Chapter 1

Introduction

In this thesis we explore new modularization paradigms and component integration technologies for handling the introduction of new features into legacy systems. This introductory chapter sets the context for this thesis and presents an overview of our research domain.

- In component-based design, a *component* is defined as self-contained structure with a clear function which communicates through a pre-determined interface.
- In a smaller scale, this is anologous to the class concept in the object-oriented design paradigm.
- In fact a component can even be a single class.
- Such modules are used to decompose software into maintainable and reusable parts.
- In order to obtain a fucntional software, modules which contain various features of a software should be integrated.
- The software architecture defines how these modules are connected to each other.
- However the requriements of software change after its deployement.

Chapter 2

The Adapter Framework

2.1 Introduction

TODO:Assumptions should be explicit in the beginning Changing requirements force software systems to include new components to extend their functionality. Software components are self-contained entities with a well-defined interface and behavior. For two components to co-operate they must be *integrated* through a compatible interface. When components are developed separately the compatibility is not a given. Therefore it should be, often manually, established through additional programming.

The integration task also entails creating dependencies between the components. These dependencies are... Once the object is represented through a compatible interface it is assigned to a target field. This integration also requires keeping the referenced data synchronized. ...

The problem of incompatible interfaces is a common software engineering problem and a possible solution is captured in the *adapter pattern* (cite Gamma).

Gamma et al. defined the adapter pattern as follows... one object makes a reference to a data value it requires a specific type, a target type. another object repersents the desired data and behaavior but this object is an insance of a source type, which is different of the target type then the adapter pattern tells you how to make this source object compatible with the expected target type. Adapter implements the target type and wraps the source object, it implments the methods of the target type by invoking methods defined in the source type. **TODO:make it better**The adapter pattern makes two interfaces compatible by adapting the instances of a source interface

to work with a target interface. This means when an object of the source interface should be assigned to an field of target interface, an adapter object which wraps the source object and implements the target interface is created and passed on as the target object.

The structure which establishes this compatibility is called an adapter and the source interface is referred as the *adaptee*. An adapter introduces a level of indirection between the incompatible interfaces.

There are some issues attached to the traditional adapter pattern. A class adapter pattern is most efficiently implemented with a language that allows multiple inheritance. Developers of the ingle inheritance programming languages need to use workarounds to achieve the same effect. For example in Java multiple inheritance is simulated with the usage of interfaces since a class can implement multiple of them. In order to inherit from two concrete types using Java, we need to extract the interface of one of the types and add this interface to the list of super-interfaces. **TODO:Limitation, we don't always have control over source code** Then the concrete implementation of the interface is repeated in the sub-class.

TODO:There are two problem statements, better structuring separate into two paragraphs The implementation of the object adapter pattern is more flexible, since it only needs to inherit from the target type. An issue is that the adapter pattern adds a level of indirection between the source and the target, which needs to be maintained **TODO:Maintained why?**. The last issue is related to 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 need for specific references has two side effects; first, the implementation contains a direct reference to the adapter type instead of just the source and the target types, which introduces an additional maintenance efforts. **TODO:It should be explicit that the user wants to reuse the adapter** Second, the user is required to know exactly which adapter is responsible for a establishing type compatibility. This may be a problem when adapters are not planned to be reusable and documented poorly. One has to search the type hierarchy.....to find the classes implementing the target type... In such cases the information about which interfaces are adapted 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. **TODO:How would you know if it is an adapter if it is not indicated with naming conventions?** 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.

Let us summarize the problems we have identified so far:

P-1 Implementation of the adapter pattern can be hindered by programming language properties.

P-2 Adapter pattern introduces additional dependencies, which increases maintenance efforts.

P-3 Adapters require additional knowledge and effort from the side of the developer to be used properly.

In light of the issues discussed, we have devised the requirements below. The first three requirements refer to a specific problem in the above itemization. The last two requirements are concerned with the binding.

R-1 Inheritance-free adapter structures. (P-1)

R-2 Use adaptation without creating dependencies to specific adapter classes. (P-2)

R-3 Adapters are found automatically given a source object and a target type. (P-3)

R-4 Means to separate the adaptation logic from the integration logic during development time.

R-5 Binding mechanisms that provide loose coupling, which can work with the adaptation logic.

