Handcraft Objects Tool

The Handcraft Objects Tool (HOT) aims to make it safer and easier to create serialized java Virtual Machine (JVM) objects. The tool makes it effortless to create these objects via a Graphical User Interface (GUI) written in kotlin with javaFX.

What is serialization?

Serialization is the process of taking an object and transferring it outside the program it was originally created in [1]. The opposite of serialization is called deserialization. This is the process in which a program takes serialized data and interprets it into (hopefully) a clone of the original serialized object. Serialized objects are a simple way for machines to communicate objects between each other. Saving an object to disk makes it possible to restore previous states of the program.

There are multiple different data formats to use for serialization. java has its own system called `java.io.Serializable`. It stores objects in a binary format, which is easy for programs to interact with, but hard for humans to read. Another, text based, serialization format is json. It is commonly used as the data format in web applications [4]. The tradeoff between the two forms of storage is how fast a machine can process the information and how much space it requires to store it.

Example: Conversation and Response objects

The following example is included to help the reader understand how we can serialize an object. The example object is modelling a conversation between the user and the computer. Each conversation has a prompt, i.e. a question or a greeting, and zero or more responses to this prompt. When responding the user is entering a new conversation. The pattern will look like a graph as is outlined in figure 1 below. As seen there are two possibilities for what happens after choosing a response. Either the conversation ends (response 3) or another conversation begins (Response 1, 2 and 4). Note that response 1 is recursively selecting the same conversation while response 2 selects a different conversation. In addition there might be some prerequisites to select a certain response, as shown with response 4. If a response has any prerequisites(s) they must be evaluated to `true` for the user to select the response.

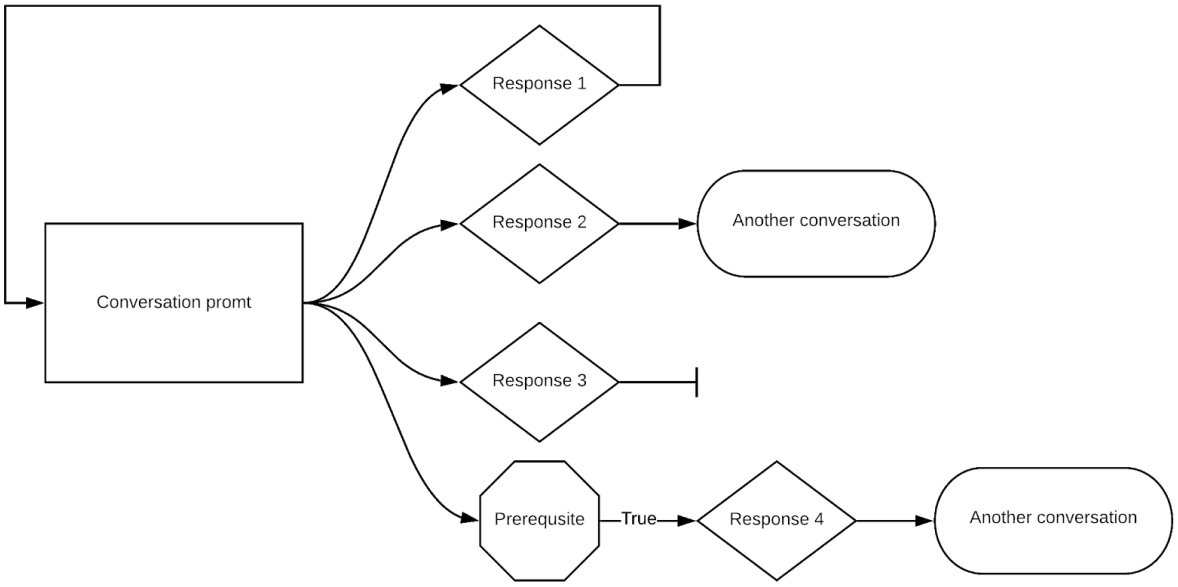


Fig. 1 *Conversation flow chart showing response possibilities.*

An example application following this example was created [19] to allow for a somewhat realistic testing. See figure 2 for a detailed overview of all serialization properties for conversation and response.

