An approach to configure a program by using context-aware features

Martin Schleiss
Email: e1126579@student.tuwien.ac.at
Seminar in Programmiersprachen SS 2016

***Abstract*—Nowadays, most of the programs use configuration files to customize their behavior. Unfortunately this can be error-prone, e.g. because of typos made by developers. To address this issue, we are going to use a solution consisting of 3 parts. Firstly, context-oriented programming which became more and more important due to the rise of ubiquitous computing as well as mobile computing. Secondly, contextual values which change their stored value depending on the current context. This combined with the LibElektra library which is responsible for storing the contextual values should improve our development experience.**

I. Introduction

By using context-oriented programming we can incoporate additional information about the program's currenct location, settings etc. in order to adapt its behavior accordingly. This became more and more important in the last few years because mobile devices have become a fundamental part of our everyday lifes. Hence applications have to run under different environments. Therefore developers have to design their programs in such a way that they can react to their environments intuitively. Particularly when it comes to configuring their programs. Therefore we are going to explain a possible solution where layers, a very common solution where additional program behavior is activated around the main instructions, combined with contextual values, which change their values depending on their current context (represented as layers), can be of use to developers if they want to integrate configuration values into their programs.

First we are going to explain how layers work. Layers are used in most of the available context-oriented programming languages nowadays [1]. There are different solutions which are written in Lisp[2] or even Smalltalk[3] but since the author has only little experience in these languages, we will use newer languages like Java in our examples. Additionally we are going to indicate some of the key aspects in which context-oriented programming languages differ from each other and how we can discern them. There are several differences, but we are only going to pick af few in order to keep this paper to a compact size. Secondly, we are going to take a look at contextual values. These are similar to thread locals, because they can change their contained value depending on the current context. In the last section we are going to take a look at Raab's solution [8] which tries to combine the concept of layers from the realm of context-oriented programming with contextual values. This combination added with the capabilities of LibElektra [7], also a solution provided by Raab, will hopefully give us the right tools to configure the programs of tomorrow.

II. Context-oriented programming with layers

A. Overview

Most of the context-oriented languages provide layers, which help developers to modularize their programs. Layers are a modularization technique in order to define behavioral variations which can be distributed across several classes. They achieve this by surrounding the core behavior with their respective variations, for instance additional code can be executed before and after a certain method.

There are 2 techniques for programs which use layers: either a layer-in-class modularization or a class-in-layer modularization. Layer-in-class means that layer specific behavior for a specific class is definied within the definition of a certain class. As a result the entire program still remains modularized by means of classes. As a side effect developers can access private information from the enclosing class because layers are defined inside of classes. On the contrary class-in-layer context-oriented programming languages can group layer specific behavior inside a separate module. Hence, developers do not have to alter the code of existing classes if they want to add new layers to a program.

In order to activate layers we can use either block syntax or globally activate certain layers. Using a block syntax the layers are only active within the scope of the given block of code. In contrast, global activation of layers affects all threads and does not require to define a block. Furthermore it should be noted that a layer is active for all direct and indirect calls which are enclosed by the same *with*-block. This called the dynamic scope where several function calls could happen between different function calls. Also it is possible to have a "stack" of different layers where the innermost *with*/*without* defines the activation or deactivation of a layer (a *without*-block is used to deactivate a possibly, previously activated layer). Between the different *with* and *without* calls, it is possible that function calls To visualize this, take Listing 1. Each consecutive line indicates an activation or deactivation of a layer.

```
with(Security) {
  with(Logging) {
    with(BatteryLow) {
      without(Logging) {
```

```
 5         with(BatteryLow) {
 6           /** do something here */
 7         }
 8       }
 9     }
10   }
11 }
```

Listing 1. Stack of layers

On the sixth line of Listing 1 *BatteryLow* and *Security* are the only layers which are still active. *Logging* has been disabled by the call on the fourth line using *without*.

To better distinguish between methods which are defined inside of layers and classic methods which are also defined inside of classes, Appeltauer et. al. [1] coined the definitions for *plain method definition* and *layered method definition*. Plain methods are methods which are not influenced by the behavior of layers, meaning at the time they are executed, no layer is active. In contrast layered method definitions consist of 2 parts. The first part is called *base method definition* which will be executed if no layer defines a corresponding partial method. The second part consists of at least one partial method definition, additional ones are possible.