In order to satisfy these requirements we have designed the `zamk` framework, which unites dependency injection with *under-the-hood* adaptation logic. For our framework we do not employ the traditional adapter pattern and we have 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

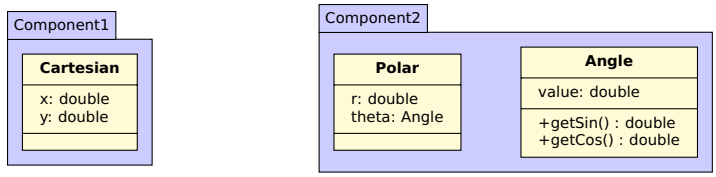


Figure 2.1 – UML diagram for the two components

intertwined with the adaptation logic which uses the conversion 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 be explained throughout the chapter.

2.1.1 Motivating Example

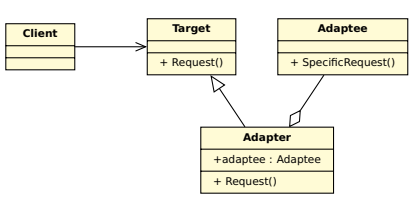
In the previous section we have identified the problems attached to the traditional adapter pattern[1] and we have come up with requirements for a solution that would remedy these problems. In this section we will illustrate the said problems with an hypothetical example.

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 has 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 2.1.1). A new requirement is received which states that the software must also support polar coordinates, and the user must be able to view plot points in a selected view. (Cartesian or polar).

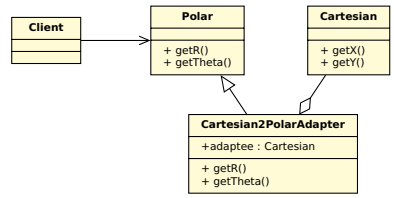
In order to support polar coordinates, a new component which contains classes to represent such data is introduced (`Component2` in Figure 2.1.1). The class `Polar` contains two fields `r` representing the radius and `theta` of type `Angle`. This component is to be integrated with the `Cartesian` component. 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 2.2a. 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

[1] Throughout this chapter, the term adapter pattern will refer to the object adapter pattern. References to 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

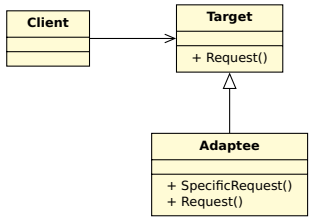
```
1  public class Cartesian2PolarAdapter extends Polar{
2    Cartesian adaptee;
3    public Cartesian2PolarAdapter(Cartesian c) {
4      this.adaptee = c;
5    }
6    public double getR()
7    {
8      return Math.sqrt(Math.pow(adaptee.getX(), 2) +
           Math.pow(adaptee.getY(), 2)));
9    }
10   public Angle getTheta()
11   {
12     return new
           Angle(Math.atan(adaptee.getY()/adaptee.getX())));
13   }
14 }
```

(c) The implementation for the `Cartesian2PolarAdapter`

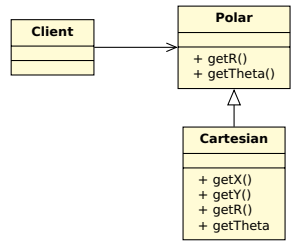
Figure 2.2 – The diagram of the object adapter and the corresponding Java implementation

to polar adapter (`Cartesian2PolarAdapter`) which takes a `Cartesian` object as an *adaptee* and extends the `Polar` class to override its methods (Figure 2.2b). An implementation for this adapter is given in listing 2.2c.

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 diagram shown in Figure 2.3a. 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 2.3c. The inter-type declaration on line 3 declares the inheritance relation and the subsequent method implementations



(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

```
1  public aspect Cartesian2PolarAdapter {
2
3    declare parents: Cartesian extends Polar;
4
5    public double Cartesian.getR()
6    {
7      return Math.sqrt(Math.pow(this.getX(), 2) +
           Math.pow(this.getY(), 2));
8    }
9
10   public Angle Cartesian.getTheta()
11   {
12     return new Angle(Math.atan(this.getY()/this.getX()));
13   }
14 }
```

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

Figure 2.3 – The diagram of the object adapter and the corresponding Java implementation

are woven into the `Cartesian` class.

The obvious problem with this implementation is that, since `Polar` is a concrete type, 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. Another limitation comes from the semantics of inter-type declarations. Only *sibling* types, i.e. types which share a parent class, can be used in an inter-type declaration. In this example both classes' super-type is `Object`, which allows us to use the inter-type declaration.

