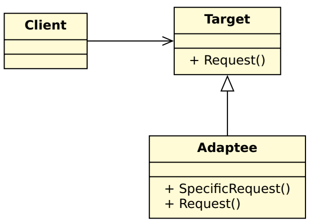
(c) The implementation for the `Cartesian2PolarAdapter`

Figure 1.2 – The diagram of the object adapter and the corresponding Java implementation

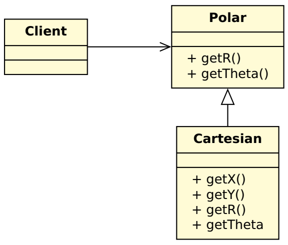
diagram shown in Figure 1.3a. For our example case the `Cartesian` class will directly have to subclass the `Polar` class and implement its method using its `x, y` field values. This is depicted in the aspect shown in Listing 1.3c. The inter-type declaration on line 3 declares the inheritance relation and the subsequent method implementations are woven into the `Cartesian` class.

The obvious problem with this implementation is that it would quickly become unusable due to Java's single inheritance. If the `Cartesian` class were extending any other class, that it would not be possible to declare that it also extends `Polar` class. Also such a declaration means that a `Cartesian` object is also a `Polar` object, which is prone to create ambiguities in an object-oriented setting.

In order to integrate the polar coordinates into our plotting software, we need to alter some code. According to the selection made



(a) Abstract view of the object adapter pattern with inter-type declarations



(b) Concrete view of the aspect-oriented adapter pattern for the example case

```java
public aspect Cartesian2PolarAdapter {

 declare parents: Cartesian extends Polar;

 public double Cartesian.getR()
 {
  return Math.sqrt(Math.pow(this.getX(), 2) +
      Math.pow(this.getY(), 2));
 }

 public Angle Cartesian.getTheta()
 {
  return new Angle(Math.atan(this.getY()/this.getX()));
 }
}
```

(c) The implementation for the `Cartesian2PolarAdapter` in AspectJ

Figure 1.3 – The diagram of the object adapter and the corresponding Java implementation

in the GUI, the information box should display the coordinate value in the correct format. An hypothetical example for the integration code is given in Listing 1.1. On line 8 we see an explicit reference to the class `Cartesian2PolarAdapter`. In this simple example, this reference is manageable. However if several such extensions are to be made, there will be several such references. This requires keeping track of all referenced classes; if one is deleted then we would have errors in our code, due to broken dependencies. Also the user performing the integration has to know all the specific classes that are responsible for adaptations; depending solely on naming is error-prone. This slows down the integration process.

We have used the plotter example to illustrate some of the issues related to the traditional adapter pattern and component integration

```java
public void viewPointValue(Point selected)
{
 if(GUI.format == CARTESIAN)
  GUI.createNewValueBox(selected.loc(),
     selected.getCoordinates().toString());
 else if(GUI.format == POLAR)
 {
  Polar p = new
     Cartesian2PolarAdapter(selected.getCoordinates());
  GUI.createNewValueBox(selected.loc(), p.toString());
 }
}
```

Listing 1.1 – The integration of Polar coordinates

using these adapters. Our goal is to provide a solution to the these issues in the form of a framework. Our framework has the following requirements:

I. Use adapters without creating dependencies to specific adapter classes.

II. Means to separate the adaptation logic from the integration logic during development time.

III. Inheritance-free adapter structures.

IV. Adapters are found automatically given a source object and a target type.

V. Dependency injection based binding via a declarative language that works with the adaptation logic.

1.3 Approach

We have concluded with a set of requirements in Section 1.2. In this section we will explain the details of the *zamk* framework which is designed according to these requirements.

zamk is a development framework specifically tailored for adapter-based integration. It offers a new and a light-weight way of defining adapters; these light-weight structures are called *converters*. Converters are user-defined classes that are free of inheritance and they are stateless. *zamk* runtime is responsible for finding the correct converter, given a source and a target object. This means the only additional dependency we have to include in the implementation is the *zamk* runtime API. Since the user doesn't have to refer to specific adapters, *zamk* allows separation of adaptation and binding concerns during integration. *zamk* also comes with its own binding language, so-called *Gluer*, which uses dependency injection under-the-hood. *Gluer* is a domain-specific language and its declarative nature allows compile-time checks. The user is flexible in how she chooses to use *zamk*, she can either use the *Gluer* language or she can call the runtime API directly. Integration using *zamk* also helps separating the binding concern and the adaptation concern from each other.

In Figure 1.4 shown an overview of the stages that are necessary to perform an integration using *zamk*. *zamk* requires user-defined input to perform an integration.

Development Time The user-defined input is a produced during *development time*. In the figure we show four inputs that are fed to the *zamk* compile-time; there are (in the order shown in the figure):

Gluer files: Gluer files contain the *Gluer* statements that define which objects are going to be injected to which fields. *Gluer* statements are used to generate conversion requests, which are *zamk* API calls that trigger the *zamk* adaptation work-flow. The *Gluer* language is discussed in Section 1.3.2.1.

Application Classes: These classes are scanned to retrieve information for integration. This information is used in checking. Also applications classes are instrumented with conversion requests that are derived from *Gluer* statements.

Converter: These are the user-defined classes that adhere to a specific structure. Converters contains methods necessary to convert one type to the other. These are discussed in Section 1.3.2.2.

Converter Precedence Declarations: these are used to resolve conflicts when more than one conversion is found for a specific request. In this case the conversion which is contained by the converter class that has the higher precedence is used.

It is also possible to write plain Java code that calls the *zamk* API directly. However these statements are not processed during compile-time, they are considered as ordinary Java code. That is why we do not list them as compile-time inputs. During the *final compilation* phase, they are compiled with all the code that is provided by the user and generated by *zamk*.