Conversation Properties		
Property Name	Type	Required
prompt	String	Yes
responses	Collection of Responses	No

Response Properties		
Property Name	Type	Required
text	String	Yes
prerequisite	Prerequisite	No
conversation	Conversation	No

Fig. 2 *Property name and type for each property in conversation and response objects.*

How to serialize an object

When serializing and deserializing objects you need to know three pieces of information: The type, identification, and the properties of the object. Not all three are equally important, the two first are sometimes optional, while the third is essential.

A serialization *property* is a piece of the serialized object. It usually is a field variable, but not exclusively. In the example provided above, *prompt* and *responses* are the serialization properties of *Conversation*. All serialization properties must also be recursively serialized. This can be visualized like a graph (see figure 1), and due to this recursive references are of course allowed. Not all properties might be required to create an object. For example it is necessary that a conversation have some prompt text to show the user. Without it “a

conversation” does not make any sense. However a conversation does not need responses to make sense. Therefore the prompt text is required, but not the responses.

The type of the object is readily available when serializing. You inherently know the type of the object as you are creating (in java this information is available with the `getClass()` method). However when deserializing, the exact type to deserialize the object to might not be known, for example when deserializing an interface or abstract class. In such cases the *type information* must be stored to know exactly what type the object is an instance of.

The second piece of information is how references are stored. A reference is simply a pointer to an object, if multiple properties are pointing to the same object we do not want to write out all its properties multiple times. Not only will it make the resulting document unnecessary large, but it might not ever work if the reference is recursive. To avoid this an object identification tag is bundled with each object where this might happen. When serializing and encountering the object for the first time, the object is written and tag included for deserialization. If a second encounter happens, only the tag is written. During deserialization this saved information will be used to restore the objects relationships.

Conception of Handcrafted Objects Tool

It is difficult to create serialized objects without inner knowledge of the application you are using. You need to know what properties are required, the type of each property, what data format the program is expecting the serialized object in, and what each property really means. The syntax of the data format used can also be an obstacle to the end user, they may not know how to write the correct syntax and bundle the required metadata information, such as type information and object id references.

The idea of a tool to help with these problems arose when I had to do this myself. I was creating a program in which you could create quests, conversations, and other typical RPG components. As I had created the entire system I knew how to create these quests and conversations, but I wanted to let others be able to create their own without intricate knowledge of the codebase. The example bundled with the tool is a simplified version of the conversation system from the original object. It has all the problems discussed above to make it a realistic goal to strive for. At the beginning of the project a list of wanted features was compiled [17]. I am pleased to say that almost all of them have been implemented, including several features from the extra list and even some features that were not listed at all.

Using Handcraft Objects Tool

Overview of the object creation process

When a user wants to create an object they must first select the type. All standard class definitions and primitives are already known, but to load custom class definitions the user has to supply one or more jar files. When the wanted jar files are loaded and the user has chosen the class, the tool (with help from Jackson) will find all properties to be serialized in

this class. The user may also select their own Object Mapper (see Jackson framework section), but this is optional. A new tab will be created and the user will be able to edit the properties of the given class. If the property has a valid default value it is already the value of the property. When the user has edited the object to the desired values and the object is valid it can be serialized and saved to disk. I will demonstrate how this works in practice in the upcoming presentation.

How Handcraft Objects Tool solves the outlined problems

As stated above the challenges with serializing are syntax, correctness, documentation abstraction, type information, and references.

Syntax, correctness, and documentation

As the tool takes care of the serialization it is not possible to create syntax errors. By using this tool it is also possible to seamlessly switch between different data formats, without having any knowledge of their syntax. The correctness of the created object is currently achieved only for a few predefined classes and all primitives. Numeric primitives, for example, only allow values within the valid limits, but there is currently no way for a user to specify a custom range. This can be solved using json schematics, see future development. Documentation can also be automated by making sure each property has a documentation annotation present, it will be shown to the user when editing the property.

Abstraction and type information

When the object to serialize is, or contains a property that is, abstract, an interface, or even a non-final class, the serialized data must reflect what the concrete class is. Doing this by hand is very error prone and tedious, as the author must have the precise knowledge on how the type information is serialized. When serializing with the tool the user does not have to think of this aspect as it is handled automatically during serialization. Assuming the object is capable of handling type information properly, it will be serialized correctly.