In order to integrate the polar coordinates into our plotting software, we need to alter some code. According to the selection made in the GUI, the information box should display the coordinate value

```
2  public void viewPointValue(Point selected)
3  {
4    if(GUI.format == CARTESIAN)
5      GUI.createNewValueBox(selected.loc(),
           selected.getCoordinates().toString());
6    else if(GUI.format == POLAR)
7    {
8      Polar p = new
           Cartesian2PolarAdapter(selected.getCoordinates());
9    GUI.createNewValueBox(selected.loc(), p.toString());
10   }
11 }
```

Listing 2.1 – The integration of Polar coordinates

in the correct format. An hypothetical example for the integration code is given in Listing 2.1. 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 developer performing the integration has to know all the specific classes that are responsible for adaptations. If the adapter class is not known, then the names of the adapter classes gives the developer hints about the types the adapter adapts from and to. However depending solely on naming is error-prone. In this case the developer has to go into the class to understand the types included in the adapter. This results in a slower integration process.

In this section we have used the plotter example to illustrate the problems we have identified in Section 2.1. Firstly we have described how programming languages can affect the implementation of the adapter pattern by using Java and AspectJ (P-1). Secondly we have shown the disadvantages of introducing adapter dependencies in the integration code, by using the introduction `Polar` view extension to the plotter software (P-2). Lastly, using the same example, we have discussed why the developer needs adapter specific knowledge and how it slows down the integration process (P-3).

2.2 Approach

In this section we will explain the details of the *zamk* framework which is designed according to the requirements specified in Section 2.1.

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 do not need use sub-class any of the types to implement adaptations. Converter classes 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 does not 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.

Figure 2.4 shows 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; they 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 2.2.2.1.

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

Converters: 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 2.2.2.2.

Converter Precedence Declarations: these are used to resolve conflicts when more than one conversion is found for a specific