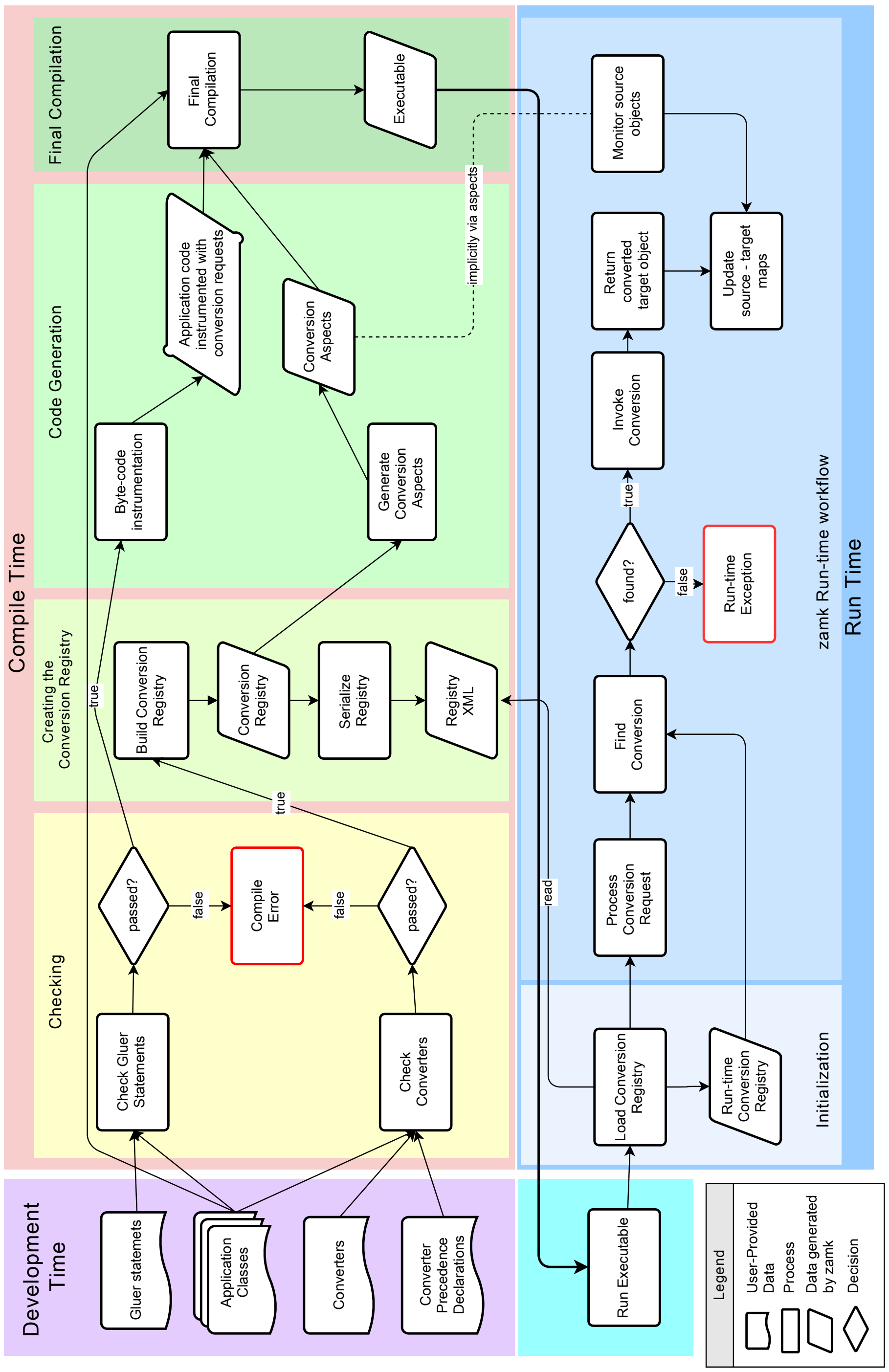


Figure 1.4 – An overview of the *zamk* framework

Compile Time Once this input is provided, the *zamk* compiler work-flow starts. This work-flow is composed of the following sequential phases. The descriptions below follow the same structure and the order as in upper-half of the Figure 1.4.

Checking: There are two checkers which are responsible for checking the *Gluer* files and converter classes. The gluer checker performs the syntax checking; it also checks if the references made in the *Gluer* statement actually exist in the application code. The converter checker performs type checking and well-formedness checking. If any of these checks fails, a compile-error is produced.

Conversion registry: When the converter checking is finished without any problems, *zamk* builds a conversion registry[2] in the form of a data structure to be used in the next step of compile-time work flow; *code generation*. Also during this phase the conversion registry data structure is serialized in XML format creating registry XML.

Code Generation: This step consists of two separate generation processes. Byte-code instrumentation is responsible for inserting *zamk* conversion requests to the places indicated as the binding points defined in the *Gluer* statements. The conversion registry created in the previous step is used to generate the conversion aspects, which are responsible for monitoring the converted objects.

Final Compilation: When the compile-time work flow is complete a final compilation step is performed, which makes sure the instrumented application classes and the generated files do not contain any errors. At the end of this step we obtain a *zamk* run-time ready code, in this figure we assume the final compilation product is an executable.

The *zamk* compile time produces two outputs; registry XML which will be loaded during *run-time initialization* and the compiled code of *zamk* generated classes and application classes.

Run-time *zamk* run-time consists of two parts, a one time *initialization* and a *run-time work flow* which is executed for every conversion request (lower-half of Figure 1.4).

[2]One converter class may include multiple conversions

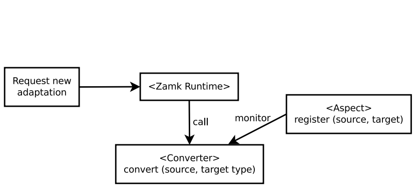
Initalization: The *zamk* run-time starts with an initialization step, which loads the conversion registry that is serialized during runtime. The loading process produces a data structure called run-time conversion registry, which is used by the run-time work flow to locate conversions.

Run-time workflow: The compiled program contains *zamk* conversion requests which trigger the conversion finder. A conversion request contains the source object and the desired target type to which the source object should be converted. The *find conversion* process searches for the correct conversion by using the type information included in the conversion request. If a suitable conversion cannot be found, *zamk* produces a run-time exception indicating the error. If a conversion is found, one of two things can happen. Either the request may result in a new conversion, i.e. it may trigger the process to create a new source–target pair. In order to create a new source–target pair, the corresponding conversion method is invoked and the newly created target object is returned. The conversion aspects are responsible for monitoring the source objects which are associated with a target object. Or *zamk* may find that a request to the same target type was processed before with the given source object. In that case the existing target object is retrieved and returned.