References

The greatest technical challenge was recursion. To truly allow users to create objects that can be used in a real world setting there has to be the option to reference other objects, including recursive references. This is a notoriously hard thing to do correctly and Jackson has throughout its lifetime created multiple different ways to tackle this challenge [5,6]. It is the more complex solution the tool is using, which supports complete traversal of cyclic graphs [7].

Technical specification

In this section the internal serialization process will be explained. As a quick overview the process can be seen as twofold. Firstly the given class is analysed to find what properties are expected to be serialized, and store them in a wrapper object called Class Builder. Then the user will edit these properties to the desired values. When finished the tool converts the wrapper objects to the wanted object and then it is serialized.

The Jackson framework

Jackson is a suite of data-processing tools for java (and the JVM platform), and of special interest to us is the flagship streaming json [9] parser/generator library together with matching data-binding library (POJOs to and from json).[3] It also has a large module community consisting of first and third party modules for additional features and compatibility. Using Jackson to handle serialization allows the tool to be usable with every application that is also using Jackson.

There are three different ways of handling serialized objects: iteration, data binding, and tree traversal [13]. Handcraft objects tool uses the second one to make sure the serialization of the objects created are as close to what an object would have looked like had it been serialized. This is done by representing the created object as their own classes (called class builders) and converting them to the actual type. By converting the object to the real deal before serialization we are sure it is correctly serialized, but this can be disabled with the *unsafe serialization* setting. This allows the user to see if there are problems with how they serialize objects.

Class builders

The interface *class builder* represents all possible types that can be created (see Appendix 1 for difference between classes and types). This interface holds information about the type it is representing, who the parent class builder is and the name of the property it is representing. Most importantly it has a field variable which represents the current value of this property, called the *serialization object*. Each implementation of class builder might have a different way of handling the serialization object.

The simplest implementation of class builder is the *simple class builder*. It represents what is considered simple types like primitives, string, and enum. These *simple* types can be represented by a single object allowing the serialization object to simply be what they represent. This type of class builder can be seen as leaves within the class builder graph.

The *parent class builder* is an abstract implementation of the class builder. It is the superclass of all class builders that have child properties of their own. Currently there are three types of parent class builders: map, collection, and complex. Collection class builders represent list, set, and array types. The collection type fulfils the same role as arrays do in java: it allows for an arbitrary number of elements to be specified with a single type. However unlike actual arrays a collection class builder can have a variable number of elements. The map class builder can be seen as a special case of the collection class builder.

The *complex class builder* represents all classes (except for a few special cases). It fulfills the same role as a class does in java: a complex type built by piecing multiple other types together. It gets its properties from the base class it is representing, and it is not possible to add or remove any properties from this type, as all properties are derived from the class it is based on.

Lastly there is a special class builder called *reference class builder* which handles references between objects. It works by pretending to be the object that is being referenced during serialization. Internally no actual object is stored, but actually queried when needed. This weak link allows the referenced object to be changed without desyncing from the object graph.

Serialization of cyclic references

The problem starts with the type of the object being created. Using the conversation example seen previously. A response's conversation variable can be the original conversation (Response 1 in figure 1). When this is represented with the internal class builder structure then the link back to the conversation is a reference class builder. It acts as a dummy node when editing, but when serializing the object it simply delegates itself to be the same object as what it is representing. By doing this Jackson sees the same object twice and will denote it as a reference to the first encountered object.

But it is not the same object, a conversation object will be represented using a complex class builder while the reference is, well, reference class builder. Custom serializers are used to change how Jackson serializes class builders. This custom made serializer does not serialize the object given but rather the object that the class builder is representing. For the most basic usage the implementation looks like the following

```
override fun serializeMaybeWithType(
      cb: ClassBuilder, gen: jsonGenerator,
      provider: SerializerProvider, typeSer: TypeSerializer?
) {
      val ser: jsonSerializer<Any> = provider.findValueSerializer(cb.type)
      ser.serialize(cb.serObject, gen, provider)
}
```

Fig. 3 *Serialization of simple Class Builder.*

It takes the object the class builder is representing (`serObject`) and serializes it with the normal serializer for that object (`cb.type`). This would have worked for the most simple object, but there are some problems when more advanced features are used. Firstly it does not support serializing any type information needed for generic types. Secondly the object that registers if an object has been seen before are created each new time, resulting in infinite recursion. After a week of full time used to solve this problem, the solution is to copy Jackson's implementation and modify it a bit to fit our solution representation.

