Usability Evaluation of Configuration-Based API Design Concepts

Thomas Scheller and Eva Kühn

Institute of Computer Languages
Vienna University of Technology
1040 Wien, Austria
{ts,eva}@complang.tuwien.ac.at

Abstract. Usability is an important quality attribute for designing APIs, but usability-related decision factors are often unknown. This is also the case when looking at APIs for configuration tasks, like for dependency injection or object-relational mapping. In these areas three different API design concepts can be found, which are annotations, fluent interfaces, and XML. There exists no research concerning usability-related characteristics and differences between these concepts.

In this paper, we present a usability study that identifies such characteristics and differences between the three concepts, by comparing three different variants of an API for dependency injection. From the study results we evaluate advantages and disadvantages in different use cases, and show how to build more usable configuration-based APIs.

Keywords: API Usability, API Design, Fluent Interfaces, Annotations, XML.

1 Introduction

There are many areas in programming where APIs (application programming interfaces) are used for some kind of configuration. Examples are APIs for object-relational mapping like Hibernate[1], APIs for dependency injection like Unity[2], or APIs for communication like the Windows Communication Foundation[3] (see Table 1). Next to logging and unit testing, these are the most prominent areas where external APIs are used. When looking at such "APIs for configuration", many of them share the same basic design concepts, independent from the area of usage. We identified three design concepts that are used for such APIs: The first is *XML*, where the whole configuration is not written in code, but stored in a separate XML file. The second is *annotations*, where the configuration is done by annotating code elements with additional information. The third is *fluent interface* [1], which has only recently become popular, and tries to make use of the natural language by defining methods that are concatenated to form a readable sentence.

[1] http://www.hibernate.org
[2] http://unity.codeplex.com
[3] http://msdn.microsoft.com/en-us/library/dd456779.aspx

A. Holzinger et al. (Eds.): SouthCHI 2013, LNCS 7946, pp. 54–73, 2013.

Table 1. Examples for configuration-based APIs

API for	Configures	Then does
Dependency Injection	bindings and injections	create instances
Serialization	which fields are serialized	serialize/deserialize objects
Object-Relational Mapping	mappings from code to db	insert, update and read data to/from the db
Communication	which messages are sent, which transport layer is used	send/receive messages

When evaluating the usability of such an API, it is therefore important to understand the usability implications of the used design concept(s). There is not yet any research concerning the usability of XMLs, annotations, or fluent interfaces in this context. Some existing papers deal with the usability evaluation of APIs [2,3,4,5]. They analyze the impact of parameters in objects' constructors, compare the usability of constructors and static factory methods, check the implications of method placement on API learnability, and evaluate the usability implications when using classes and methods in different scenarios. These papers cover well the usage of basic elements of object oriented APIs, but do not take higher level design concepts into account like the ones mentioned above. In the context of XML there are a few papers dealing with usability [6,7], but they are either too problem specific or not applicable in the context of programming. Further, there is a number of API design guidelines like [8] and [9], which give a good overview how to build usable APIs. While such guidelines show how to build APIs with each of the mentioned design concepts, there are none comparing them and/or saying when it is best to use which one. Further, they don't have a scientific basis. Today, when a programmer e.g. wants to choose a dependency injection framework, he/she has a large selection of different APIs with different design concepts. It has often been shown that usability is an important factor [10,11,12], both for API developers who need to know how to design an API, as well as for API users who want to choose the best usable API. But usability can only be taken into account if the according differences between design concepts are known.

Therefore, in this paper we want to evaluate the usability of the three API design concepts *XML*, *annotations* and *fluent interface*, and compare them to each other. We therefore conducted a usability study with three different APIs, each implementing one of these design concepts. As a result of the study, we want to show which designs perform better or worse in which situations, and which factors influence usability for each design concept.

In section 2 we present the design of the study, including the used APIs and details about the test execution and data analysis. Section 3 presents the study results, with statistical evaluations for each measured performance detail. An interpretation of the results is shown in section 4, where we present advantages, disadvantages and measurable properties for each design concept.

2 Design of the Study

To compare the three design concepts XML, annotations and fluent interface, we implemented an API for each one, all providing the same functionality. From the areas shown in Table 1 we chose dependency injection [13], because it is widely known, tests are easy to setup because there are no external dependencies (e.g. for testing an object-relational mapping API, a database would be needed) and there are many different APIs available for all three design concepts. To identify which tasks are most often done with a dependency injection (further in short: DI) API, as well as to see how the three design concepts are used in this context, we looked at a number of existing APIs. We examined Spring, Java EE, Google Guice and PicoContainer on the Java side, as well as Unity, Ninject, Castle Windsor, StructureMap, AutoFac and the Managed Extensibility Framework (MEF) on the .Net side. We concentrate our research on Java and C# because they are two of the most widely used object oriented programming languages.

2.1 Design of the APIs

Concerning the tasks the users should solve during the study, we chose the ones that were most common with existing DI APIs and ordered them so that users can use their knowledge from previous tasks, allowing us to monitor the learning effect. The following 6 tasks had to be solved in the given order:

1. creating a simple binding from an interface type to an implementation type
2. creating named bindings
3. defining for a class which constructor is used (constructor injection)
4. defining for a class which constructor parameters are injected, when it is instantiated using the DI API (constructor injection)
5. creating a binding with a singleton scope (so only a single instance is created)
6. applying all of the functions used in previous tasks on a more complex class structure, with three different bindings and two constructor injections

For each task, we evaluated the cognitive steps for solving it, e.g. for the first task this would be (1) choosing the interface type for the binding, (2) choosing the implementation type. The three APIs were then implemented following these cognitive steps as closely as possible, also using the same terms (e.g. “bind/binding”, “inject”, “instantiate”) wherever possible, so that there is no impact on the study results from such differences. We also closely followed the domain language dictated by existing APIs. Figure 1 shows for two examples how functionality was implemented in the three different APIs. To be able to test a more diverse range of programmers we implemented our APIs both in Java and C#.