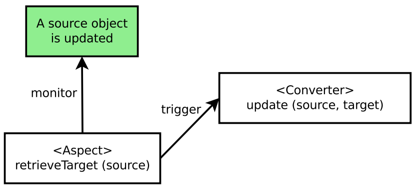
1.3.1 Decomposition of the adapter pattern

In the traditional adapter pattern, the adapter class has an adaptee; if this adaptee's value is changed, it directly effects the return values of the method in the adapter's interface. We have decided to decompose adaptation as a one-time conversion and a series of updates during an objects life-cycle. Our decomposed adapter pattern consists of two parts, first is a class that has two methods, a convert and an update method. The second part is an *aspect* which contains a hashmap of source(adaptee) and target objects. The responsibility of this aspect is to monitor the source (adaptee) objects and update the target objects if a source object changes.

In Figure 1.5a the first step of the adaptation process is shown. A new adaptation requested by giving a source object and a target type. In the traditional adapter pattern this source object is the adaptee and an adapter which is a subtype of the target type is instantiated that aggregates this source object. In our adaptation process this source object is passed onto the convert method of an appropriate


Request new adaptation
<Zamk Runtime>
call
monitor
<Aspect> register (source, target)
<Converter> convert (source, target type)

(a) A new adaptation request


A source object is updated
monitor
trigger
<Converter> update (source, target)
<Aspect> retrieveTarget (source)

(b) Keeping source and target objects synchronized

converter (automatically found by the framework, section 1.3.3.3) and this method returns the corresponding target object, which is initialized according to the values provided by the source object. When a new target object is created, the *zamk* runtime registers the source-target tuple in a hashmap. This hashmap replaces the *has-a* relationship between the adapter and the adaptee.

Once a target object is linked to a source object, they are kept synchronized. This is ensured by the monitor aspect. This aspect monitors the events which change the source objects, when such an event is encountered it retrieves the corresponding target object. Then it triggers the *update* method in the converter class to update the target object that is linked to the changed source object.

In this method of adaptation, there's no need to create an intermediate adapter type. The converter directly creates an object of the target type and keeps it up-to-date. As we will explain in section 1.3.2, the user only has to define the converter class. The monitor aspect is generated during compile-time by *zamk* framework.

1.3.2 Compile-time

In this section we will explain the elements and modules that are involved during the compile-time of the framework. In figure 1.5 a compile-time work-flow is shown. The user is responsible with providing converters that are specific to her application. The converters are required to adhere to a specific structure, which is discussed in section 1.3.2.2. If *zamk*'s built-in dependency injection module is to be used, then the user should also provide a *Gluer* file that includes the necessary *Gluer* statements.

All of the provided input is then checked by *zamk*'s compile-time checker. If any problems are found, then this is reported as a compile error or a warning. When the checking is passed without any problems *zamk*'s code-generator is invoked. *Gluer* files are processed by the byte-code generator, which performs *weaving* of the dependencies to the target components. The converters are processed by the

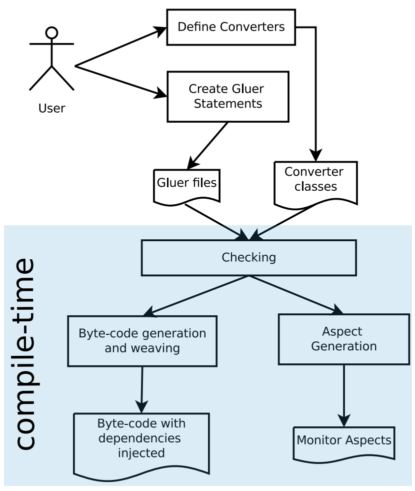


Figure 1.5 – The user provided input and the compile-time workflow of `zamk`

aspect-generator which generated the monitoring and management aspect for each converter, concluding the compile-time work-flow of `zamk`.

1.3.2.1 Gluer DSL

The *Gluer* language is a concise DSL that is designed to declare dependencies between fields and objects. We have developed the *Gluer* language to offer an external, non-intrusive way of declaring bindings. Essentially the *Gluer* is an external dependency injection declaration which creates the objects to be injected, but is connected to an adaptation logic which can process the created objects before injection. Therefore the *Gluer* is not a plain dependency injection language. This is why we chose the keyword `glue` instead of `inject`. A *Gluer* program is stored in a file with the extension '.gluer'.

A simple gluer statement looks like the following:

```
glue field <target-field> with <source-class> [using <converter>]
```

<target-field> The target field is a fully qualified name of a non-static field of a class. It can refer to any type, including primitive types.

<source-class> The source class represents the class to be instantiated and injected to the target field. There are several options for creating this objects.

new The `new` statement is followed by a fully classified name of a class. This means, whenever an object is to be injected, it should be newly created using the *default constructor* of the source class.

single Similar to `new`, `single` statement is followed by a fully qualified name of a class, which is instantiated when an injection is triggered. The difference is instead of creating a new object each time, a *single* object is reused among injections. **TODO:possible use case**

retval Short for “return value”, this keyword is followed by a fully qualified reference to a method, which returns the object we would like to inject. When an injection is triggered, the method is called and the returned value is glued to the injection field.

using keyword The `using` keyword is optional and can be used to override the automated converter finding logic. With this keyword the user can point to a specific converter to be used while converting the source object to the target type, before the injection.

Checks *Gluer* also allows us to perform some compile-time checks to ensure the correctness of the `glue` statements. All of these checks result in compile errors that contain specific information about the place and the cause of the error.

- Target-field is checked to see if it actually exists.
- For creating the source object with the `new` and `single` keywords, the framework requires that the source class contains a no-argument constructor. For the `retval` keyword, the framework checks if the referred method exists.
- The `using` keyword triggers two checks. First one checks if the referred converter exists and the second one checks if the converter is suitable for converting from the source class to the target field.
- If the *Gluer* statements are error free up to this point, then a conflict check is performed to see if any two *Gluer* statements try to inject into the same field.

1.3.2.2 User-defined Converters

The users of *zamk* are responsible for creating a converter, which we classify as a domain-specific aspect. Although the user creates a class, it adheres to a specific structure and it is annotated order to be managed as part of an aspect.

A converter has three important requirements:

1. It should be annotated with the `@Converter` annotation

2. It has to include at least two *static* methods:

   a) A *convert* method that is annotated with `@Convert`. This method takes a single parameter and cannot be void.

   b) An *update* method that is annotated with `@Update`. This method takes two parameters of the same type and is void.