As it turns out this is all the code needed to handle all recursive objects. When using primitive types and strings the value can be copied without fear for infinite recursion. As they cannot have any references of their own and are usually a built in type in the data. When using a collection, array or map each element is what is referenced, and not the map/collection itself. This is also how Jackson behaves when referring to lists.

Retrospective

Lessons learned

During the course of the project I have faced many challenges. The greatest of them I would say is allowing objects to be referring to other objects. Due to its recursive nature, making sure it was correctly serialized without throwing a stack overflow. Overcoming this issue took me about seven days to do correctly

This is also the first project I have been using kotlin and Jackson, so getting to know a new language on top of using a new framework was challenging at times. However as I got more used to kotlin, I have begun to favor it over java. Going back to using java after this project, allows me to appreciate how much more modern kotlin is as a language.

The third lesson learned is how complex serialization can be. There are multiple ways of just storing metadata (like type information and object identification). A user also have to make a choice of data format, what framework to use (Jackson, gson, `java.io.Serializable`), how to handle unknown properties, what structure to read data in (tree, stream, object binding), how to handle missing properties (use a default value or fail), making sure external dependencies can be serialized, and much more.

Future development

In the future a range of new features can be implemented to improve the quality of this tool. The two main new features that should be implemented is the ability to load existing serialized objects with references and using json schematics to load class definitions and additional constraints for properties.

Loading of serialized objects

The ability to continue to edit a serialized object was one of the original goals of the project. It was also the last feature to be implemented, resulting in unsatisfactory results. It does not serialize references at any level. When a serialized object with references is loaded, they are simply ignored. There is a simple detection mechanism to let the user know that their references did not survive the loading. It should not be hard to implement a scheme to allow the user to manually select these references when found, but currently only a warning is displayed.

To properly solve this deserialization problem a custom Jackson deserializer must be created. Currently a manual scheme is used when creating the class builders. When the serialized object is loaded we deserialize it and pass it along when creating class builders for each property. This is a slow process, and only really fit to be a temporary solution before rewriting it.

External classes

Handcraft objects tool works by loading classes from external jar files. To do this the user needs to be able to export their projects as a jar file with all the dependencies it has. If the user wants to use a custom Object Mapper it also needs to specify that [10]. To achieve this the tool is using a custom class loader that is able to contain all classes loaded. In the future the tool should be able to load classes in a less cumbersome way. One proposal is to use json schemas to define classes. A class definition would then look like

```
{ "type":"object",
  "properties": {
    "foo": { "type": "string" },
    "bar": { "type": "integer" },
    "baz": { "type": "boolean" }
  }
}
```

Fig. 4 *Example class definition with json Schema. See [8]*

This could also allow for the tool to create and export custom types with json schema.

Custom validation

One of the main strengths of the tool is that it is near impossible to create an object that is not valid. There are range checking primitives and enums, but there is no way to only allow say positive integers. What would be a good addition to the tool is a custom validation scheme either using json Schema validation [11], annotations [12], or custom class builders.

Customization of class builders

Users should be able to create their own class builders. For example there is no good way to select a time or date. Implementing this allows the tool to be more versatile and allow for wizard creation of complicated objects (like time and date!).

Support more types of serialization libraries

Currently the only way to correctly create objects is by using Jackson. However there are other large serialization libraries available for example gson [16].

Conclusion

Working on this project has been challenging but very rewarding. The project consists of 10000 lines of code spread over more than 400 commits. It fulfills almost all of the original goals I initially wanted it to satisfy. Serialization of arbitrary complex objects is not easy to handle, but as long as it can be serialized with Jackson it should be possible to edit it using this tool. Dealing with references was not something that was on the original list of features but I think it will come in handy. Handcraft objects tool fulfils what I wanted to create for my own needs. It is a program I am proud to have created.