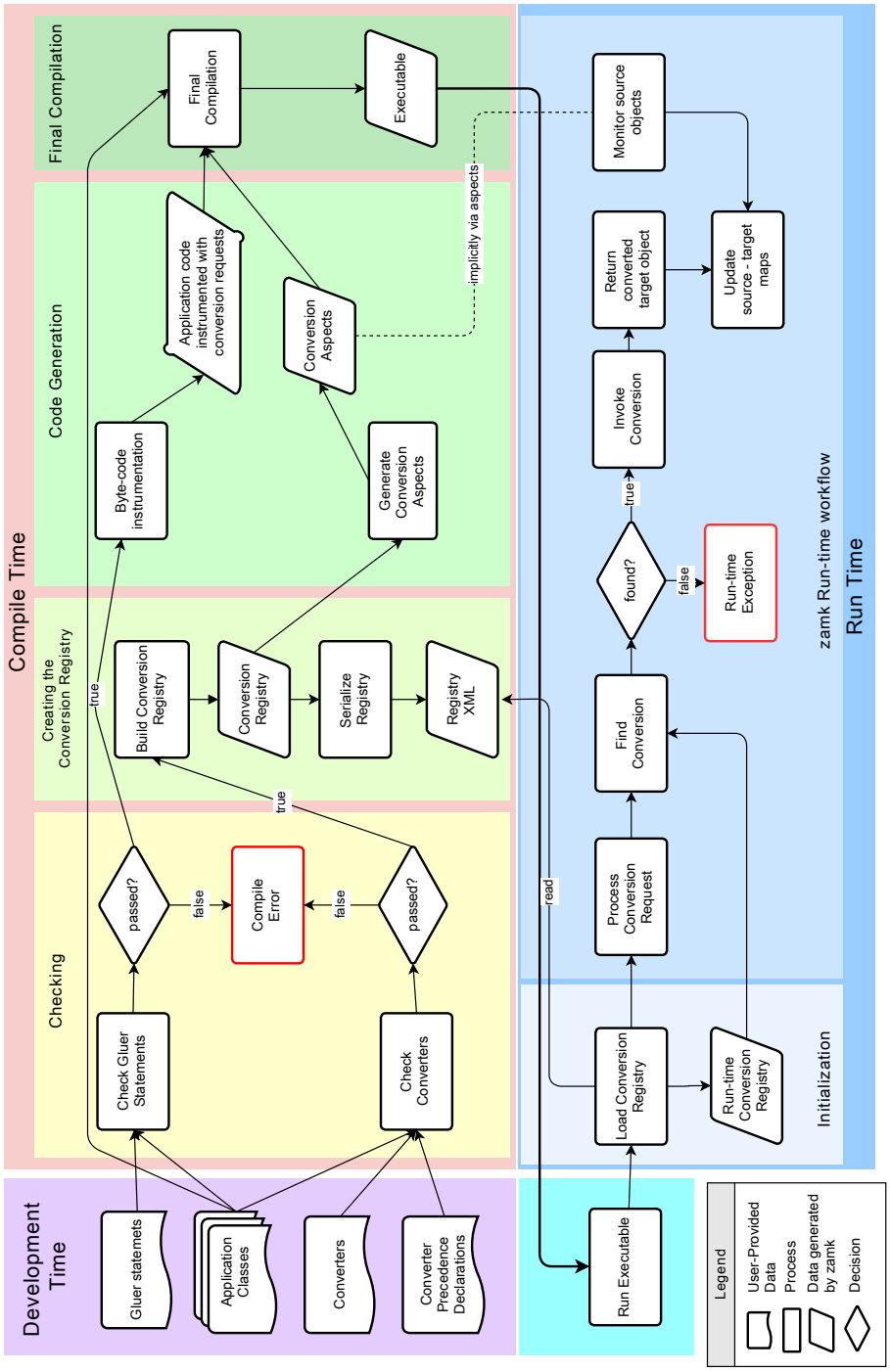


Figure 2.4 – An overview of the *zamk* framework

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; plain Java statements. 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*.

Compile Time Once the 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 2.4.

Checking: There are two checkers; one responsible for checking the *Gluer* files and the other for 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 the 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 that is 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* runtime ready code, in this figure we assume the final compilation product is an executable.

[2] One converter class may include multiple conversions

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 2.4).

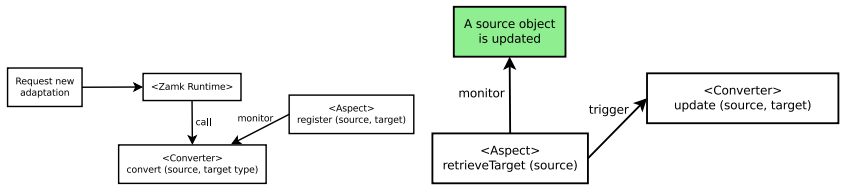
Initalization: The *zamk* run-time starts with an initialization step, which loads the conversion registry that is serialized during runtime. The loading process produces a data structure called run-time conversion registry, which is used by the run-time work flow to locate conversions.

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. Alternatively, *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.

2.2.1 Conversions Instead of Adapters

In the traditional adapter pattern, the adapter class has an adaptee; if this adaptee's value is changed, it directly affects the return values of the methods that are implemented by the adapter. In order to eliminate the dependency to the adapter type, we have created the notion of conversions. Conversions are defined in user-provided converter classes. Conversions simulate adaptation as a one-time conversion and a series of updates during an objects life-cycle.

A conversion consists of two parts, the first part consists of two user-defined methods: a convert and an update method. The second



(a) A new adaptation request (b) Keeping source and target objects synchronized

part is an *aspect*, so-called conversion aspect, generated by *zamk* which contains a hash-map of source (adaptee) and target objects. The conversion aspect is generated during compile-time[3]. The responsibility of this aspect is to monitor the source (adaptee) objects and update the target objects if a source object changes. The update operation is performed by calling the user-defined update method.

In Figure 2.5a the first step of the conversion process is shown. A new conversion 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 conversion process this source object is passed onto the convert method of an appropriate conversion (automatically found by the framework, section 2.2.3.3) 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 pair in a hashmap. This hashmap represents 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 generated conversion aspect. The conversion 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 is 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 2.2.2, the user only has to define a conversion in a converter class.

[3] Please refer to the code generation step of Figure 2.4

2.2.2 Compile-time

In this section we will explain the elements and modules that are involved during the compile-time of the framework. In figure 2.5 a simplified version of the compile-time work-flow is shown. In this figure we have numbered the steps taken in the compile-time, since they are performed sequentially.

As mentioned before, the developer 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 2.2.2.2. The correctness of the converters are checked by the process Check Converters, in the Checking step. All of the processes in this step take the application classes as input, since the checking operation uses these classes to investigate the existence of the dependencies. *Gluer* files are checked by the Check Gluer Statements process. This process, the related processes and the input and outputs are shown in dashed style, since providing *Gluer* files is an optional step. As mentioned before it is also possible to call *zamk* API directly from Java.

Once the provided input is checked and proved to be error-free, the checked converter classes are input to the Build Conversion Registry process, inside the Creating the Conversion Registry step. The processes included in this step are also numbered, since the Serialize Registry process requires the conversion registry data structure produced by the Build Conversion Registry process.

The conversion registry data structure is also passed to the third step Code Generation, which contains two processes. First one is the optional process Byte-Code Instrumentation, which takes the checked *Gluer* files as input. This step is not performed if no *Gluer* files are provided. Second one is the Generate Conversion Aspects process, which takes the conversion registry as input and outputs the conversion aspects.

The fourth and the final step is the compilation of all code that is generated by *zamk* and provided by the developer.