Concerning the XML-based API, we observed that a majority of the available APIs do not directly provide an XML schema. Even if a schema is provided, it needs to be added manually to the IDE in a separate step from referencing the API (e.g. in Visual Studio this feature is rather hidden). We therefore decided that it would be more representative for existing APIs to not provide an XML schema for the study participants.

Class for binding:

```
public class ConsoleLogger
  : ILogger {}
```

Annotations:

```
[BindingFor(typeof(ILogger))]
public class ConsoleLogger...
```

Fluent Interface:

```
container.Bind<ILogger>()
   .To<ConsoleLogger>();
```

XML:

```
<DIContainer>
<Bindings>
<Binding type="Classes.ILogger"
bindTo="Classes.ConsoleLogger"/>
</Bindings>
</DIContainer>
```

Class for constructor injection:

```
public class Service {
  public Service() {...}
  public Service(ILogger logger)
       {...}
}
```

Annotations:

```
[Inject]
public Service(ILogger logger)
    ...
```

Fluent Interface:

```
container
 .WhenInstantiating<Service>()
 .UseConstructorWithTypes<
     ILogger>();
```

XML:

```
<DIContainer>
  <Instantiations>
    <Instantiation
     type="Classes.Service">
      <Constructor>
        <Param name ="logger" />
      </Constructor>
    </Instantiation>
  </Instantiations>
</DIContainer>
```

Fig. 1. Code examples in C# for (a) creating a binding and (b) defining an injection constructor for all three APIs

In addition to the API itself, a short tutorial document was created for each API. Again attention has been paid to make the three tutorials as similar as possible, to minimize influence of the documentation structure on the study.

2.2 Participants and Measurement

Our study included 27 programmers (9 per API). 17 were Java programmers using Eclipse as IDE, 10 were .Net programmers using Visual Studio. The study involved both programmers with academic and industrial background, which had experience with a variety of different APIs. The participants were between 22 and 46 years old and had between 2 and 20 years of programming experience. All programmers were recorded using a screen capturing software, and a supervisor was sitting next to the programmer to explain each task. The tests were designed as unittests, so users could evaluate the code at any time (a successful test run marks the end of each task). Additionally, an eye tracker was used to capture the participants' eye movements, allowing us to add gaze replays to the screen capture data. A between-subjects design was used for the study, meaning each participant was only tested with one of the three APIs, to prevent a falsification of the data from cross-API learning effects.

Overall 180 values were measured per participant. From the captured video data, we extracted times for specific steps that programmers had to do in each

task, in the same way as we did in [3]. These steps include instantiating the DI container, creating a certain binding, defining the constructor injection for a certain class, reading a certain part of the tutorial, running a test and searching for errors. This way of fine grained evaluation removes unnecessary noise and by that allows a more accurate statistical evaluation.

When participants used the tutorial, we focused on identifying how long they took to understand certain parts of it, and how long they spent on reading either the running text or the included code examples. In this case the eye tracking data was especially helpful, because participants would sometimes put code and tutorial side-by-side on the screen, which would have made it impossible to detect what the user was looking at without this data.

In addition to performance times some other values were evaluated, like the number of switches between tutorial and code, the number of switches between classes in the IDE, and the number of needed test runs. At the end of the study participants had to fill out a short questionnaire concerning their experience with other DI frameworks, their satisfaction with the used API, and what they thought were advantages and disadvantages of each of the three design concepts.

3 Study Results

We evaluated all our results with statistical data analysis methods. If not mentionend otherwise, a *t-test* was used, which is a parametric statistical hypothesis test for assessing whether one of two samples of independent observations tends to have larger values than the other. For each applied test the p-value is given below, which represents the probability that there are significant differences between the two samples (a significant difference is present when $p<0.05$).

In the following sections, the letters A, F and X are used as short forms for the three API variants (annotations, fluent interface, XML).

3.1 Performing Simple Tasks (Creating a Binding)

The action of creating a binding represents a simple task which can be solved using a single annotation, or only very few methods or xml elements. The basic code for this task is shown in Figure 1(a). In the first task, users had to write exactly this code, in successive tasks they had to additionally define a binding name, with one additional value/method/xml attribute needed in the code.

Figure 2(a) compares the times for all tasks that were needed for first reading the tutorial section about bindings and then creating the binding. The median times are 42s for A, 34s for F and 65s for X. A check for statistical significance shows that the times for both A and F are significantly smaller than X (t-test: $p<0.001$). When removing one extreme outlier of F, the times for F are also significantly smaller than A ($p=0.042$). So, for simple tasks users performed best with the fluent interface, closely followed by the annotations API. The large gap from X to the other two APIs is not unexpected, as users of the XML API spent much time writing the XML configuration without auto completion features available.

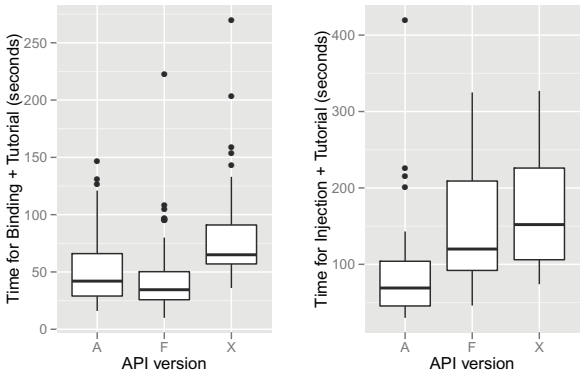

Fig. 2. Boxplots: (a) Time needed for creating a binding incl. reading tutorial, (b) time needed for constructor injection incl. reading tutorial.

3.2 Performing Complex Tasks (Constructor Injection)

Performing constructor injection represents a more complex task. Users had to use multiple annotations, chain a higher number of methods in the fluent interface, and use more xml elements and attributes. The basic code for this task is shown in Figure 1(b). Users first had to use constructor injection in task 3, and later in tasks 4 and 6. In tasks 4 and 6 users additionally had to specify which bindings are used for the injection of the constructor parameters, resulting in additional complexity.