B. Example using ContextJ

From the example in Listing 2 using ContextJ we can deduct that is has to be a layer-in-class context-oriented programming language. Inside of the classes we have 2 layers: *PrintIdentification* and *PrintUniversity*. Both add additional behavior to the existing *toString()* methods of each class. Inside of Listing 3 we can see the result of activating both layers. Both, the original output and the additional output are included in the end. To achieve this effect we have to use the method *proceed()* which calls the enclosing layer's behavior of the currently executing method. If we are calling *proceed()* in the outermost layer, the behavior of the class will be executed.

```
class University {
  private String id;
  private String name;

  University(String id, String name) {
    this.id = id;
    this.name = name;
  }

  String toString() {
    return "University: " + name;
  }

  layer PrintIdentification {
    String toString() {
      return proceed() + " - ID: " + id;
    }
  }
}

class Student {

  private University university;
  private String name;
  private String id;

  Student(University university, String name,
      String id) {
    this.university = university;
    this.name = name;
    this.id= id;
  }

  String toString() {
    return "Student: " + name;
  }

  layer PrintUniversity {
    String toString() {
      return proceed() + " - " + university;
    }
  }

  layer PrintIdentification {
    String toString() {
      return proceed() + " - ID: " + id;
    }
  }
}
```

Listing 2. Modified version from [4]

```
University u = new University("1", "TU Wien");
Student s = new Student(u, "Martin",
"1126579");

with (PrintIdentification) {
  with(PrintUniversity) {
    System.out.println(s);
  }
}

Outputs: Student: Martin - ID: 1126579 -
    University: TU Wien - ID: 1
```

Listing 3. modified version [4]

```
Student.PrintUniversity.toString()
  Student.PrintIdentification.toString()
    Student.toString()
      University.PrintIdentification.toString()
        University.toString()
```

Listing 4. Execution order of Listing 3

C. Comparision of context-oriented programming languages

Other context-oriented programming languages also use layers as a modularization technique. As already mentioned before, there are 2 different types for layer modularization: layer-in-class and class-in-layer. These are one of the main distinguishing aspects among context-oriented programming languages. [1]

As we can see from Table I, most of the languages either opt for the layer-in-class method or they implement both possibilities. Only ContextS and PyContext use class-in-layer exclusively. Most of the context-oriented programming languages in Table I use libraries to provide context-aware features. There, layers are defined by using classes because most of the implementations are situated in an object oriented language. Though some of them, for instance ContextJ, provide an extended syntax in the respective language by using a seperate compiler. The advantage is of course a more concise and readable program code but the compatibility with newer compilers reduces greatly. For instance in ContextJ, we can

TABLE I
COMPARISON OF CONTEXT-ORIENTED PROGRAMMING LANGUAGES ADAPTED FROM [1]

Language	class-in-layer	layer-in-class
ContextL	x	x
ContextS	x	
ContextJ		x
ContextLogicAJ		x
PyContext	x	
ContextPy		x
ContextR		x
ContextJS	x	x
ContextG		x
cj	x	x

imagine that an update to a newer Java version can only be done if ContextJ's compiler has been updated as well.

Another key point which differentiates most of the context-oriented programming languages is the manner in which a certain layer becomes activated or deactivated. Most of the solutions in Table I adopt the strategy of *dynamic-extent activation* [1]. Here we have to declare a block. During the dynamic-extent of the given block, the context-oriented language will either activate or deactivate a given layer. This technique may also be called *dynamically scoped layer activation*. Other context-oriented languages either opt for a global activation of layers or a thread based activation. It should be noted though, that these strategies are not an exclusive solution by any means. For example, ContextJ supports dynamic-extent based, thread based as well as global activation strategies [1].