2.2.2.1 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 *Gluer* is an external dependency injection declaration language which creates the objects to be injected and is connected to an adaptation logic which can process the created

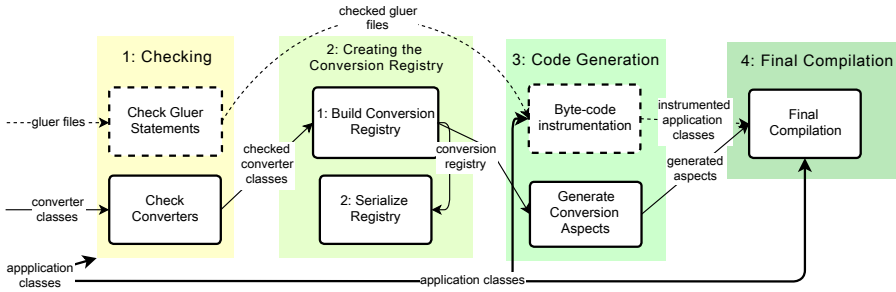


Figure 2.5 – The compile-time workflow and dataflow of *zamk*

objects before the injection happens. A *Gluer* program is stored in a file with the extension '.gluer'.

Syntax and Semantics We chose the keyword `glue` instead of `inject` since *Gluer* does more than dependency injection.

The syntax of the *Gluer* statements is defined as follows:

```
glue <target-field> with
[[new|single] <source-class> | retval <method-ref>]
[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 fully-qualified name of 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.

<method-ref> It is also possible to use the return values of methods as source objects by referring to the fully-qualified method name of a class.

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 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 *zamk* performs some compile-time checks to ensure the correctness of the *Gluer* statements. The checks which result in compile errors 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. The first one checks if the referred converter exists and the second one checks if any of the conversions in that 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.

2.2.2.2 User-defined Converters

The users of *zamk* are responsible for creating a converter, which we classify as a domain-specific aspect. Converter classes contains conversions; they must adhere to a specific structure and they are annotated in order to be managed by *zamk*.

A converter has three important requirements:

1. It must be annotated with the `@Converter` annotation
2. It has to include at least one conversion which is comprised of two *static* methods:

a) A *convert* method that is annotated with `@Convert`. This method takes a single parameter and must return an object value.

b) An *update* method that is annotated with `@Update`. This method takes two parameters of the same type and does not return any value.

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

Since the methods are annotated there are no restrictions imposed by *zamk* on the method naming. 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 creates the corresponding target object. This method is invoked by *zamk* when an adaptation is requested for a *new* source object, i.e a source object that was not adapted before to a given target type. If the requested adaptations has been performed before and *zamk* has a matching source–target pair in the registry, then the existing target object is returned.

The update method contains the logic for updating a target object. It takes two arguments of the type target; the first is the existing registry object which will be updated due to its associated source object's state change. The second argument is the updated state of the registry object based on the new state of the associated source object. In the update method, the developer specifies how an object of the target type is assigned a new state. The reason we update the values of an existing object is, if we simply replace the object with a new one then the dependencies to the old object will be outdated. Note that if we implemented this method in a language that supported pointers, then we would simply keep a registry of pointers and update the values they point to. However with Java, we require that an additional update operation is defined.

It is also possible to declare two-way conversions in a converter class. For a conversion from A-to-B, if the inverse conversion B-to-A is defined in the same converter class, than *zamk* registers this as a two-way conversion.

Referring back to our example given in section 2.1.1, a user-defined adapter for the `Cartesian-Polar` conversion that conforms to the requirements above can be defined as shown in listing 2.2. In this example we have defined two methods; `cart2polar` which is annotated as the convert method. It takes a source object of type `Cartesian` and returns a `Polar` object, the `Polar` object is created