Figure 2(b) compares the times for all tasks that were needed for reading the tutorial section about constructor injection and configuring the injection. The median times are 69s for A, 120s for F and 152s for X. A check for statistical significance shows that the times for A are significantly smaller than both F and X (t-test: $p=0.017$ for $A<F$ and 0.002 for $A<X$). The times for F are not significantly smaller than X ($p=0.195$). The result of the annotations API being the best was expected because in this case much work is done implicitly just by choosing on which code element the annotation is placed. E.g. when the injection for constructor parameters needed to be defined, an annotation needed to be placed directly on the constructor parameter. With the other two APIs, users first needed to explicitly define the class and the constructor before they could even start with defining the injection of the parameters. This leads to the conclusion that the deeper within a hierarchy something needs to be configured, the greater the advantage of annotations against the other API variants will be.

What is surprising is that compared to simple tasks (see above) users with the fluent API were now much slower. This may indicate that a more complex method chain with a higher number of chaining options imposes a significant difficulty on the users.

3.3 Overall Times

Interesting details can also be found when looking at how much time users needed for each task depending on the used API. Figure 3 shows boxplots for the overall times per task, for each task comparing the three different APIs (see section 2.1 for a description of the tasks).

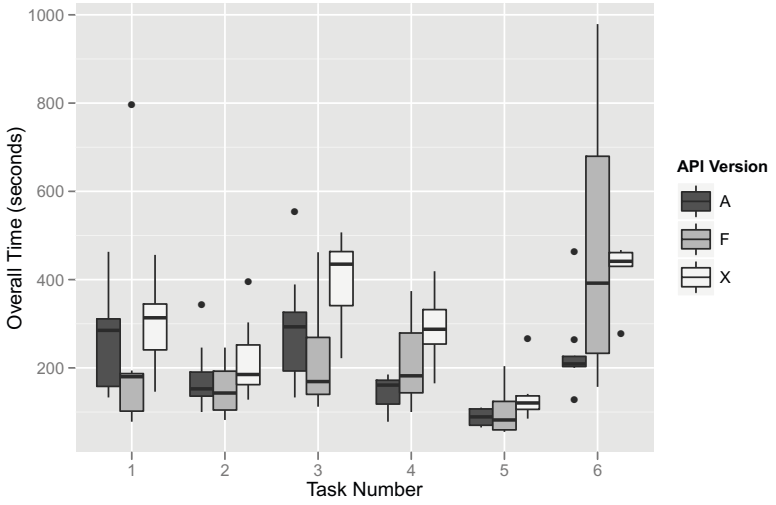

Fig. 3. Boxplot: Overall times per task and API version

The first thing that can be noticed is the learning effect. This can best be observed when comparing tasks 1 and 2 (users needed to create a binding), as well as when comparing tasks 3 and 4 (users needed to use constructor injection). While both the annotations and the XML API show a clear speedup, there is hardly one to see for the fluent interface. This implies that the fluent interface has a more flat learning curve than the other two.

A second thing that confirms this suspicion are the times for task 6. For this task, users had to use everything they had previously learned, and we asked them to try to do it without looking into the tutorial if possible. Only for the XML API 5 of the 9 users had to look into the tutorial to solve the task, which was simply because they did not remember all needed XML elements and attributes any more, and were not able to use auto completion for this. When looking at the times for task 6, it can be noticed that while the spread is very small for A and X, it is very large for F. From this we interpret that users of A and X all had at this point gained a complete understanding of the APIs and knew exactly how to use them – so solving a task of this complexity takes about 200-230 seconds with the annotations API, and about 420-470 seconds with the XML API. On the other hand, with the fluent interface some users were very fast (about 200

seconds), but others were extremely slow (up to 15 minutes), which means that many users still did not understand the API well enough to use it in a fast and efficient way. This could also be observed during the test, where users that had a good understanding of the API were very focused on the task and used API functions intuitively, while others needed to stop and think about how something is done with the API and also needed much more time searching for errors that were related to wrong API usage.

3.4 Information Sources

For each action the users needed to take during the test we analyzed where he/she acquired the information needed for that action. We distinguished between three different sources of information: The first was that users needed to look into the tutorial, the second that they used the auto completion mechanisms of the IDE, and the third was that they already had all the information they needed and therefore didn't need to look anywhere. For auto completion we expected that it would be most useful for the fluent interface, since it always presents the methods that can be chained next.

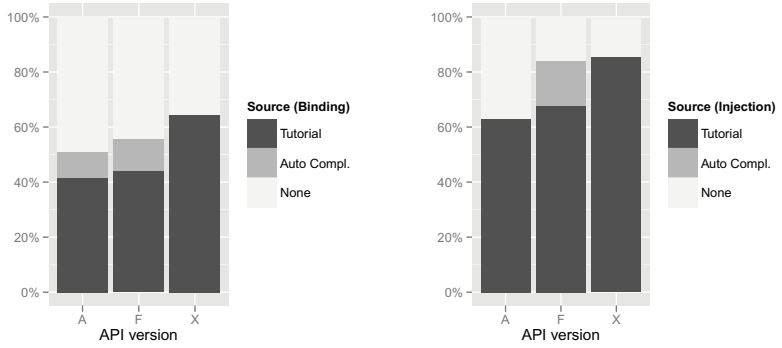

Fig. 4. Which information source was used how often, compared by Task, for the action of (a) creating a binding, (b) constructor injection

Figure 4 shows the evaluation of information sources for the actions of creating a binding (the simple task as explained in section 3.1) and constructor injection (the complex task, see section 3.2). Unsurprisingly the tutorial was used most often with the XML API (65% of the time for creating a binding, 85% for constructor injection), which is partly because of the missing auto completion. The users of the annotations API needed the tutorial the least often (40% and 60%), especially for the task of injection, where 20% more of the tasks were solved without using any information source than with the other APIs. Auto completion was most used with the fluent interface as expected (most clearly visible in Figure 4(b)), but an advantage due to using auto completion can only be seen compared to XML.

