Integration framework

Kardelen Hatun, Christoph Bockisch, and Mehmet Akşit

TRESE, University of Twente
7500AE Enschede
The Netherlands
http://www.utwente.nl/ewi/trese/
{hatunk,c.m.bockisch,aksit}@ewi.utwente.nl

1 Introduction

Systems are often faced with new requirements for adding new functionality or updating what is already there. A preferred way of extending or updating system behavior is through adding components. A software component can be described as a self-contained entity with a well-defined interface and behavior. Components can be plugged into legacy systems to extend the system with their provided functionality. In order to make the composed system functional, proper connections between the new component and the target system should be established; this operation is usually referred as component integration or component binding.

A specific issue regarding the extensibility of legacy systems is the integration of an *unanticipated concern*. It is common that some requirements are not taken into account, or even not foreseeable, when the systems are being designed. Depending on the extensibility of the legacy system such an integration task can be very difficult and requires extensive manual labor. This is due to the fact that, usually the interfaces of the legacy system and the new component is incompatible. This is a known problem and its solution is captured in the *adapter pattern* (cite Gamma). Typically adapters are tailored for a given integration. The complexity of the adapters depend on the complexity of the structures they adapt from and to. It is common that adaptation is seen as a one time task, which results in adaptation code that is fragile against software evolution and is non-reusable.

Another important aspect of component integration is establishing instance-level dependencies. Hence the other step of integration is assigning the data values to the relevant fields. Keeping the system and the components loosely coupled is an important principle of component-based design. 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.

In this paper we introduce our component integration framework, **zamk** , which unites dependency injection with *under-the-hood* adaptation logic. The **zamk** framework creates a registry of user-defined adaptation modules, which are composable entities. **zamk** framework also comes with a concise domain-specific language called gluer to define dependency injections. Given a dependency injection statement the framework tries to build the *type path* between the injectee and the injection field. We automate the adaptation process by exploiting the type hierarchies and provide checks and context-relevant messages for correct integration.

The rest of the paper is organized at follows. In section 2 layout the problem statement in detail and motivate our conceptual solution. In section 4 we go into the details of the **zamk** integration framework. The checks performed by **zamk** is shown in section 6.

2 Motivation and Goals

The goal of component integration is to make two components work together to provide a required functionality. However component integration efforts can be difficult to estimate, since the design and implementation of components vary greatly. Component models try to remedy this situation by defining the term component as "...a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard. [1]" Unfortunately component models do not guarantee well-structured binding, since one cannot guarantee that every used component conforms to a single component model. The incompatibility starts on the model level which translates to the interface level.

The code that binds components together is referred to as *glue code*. This is an umbrella term which refers to all code that facilitate the "hooking" the components together and establish channels for information exchange. The overall effort spent on the glue code varies and is dependent on many other factors. In [2], the preliminary findings of a software integration life-cycle cost model is presented. The authors state that the largest efforts in commercial-off the shelf integration is spent on glue code development. Typically glue code is viewed as a one-time task during integration, which results in non-reusable implementation.

Let us illustrate our claim with a simple example. In Figure 1[1] two components are shown. component1 contains one class called Cartesian. Now we would like to support also polar coordinates and component2 provides this support. However all the existing data is in Cartesian coordinates and we need to convert in to polar coordinates. In order to make this conversion, one can write an adapter that looks like the one shown in Listing 1.1. The class Cartesian2Polar is an object adapter and it is a valid solution to the problem at hand. Note that component2 also provides support for cylindrical and spherical coordinates, which are not used, since component1 does not support 3D coordinates.

Now let's take a look at the situation where this system evolves. Now we would like to support three dimensional Cartesian coordinates. Therefore we extend the class Cartesian as Cartesian3D. This new feature gives us the opportunity

[1] In order to save space we do not show the setter and getter methods, assume these methods are also members of the shown classes.

