Domain Specific Languages

Conrardy Aaron, Lakhoune Ben

May 3, 2019

Abstract

Modern software systems are becoming ever more complex and specialized in their respective domain. A better communication between software developers and domain experts to formally specify those systems is needed. The use of a formal language for communication describes the concept of domain-specific languages.
In this report we explain the term domain-specific language in the field of programming languages. We give a brief explanation of the term domain-specific as well as the criteria for a programming language to be considered a domain-specific language. We distinguish between an **internal** and an **external** domain-specific language by stating the key differences between the two. Following this we explain the workings of a domain-specific language and apply the concepts on a few examples. We analyze the advantages and disadvantages of using a domain-specific language. To conclude, we give a short guidance on how to implement domain-specific languages.

1 Introduction

One of the key elements to success for a project is the level of communication between the developers and the customers. The customers come from a variety of domains. The term *domain* means a precise field of expertise, e.g. the medical field, biological field etc. As developers, our main goal is to please the customers. Even though the customers are experts in their own domain, their ability to understand complex software code is usually limited. This means we have to adjust the code so that they can understand it. This can be achieved by writing the code in a programming language that is easy to read. Another way to adjust to them is by creating a language, which uses the terms of the respective domain.
Generally, when a developer deals with a problem, he takes one of two approaches: the generic or the specific approach [5]. The generic approach provides a generalized solution for a set of problems of the same type, even if more optimal solutions exist. This is a very common approach used in *domain engineering.* Domain engineering is the process of reusing domain knowledge in the production of new software systems [1]. One example would be design patterns in software development, which describe a general way to deal

with common problems.
The specific approach concentrates on a much smaller but more efficient set of solutions. This specific approach is what led to the idea of domain-specific languages. Domain-specific languages have been part of the programming world for some time now, but the term has only recently been popularized.
There is no exact definition of the term domain-specific language. For this report, we will use the definition proposed by Martin Fowlers in his book on "domain-specific languages" [3] :
A **Domain specific language** is "a computer programming language of limited expressiveness focused on a particular domain."
We can divide this definition into four key elements :

- **Computer programming language**: Although a domain-specific language is created for humans to instruct the computer to do something, and that in a far simpler manner than a general-purpose language, it should be readable by a human and still be executable by a computer.
- **Language nature**: The expressiveness of a domain-specific language comes from combined expressions rather than individual ones, giving the domain-specific language a certain fluency like a real language.
- **Limited expressiveness**: The purpose is not to build an entire software system in a domain-specific language. It should rather just be a part of one particular aspect of the system. The domain-specific language supports only a bare minimum of features needed to support its domain and should not be turing-complete.
- **Domain focus**: The purpose of a domain-specific language is not to be a general-purpose language. The purpose of the domain-specific language is to be applied in one small domain; meaning the use of the domain-specific language in that context makes it feel more natural.

To further demonstrate the concept of a domain-specific language, we introduce the following example.

2 Introductory example

Let's assume we work for a company that builds security-systems that open and close secret panels inside of a room. It is our job, as developers, to program a controller for those security-systems. The controller works as follows: First it waits for a certain sequence of events to happen. It then locks or unlocks a panel in case this sequence of events is triggered. For a client a secret room, which opens through a hidden door, needs to be installed.The sequence of events should be the following: first the front door is closed then the second drawer of the bedside chest is opened and her bedside light is turned on, in either order. After these events a secret door opens in the kitchen. To implement the security systems, little sensory devices are installed. Those devices send four-character messages when certain events happen. Little control devices are also installed. Those devices respond to four-character command-messages by executing certain instructions, for example opening a door. One main controller listens to event messages, sends command messages and decides on what to do next. To demonstrate the workings of a domain-specific language, we will focus on the controller program. We can illustrate the before-mentioned sequence to access the treasure room with a state diagram.

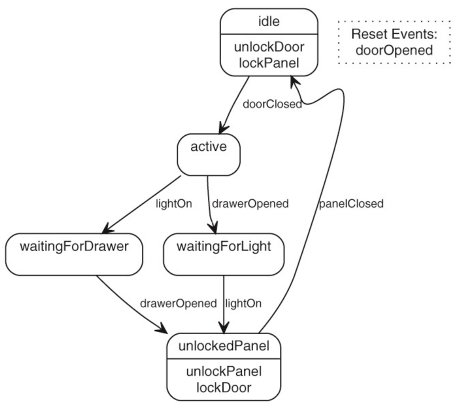


Figure 1: Controller

The controller can be in different states, represented by the boxes. Depending on the state and the next event, the controller will transition to another state with different transitions. If a sequence of events happens, we jump from one state to another state. In our state diagram model, we also have an "idle" state. The controller spends most of its time in that state. It represents the state that waits for the door to be closed to start of the chain of events, which lead to the secret door being opened. Additionally, the state machine has a reset event. In our case, opening the door resets everything and sends us back to the idle state. The first thing to do is to choose a model for our implementation. We choose the *State Machine* model. We will configure the state-machine's Controller using different languages or styles.