Generally, the main advantage of using auto completion was actually not that users could search for a method within the IDE, but simply that they didn't need to remember the whole class or method name. If the users only remembered the first few letters, they could still easily use a method by just auto-completing it. So, although it may appear in Figure 4 that auto completion was rarely used, it was actually used very often, but only in 10-20% of the cases it was really used for searching. For simple completion of class, method and property names it was used in about 80% of all cases.

3.5 Time Spent in Tutorial

We analyzed how much time users spent in the tutorial and what they were reading there. The result can be found in Figure 5(a). It shows that users spent about 30 to 35 seconds per task in the tutorial, with no significant differences between the three APIs.

The eye tracking data additionally allowed us to evaluate how much time user spent reading the running text, and how much reading only the headers and code examples (in all tutorials about 50% was running text, and the other 50% code examples). By doing that we intend to gain evidence on the self-explainability of each API. We observed that users almost always tended to look at the code examples first (after finding the correct section by scanning the headers), and only look at the running text when there was something they didn't understand or the code behaved not as expected. Figure 5(b) shows the percentage of time that was spent in the tutorial only reading headers and code examples. While the median percentage is 81% for A and 63% for F, it is 100% for XML, meaning in over half of the cases users spent no time at all reading the text in the tutorial. This can also be seen when looking at Figure 5(c), which shows the time spent reading running text. The median values are 9s for A, 21s for F and 0s for X. In this case we used the non-parametric *Wilcoxon Rank Sum Test* to check for statistical significance because the data shows a strong floor effect and is therefore not normally distributed. X is significantly lower than F ($p=0.010$), but not significantly lower than A ($p=0.155$), and A is very close to being significantly lower than F ($p=0.052$). This leads us to the conclusion that the XML code has the highest self-explainability because most of the time users didn't need to read any additional text to understand it. On the other hand, users of the fluent interface did not get enough understanding from just the code examples and therefore needed to read the whole tutorial more thoroughly.

The question may arise why the overall tutorial time for XML was not lower than the others. One reason is that due to the missing auto completion users often needed to switch into the tutorial just for looking how XML elements and attributes are written or how the exact structure looks like. Users of the XML API also switched into the tutorial much more often because of that.

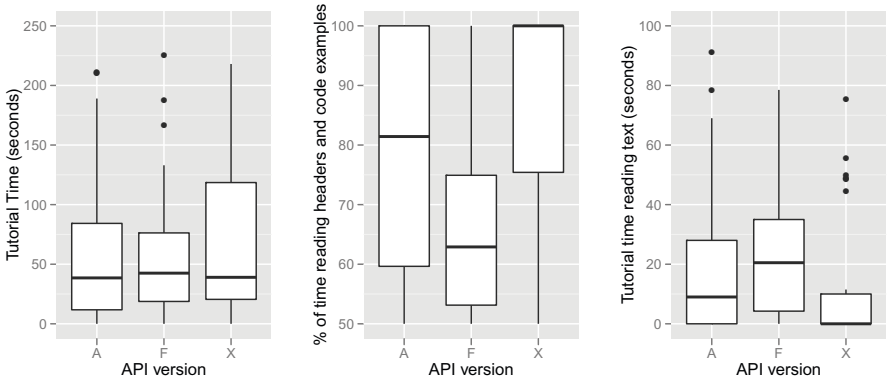

Fig. 5. Boxplots: (a) Overall tutorial times per API, (b) Percentage of time spent reading only headers and code examples, (c) Time spent reading running text

3.6 Switches between Classes

We observed that the three APIs differed in the number of times that users needed to switch between different classes in the IDE. Figure 6(a/b) compares how often users switched between classes, for the tasks of creating a binding (a) and constructor injection (b). While A and X show only a slight difference between the two different tasks, there is a large one with F. This is because with the fluent interface, users only switched between classes when they wanted to take a look at the class contents. This was mostly when doing constructor injection, to look at the available constructors and their parameters, but hardly ever happened for creating a binding. When looking at the large time differences between simple and complex tasks for the fluent interface (see Figure 2), it is highly probable that this is also due to an impact from the number of class switches. While the switching itself doesn't take much time, it is the act of looking up information in the configured class (e.g. parameter types and names of the constructor) that is time-consuming.

A check for statistical significance shows that for creating a binding (Figure 6(a)), the number of switches with F is significantly lower than for A and X (t-test: both $p<0.001$). In the other hand, for constructor injection (Figure 6(b)) the number of switches for A is the lowest ($p=0.062$ for A<F and 0.002 for A <X). The result that X has equal or more switches than A was unexpected, since with the annotations API users needed to switch a lot because annotations need to be applied directly to the targeted classes. The reason for the XML API needing so many switches was that most users opened every class to copy-paste the class name and often a second time to copy-paste the package/namespace name, because there is no auto completion of class names in the XML. Especially when doing constructor injection, users of F and X needed to switch back and forth multiple times, while with A switching to the class once was often enough.