3. There cannot be two convert methods that has the same argument and return types in a single converter. Same is true for the update method.

The convert method contains the logic for converting a source object to a target object. It takes a source object as its single argument and returns the corresponding target object. The update method contains the logic for updating a target object. It takes two arguments of the type target; the first represent the old value of the object and the second the updated value. Inside the method the second argument is used to update the first argument.

Referring back to our example given in section 1.2, a user-defined adapter for the `Cartesian-Polar` conversion that conforms to the requirements above can be defined as shown in listing 1.2. In this example we have defined two methods; `cart2polar` which is annotated as the convert method takes a source object of type `Cartesian` and returns a `Polar` object, the values of which is calculated using the source object. The second method is `updatePolar`, which updates the `old` `Polar` object using the field values of the `newValue` `Polar` object. Note that since the methods are annotated there are no restrictions on the method naming.

This converter can also include methods which convert from `TypeA` to `TypeB` following the same rules. The `@Converter` annotation simply marks a class to be found by *zamk*. When *zamk* finds a converter class, it expects that it has one or multiple *pairs* of convert and update methods. If a convert methods is found to be without an update

```java
@Converter
public class Cartesian2PolarUser{
  @Convert
  public static Polar cart2polar(Cartesian source)
  {
    //calculations
    return new Polar(..));
  }
  @Update
  public static void updatePolar(Polar old, Polar newValue)
  {
    old.r = newValue.r;
    old.the = newValue.the;
  }
}
```

Listing 1.2 – A converter defined for converting a Cartesian object to a Polar object

methods or vice versa, this results in a compile error. The convert-update method pairs should be declared in the same converter. *zamk* does not merge methods from separate converter classes.

If the developer declares another pair of convert-update methods in the same converter which is the inverse of the previously declared one, this conversion is registered as a *two-way* conversion. For example if we add the convert and update methods for `Polar` to `Cartesian` conversion to the converter in listing 1.2 then *zamk* will register a two way conversion between these types. From the developer's perspective implementation requirements do not change, *zamk* handles the operations required to keep converted objects synchronized.

When a pair of convert and update method are found, some type checks are performed. In order to register a conversion *zamk* looks a the convert method's source (single parameter) and target (return) types. The accompanying update method *must* take arguments of exactly the target type. Otherwise a compile error indicating the situation is given to the user.

Converters do not need to use inheritance to perform adaptations. Since all the adaptation related methods are static, they do not need to be initialized to function; they are stateless. Their well-structured definition allows the developer to implement only what is necessary to perform an adaptation. Converters are concise and light-weight structures that encapsulate the adaptation concern.

1.3.2.3 Conversion Registry

During compile-time a converter registry is created and serialized using the annotated converter classes. The procedure for creating the converter registry can be seen in Procedure 1.

Procedure 1 Creating the conversion registry

```
1: procedure createRegistry
2:     $converters \leftarrow$ all classes annotated with @Converter
3:     for all $c \in converters$ do
4:         $cMethods \leftarrow$ all methods in $c$ annotated with @Convert
5:         $uMethods \leftarrow$ all methods in $c$ annotated with @Update
6:         for all $cm \in cMethods$ do
7:             $sourceType \leftarrow$ the argument type of $cm$
8:             $targetType \leftarrow$ the return type of the $cm$
9:             for all $um \in uMethods$ do
10:                if $um.argumentType = targetType$ then
11:                    $conversion \leftarrow (sourceType, targetType, cm, um)$
12:                    add $conversion$ to $record$
13:                    remove $cm$ from $cMethods$
14:                    remove $um$ from $uMethods$
15:                end if
16:            end for
17:        end for
18:        registerTwoWayConversions(record)
19:        add $(c.FQN, record)$ to $registry$
20:    end for
21:    serialize($registry$)
22: end procedure
```

This procedure assumes the converter class is structurally correct, i.e. the methods are properly annotated and there is exactly one matching update method for each convert method in a converter. The createRegistry procedure first finds all classes annotated with `@Converter`. From each converter class it find the convert and update methods and puts them into separate lists. For each convert method found in the converter the variables $targetType$ (the return type of the convert method) and $sourceType$ (the argument type of the convert method) are initialized. Then the matching update method is found by iterating over the list of update methods and comparing their argument the target types. When an update method is found and then the record of the conversion is created.

The $record$ data structure contains the list of $conversion$ items a converter contains. A $conversion$ consists of the the source and the target types, the name of the convert and update methods. The convert and update methods included in the created $conversion$ are removed from the corresponding lists. When all conversions in a converter is found, the $record$ for a single converter class is complete. However at this point the $record$ does not contain any information about two-way converters.

The registerTwoWayConversions(shown in Procedure 2) procedure processes the $record$ list and changes its contents if it contains any two-way conversions. This procedure iterates over the conversion list $record$ and tries to find conversions which have the inverse source and target types. Once a pair of such conversions are found, a $newConversion$ which contains the source and the target types and the convert and update method names of both conversions is created. The $newConversion$ items are collected in a separate list called $newRecord$ and the individual conversions forming a two-way conversion are marked in the $record$ list. After all two-way conversions are found, the marked entries from $record$ are removed and the new entries collected in $newRecord$ are added to list $record$. This procedure is also responsible for assigning the unique IDs to each conversion, which is done in the for loop shown on line 16. This unique ID is later used in the aspect generation as the aspect unique ID. Once the $record$ list is in its final form it is mapped to the $registry$with the converter's fully-qualified name as the key. After the list of converters is exhausted the $registry$ is fully populated. The last operation is the serialization of the $registry$ to an XML file, which is done by the serialize operation at the end of the procedure.

The registry is serialized as an XML file, with the format shown in Listing 1.3. The XML structure adheres to the class structure; multiple conversion tags are enclosed with a converter tag, which takes the fully-qualified name of the converter as a value. The conversion tag marks if the conversion is a two way conversion and contain the source–target types for the conversion, and the convert–update method pair for each conversion direction (source-to-target and target-to-source). The convert and update tags contain the method names for convert and update methods respectively. The tags marked with *1* are the methods responsible for the conversion from source-to-target, and following tags marked with *2* are the convert and update methods for the inverse conversion, if the conversion is indeed a two way conversion.