If we take a look on the left code of the following page, the controller was configured in basic Java. It does not have any particular traits and the structure looks like ordinary Java code. This wouldn't be considered a domain-specific language since it feels like stitching together with an *API* and is not a fluent language.

```
public class StateJava{

  Event door = new Event("doorClosed", "D1CL");
  Event drawerOpened = new Event("drawerOpened", "D2OP");
  Event lightOn = new Event("lightOn", "L1ON");
  Event panelClosed = new Event("panelClosed", "PNCL");

  Command unlockPanelCmd = new Command("unnlcokPanel",
        "PNUL");
  Command lockPanelCmd = new Command("lockPanelCmd", "
        PNLK");
  Command lockDoorCmd = new Command("lockDoorCmd", "
        D1LK");
  Command unlockDoorCmd = new Command("unlockDoorCmd
        ", "D1UL");

  State idleState = new State("idleState");
  State activeState = new State("activeState");
  State waitingForLightState = new State("waitingForLightState
        ");
  State waitingForDrawerState = new State("
        waitingForDrawerState");
  State unlockPanelState = new State("unlockPanelState");

  SateMachine machine = new StateMachine(idle);

  idle.addAction(unlockDoorCmd);
  idle.addAction(lockPanelCmd);
  idle.addTransition(doorClosed, active);

  activeState.addTransition(drawerOpened,waitingForLightState)
        ;
  activeState.addTransition(lightOn,waitingForDrawerState);

  waitingForLightState.addTransition(lightOn,unlockPanelState);
  waitingForDrawerState.addTransition(drawerOpened,
        unlockPanelState);

  unlockPanelState.addAction(unlockPanelCmd);
  unlockPanelState.addAction(lockDoorCmd);
  unlockPanelState.addTransition(panelClosed,idleState);

  machine.addResetEvents(doorOpened);

}
```

```
public class StateJava{

  Event doorOpened, doorClosed, drawerOpened, lightOn,
        panelClosed;

  Comand unlcokPanelCmd, lockPanelCmd, lockDoorCmd,
        unlockDoorCmd;

  State idleState, activeState, waitingForLightState,
        waitingForDrawerState

  void defineStateMachine(){

    doorClosed.code("D1CL");
    drawerOpened.code("D2OP");
    lightOn.code("L1ON");
    panelClosed.code("PNCL");

    doorOpened.code("D1OP");

    unlcokPanelCmd.code("PNUL");
    lockPanelCmd.code("PNLK");
    lockDoorCmd.code("D1LK");
    unlockDoorCmd.code("D1UL");

    idleState
      .action(unlockDoorCmd,lockPanelCmd)
      .transition(doorClosed).to(activeState)
      ;

    activeState
      .transition(drawerOpened).to(waitingForLightState)
      .transition(lightOn).to(waitingForDrawerState)
      ;

    waitingForDrawerState
      .transition(drawerOpened).to(unlcokPanelCmd)
      ;

    unlcokPanelCmd
      .action(unlcokPanelCmd, lockDoorCmd)
      .transition(panelClosed).to(idleState)
      ;
  }
}
```

The right code, while still being Java, has a more declarative feel to it. For Java standards this code is formatted oddly and even uses unconventional programming rules like *method-chaining* (more on this later). However, if we have the state machine in mind, the code becomes more intuitive. Even if this code does not completely fulfill the definition of a domain-specific language, it is still considered one because of the language nature, domain focus and it is still code that is executable by a computer. We will now see a custom syntax created by Martin Fowler for this specific example.

```
events
  doorClosed D1CL
  drawerOpened D2OP
  lightOn L1ON
  doorOpened D1OP
  panelClosed PNCL
end
resetEvents
  doorOpened
end
commands
   unlockPanel PNUL
   lockPanel PNLK
   lockDoor D1LK
   unlockDoor D1UL
end
state idle
  actions {unlockDoor lockPanel}
  doorCLosed => active
end
state active
  drawerOpened => waitingForLight
  lightOn => waitingForDrawer
end
state waitingForLight
  lightOn => unlockedPanel
end
waitingForDrawer
  drawerOpened => unlockedPanel
end
state unlockedPanel
  actions{unlockPanel lockDoor}
  panelClosed => idle
end
```