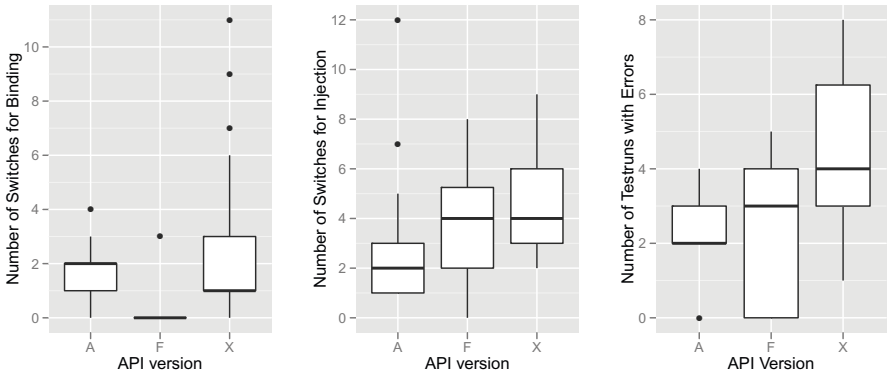

Fig. 6. Boxplots: (a) Number of switches between classes in IDE for creating a binding, (b) Number of switches for constructor injection, (c) Number of test runs with errors

3.7 Error Rate

To check if more errors were made with one API than with another, we compared the number of test runs that were needed to finish all tasks, as shown in Figure 6(c). All tasks were designed as unit tests, and to show that a task was finished correctly (or to check for runtime errors), users had to run the tests. The median number of test runs with errors was 2 for A, 3 for F and 4 for X. So with the XML API an error happened twice as often as with the annotations API. The number of errors with XML is significantly higher than with the other two (t-test: p=0.019 for X>A, p=0.043 for X >F). There is no significant difference between A and F (p=0.485 for A<F).

The main reason for this is of course the missing auto completion in the XML configuration, and that errors are therefore also not discovered at compile time. Errors happened especially often for longer and more complex words like "Instantiation". Another typical error was that the package/namespace of a class was incorrect, because the classes were sometimes placed in a different namespace/package than the one that was normally used (which was actually prepared exactly to show that this can lead to errors in the XML configuration, while not affecting the other two API variants).

3.8 Programmer Satisfaction

After the test we asked each programmer to rate 3 aspects of the used API on a 5 point differential scale as suggested in [14]. The measured aspects were simplicity (Simple ... Complicated), satisfaction (Pleasing ... Irritating/Frustrating) and speed (Fast to use ... Slow to use).

The results are shown in Figure 7. For *Simplicity*, both A and X were rated with a median value of 5 (the highest rating). F achieved a median value of 4, which is significantly lower than the other two (t-test: p=0.016 for F<A and

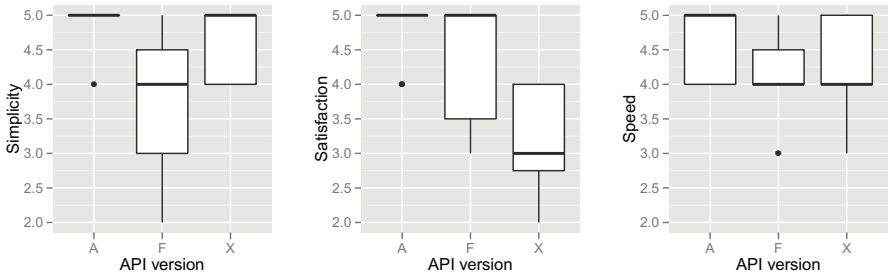

Fig. 7. Boxplots displaying the results of the post-test questionnaire (1-5, 5=highest): (a) Simplicity, (b) Satisfaction, (c) Speed

0.041 for F<X). On the first look it may be unexpected that the XML API was rated significantly higher in simplicity than the fluent interface. But this result also supports the assumptions made in section 3.5, that XML has an especially high self-explainability and can be easily understood. With the fluent interface users had difficulties in this area – often users stated that they were unsure which methods can be chained or which must be used consecutively, especially when using more complex functions.

For *Satisfaction*, A and F were rated with a median value of 5, while X was rated significantly lower with a median of 3 ($p<0.001$ for X<A and 0.014 for X<F). The reason for XML being rated worst is clearly the missing auto completion, both for typing XML element and attribute names, and for typing class names in the XML. Although working with the fluent interface was less simple, it was still more satisfying because nothing needed to be typed in manually.

The smallest differences are shown for *Speed* with a median value of 5 for A and 4 for F and X. There are no significant differences in the speed values, with F<A being the one closest to significance ($p=0.060$). Especially the fluent interface and XML APIs were rated very equally, although when looking at the performance the fluent interface actually was almost always significantly faster.

When combining the results for all three rated aspects, the annotations API was rated significantly higher than the fluent interface and XML APIs, which were both rated equally.

3.9 Programmer Preferences

In addition to the questionnaire, we asked users about their general opinions on the used API and, in case they had used the other two design patterns, which one they preferred and why. From the 27 test users, 10 users had never used a DI API before, 5 had worked with an annotations-based API, 9 had used an API with a combination of XML and annotations, and 3 a fluent interface. When asked which kind of API they preferred, 9 users had no preferences because of insufficient experience, 12 preferred an annotations-based API, and 6 and API with a fluent interface. None of the users gave a preference for XML.

For annotations the by far most often mentioned advantage was that they can be seen directly in the code that is configured, which makes it easy to understand how the code behaves because you don't need to look somewhere else for the configuration. Only few users mentioned other advantages, e.g. that annotations are safer when refactoring code (like renaming parameters) than the other variants, and that no additional file needs to be managed that holds a configuration. The most often mentioned disadvantage was that annotations are very static and don't allow different configurations for the same class, like having one configuration for unittesting and another for the production environment. Further, some mentioned it was a disadvantage that you have to look into every file separately to see the configuration, so it can be difficult to gain an overview.

Some advantages were mentioned for both the fluent interface and XML, which is not surprising because they are similar in certain points. These advantages were that the whole configuration is in one place and is therefore easier to overview (which is interestingly the opposite of the most popular advantage of annotations), and that the configuration is independent from the configured code, also allowing different configurations if needed.