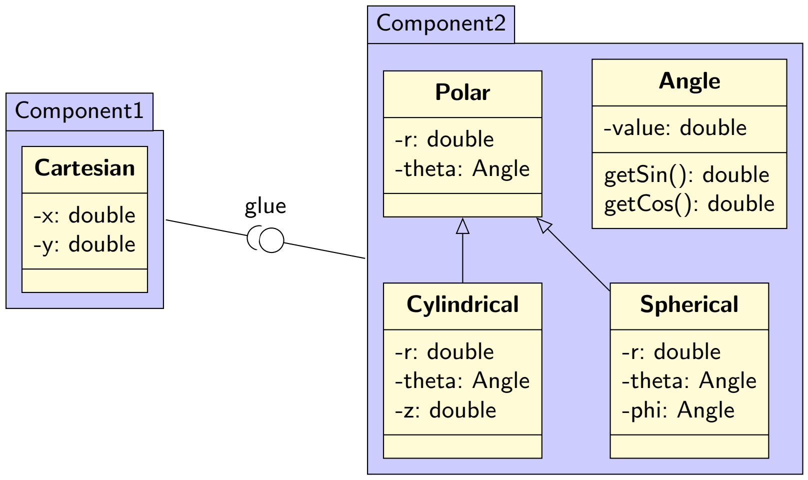


Fig. 1. Plugging a polar coordinate component to a cartesian component system

```
public class Cartesian2Polar extends Polar{
  Cartesian adaptee;
  public Cartesian2Polar(Cartesian c) {
    super();
    this.adaptee = c;
    super.setR(Math.sqrt(Math.pow(adaptee.getX(), 2) + Math.pow(adaptee.getY(),
         2)));
    super.setTheta(new Angle(Math.atan(adaptee.getY()/adaptee.getX())));
  }
}
```

Listing 1.1. A Cartesian to polar adapter

to use the cylindrical and spherical coordinates provided by component2. Once again we need to write adapters to convert between Cartesian3D and Spherical, Cylindrical classes (**TODO:Figure evolved model**).

The implementation for Cartesian3D2Cylindrical and Cartesian3D2Spherical are shown in Listing 1.2 and Listing 1.3 respectively. All of the adapters presented in these implementations have repeated lines of code. In Listing 1.1 on line 6, in Listing 1.2 on line 6 and in Listing 1.3 on line 6 all include the same calculation. In fact the calculations made in Cartesian2Polar adapters appear in Cart3D2Cylindrical and Cart3D2Spherical. However the way Cartesian2Polar is defined makes it impossible for us to reuse these calculations.[2]

[2] If Java supported multiple inheritance, we could have reused the Cartesian2Polar adapter's code by declaring Cart3D2Cylindrical as a subclass of both Cartesian2Polar and Cylindrical. However the lack of reuse would still be present in Cart3D2Spherical.

```
public class Cart3D2Cylindrical extends Cylindrical {
  Cartesian3D adaptee;
  public Cart3D2Cylindrical(Cartesian3D adaptee) {
    super(0, null, 0);
    this.adaptee = adaptee;
    super.setR(Math.sqrt(Math.pow(adaptee.getX(), 2) + Math.pow(adaptee.getY(),
         2)));
    super.setTheta(new Angle(Math.atan(adaptee.getY()/adaptee.getX())));
    super.setZ(adaptee.getZ());
  }
}
```

Listing 1.2. Adapter for converting three dimensional cartesian coordinates to Cylindrical coordinates

```
public class Cart3D2Spherical extends Spherical {
  Cartesian3D adaptee;
  public Cart3D2Spherical(Cartesian3D adaptee) {
    super(0, null, null);
    this.adaptee = adaptee;
    super.setRho(Math.sqrt(Math.pow(adaptee.getX(), 2) +
         Math.pow(adaptee.getY(), 2) + Math.pow(adaptee.getZ(), 2) ));
    super.setTheta(new Angle(Math.atan(adaptee.getY()/adaptee.getX())));
    super.setPhi(new
         Angle(Math.atan(adaptee.getZ()/(Math.sqrt(Math.pow(adaptee.getX(), 2) +
        Math.pow(adaptee.getY(), 2))))));
  }
}
```

Listing 1.3. Adapter for converting three dimensional cartesian coordinates to Spherical coordinates

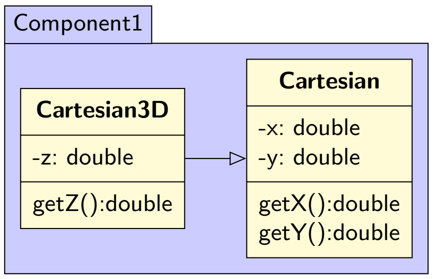


Fig. 2. component1 is evolved to include 3D coordinates

Let us re-design the adapter shown in Listing 1.1. Then we can have the decomposition shown in Listing 1.4. Using this decomposition we can create a P olar object given a Cartesian object c by writing:

```
Polar p = new Polar(new Angle(c), (new Pythagoras(c)).getAdaptedValue());
```

Furthermore we can reuse these classes to create a Cylindrical object given a Cartesian3D object c3d as follows:

```
Cylindrical cyn = new Cylindrical(new Angle(c3d), (new
    Pythagoras(c3d)).getAdaptedValue(), c3d.getZ());
```

By decomposing the glue code into two reusable classes we were able to make the coordinate conversions in a modular manner. The classes we have defined so far are not sufficient for converting from Cartesian3D to Spherical. Still we can reuse these classes by extending them as seen on listing 1.5.