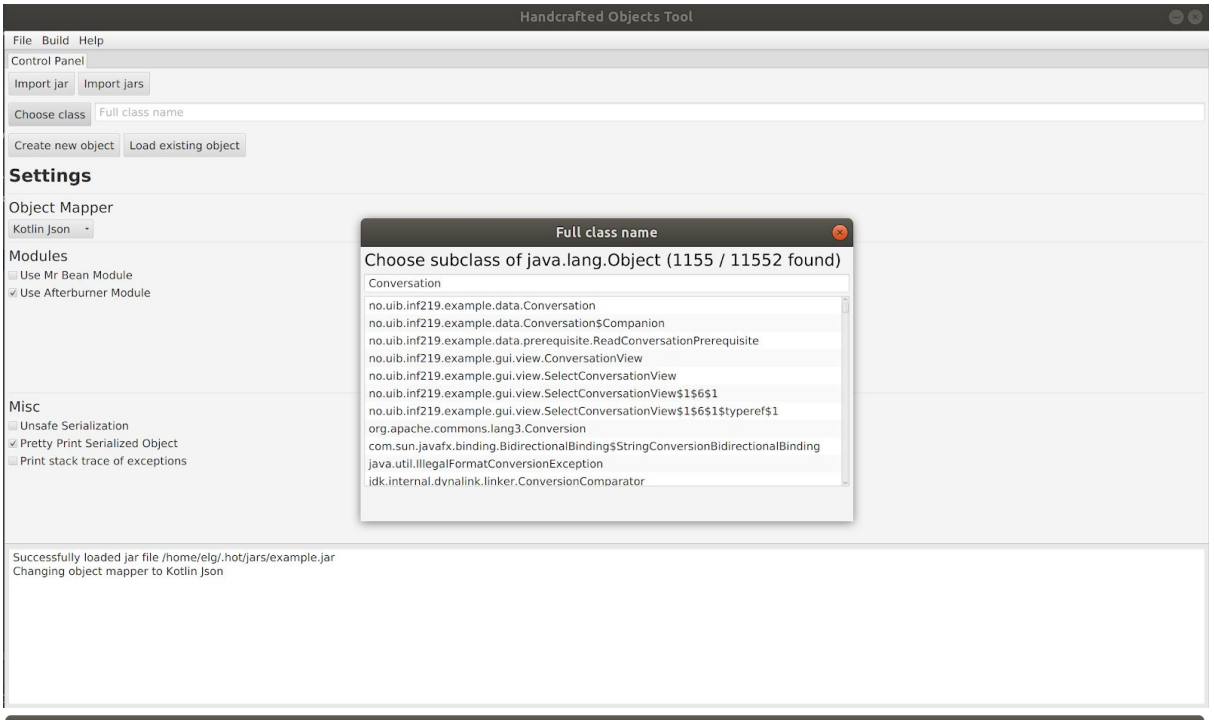
Appendix

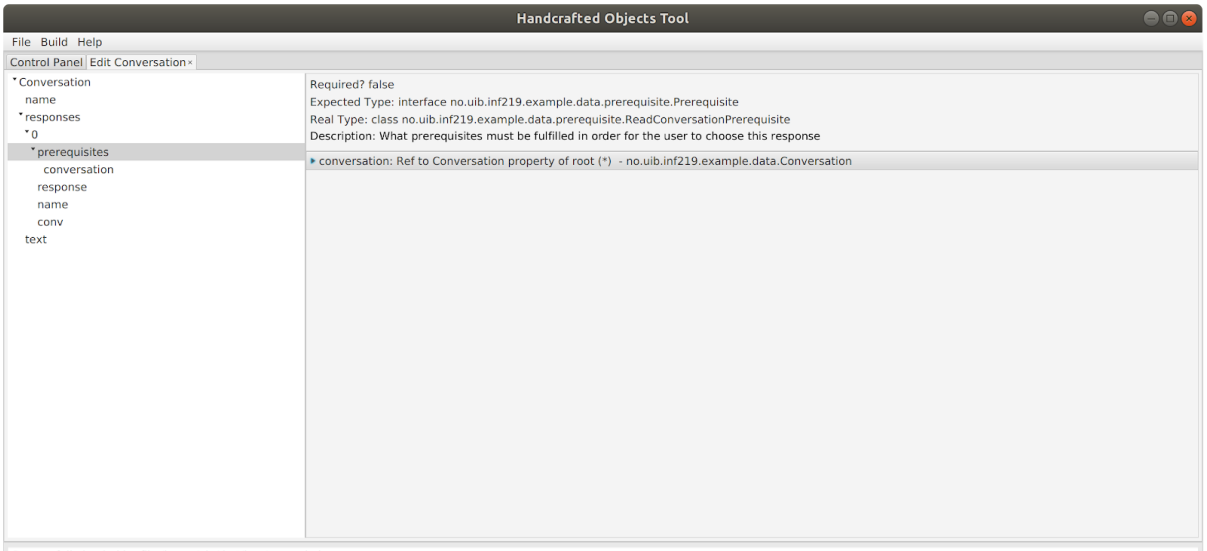
Appendix 1 - Difference between class and type