For the fluent interface, other mentioned advantages were that the chaining of methods is very comfortable and easy to use once you are familiar with it, and that you can do almost everything with auto completion (unlike with XML). Especially for users that mentioned the former it could often be observed that they were mumbling to themselves in sentences similar to the method chains. For example, when users needed to write the code for constructor injection, which starts like `container.WhenInstantiating<Service>().UseConstructorWithTypes<...`, they mumbled something like "when I am instantiating the class Service, then I want to use the constructor with the types ...". We also see this as a confirmation that once the structure of a fluent interface is understood, it allows coding by building such sentences very intuitively. The main mentioned disadvantage of the fluent interface was that as soon as you need to configure something that cannot be accessed with methods and generics, like constructor parameters, you need to use strings which feels very unpleasant and is not safe to refactoring (unlike annotations, which don't have this problem). All other disadvantages were centered around the concept of chaining methods, and displayed well the problems we observed during the tests. Users said that it was unclear in the beginning which methods could be chained with which others, that they were unsure if a method chain was complete or if they had forgotten anything, and that longer method chains (4+ methods) were increasingly complex.

For XML, a mentioned advantage was that the configuration can be changed without recompiling, though on the other hand favourers of the fluent interface mentioned that this was completely unnecessary in most cases. Further, the actual API that is needed in the code is very small and simple: For the DI API it only needed one function, which was creating instances. The main disadvantage was unsurprisingly the missing auto completion, and that due to this there is also no compile-time error checking. A few users noted that this would not have been as much a problem if a schema file had been provided. Some additionally

mentioned the problem of refactoring, that most of the time the configuration file is not included in the refactoring process, so refactorings can easily destroy the configuration without directly being noticed.

3.10 Impact of Programmer Experience

An important question is if the study results can be applied to both experienced and inexperienced users. In [3] we analyzed if the years of programming experience and/or the domain knowledge had any impact on performance, with the result that there was none. We did the same tests for this study, with the same results. Additionally we asked users if they were using any DI framework frequently. A performance comparison shows that users that were using a DI framework regularly were not significantly faster while writing code (all needed about 16 minutes overall). But they spent significantly less time in the tutorial (t-test: $p<0.001$): The median times are 338s for non-regular DI users and 201s for regular ones. It should be noted though that the significance of these results is limited since only 4 of the 27 users were regular DI framework users.

4 Interpretation of Results

Based on the study results we can now present an overview of advantages and disadvantages of the three design concepts, as well as an analysis which one should be used in which cases. Further we analyze the possibilities for automatically measuring usability and/or making suggestions on the usage of these concepts, which would be necessary for integration into an automated measurement method as we proposed in [15].

4.1 Advantages and Disadvantages

Annotations. The annotations API performed either best for almost all measured results. Users were fastest especially when performing more complex tasks, the API was well understood from the tutorial, the learning curve was high, the error rate low, and it was also rated overall best for simplicity, satisfaction and speed. As the main reason for this we see the fact that considerable amount of information is given simply by the fact where the annotation is placed. So, while with the other APIs the users needed to explicitly say "I want to configure the class Service, and use the constructor with the following parameters...", this was naturally defined with annotations just by the fact that the annotation was placed at this constructor. Therefore also much fewer code elements were needed, e.g. only a single parameterless annotation for defining the injection constructor, compared to two different methods with multiple parameters with the fluent interface. This advantage was stronger, the deeper in a hierarchy the annotation was placed. It was strongest when it needed to be placed at a constructor parameter, which is in the third hierarchical level (class>constructor>parameter). An additional reason for the performance advantage was found in the number of

class switches. For complex tasks users needed to look into the classes to create the configuration, which made it necessary for users of the fluent interface and XML to switch back and forth between class and configuration. For annotations this was not an issue, since the class and configuration were at the same place. Only for very simple tasks (operating on the first hierarchical level) the fluent interface had a small performance advantage against annotations (as shown in section 3.1), in which case the users could solve the task without ever switching into the configured class.

The simpleness of annotations was also well expressed by the opinions of the users, since a majority of them gave a preference to annotations. What users like most about annotations, and what was also the most often made statement concerning preferences, is that they are applied directly to the code that is configured. By having the configuration and the element that is being configured at the same place, the code is especially easy to understand, and there is no need to look somewhere else.

There are only few mentioned disadvantages, one being that it gets complicated to get an overview in larger projects because the configuration is spread out across a large number of classes. This was out of the scope of this study, but would be interesting to evaluate in future studies. In general it can be said that from a usability viewpoint, if the use case allows it, an annotations based API should be preferred to fluent interfaces and XML.

Fluent Interface. The fluent interface showed its main advantages in type safety (especially when compared to XML), as well as in the intuitive way that a method chain can be formed like a sentence in natural language. This enabled users to program in the same way they were thinking, e.g. when thinking "I want to bind ILogger to ConsoleLogger", the code they needed to write was `container.Bind<ILogger>().To<ConsoleLogger>()`.

On the other hand, the chaining of methods was also the main source for complications. While there was no problem chaining only two or three methods, users got confused when more methods were needed, and were wondering whether there was a needed order or if a certain method needed to be used or not. Most users seemingly thought that it was not ensured by the API itself that methods can only be called in a valid order (which the API of course did), and that they needed to do that themselves. To illustrate that, Figure 8 shows the chaining tree for the methods that were available in the study. The arrows show which methods can be called in succession, e.g. to define a named binding with singleton scope, the methods `Bind`>`To`>`Named`>`InSingletonScope` can be chained. In this case, either of the methods `Named` and `InSingletonScope` could be called first, or only one of them could be used alone if the other was not needed. Most often this problem occurred with the more complex chaining tree starting with `WhenInstantiating`. Users were often unsure if `UseConstructorWithTypes` was needed before defining injections for the parameters with `ForConstructorParam`, although the tutorial even contained examples where one was used without the other.