Compared to the two previous codes, it is less noisy. This means that it uses an intuitive structure without quoting. Thus making the code easier to read and to write. This syntax was created especially to configure this type of state machine, resulting in a very limited programming language. We might not have the same level of language nature as the previous examples when concatenating methods, but we still have enough of it for it to make sense in a logical context. We got ourselves a programming language that is executable by a computer, has a slight sense of sentence-like flow, is limited and focuses on only the domain of this kind of state machine. This code can definitely be described as a domain-specific language.

3 Workings of a domain-specific language

In our example, we saw how one might choose different implementations for the configuration of the controller. To understand the workings of a domain-specific language we need to distinguish between an *internal* and an *external* domain-specific language.

3.1 Internal and external domain-specific languages

As a domain-specific language is developed, it is important to choose a syntax. On the one hand the syntax of a general-purpose language seems like a good option, as we are most familiar with it. In this case the domain-specific language is called an internal domain-specific language as it uses the same syntax as the main development language. The difference being that it only uses a subset of the general-purpose language's features. Furthermore internal domain-specific languages use a concept called **method-chaining**. Method-chaining is a method to write code that invokes a sequence of methods. Each method returns an object, which the next method is called on. If we take a look at the Java code from before, we see the difference between the use of a general-purpose language on the left and an internal domain-specific language on the right. If we look at the following line

```
idle
.actions(unlockDoor, lockPanel)
.transition(doorClosed).to(active)
;
```

we see how method-chaining makes the code much more readable than if we were to write the following:

```
idle.addTransition(doorClosed, activeState);
idle.addAction(unlockDoorCmd);
idle.addAction(lockPanelCmd);
```

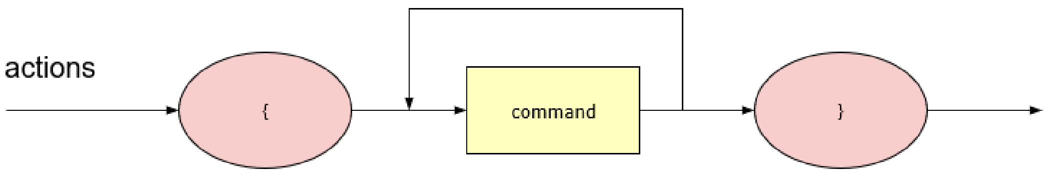
The code is not a set of instructions, separated by semi-colons, but has a sentence-like structure, that only makes sense if you consider it as a whole. This results in the language being much more fluent. This is why internal domain-specific languages are also called *fluent interfaces*.
On the other hand, the domain-specific language can be written in another language than the base structure. In this case the domain-specific language is called an external domain-specific language. This difference between the syntaxes of the base language and the domain-specific language result in a much clearer separation than in the case of an internal domain-specific language . The code, written in an external domain-specific language ,needs to be parsed into the language of the base structure.

3.2 The workings of a parser

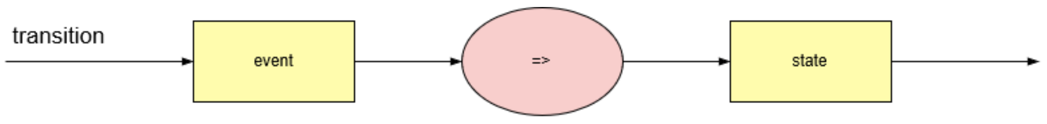
The parser is a program that gets a file in one language as an input and translates it into another language. In the case of the domain-specific language it scans the text for keywords and translates them into method-calls using the rules defined by the domain-specific language. It associates the keywords of the domain-specific language with *method-calls* in the host language.
In the example of our custom domain-specific language introduced in Section 2, the expression

```
state idle
  actions {unlockDoor lockPanel}
  doorClosed => active
end
```

would be parsed into Java code as follows: The keyword "state" is a declaration of the context. The parser would store the context, in this case the specific state "idle" inside a Variable. This Variable is called the **Context Variable**. The keyword "actions" is defined by the following rule:



The keyword *doorClosed* was defined as an event in an earlier stage. The parser looks it up inside the symbol table. The keyword => is defined as a transition. Transitions are defined by the following rule:



Now, the parser can translate the actions into the Strings `"idle.addAction(unlockDoorCmd);"` and `"idle.addAction(lockPanel);"` and the transition into the String `"idle.addTransition(doorClosed, activeState)"`. This can now be interpreted by a Java program.