III. CONTEXTUAL VALUES

Éric Tanter introduced the notion of contextual values in 2008 [9]. There he uses Common Lisp to introduce the concept of contextual values. These contextual values will be stored by a library written in Common Lisp and later tries to achieve language support for contexual values in Lisp. In this paper we are only going to cover the concepts which build the fundation of contextual values. First it should be noted, that contextual values change their stored value depending on the context they are in. In order to understand contextual values we first have to take a look at thread local values. These are avaible in most programming languages.

```
ThreadLocal<Integer> local = new ThreadLocal<
    Integer>() = {
  new ThreadLocal<Integer>() {
    @Override protected Integer initialValue()
        {
      return 0;
    }
  }
}
int c0 = counter.get();
```

Listing 5. Thread local variables

In Listing 5 each thread instance executing the code will have its own instance of *ThreadLocal*. On the eighth line the value of *c0* will be 0 for all thread instances. Furthermore passing a reference of a *ThreadLocal* to another thread is not possible. The value which is stored inside an instance of *ThreadLocal* is only accessible to the instantiating thread (= context) [5].

According to Tanter, contextual values consist of two parts: a context function and a value mapping. In order to retrieve the value for the current context we have to apply the context function to get the key for the value mapping. In this case the context of the thread local is defined by the thread's id. Now, contextual values generalize the idea of thread locals. They allow any computational value to be a context to themselves. Hence Tanter defines contextual values as a tuple of a context function *ctx* and a place for storing values *vals*.

```
cv = <ctx, vals>
```

To retrieve the value for the current context we have to apply *ctx*. This yields a key for the context *ctx*. Using this key we can find the corresponding value for *ctx* in *vals*. Also, as context any value can be used as long as it is computationally-accessible. If we want to update or create a new entry in *vals*, we also have to retrieve the key by applying *ctx* first and then query *vals* by using the just retrieved key [9]. In comparison to thread locals which will not be accessible after the thread finished his work, contextual values can be stored beyond their context. This means if a context happens to be active once, stores a value in *vals* and later becomes active again, the value will still be accessible because the context is the same.

Other implementations of contextual values are similar to those of Tanter. For instance in PyContext, Löwis et. al. [10] introduced the concept of context variables where combined with *with*-statements these variables become newly initialized by the time they enter the scope of a new *with*-block (= new context) and reverted as soon as they leave the scope. In comparison Tanter's approach uses any computationally-accessible value as context, not just the dynamic-extent or in other approaches the thread id. Tanter even goes a step further, he allows contextual values to serve as a context to other contextual values. Unfortunately explaining this concept would go beyond the purpose of this paper and is therefore left to the interested reader for self-study.

IV. CONTEXTUAL VALUES COMBINED WITH LAYERS

Using contextual values combined with layers we have the appropiate tools to incorporate configuration data into our programs. Raab et. al. [8] connect the notion of contextual values as a container for configuration data and layers from the field of context-oriented programming, which act as context to the contextual values. Combined with Raab's Elektra Library [7] which serves as a middle man between the program's execution environment and the contextual values, as indicated in Figure 1, we have the tools we need to incoporate configuration data into our programs. In Figure 1 we an see that the Elektra library serves as a middle man. It loads and stores values from the execution environment. Furthermore we can see a code generator which translates configuration specification files into executable code. This executable code then accesses the key set where our data is stored.

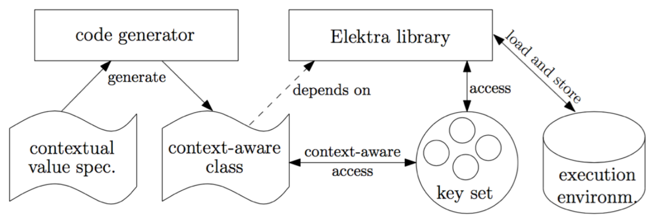

Fig. 1. Architecture [8]

First of all, in order to get started we have to declare a config file that acts as specification for our contextual values as in the following:

```
1  [/device/battery/%language%/low]
2  type=string
3  default=""
4  [/%user%/logins]
5  type=Integer
6  default=0
```

Listing 6. Configuration specification

Here we define a contextual value named *low* below 2 other contextual values named *device* and *battery*. The value is supposed to tell the user in the device's language that the battery is almost empty. The keyword type is used by GenElektra to generate appropriate classes for the specification. The default value is mandatory in the current implementation of LibElektra and is used if the provided configuration does not specify a value.