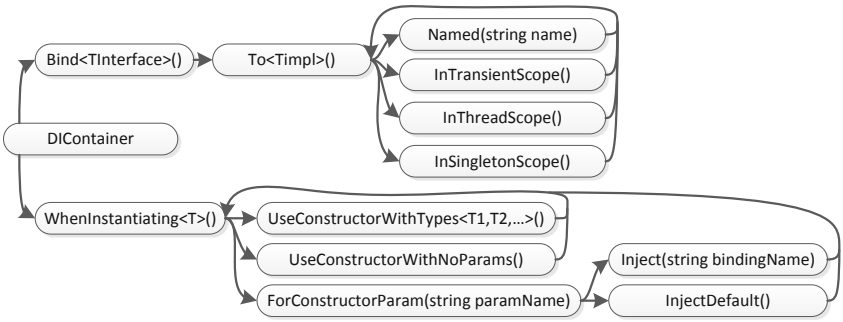

Fig. 8. Method chaining tree of the fluent interface API

In addition to not knowing if the order is correct or a certain method was needed, users had problems because they were missing some kind of visible ending to a method chain. This is certainly a problem with fluent interfaces, because you can of course just stop writing after any method without the compiler marking any errors. So, an error because of an incomplete method chain would only be found at runtime. In some fluent interfaces we evaluated, an ending method was used to mark the ending of a configuration (like "Start" or "Do‘"), but with such an ending the chain unfortunately looses readability, since a natural sentence would never end with such a word.

It is to be expected that many of these problems occurred because most of the users had never used a fluent interface before. But rather than that being a threat to the validity of the study, we think that this just shows that fluent interfaces are not widely known, which makes them harder to learn and use.

When looking at these results and the statements of the users, there are some things that can be done to maximize the usability of fluent interfaces: First, users performed much better with short chains (up to 3-4 methods), so if some use cases would require the users to build especially long chains, this could be improved by changing the methods so that the chain is shortened. Second, to minimize the insecurity of users concerning the order and selection of chained methods, the API should whenever possible prevent the user to chain methods that would build an invalid chain. This can be a very difficult task, especially when the API has a large number of chainable methods, but can prevent many potential usability problems.

XML. As a main advantage of XML we identified its understandability and self-explainability. This can be seen when looking at the times spent in the tutorial, where most of the time the code examples alone were sufficient for understanding how to create the XML configuration. We see the reasons for that in the simpleness of the XML language itself (there are just two components that need to be understood, namely elements and attributes, which is far less than in a programming language like Java or C#), as well as the fact that nearly every programmer has already used XML in some way. This makes the users

comfortable with XML from the start, which is also shown in the ratings for simplicity, where XML showed good results.

Unfortunately, this cannot compensate the big drawback of missing auto completion. Again it needs to be said that a result of the evaluation of various XML based APIs was that most of them don't offer an XML schema. Even if they offered one, it is not automatically integrated into the IDE, and manual integration is often not done. Although many users criticised that an XML schema was missing, e.g. for Visual Studio none of them even knew how a schema could be integrated (it is a function hidden deep in the options menu). Since we wanted to show the most common use case, we therefore decided that the study should also not contain an XML schema.

If a schema is present, this would definitely improve the performance of XML since it makes creating the configuration faster and prevents typing errors. But it doesn't remove all disadvantages, because there is still no checking of package, class and parameter names. Especially when creating a larger configuration, users rely strongly on copy-pasting the XML elements (this was also observed in the study), which minimizes the problem of typing errors for XML elements, but doesn't change anything about the problems with package/class/parameter names. To prevent this problem, users in the study often copy-pasted the class and package names, which was a significant slowdown. These problems are also displayed in the ratings for satisfaction, where the XML API was rated lowest.

From the performance results it can be said that in general one of the other two design concepts should be preferred to XML, as long as the use case allows it. In all performance ratings except for the tutorial, XML performed worst of the three, sometimes even taking twice as much time, because of the mentioned disadvantages (Figure 3 makes this especially obvious). If XML needs to be used, the study has shown that especially an intuitive and simple XML structure with short and easy to memorize element and attribute names can improve working with the API and prevent unnecessary errors. Further, an XML schema should always be made available.

4.2 Suitability Per Use Case

The design concept that has been identified as easiest to use is annotations, so it should be the first choice when deciding for a configuration-based API to use. An annotations API can be used whenever the targets of the configuration are classes (e.g. for the areas shown in Table 1), and when it is possible to statically apply the configuration directly to the classes (which can only be done when the source code of the classes is accessible and can be changed). A fluent interface can be considered alternatively, if the configuration is not too complex (if the configuration of constructors, methods and parameters is not involved). The fluent interface is not limited to only a single configuration, so for simple configurations it could be preferred to annotations.

The only reason to use XML would be a case where it is absolutely necessary that the configuration can be changed without recompiling. Such a case would for example be logging. To be able to switch logging on and off (or change the

log level), it would make no sense to compile the configuration into the program. In all other cases, where Annotations cannot be used, fluent interfaces should be the first choice, since despite being harder to learn the study showed that they provide significantly better usability than XML. Examples where a fluent interface can be the best choice are APIs for unit testing and mocking.

Many existing APIs show that also a combination of multiple design concepts can make sense, e.g. to have a fluent interface and combine it with annotations for more complex class configurations.

4.3 Automated Measurement

One of our goals is to find properties that could be integrated into an automated measurement method as described in [15]. Usability is evaluated by analyzing the structure of an API as well as usage examples.

Automatic recognition of the three design concepts can easily be done: Annotations and XML can both be recognized by the presence of the corresponding elements. A fluent interface can be recognized in the code examples by two or more methods of the API being chained together. In case of the presence of XML, a recommendation could be given to use either annotations or a fluent interface instead. In case of a fluent interface, if longer method chains are present annotations could be recommended as an alternative, if the configuration involves classes (this could be recognized by checking if one of the fluent methods takes a class as parameter type).

For XML, the understandability of the structure is very important. One measurable parameter can be the number of elements and attributes necessary to solve a certain task. Especially when compared to a fluent interface, elements and attributes are mostly equivalent to methods and parameters, not only in their structure but also in the necessary numbers for solving a task. Overall, the usage the XML took about 50% more time than the fluent interface, and should be rated accordingly more complex. Additionally, the fact that more complicated element names were especially hard to memorize could be considered by checking the length of the names. In the study, names with more than 11 letters were perceived especially difficult.