```
1  @Converter
2  public class Cartesian2PolarUser{
3    @Convert
4    public static Polar cart2polar(Cartesian source)
5    {
6      double r = Math.sqrt(Math.pow(source.getX(), 2) +
           Math.pow(source.getY(), 2))
7      Angle a = new
           Angle(Math.atan(source.getY()/source.getX()));
8      return new Polar(r,a);
9    }
10   @Update
11   public static void updatePolar(Polar registryObject, Polar
         newValue)
12   {
13     registryObject.r = newValue.r;
14     registryObject.the = newValue.the;
15   }
16 }
```

Listing 2.2 – A converter defined for converting a Cartesian object to a Polar object

using the source object (lines 6 –8). The second method is `updatePolar`, which updates the `registryValue Polar` object using the field values of the `newValue Polar` object. If we add the convert and update methods for `Polar` to `Cartesian` conversion to the converter in listing 2.2 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.

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 method is found to be without an update method or vice versa, this results in a compile error. The convert-update method pairs must be declared in the same converter. *zamk* does not merge methods from separate converter classes.

When a pair of convert and update method are found, 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, since the update method will only be used with that exact type by *zamk*. 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 focus the implementation efforts on what is necessary to perform an adaptation. Converters are concise and light-weight structures that encapsulate the adaptation concern.

2.2.2.3 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 ← all classes annotated with @Converter
3:      for all c ∈ converters do
4:          cMethods ← all methods in c annotated with @Convert
5:          uMethods ← all methods in c annotated with @Update
6:          for all cm ∈ cMethods do
7:              sourceType ← the argument type of cm
8:              targetType ← the return type of the cm
9:              for all um ∈ uMethods do
10:                 if um.argumentType = targetType then
11:                     conversion ← (sourceType, targetType, cm, um)
12:                     add conversion to record
13:                     remove cm from cMethods
14:                 end if
15:             end for
16:         end for
17:         registerTwoWayConversions(record)
18:         add (c.FQN, record) to registry
19:     end for
20:     serialize(registry)
21: 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 source and the target types, the name of the convert and update methods. The update method 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 2.3. 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

Procedure 2 Finding the two-way conversions

```
1: procedure registerTwoWayConversions(record)
2:     for all unmarked x ∈ record do
3:         for all unmarked y ∈ record do
4:             if y.sourceType = x.targetType then
5:                 if y.targetType = x.sourceType then
6:                     newConversion ← (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 ∈ record do
17:        assign uid to r
18:    end for
19: end procedure
```

```
<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 2.3 – The XML code for a registry item

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.

2.2.2.4 Code Generation

There are two separate code generation modules included in *zamk*. The first one is the byte-code generation and weaving module which is used to generate code from *Gluer* statements. The second one is the aspect generator, which uses user-defined converters to generate

```
1  public privileged [name][source-type]2[target-type]GenAspect
       extends ZamkRuntime{
2    private static String aspectUID = [..];
3    [name][source-type]2[target-type]GenAspect()
4    {
5      ZamkRuntime.register(aspectUID, map);
6    }
7    Map<[source-type], [target-type]> map = new
         WeakHashMap<[source-type], [target-type]>();

9    pointcut updateObserverThis(Object c): updateObserver(c) &&
         if(c instanceof [source-type]);
10   after(Object c): updateObserverThis(c)
11   {
12     [source-type] obj = ([source-type])c;
13     if(map.containsKey(obj)){
14       [name].[update-method](map.get(obj),
             [name].[convert-method](obj));
15     }
16   }
17 }
```

Listing 2.4 – The code generation template for producing an adaptation-specific aspect

conversion aspects for each converter.

Byte-code generation and Instrumentation The *Gluer* statements are parsed and transformed into Javassist (cite) library calls, which is used to insert byte-code into class files. Since *Gluer* is a proof of concept implementation it only supports constructor injections; the target field of the injection is initialized during object creation with this type of injection. It is also possible to extend the grammar and the byte-code generator to implement setter injections.

Aspect Generator For each conversion in a user-defined converter a specialized aspect is generated. The template for this aspect is shown in listing 2.4.

Let us explain this template in detail. `[name]` is 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]`. Since a single converter class can include multiple conversions, the aspect names also include the type information in their names. This naming convention provides a unique fully-qualified name for each generated aspect. Every gener-

ated aspect extends the abstract aspect `ZamkRuntime`. The details of this aspect will be covered in section 2.2.3.

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 and is the fully-qualified name of the aspect. Inside this constructor the aspect registers its `map` to the *zamk* runtime with its unique ID; this map is used for storing the source–target pairs. The declaration of the `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 2.2 is shown in listing 2.5. 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 2.6. 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

```
1  public privileged Cartesian2PolarUserCartesian2PolarGenAspect
       extends ZamkRuntimeAJ{
2    private static aspectUID = "myID";
3    Cartesian2PolarUserCartesian2PolarGenAspect()
4    {
5      ZamkRuntime.register(aspectUID, map);
6    }
7    Map<Cartesian, Polar> map = new WeakHashMap<Cartesian,
         Polar>();

9    pointcut updateObserverThis(Object c): updateObserver(c) &&
         if(c instanceof Cartesian);
10   after(Object c): updateObserverThis(c)
11   {
12     Cartesian obj = (Cartesian)c;
13     if(map.containsKey(obj)){
14       Cartesian2PolarUser.updatePolar(map.get(obj),
             Cartesian2PolarUser.cart2polar(obj));
15     }
16   }
17 }
```

Listing 2.5 – The aspect generated for the Cartesian to Polar converter in listing 2.2

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

Listing 2.6 – `updateObserverThis` pointcuts for two-way conversion

listing 2.7.