1.3.2.4 Code Generation

There are two separate code generation modules included in `zamk`. The first one is the byte-code generation and weaving module which

Procedure 2 Finding the two-way conversions

```
1:  procedure registerTwoWayConversions($record$)
2:      for all unmarked $x \in record$ do
3:          for all unmarked $y \in record$ do
4:              if $y.sourceType = x.targetType$ then
5:                  if $y.targetType = x.sourceType$ then
6:                      $newConversion \leftarrow (sourceType, targetType, x.convert,$
7:      $x.update, y.convert, y.update)$
8:                      mark $x$ and $y$
9:                      add $newConversion$ to $newRecord$
10:                 end if
11:             end if
12:         end for
13:     end for
14:     remove marked conversions from $record$
15:     merge $newRecord$ and $record$
16:     for all $r \in record$ do
17:         assign $uid$ to $r$
18:     end for
19: end procedure
```

```xml
<converter = [FQN]>
  <conversion uid=".." twoway = [true|false]>
    <source>[source-type]</source>
    <target>[target-type]</target>
    <convert = "1">[convert-method]</convert>
    <update = "1">[update-method]</update>
    <!--For two-way conversions-->
    <convert = "2">[convert-method 2]</convert>
    <update = "2">[update-method 2]</update>
  </conversion>
  <conversion...
</converter>
```

Listing 1.3 - The XML code for a registry item

```
public privileged [name][source-type]2[target-type]GenAspect
    extends ZamkRuntime{
  private static aspectUID = [..];
  [name][source-type]2[target-type]GenAspect()
  {
    ZamkRuntime.register(aspectUID, map);
  }
  Map<[source-type], [target-type]> map = new
      WeakHashMap<[source-type], [target-type]>();

  pointcut updateObserverThis(Object c): updateObserver(c) &&
      if(c instanceof [source-type]);
  after(Object c): updateObserverThis(c)
  {
    [source-type] obj = ([source-type])c;
    if(map.containsKey(obj)){
      [name].[update-method](map.get(obj),
          [name].[convert-method](obj));
    }
  }
}
```

Listing 1.4 – The code generation template for producing an adaptation-specific aspect

is used to generate code from *Gluer* statements. The second one is the aspect generator, which uses user-defined converters to generate the monitoring and managing aspects, so-called conversion aspects, for each converter.

Byte-code generation and Weaving **TODO:get the details from Arnout** Since *Gluer* is a proof of concept implementation it only supports constructor injections. This is indicated by the `field` keyword. It is also possible to extend the grammar and the byte-code generator to implement setter injections. **TODO:discuss**

Aspect Generator For each convert-update method pair in a user-defined converter a specialized aspect is generated. The template for this aspect is shown in listing 1.4.

Let us explain this template in detail. `[name]` represents the user-defined converter's class name. We concatenate the the source and the target type names to create `[source-type]2[target-type]` and `GenAspect` at the end of `[name]` to create the unique name for the generated aspect. Since a single converter class can include multiple convert-update pairs, the aspect names also include the type information in their names. Every generated aspect extends the abstract aspect `ZamkRuntime`. The details of this aspect will be covered in section 1.3.3.

Each aspect has a unique ID called `aspectUID` (line 2) and a constructor which is called create a *singleton* instance of the aspect (lines 3–6). The `aspectUID` is the conversion unique ID which is assigned during the generation of the conversion registry. Inside this constructor the aspect registers its `map` to the *zamk* runtime with its unique ID, which is used for storing the source–target pairs. The declaration of this `map` is shown on line 7. It is constructed using generics notation; the `[source-type]` is the type of the source object and the `[target-type]` is the type of the target object. These types are determined by looking at the argument type and the return type of the convert method, respectively. The generated aspect declares a single pointcut.

The `updateObserverThis` pointcut selects the join-points when fields of an object which is of `[source-type]` is `set` (line 9). This pointcut reuses a pointcut that is declared in it's super-aspect, called `updateObserver` and narrows the scope of this pointcut by composing an `if` pointcut that checks if the updated object is indeed an instance of the `[source-type]`. The `after` advice that follows uses this pointcut and calls the update method of the user-defined class at the selected join-points(line 14). This operation makes sure the target object associated with the updated source object is updated in the `map`.

The generated aspect for the plotter example's converter shown in listing 1.2 is shown in listing 1.5. The `updateObserverThis` pointcut monitors all the `Cartesian` objects and selects the join-points where they are changed. The after advice following this pointcut, calls the `updatePolar` method of the `Cartesian2PolarUser` converter to update the corresponding `Polar` object.

In case of two-way conversions the generated aspect slightly changes. Instead of a `WeakHashMap` we use a `HashBiMap` (**TODO:cite Google Guava)**. A `HashBiMap` has two underlying `HashMaps` with inverse type parameters and it preserves the uniqueness of its values as well as its keys. The constructor of the aspect does not change.

We generate two `updateObserverThis` pointcuts for each source type. For the plotter example these pointcuts are shown in listing 1.6. The after advices that use these pointcuts also use the corresponding update and convert methods that are declared for each one-way conversion.

The resulting two way aspect for the plotter example is shown in listing 1.7.

```
public privileged Cartesian2PolarUserCartesian2PolarGenAspect
    extends ZamkRuntimeAJ{
 private static aspectUID = "myID";
 Cartesian2PolarUserCartesian2PolarGenAspect()
 {
  ZamkRuntime.register(aspectUID, map);
 }
 Map<Cartesian, Polar> map = new WeakHashMap<Cartesian,
     Polar>();

 pointcut updateObserverThis(Object c): updateObserver(c) &&
     if(c instanceof Cartesian);
 after(Object c): updateObserverThis(c)
 {
  Cartesian obj = (Cartesian)c;
  if(map.containsKey(obj)){
   Cartesian2PolarUser.updatePolar(map.get(obj),
       Cartesian2PolarUser.cart2polar(obj));
  }
 }
}
```

Listing 1.5 – The aspect generated for the Cartesian to Polar converter in listing 1.2

```
pointcut updateObserverThisCartesian(Object c):
    updateObserver(c) && if(c instanceof Cartesian);
pointcut updateObserverThisPolar(Object c): updateObserver(c)
    && if(c instanceof Polar);
```

Listing 1.6 – `updateObserverThis` pointcuts for two-way conversion

Note In *zamk* updates made for one object can trigger other updates, which may result in infinite update loops. For example in the two-way converter case, when a source object is changes its target is also updated, which again triggers an update to the source object which started the update cycle in the first place. This is handled by *zamk* runtime, which has mechanisms to detect such loops. The details of the detection of update loops will be discussed in section 1.3.3.5.