In Listing 6 for the first entry *[/device/battery/%language%/low]* 3 classes are generated - one for *Device*, one for *Battery* and one for *Low*. For the second entry, GenElektra[7] generates a class *Logins*. We can see the simplified output for *Logins* in Listing 7 which GenElektra generated for us.

```
class Logins : public Integer
{
public:
  Logins:(KeySet &ks, Context &context) :
    Integer(ks, context, Key("", KEY_VALUE, "/%
         user%/login", KEY_META, "default", "0",
          KEY_END)) { }
    using Integer::operator=;
}
```

Listing 7. Generated Logins class [8]

Having a tool like GenElektra should save a lot of boiler plate code which the developers would have to write and avoids type mismatches between the configuration and the code. This is due to the fact that the specification of our contextual values is more than enough to produce simple classes for our contextual values. To further illustrate how this configuration specification works, we will switch to the class *Device*. Here we have a member variable of type *Battery* and for each language which will be added. For the second entry in Listing 6 on the fourth line a class *Logins* should be generated. Because a default value is defined for *Logins*, new instances of this class are initialized to 0. Furthermore we have to provide actual values for different contexts inside of a configuration file:

```
/device/battery/%/low=10% of battery remaining
/device/battery/German/low=Nur mehr 10% der
    Batterie sind verfuegbar
/device/battery/English/low=10% of battery
    remaining
```

In the first line the % character defines an empty name [8] which basically tells the system to use this value as the default one. In the next two lines we define the alert message for the German as well as the English language.

Next we have to define classes for each and every entry. These classes will serve as layers in our program later. In the following Listing 8 we can see a class definition for the English alert message:

```
class DeviceBatteryEnglishLow : public Layer
{
  public :
    string id() const {
      return "language";
    }

    string operator()() const {
      return "English";
    }
}
```

Listing 8. Layer for DeviceBatteryEnglishLow

Here the method *id()* is used to associate the given layer with the layer specification in Listing 6, in our case the placeholder *%language%*. It should be noted, that this identifier has to be globally unique otherwise it is not possible to deterministically reference the associated placeholder. In order to reduce the amount of classes we have to write, we could also define a *DeviceBatteryLanguageLow* class which serves as a foundation for all languages:

```
class DeviceBatteryLanguageLow: public Layer
{
  public:
    DeviceBatteryLanguageLow(string language) :
         language(language) { }
    string id() const {
      return "language"
    }

    string operator()() {
      return language;
    }
    private:
      string language;
}
```

Listing 9. Layer class for any language

After defining our layer specification, our values for each layer and a layer class for all languages, we are ready to access our configuration data:

```
void batteryAlertLow(Low &l) {
  l.context().with<DeviceBatteryLanguageLow>("
      German")([&] {
    count << "Alert: " << l;
  });
}
```

Due to the fact, that we use a universal layer class for every language, it would be easy to switch to a different language by just replacing the string within the parenthesis. There is no need to define separate classes for every language. As a side note, it is also possible to access command line arguments using LibElektra by using a configuration specification as in Listing 10.

```
[/server/%port%/number]
type=Integer
opt=p
opt/long=port
range=0-65535
default=0
readonly
```

Listing 10. Command line configuration specification

Here we state that for the layer port only values between 0 and 65535 are valid and that we can provide the number by either using the short version *-p* or the long version *--port* for the command line argument. Addtionally we declare this contexual value as *readonly* so that the application itself can not alter the port from within the code [7].