2.2.3 Runtime

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* 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.

Figure 2.6 shows how *zamk* processes a conversion request. For each request the appropriate conversion is found by the Find Conversion process. When this process completes successfully, the conversion map of the found conversion is retrieved. If the source object already exists in the conversion map, the corresponding target object

```
1  public privileged Cartesian2PolarUserCartesian2PolarGenAspect
       extends ZamkRuntimeAJ{
2    private static aspectUID = "myID";
3    Cartesian2PolarUserCartesian2PolarGenAspect()
4    {
5      ZamkRuntime.register(aspectUID, map);
6    }
7    HashBiMap<Cartesian, Polar> map =
         HashBiMap.create<Cartesian, Polar>();

9    pointcut updateObserverThisCartesian(Object c):
         updateObserver(c) && if(c instanceof Cartesian);
10   after(Object c): updateObserverThisCartesian(c)
11   {
12     Cartesian obj = (Cartesian)c;
13     if(map.containsKey(obj)){
14       Cartesian2PolarUser.updatePolar(map.get(obj),
             Cartesian2PolarUser.cart2polar(obj));
15     }
16   }
17   pointcut updateObserverThisPolar(Object p):
         updateObserver(p) && if(p instanceof Polar);
18   after(Object p): updateObserverThisPolar(p)
19   {
20     Polar obj = (Polar)p;
21     if(map.inverse().containsKey(obj)){
22       Cartesian2PolarUser.updateCartesian(map.inverse().get(obj),
             Cartesian2PolarUser.polar2cart(obj));
23     }
24   }
25 }
```

Listing 2.7 – The aspect generated for the Cartesian to Polar two-way converter

is retrieved. Otherwise a new target object is created by invoking the convert method of the conversion. The conversion map is also updated to include the new source–target pair. 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. *zamk* runtime is also responsible for managing source–target object pairs, by monitoring the source objects and updating the maps accordingly.

2.2.3.1 Initialization

In order to process conversion requests, *zamk* performs a one-time initialization step at the beginning of the run-time which consists of

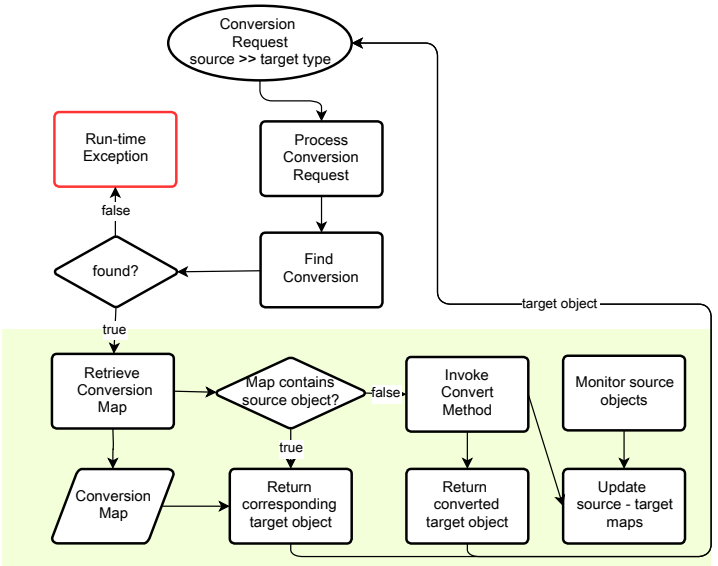


Figure 2.6 – The process triggered by a conversion request

loading the conversion registry and conversion aspect registration.

zamk creates a registry of conversion during compile-time (Section 2.2.2.3). 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 Figure 2.7.

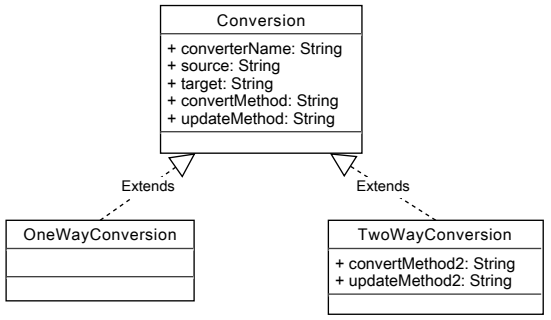


Figure 2.7 – The run-time `Conversion` objects type hierarchy

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, the contents of the maps are managed by the `ZamkRuntime`.

2.2.3.2 *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 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 2.2.3.3.

Using uid the source–target map for the conversion is retrieved from `mapPerConversion` hash-map (line 4). This operation works like lazy initialization. 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. Otherwise 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.