For the fluent interface, an important parameter is the number of chained methods. The complexity showed more an exponential than a linear increase, especially for 4+ methods. Other than that, the number of chaining possibilities (size of the chaining tree) could be an indicator for a hard to understand fluent interface.

The study shows that while the number of code elements increases with more complex tasks when using XML or fluent interfaces, it stays mostly the same for annotations, because they are simply placed deeper in the hierarchy. So, this advantage could be measured by the number of needed code elements. While applying an annotation needs more effort than calling a single method, the advantage is simply achieved by the small number of code elements. According to the study results, using one annotation with a single parameter is about equal to two methods with a single parameter each in the fluent interface.

4.4 Validity and Applicability of the Study Results

The general validity of the study has been ensured by the usage of common statistical data analysis methods, as well as by having a sufficient number of participants with different backgrounds and different levels of experience.

To ensure that our study is not only applicable to APIs for dependency injection, we evaluated a much wider range of APIs, for the areas that were shown in Table 1. For all of these areas, there are implementations available using the three presented design concepts. The evaluation showed that the general structure, as well as the usability-specific differences between the different design concepts are basically the same for all of these areas. We therefore conclude that our study results are also applicable to at least these areas.

Of course there are also limits to what can be checked with such a study, like how efficient users can work with an API after they having used it for several hours, days or weeks of experience. Especially for the fluent interface, it can be expected that users would have been able to improve further (see section 3.3), although it is unlikely that it would perform better for complex tasks than annotations, so there would be little change to the overall results.

While the study focused on evaluating how programmers work with an API, an additional interesting fact to research in the future would be reading and understanding existing code. This could further strengthen the study results and/or show additional interesting usability aspects. Also a usability evaluation in the context of large projects could bring additional valuable results. Especially the missing support for refactoring in an XML could become a serious problem in this case, and with Annotations the scattering of the configuration over a large number of classes could make it difficult to maintain.

5 Conclusions

In this paper, we presented a study for evaluating three configuration-based API design concepts: annotations, fluent interfaces and XML. Users performed best and were most satisfied with annotations, while they performed worst with XML. Annotations proved especially useful when configuring elements deeper in a hierarchy, but can in general only be used when the targets of a configuration are classes. For most other cases, the best choice is a fluent interface, where users performed very well with small method chains, but had often problems when four or more methods needed to be chained. Also, the fluent interface paradigm proved hardest to understand, partly because of the fact that it is the least common one. XML showed its strengths by being very easy to understand and self-explainable, but proved the most cumbersome to use because of its missing auto completion, which was an issue both for XML elements and attributes, as well as for class and package names. Based on these results, we gave suggestions how usability could be improved, and extracted several properties that could be integrated into an automated measurement method. The paper hereby presents an important step towards a better understanding of API usability.

For future work, we plan to conduct further studies to gain more understanding of API usability. In the area of configuration-based APIs, a study on the understandability of existing code and suitability for larger projects would be especially interesting.

Acknowledgments. The work is funded by the Austrian Government under the program BRIDGE (Brückenschlagprogramm der FFG), project 827571 AgiLog – Middleware technologies to reduce complexity for agile logistics. Special thanks go to our project partner pcsysteme.at, all study participants and the Interactive Media Systems Group at TU Vienna for lending us their eye tracker.

References

1. Fowler, M.: Domain-Specific Languages. In: The Addison-Wesley Signature Series. Addison-Wesley (2010)
2. Ellis, B., Stylos, J., Myers, B.: The factory pattern in API design: A usability evaluation. In: Proc. of the 29th International Conference on Software Engineering, ICSE 2007, pp. 302–312. IEEE Computer Society, Washington, DC (2007)
3. Scheller, T., Kühn, E.: Influencing factors on the usability of api classes and methods. In: 19th Annual IEEE International Conference and Workshops on the Engineering of Computer Based Systems, ECBS 2012, Novi Sad, Serbia, pp. 232–241. IEEE Computer Society (2012)
4. Stylos, J., Clarke, S.: Usability implications of requiring parameters in objects' constructors. In: Proc. of the 29th International Conference on Software Engineering, ICSE 2007, pp. 529–539. IEEE Computer Society, Washington, DC (2007)
5. Stylos, J., Myers, B.A.: The implications of method placement on API learnability. In: Proc. of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT 2008/FSE-16, pp. 105–112. ACM, New York (2008)
6. Graaumans, J.: A qualitative study to the usability of three xml query languages. In: Proc. of the Conference on Dutch Directions in HCI, Dutch HCI 2004, pp. 6–9. ACM, New York (2004)
7. Sapienza, F.: Usability, structured content, and single sourcing with xml. Technical Communication 51(3), 399–408 (2004)
8. Cwalina, K., Abrams, B.: Framework design guidelines: conventions, idioms, and patterns for reusable. net libraries, 1st edn. Addison-Wesley Prof. (2005)
9. Tulach, J.: Practical API Design: Confessions of a Java Framework Architect, 1st edn. Apress, Berkely (2008)
10. Clarke, S.: Measuring API usability. Dr. Dobb's Journal 29, S6–S9 (2004)
11. Henning, M.: API design matters. Queue 5, 24–36 (2007)
12. Robillard, M.P.: What makes APIs hard to learn? answers from developers. IEEE Software 26, 27–34 (2009)
13. Fowler, M.: Inversion of Control Containers and the Dependency Injection pattern (January 2004), `http://martinfowler.com/articles/injection.html`
14. Nielsen, J.: Usability Engineering. Morgan Kaufmann, San Francisco (1994)
15. Scheller, T., Kühn, E.: Measurable concepts for the usability of software components. In: Proc. of the 37th EUROMICRO Conference on Software Engineering and Advanced Applications, SEAA 2011, pp. 129–133. IEEE Computer Society, Oulu (2011)