1.3.3 Runtime

zamk runtime is responsible for applying correct conversions and managing source–target object pairs. The *zamk* run-time is triggered by conversion requests. There are two ways to create such requests; first one is the *Gluer* statements which are transformed into *zamk*

```
public privileged Cartesian2PolarUserCartesian2PolarGenAspect
    extends ZamkRuntimeAJ{
 private static aspectUID = "myID";
 Cartesian2PolarUserCartesian2PolarGenAspect()
 {
  ZamkRuntime.register(aspectUID, map);
 }
 HashBiMap<Cartesian, Polar> map =
     HashBiMap.create<Cartesian, Polar>();

 pointcut updateObserverThisCartesian(Object c):
     updateObserver(c) && if(c instanceof Cartesian);
 after(Object c): updateObserverThisCartesian(c)
 {
  Cartesian obj = (Cartesian)c;
  if(map.containsKey(obj)){
   Cartesian2PolarUser.updatePolar(map.get(obj),
       Cartesian2PolarUser.cart2polar(obj));
  }
 }
 pointcut updateObserverThisPolar(Object p):
     updateObserver(p) && if(p instanceof Polar);
 after(Object p): updateObserverThisPolar(p)
 {
  Polar obj = (Polar)p;
  if(map.inverse().containsKey(obj)){
   Cartesian2PolarUser.updateCartesian(map.inverse().get(obj),
       Cartesian2PolarUser.polar2cart(obj));
  }
 }
}
```

Listing 1.7 - The aspect generated for the Cartesian to Polar two-way converter

API calls in the byte-code, second one is including direct references to the *zamk* API in the base-code. Both of these operations trigger the same conversion finding process. Once the desired target object is created or retrieved (in case the given source object is already associated with a target object), it is returned to the owner of the request.

1.3.3.1 Initialization

In order to process conversion requests, *zamk* performs an initialization step which consists of loading the conversion registry and conversion aspect registration.

zamk creates a registry of conversion during compile-time (Section 1.3.2.3). At the beginning of the runtime this registry is loaded by parsing the XML file which contains the conversions. As mentioned before the XML file contains data about two kinds of conversions, one-way and two-way. Each `<conversion>` tag is mapped to a `Conversion` object, which is the parent type for `OneWayConversion` and `TwoWayConversion`. The class structure is shown in Figure1.6.

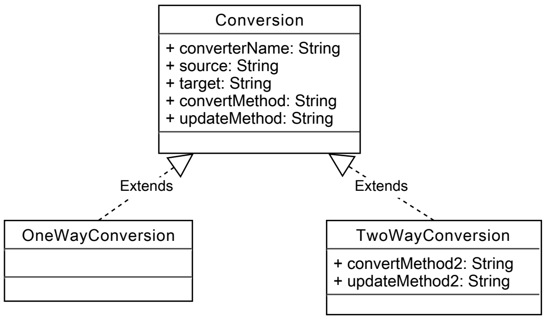


Figure 1.6 – The run-time `Conversion` objects type hierarchy

From the conversion data two separate `HashMaps` are generated; `conversionMap` and `sourceMap`. The `conversionMap` is a `<String, Conversion>` mapping; the conversions are mapped with their unique ID. The `sourceMap` is mapping from `<Class<?>, ArrayList<String> >`, where `Class` represents the source type of the conversion and the `ArrayList<String>` is a list of unique IDs of conversion which convert from the source type declared as the key. The `sourceMap` is constructed to avoid iterating over the whole `conversionMap` while searching for a conversion.

Conversion aspects register their source–target maps during initialization. This is done by calling the `register(String, Map)` method of the `ZamkRuntime`. The `String` value is the unique ID of the conversion and the `Map` is the `HashMap` or the `BiHashMap` the aspect contains. The individual conversion maps are stored in a `HashMap` called `mapPerConversion`. Even though the maps are declared and initialized in the conversion aspects, they are managed by the `ZamkRuntime`.

1.3.3.2 *zamk* Conversion Requests

A conversion request passes on the source object and a desired target type, in return *zamk* runtime provides an object which is initialized based on the value of the source object. The conversion requests are communicated to *zamk* with `getConvertedValue(Object, Class<?>)` method, which is a static method of `ZamkRuntime`. The pseudo code for `getConvertedValue` method is shown in Procedure 3. The step is to find the the suitable conversion for the given input by calling the `findConversion` method (line 3). If this operation is successful, the unique ID of the found conversion is returned and stored in the uid variable). The details of the `findConversion` method is discussed in section 1.3.3.3. Using uid the source–target map for the conversion is retrieved from `mapPerConversion` hash-map (line 4). First we check if there is a source–target entry in the conversion map for the given source object, i.e. if the source object has been converted before using this conversion. If this is true, then the corresponding target object is retrieved and returned. If the request triggers a new conversion, then the `invokeConversion` method is called with the uid and the source object to create a new target object. The new source–target pair is added to the map and the target object is returned.

Procedure 3 The `getConvertedValue` method

```
 1: procedure getConvertedValue($source, targetType$)
 2:     $sourceType \leftarrow$ source.Class
 3:     $uid \leftarrow$ findConversion(sourceType, targetType)
 4:     $map \leftarrow$ mapPerConversion.get(uid)
 5:     if $source \in map$ then
 6:         return $map.get(source)$
 7:     else
 8:         $target \leftarrow$ invokeConversion(uid, source)
 9:         add $(source, target)$ to $map$
10:         return $target$
11:     end if
12: end procedure
```

1.3.3.3 Finding a Conversion

The `findConversion` method shown in Procedure 3 implements an algorithm, which uses type information to find the *closest* conversion among *eligible* conversions for the given source type and the expected target type.

Given a conversion request from type X to type Z, the requirements for an an eligible conversion are as follows:

E-1 The conversion source type is exactly the same as or is a super type of type X and,

E-2 the conversion target type is exactly the same as or is a sub type of Z type.

The closest conversion is characterized as follows:

C-1 Among eligible conversions its source-type is the closest to the actual type X object given in the conversion request.

C-2 Among the eligible conversions with the same source-type proximity, its target-type is the closest to the Z type given in the conversion request.