2.2.3.3 Finding a Conversion

The `findConversion` method shown in Procedure 3 implements an algorithm, which uses type information to find the *closest* conver-

Procedure 3 The `getConvertedValue` method

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

sion 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.

Let us clarify these descriptions with an example. In Figure 2.8 two separate hierarchies that represent different classifications for animals. Conversions are defined between the types of these 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 eligible. 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

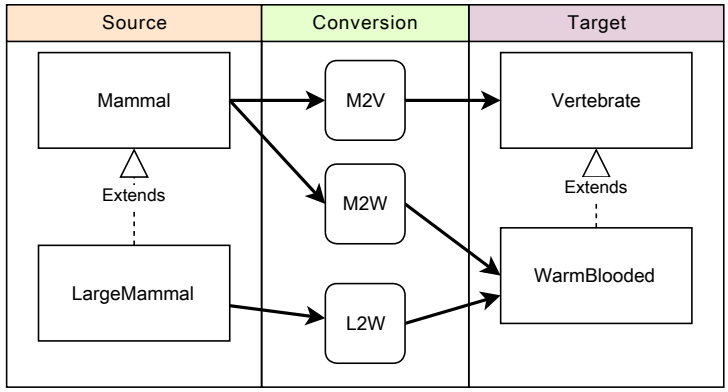


Figure 2.8 – Two type hierarchies for representing animals and conversions between them

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 2.1 – 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 2.1. 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.

Procedure 4 The procedure for finding the most suitable conversion

```
1: procedure findConversion(sourceType, targetType)
2:     eligibles ← empty list
3:     superTypesSource ← getAllSuperTypes(sourceType)
4:     while superTypesSource.hasNext do
5:         if sourceType ∈ sourceMap.keySet then
6:             candidates ← sourceMap.get(source)
7:             for all id ∈ candidates do
8:                 conversion ← conversionMap.get(id)
9:                 if conversion.getTargetType ⊂ targetType then
10:                    eligibles.add(id)
11:                end if
12:            end for
13:        else
14:            sourceType ← superTypesSource.next
15:        end if
16:    end while
17:    if eligibles = ∅ then
18:        Runtime Exception, no conversions found
19:    else
20:        found ← findClosest(eligibles, sourceType, target-
   Type, 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
```

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 these conversions, we first need to find the super types of the source type by walking their type hierarchy. This is done by the method `getAllSuperTypes` which uses reflection to populate the full set of super types (classes and interfaces) of a given type. On line 3 this method is called and the returned list is stored in *superTypeSource*. 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 $candidates$ (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 a sub-type 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 2.8. On line 2 we create an empty list which will hold the

```
1  private static List<Class<?>> getAllSuperTypes(Class<?>
       sourceType) throws ClassNotFoundException {
2    List<Class<?>> sourceTypeHier = new ArrayList<Class<?>>();
3    sourceTypeHier.add(sourceType);
4    Class<?> superType = sourceType.getSuperclass();
5    Class<?>[] itc = superType.getInterfaces();
6    for(int i=0; i<itc.length; i++)
7    {
8      sourceTypeHier.add(itc[i]);
9    }
10   while (!superType.getName().equals("java.lang.Object")) {
11     sourceTypeHier.add(superType);
12     itc = superType.getInterfaces();
13     for(int i=0; i<itc.length; i++)
14     {
15       sourceTypeHier.add(itc[i]);
16     }
17     superType = superType.getSuperclass();
18   }
19   sourceTypeHier.add(Class.forName("java.lang.Object"));
20   return sourceTypeHier;
21 }
```

Listing 2.8 – The source code for the `getAllSuperTypes` method

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 5. 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 2.1 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**

`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**

2.2.3.4 Target Object Creation and Retrieval

In section 2.2.3.2 we have mentioned the steps taken after a conversion is found. In this section we will discuss these steps in detail.

Procedure 5 Partial view of the `getConvertedValue` method

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

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 call the `invokeConversion` method, which reflectively invokes the convert method of the found conversion. The source code for this method is given in listing 2.9. 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. We first create a `Class` object for the converter that contains the `conversion` (line 4). After constructing an array for argument types of the convert method (line 5), the convert method for the `converter` is loaded from the `Class converter` (line 6). The retrieved method is invoked

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

Listing 2.9 – The `invokeConversion` method which reflectively invokes the convert method of a given conversion

by calling the `invoke` method and passing on the argument `source` (line 7). The result of the method invocation is stored in the variable `trgt` which was initially null and returned.

2.2.3.5 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 2.2.2.4. 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 changed 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.

2.2.3.6 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

2.3 Related Work

2.4 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.