3.3 Domain-specific languages populate models

As we see with the example of the treasure room a domain-specific language can be used as a configuration tool for a system. This approach of separating the configuration from the base system is very typical. The base system would be the **abstract** state machine. The output, that this configuration gives, is a **concrete** state machine. It is also commonly called the model generated by a domain-specific language. This model is also called the *Semantic Model* and it describes one specific output.
In the case of our leading example, the controller is configured through the domain-specific language. The domain-specific language describes a concrete state machine. It describes the states of the controller and the different transitions between them.

3.4 Types of models

For many programmers it might come as a surprise that there is more than one (computational) model. In fact most general-purpose languages, like JAVA, or C++ follow a concept that we refer to as the *imperative model.* This approach focuses on sequence of commands that the computer then sequentially executes.
Another model is the declarative model. This model is widely spread among functional and logical programming languages, such as Prolog or Haskell. Instead of giving a set of clear instructions the solution to the problem is only defined.
Those two are the principle programming paradigms. However a domain-specific language is a great tool to easily implement other computational models into a sequentially oriented programming language, such as JAVA. The state machine is only one example for this.

4 Cascading Style Sheets

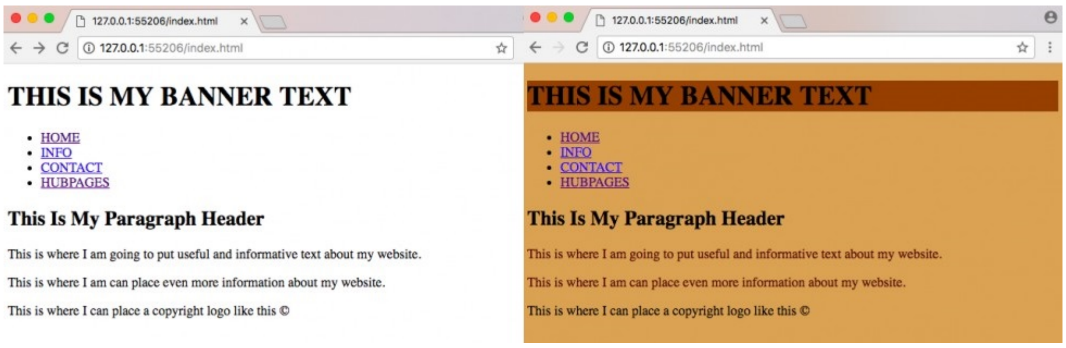
Using only `HTML` for web-pages results in websites looking very plain. This is due to the fact that `HTML` only describes the structure of web-page content. The browser uses a default style sheet for displaying those types of web-pages. This leads to the problem that the web-pages look different on different browsers and on different types of devices. To

give authors more control over the look of their web-pages, languages for style sheets were developed. CSS is a style sheet language that allows the combination of different style sheets into one representation.
CSS is an abbreviation for Cascading Style Sheets. The *cascading*-mechanism manages conflicts between style sheets by origin and specificity. Style sheets can come from three different origins: the author, the user or the browser. By default the priorities are in that order. This means that the style sheet of the author has the highest priority. The combination of different style sheets also allows partial style sheets. This has some great advantages namely smaller files and therefore faster transfer speed over the Internet. The specificity criterion allows for an easy manipulation of different elements. It means that specific properties overrule global properties.

```
    #banner {
    background-color: saddlebrown;
}
    body {
    background-color: rgb(209, 162, 98);
}
    .bodyText {
    color: #5b120c;
}
```

Looking at this very simple example [2] we can see why CSS is a domain-specific language ; It is very intuitive to read, it uses the vocabulary of the domain, namely words like color and background. CSS was developed specifically to make life easier for web developers and thus marks a really good choice for a domain-specific language .

We can use this CSS code to manipulate a web-page . In this figure we see the output file:



THIS IS MY BANNER TEXT

- HOME
- INFO
- CONTACT
- HUBPAGES

This Is My Paragraph Header

This is where I am going to put useful and informative text about my website.

This is where I am can place even more information about my website.

This is where I can place a copyright logo like this ©

THIS IS MY BANNER TEXT

- HOME
- INFO
- CONTACT
- HUBPAGES

This Is My Paragraph Header

This is where I am going to put useful and informative text about my website.