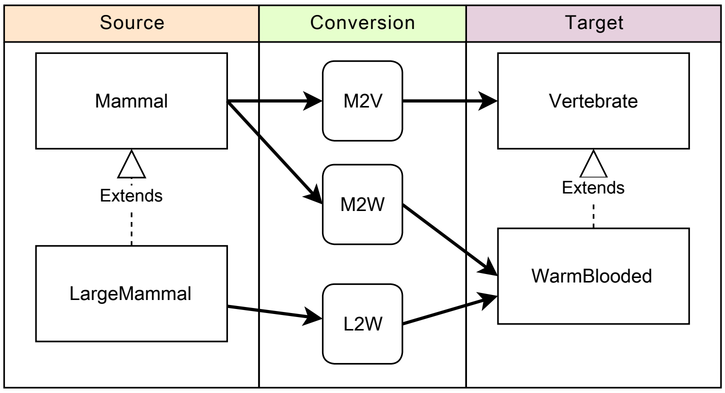


Figure 1.7 – Two type hierarchies for representing animals and conversions between them

Let us clarify these descriptions with an example. In Figure 1.7 two separate hierarchies that represent different classifications for animals. Conversions are defined between the types of the hierarchies. Consider the conversion request `getConvertedValue(largeMammal, Vertebrate.class)`. According to our description of the eligible conversions, we can check each conversion to determine if they are in fact able to perform this conversion. Starting from the bottom of the figure:

- The `L2W` conversion converts from `LargeMammal` to `WarmBlooded`. Since the type of the source object (`LargeMammal`) exactly matched the conversion's source type (`LargeMammal`) it satisfies E-1, since the target type of this conversion is `WarmBlooede` which is a sub-type of `Vertebrate`, the conversion also satisfied E-2. Hence we conclude that the conversion is eligible.
- The conversion `M2W` satisfies the E-1 since `Mammal` is a super-type of `LargeMammal`, it also satisfies E-2 since its target type `WarmBlooded` is a sub-type of `Vertebrate`.
- Similarly to `M2W` the conversion `M2V` satisfies E-1 since its source type `Mammal` is a super type of `LargeMammal` and its target type

`Vertebrate` is which is exactly the same as the target type given in the conversion request, satisfying E-2.

From this analysis we conclude that all three conversions are eligible for this conversion request.

Conversion	Source	Target
L2W	0	1
M2W	1	1
M2V	1	0

Table 1.1 – The type distances of conversion's source–target types to the source–target types given in the conversion request `getConverted-Value(mammal, WarmBlooded.class)`

To identify the closest conversion, we look at the hierarchical distances of the conversion types to the types given in the conversion request. For this example the distances are listed in Table 1.1. Even though `L2W` and `M2V` have the same combined distance, the closest conversion is `L2W` since closeness of the source type has priority while deciding on the closest conversion. The information of the conversion comes from the source object. For a conversion to be more accurate, the type of the given source object should be as close as possible to the source type it converts from, i.e the source object should be as specialized as possible. That's why the closeness checks prefers the closeness of the source type over the closeness of the target type.

These operations are implemented in `findConversion`; pseudo code is shown in Procedure 4. The procedure starts by creating an empty list named $eligibles$ which will contain the eligible conversions. In order to determine the these conversions, we first need to find the super types of the source and target types by walking their type hierarchy. This is done by the method `getAllSuper-Types` which uses reflection to populate the full set of super types (classes and interfaces) of a given type. On lines 3 and **??** this method is called and the returned lists are stored in the respective variables; $superTypeSource$ and $superTypeTarget$. Note that when the `java.lang.Object` is passed as an argument to this method, it returns a list that contains a set that contains the single element `java.lang.Object`.

The while loop (lines 4– 16) iterates over the $superTypeSource$ set and checks if that source type is associated with any conversions; in the `if` statement on line 5 checks if $sourceMap$'s key set contains the $sourceType$. When this expression evaluates to true, we retrieve the candidate conversions from the $sourceMap$ and store them into the list

Procedure 4 The procedure for finding the most suitable conversion

```
1: procedure findConversion($sourceType, targetType$)
2:     $eligibles \leftarrow$ empty list
3:     $superTypesSource \leftarrow$ getAllSuperTypes(sourceType)
4:     while $superTypesSource.hasNext$ do
5:         if $sourceType \in sourceMap.keySet$ then
6:             $candidates \leftarrow sourceMap.get(source)$
7:             for all $id \in candidates$ do
8:                 $conversion \leftarrow conversionMap.get(id)$
9:                 if $conversion.getTargetType \subset targetType$ then
10:                    $eligibles.add(id)$
11:                end if
12:            end for
13:        else
14:            $sourceType \leftarrow superTypesSource.next$
15:        end if
16:    end while
17:    if $eligibles = \emptyset$ then
18:        Runtime Exception, no conversions found
19:    else
20:        $found \leftarrow$ findClosest(eligibles, sourceType, targetType, superTypesSource)
21:    end if
22:    if $found > 1$ then
23:        return resolvePrecedence(found)
24:    else
25:        return $found(0)$
26:    end if
27: end procedure
```

candidates (line 6). The *candidates* list contains the conversion IDs that converts from the *sourceType*. The conversions pointed by *candidates* only satisfy the source type criteria of an eligible conversion, therefore we still need to check the target types of these conversions to detect if they are indeed eligible. This detection is done in the for loop on lines 7– 12. For each unique id in the list *candidates*, we retrieve the corresponding conversion from the *conversionMap* (line 8) and store it into the variable *conversion*. Then we check if the target type of this *conversion* is assignable from the *targetType* passed to the conversion request (line 9). If this expression is true then we add the id of the *conversion* to the list of *eligibles*.

When the if statement on line 5 evaluates to false, then we set the variable *sourceType* to the next element in *superTypesSource* and reiterate the process until there are no more elements left in

$superTypesSource$. At the end of this iteration, we obtain a list of eligible conversions stored in $eligibles$. The next operation is to find the closest conversion from this list. First we check if the list $eligibles$ is empty (line 17); if it is an empty list then we throw a run-time exception, indicating there are no suitable conversions found for the given source and target types. If the list is non-empty then we invoke the method `findClosest` (line 20) and store the returned conversion id values in the variable $found$. It is possible that there are more than one equally close conversions for a given request, if this is the case we invoke to `resolvePrecedence` method (line 23) and return the value obtained from this method. If the `resolvePrecedence` method cannot resolve the ambiguity in the $found$ list then it throws a run-time exception, indicating there is not enough information to resolve the precedence. If the `findClosest` method returns a single conversion then we simply return the first element of the list $found$.