Using the extended decomposition we can convert a Cartesian3D object to a Spherical object in the following manner.

```
Spherical sph = new Spherical((new Pythagoras3D(c3d)).getAdaptedValue(),new
    CartesianToAngleAdapte(c3d), new Cartesian3DtoAngleAdapter(c3d))
```

Based on our observations on the reusability issues of glue code we have devised our goal to solve these issues. Our main idea is that given *atomic adaptations*, it is possible to build a path between two types automatically using static analysis techniques and generate the glue code that binds these types together. Our focus is on the structural binding, which entails forming the static dependencies between components. In its most basic form this corresponds to injecting values to certain fields. At this level the requirement is to establish the type compatibility. This is achieved by obtaining the correct type that is injectable to a field by wrapping the injected value.

Another issue we want to tackle is *overloaded adapters*. Above we mentioned atomic adapters, by this we mean an adapter is a unit that only performs adaptation and nothing else. Therefore adapters are not sufficient to realize our goal, we need to have additional composition structures which assemble, enrich or compartmentalize these adapters.

TODO:Requirements

```
public class CartesianToAngleAdapter extends Angle{
  private Cartesian adaptee;
  public CartesianToAngleAdapter(Cartesian c)
  {
    super();
    this.adaptee = c;
    super.angle = Math.atan(adaptee.getY()/adaptee.getX());
  }
}
public class Pythagoras{
  private Cartesian adaptee;
  public Pythagoras(Cartesian c)
  {
    this.adaptee = c;
    r = Math.sqrt(Math.pow(adaptee.getX(), 2) + Math.pow(adaptee.getY(), 2));
  }
  public double getAdaptedValue()
  {
    return r;
  }
}
```

Listing 1.4. Decomposition of the members of the polar coordinate

```
public class Cartesian3DToAngleAdapter extends CartesianToAngleAdapter{
  private Pythagoras pyth;
  public Cartesian3DToAngleAdapter(Cartesian3D c)
  {
    super(c);
    pyth = new Pythagoras(c);
    super.angle = Math.atan(adaptee.getZ()/pyth.getAdaptedValue());
  }
}
public class Pythagoras3D extends Pythagoras{
  public Pythagoras3D(Cartesian3D c)
  {
    super(c);
    this.r = Math.sqrt(Math.pow(super.getAdaptedValue(), 2) + Math.pow(c.getZ(),
         2));
  }
}
```

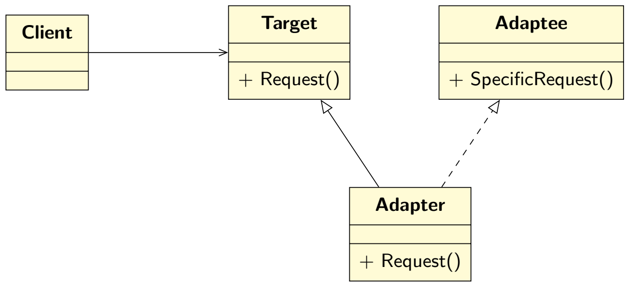
Listing 1.5. Extended versions of conversion classes

3 Background

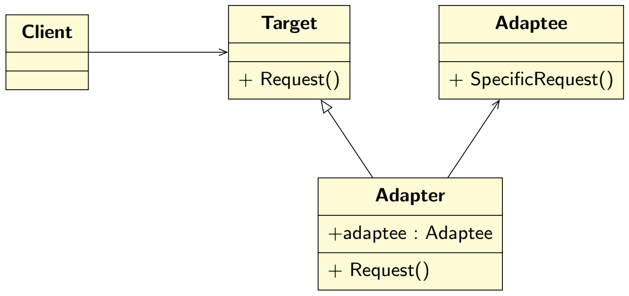
In this section we will give some background on the concepts that will be used as a part of our solution.

3.1 Adapter Pattern

Adapter pattern is a design pattern that is used to translate an interface to a compatible interface. Adapters, also known as wrappers, can be implemented in two ways: as a *class adapter* (Figure 3a), which uses multiple inheritance to adapt one interface to the other *or* as an *object adapter* (Figure 3b), which uses object composition to adapt wrapped object to a specific interface.