From the listings above we can see that there exist 2 important classes:

- Context
- Layer

In the following we will try to explain each of them. Due to the fact that Raab's solution is not an extension to the C++ language but rather a library which can be imported into any project, there exists a key class called *Context* within LibElektra [7]. This class is responsible for activating or deactivating layers on a global basis as well as defining *with* and *without*-blocks where layers should be activated or deactivated. From an object oriented standpoint the *Context* class and its related classes are organized with the observer pattern. Since *Context* is the subject in this case, it will notify any contextual values about any updates [8].

```
class Context : public Subject {
  public:
  template <typename L> void activate(...);
  template <typename L> void deactivate(...);
  template <typename L> Context & with(...);
  template <typename L> Context & without(...);
  string evaluate(string const & spec) const;
  Context & operator()(function const & f);
  // ...
};
```

Listing 11. Context interface [8]

Every layer has to implement the *Layer* interface in Listing 12 in order to work. The function *id()* is responsible for returning the placeholder in our configuration specification language which are enclosed by two percent signs. For instance, a placeholder which is called *%port%* would mean that a responsible layer has to return the given string *port* as result for the *id()* method. For distinguising between differently configured layers we have to use the *operator()* method. Here we return different instances, for example *8080* or *443*. From the low battery example above, this would mean we would either return *English* or *German*.

```
class Layer {
  public:
  virtual string id() const = 0;
  virtual string operator ()() const = 0;
};
```

Listing 12. Layer interface [8]

Furthermore developers sometimes want to react to context changes immediately. For instance, having a mobile device and the system has suddenly reached a critically low battery level, developers want to notify users the situation as well as reduce computationally intensive tasks in order to prolong the battery's remaining capacity. To achieve this, we can pass a lambda to LibElektra which will be executed by the time a certain layer has been activated [6].

```
Coordinator c;
c.onLayerActivation<DeviceBatteryLanguageLow
    >([] () {
    turnOffGPSTracking();
  }
);
```

Listing 13. Reacting to layer activations [6]

This layer activation and deactivation is not only bound to the current thread but rather to all threads. In order to synchronize the current status of all layers across all possible threads we have to either activate or deactivate another layer or call the *syncLayers()* method on our context object.

V. Conclusion

In this paper we tried to give an overview of the concept of context-oriented programming by one of the leading techniques of structuring ones program, namely layers. They are a very handy way to incoporate contextual information into a program and are easy to use. Nowadays there exist several context-oriented programming languages using layers as a main modularization technique. Therefore we tried to indicate some of the key differences among them. Additionally we took a look at contextual values which change their value depending on the current context. They are a very straight forward concept similar to thread locals but can also store their information once a context changes. Last but not least we saw a combination of the 2 solutions combined with the LibElektra library to configure programs. Using a simple configuration specification in order to define configuration data seems pretty easy. Furthermore the idea of generating classes for our configuration stores looks promising and could be a real time saver for developers.

References

[1] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, Jens Lincke, and Michael Perscheid. A comparison of context-oriented programming languages. In *International Workshop on Context-Oriented Programming*, COP '09, pages 6:1–6:6, New York, NY, USA, 2009. ACM.

[2] Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming: An overview of contextl. In *Proceedings of the 2005 Symposium on Dynamic Languages*, DLS '05, pages 1–10, New York, NY, USA, 2005. ACM.

[3] Robert Hirschfeld, Pascal Costanza, and Michael Haupt. Generative and transformational techniques in software engineering ii. chapter An Introduction to Context-Oriented Programming with ContextS, pages 396–407. Springer-Verlag, Berlin, Heidelberg, 2008.

[4] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3), 2008.

[5] Oracle. Threadlocal (java platform se 7), 2016. Available one at https://docs.oracle.com/javase/7/docs/api/java/lang/ThreadLocal.html.

[6] M. Raab. Global and thread-local activation of contextual program execution environments. In *Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 2015 IEEE International Symposium on*, pages 34–41, April 2015.

[7] Markus Raab. Libelektra, 2016. Available online at https://www.libelektra.org.

[8] Markus Raab and Franz Puntigam. Program execution environments as contextual values. In *Proceedings of 6th International Workshop on Context-Oriented Programming*, page 8. ACM, 2014.

[9] Éric Tanter. Contextual values. In *Proceedings of the 2008 symposium on Dynamic languages*, page 3. ACM, 2008.

[10] Martin von Löwis, Marcus Denker, and Oscar Nierstrasz. Context-oriented programming: Beyond layers. In *Proceedings of the 2007 International Conference on Dynamic Languages: In Conjunction with the 15th International Smalltalk Joint Conference 2007*, ICDL '07, pages 143–156, New York, NY, USA, 2007. ACM.