This is where I am can place even more information about my website.

This is where I can place a copyright logo like this ©

On the left side we see an example of a website programmed using only `HTML` and on the right side we see the same website, but now in combination with our CSS example.

5 Advantages and disadvantages of using a domain-specific language

As mentioned in section 1, the developers have to choose between the *generic* or the *specific* approach. Each of them come with their own advantages. We will now focus on the advantages and disadvantages of using the specific approach.

5.1 Advantages

One of the reasons computer scientists create domain-specific languages is to enhance the communication between developers and domain experts. The lack of communication is a common cause of project-failure. This is either because the customers do not fully understand the software or because the features they wanted do not work the way they imagined them to work.
An easy-to-read domain-specific language gets rid of this issue. The simplicity of the language makes it possible for the domain-experts to interact more with the developers and to provide them feedback on the project in an early stage. The increase of communication is accomplished by building a communication channel through the domain-specific language. By being able to read and understand the domain-specific language code, the domain-experts can help the developers to define test scenarios and spot mistakes. They can even redefine some rules of the domain-specific language and ultimately push it into a direction that is well suited for them.
Another aspect in favor of using a domain-specific language, is how it makes configuring systems easier for developers, as we saw in the example of CSS. The limited expressiveness leaves less room for potential mistakes. As the system gets more simple and has a more defined structure, it becomes easier to spot mistakes. It also makes the configuration of large systems more efficient as the actual configuration file in a domain-specific language is smaller compared to a configuration file in a general-purpose language. This is due to the fact that the parser does a lot of code generation. Another advantage is code-separation. By separating the code for the base structure, in our case the absrtact controller, from the configuration in the domain-specific language, they can be developed and upgraded independently. This also means that you can have specialized teams for each project, which enables a better project organization.

5.2 Disadvantages

A disadvantage of using external domain-specific languages is the complexity of the parser. External domain-specific languages need to be parsed. Not every language comes with a parser, which means that often times the development of a parser needs to be taken into account. This is especially the case if you want to implement your own domain-specific languages.
Furthermore parsing can also lead to less efficient code than if one were to write the code in the general-purpose language in the first place, as the parser itself only translates the code and does not do any optimization.
Another drawback is lack of tool support. Even though domain-specific languages are created to specify complex software systems, they are not efficient in practice as they are often difficult to integrate into existing revision control systems *IDE*s.[4] Lastly, using an external domain-specific language is followed by training costs. The developers have probably not had any prior experience with the language and need to be trained in using it. This needs to be taken into consideration as it costs time and money to train the developers into using a new software.

6 Implementing domain-specific languages

When developing a new system, a lot of research has to be done. As mentioned before, when implementing a domain-specific language one has to choose between an internal and external domain-specific language. It is important to be able to implement both as each of them come with their own advantages and drawbacks. One should always start by using an internal domain-specific language, as they are easier to implement.
By choosing to use an external domain-specific language, the objectives and the scope of the domain-specific language need to be specified. The developer can now look for similar domain-specific languages that have already been developed. If the existing ones do not apply to the specific case, the developer can choose to create their own language. Note that a parser also needs to be developed for external domain-specific languages. Most popular domain-specific language come with a parser. However some might not provide a parser for the host language. By developing their own domain-specific language, the developers also need to develop their own parser.

7 Conclusion

In conclusion, implementing a domain-specific language as a developer for the domain-experts may enhance the work experience for both parties. Seeing how a domain-specific language helps the developer communicate with the domain-experts, it makes up for an easier time creating something together. Following this, we got to learn the main difference between an internal and an external domain-specific language and how it is important to know how to implement both. Not only that, but we also saw how a domain-specific language can be an easy tool for non-programmers to partly understand and write code. While not always optimal, creating a domain-specific language comes with a lot of benefits and even if it is a concept that not everyone is familiar with, one should encourage the use of domain-specific languages.

References

[1] Don Batory, Clay Johnson, Bob Macdonald, and Dale Von Heeder. Achieving extensibility through product-lines and domain-specific languages: A case study, 2002.

[2] Bret Phillips. Basic web design with html and css, 2018.

[3] Martin Fowler Rebecca Parsons. *Domain-Specific Languages.* Addison-Wesley, 2010.

[4] Diomidis Spinellis. Notable design patterns for domain-specific languages. *The Journal of Systems and Software*, 56(1):91–99, 2001.

[5] Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000.