(a) The class adapter pattern



(b) The object adapter pattern

Fig. 3. Different versions of the adapter pattern

4 Approach

Our approach is an integration framework that marries dependency injection pattern with object adapters. In Figure 4 an overview of the **zamk** integration framework is shown. The framework consists of several components that work together and introduces a concise domain-specific language (DSL) called *Gluer*.

Gluer is used to define dependencies between objects. The specifications written in *Gluer* is parsed by the *Gluer parser*.

zamk uses a repository of composable *integration modules* which are defined by the user. Integration modules can be adapters, enrichers, aggregators or splitters. The details of these modules will be explained in section 4.3. When the user defines an injection between incompatible types, *integration logic* component retrieves and assembles integration modules to convert the type of the injectee object to a compatible type expected by the injected field.

Finally the *dependency injection component* performs the injection; this component supports two types of injection, constructor injection and setter injection. The details of this operation will be elaborated in section 4.4.

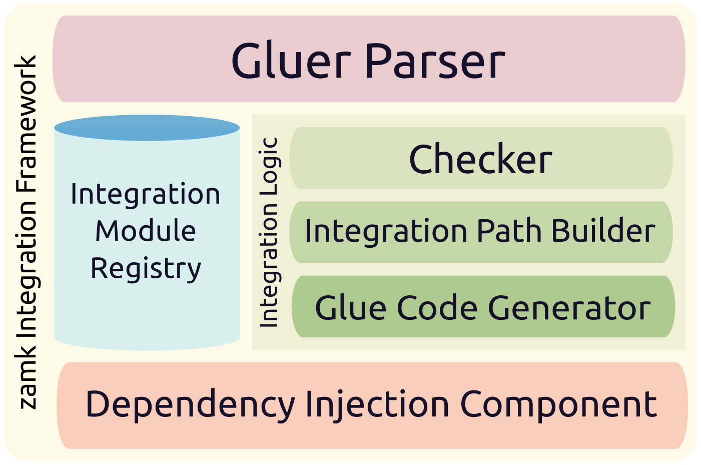

zamk Integration Framework
Gluer Parser
Integration Module Registry
Integration Logic
Checker
Integration Path Builder
Glue Code Generator
Dependency Injection Component

Fig. 4. An abstract view of the **zamk** integration framework

4.1 The Gluer Language

The *Gluer* language is a concise DSL that is designed to declare dependencies between fields and objects. Essentially the *Gluer* language is an external dependency injection declaration which triggers an internal logic that processes the injectee object. Therefore the *Gluer* language is not a plain dependency injection language. This is why we chose the keyword glue instead of inject. A simple gluer statement looks like the following:

```
glue <injection−field> with <injectee−object>
```

¡injection-field¿ The injection field is a fully qualified name of a non-static field of a class. It can refer to any type, including primitive types.

¡injectee-object¿ The injectee object represents the object to be processed and injected to the injection 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 given 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.

The *Gluer* language also supports 1–to–many and many–to–1 gluing operations. **TODO:Expand**

4.2 Integration Logic

The integration logic is responsible for wrapping the what objects with appropriate adapters to be injected to where field. The what objects may need to be preprocessed before adaptation, a what object can be split into different objects (one-to-many) or multiple what objects can be aggregated to a single object. These operations are done automatically by the integration logic. Figure **ref** shows the flowchart of operations that take place for finding splitters and aggregators for objects.

flowcharts for splitter and aggregator

4.3 Integration Modules : Splitter, Aggregator, Adapters

The integration modules are composable

Splitters

1. how are they defined; regular classes with @Splitter annotation, They should implement the ISplitter interface which comes with the split(Object) method
2. How are they registered, we may need some meta information

Aggregators

1. how are they defined; regular classes with @Aggregator annotation, They should implement the IAggregator interface which comes with the aggregate(List –Object–) method
2. How are they registered, we may need some meta information

Adapters Once the structural mapping is done, the resulting object(s) should be adapted to be injected. This step is also automated…

The flowchart from Arnout about adapter retrieval

Adapter Resolution Precedence rules
Informative error messages

4.4 Dependency Injection

5 Module Retrieval Algorithm

6 Checks

7 Evaluation

8 Conclusion

References

1. Heineman, G.T., Councill, W.T., eds.: Component-based software engineering: putting the pieces together. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2001)
2. Abts, C., Boehm, B., Clark, E.: Cocots: A cots software integration lifecycle cost model-model overview and preliminary data collection findings. In: ESCOM-SCOPE Conference, Citeseer (2000)