`getAllSuperTypes` method This is a utility method which returns the super-types of a type. We use reflection to walk the type hierarchy of the given type. The source code for this method is shown in Listing 1.8. On line 2 we create an empty list which will hold the super types of `sourceType`. Then we add the `sourceType` to this list as the first element (line 2). On line 4, by using the reflection method `getSuperClass` we store the super type of `sourceType` to the variable `superType`. The reflective method `getInterfaces` is used to return the interface a type implements, which is called on line 9. These interface are added to the list `sourceTypeHier` by iterating over the list `itc`. The while loop shown on lines 10– 18 is only executed if the variable `superType` is not `Object`, since that is the root type for all objects in Java. Inside the while loop the same operations described above are performed until the next source type is `java.lang.Object` which is added to the list after the loop terminates. Finally the `sourceTypeHier` list is returned.

`findClosest` method This method find the closest conversion from the list of eligible conversions. In order to perform this task, the method takes the eligibles list, source and target types and the lists of their super types as an argument. The method creates the table shown in Table 1.1 for each eligible conversion, and decides on the closest according to the closest conversion criteria given at the beginning of this section. **TODO:pseudo code**

```java
private static List<Class<?>> getAllSuperTypes(Class<?>
    sourceType) throws ClassNotFoundException {
  List<Class<?>> sourceTypeHier = new ArrayList<Class<?>>();
  sourceTypeHier.add(sourceType);
  Class<?> superType = sourceType.getSuperclass();
  Class<?>[] itc = superType.getInterfaces();
  for(int i=0; i<itc.length; i++)
  {
    sourceTypeHier.add(itc[i]);
  }
  while (!superType.getName().equals("java.lang.Object")) {
    sourceTypeHier.add(superType);
    itc = superType.getInterfaces();
    for(int i=0; i<itc.length; i++)
    {
      sourceTypeHier.add(itc[i]);
    }
    superType = superType.getSuperclass();
  }
  sourceTypeHier.add(Class.forName("java.lang.Object"));
  return sourceTypeHier;
}
```

Listing 1.8 – The source code for the `getAllSuperTypes` method

`resolvePrecedence` method When `findClosest` method finds multiple equally close conversion for a conversion request, the `resolvePrecedence` method is invoked. The task of this method is to process the precedence information given during compile-time and decide which of the conversions should be applied to a conversion request. **TODO:pseudo code**

1.3.3.4 Target Object Creation and Retrieval

In section 1.3.3.2 we have mentioned the steps taken after a conversion is found. In this section we will discuss these steps in detail.

We show a partial pseudo code of the `getConvertedValue` in Procedure 5. When `findConversion` method returns a conversion id, we use it to retrieve the map for that conversion. Then we check if the map contains the $source$ object, if it does we return the target object, associated with this $source$. This operation is called target object retrieval. By having this operation, we ensure that the state of a conversion is preserved, similar to an object adapter inferring its state from its adaptee.

If the $source$ object was not converted before with found conversion, we need to create a new source–target pair. To do this we must

Procedure 5 Partial view of the `getConvertedValue` method

```
 1: procedure getConvertedValue($source, targetType$) ...
 2:     $map \leftarrow$ mapPerConversion.get(uid)
 3:     if $source \in map$ then
 4:         return $map.get(source)$
 5:     else
 6:         $target \leftarrow$ invokeConversion(uid, source)
 7:         add $(source, target)$ to $map$
 8:         return $target$
 9:     end if
10: end procedure
```

```java
1  public static <T, U> T invokeConversion(String uid, U source)
       {
2    T trgt = null;
3    Conversion conversion = conversionMap.get(uid);
4    try {
5      Class converter =
           Class.forName(conversion.getConverterName());
6      Class[] argTypes = new Class[] { source.getClass() };
7      Method convert =
           converter.getDeclaredMethod(conversion.getConvertMethod(),
           argTypes);
8      trgt = (T) convert.invoke(null, source);
9    } catch (Exception e) {
10     e.printStackTrace();
11   }
12   return trgt;
13 }
```

Listing 1.9 – The `invokeConversion` method which reflectively invokes the convert method of a given conversion

call the `invokeConversion` method, which reflectively invokes the convert method of the found conversion. The source code for this method is given in listing 1.9. The argument `uid` is used to retrieve the conversion from the `conversionMap` (line 3. A `Conversion` object contains the fully-qualified name of the converter it is contained in and the names of the convert and update methods. In a `try-catch` statement, we first create a `Class` object for the converter that contains the `conversion` (line 5). After constructing an array for argument types of the convert method (line 6), the convert method for the `converter` is loaded from the `Class converter` (line 7). The retrieved method is invoked by calling the `invoke` method and passing on the argument `source` (line 8). The result of the method invocation is stored in the variable `trgt` which was initially null and returned.

1.3.3.5 Object Synchronization

The source–target pairs kept in the conversion maps are maintained by conversion aspects. We have previously discussed the structure and the members of these aspect in section 1.3.2.4. A conversion adapter contains a pointcut (`updateObserverThis`) which monitors all of the changes made to the objects of source type. When this pointcut matches, the change object is bound. The after advice that uses this pointcut, looks at its source–target map to check if the map contains the bound object as a key. If this is the case, then the corresponding target object is updated by creating a new object of target type and passing the values of the newly created object to the existing target object. The copying of the values is performed by the update method of the conversion.

1.3.3.6 Run-time API

zamk run-time API offers two overloaded methods to access its functionality.

- `getConvertedValue(U source, Class<T> targetType)` : This method invokes the fully-automated functionality of *zamk*. The run-time steps mentioned in the sections before take place, and the converted value is returned.
- `getConvertedValue(U source, Class<T> targetType, String using)` : Calling this method is equivalent to adding a `using` clause in *Gluer*. The `String` argument should be the fully-qualified name of the conversion. In this method the `findConversion` is not called.

TODO:expand this section

1.4 Related Work

1.5 Conclusion

Bibliography

[1] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications*, pages 161–173. ACM Press, 2002.