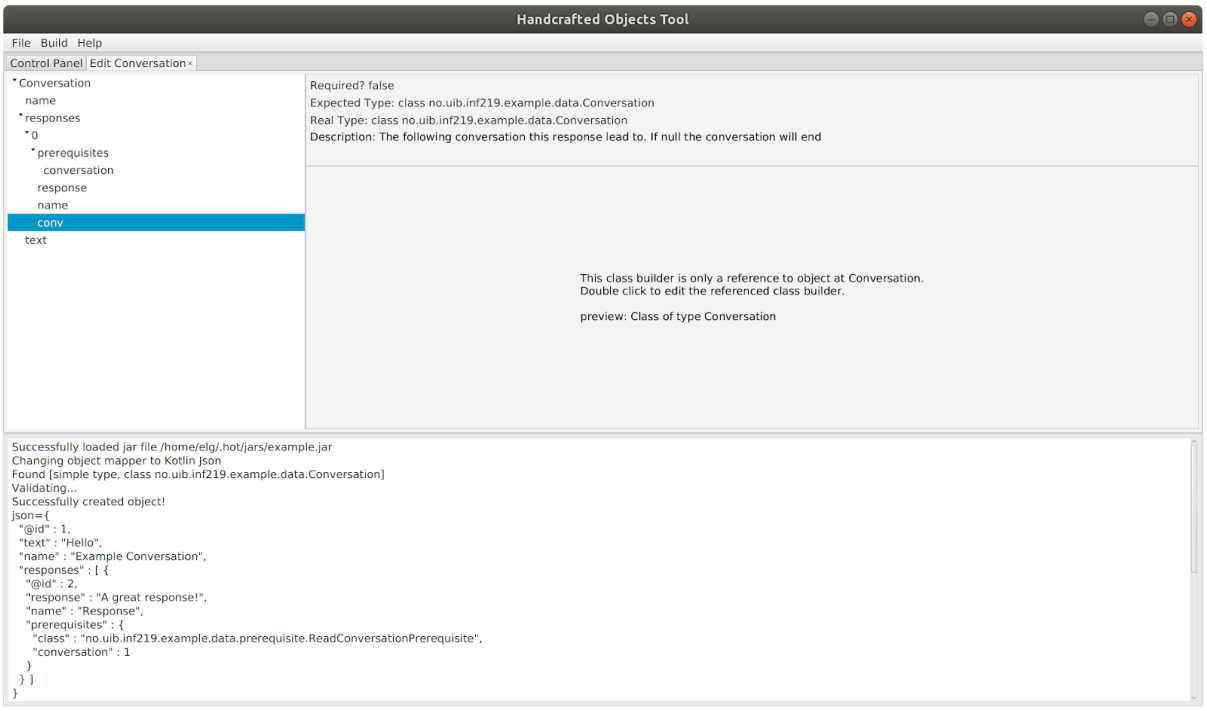
Explained in a single sentence: A class is a specific type. Every primitive, class, interface and array is a type [14,15]. This means that the interface of the wrapper types *ClassBuilder* should really have been named *TypeBuilder*, and *ComplexClassBuilder* named *ClassTypeBuilder*. However, then the other parent class builder would have been named *CollectionTypeBuilder* (or maybe *ArrayTypeBuilder*) and *MapTypeBuilder,* but this does not make much sense as neither map nor collection is a type, they are just classes. In the end all primitive types are converted to their object equivalent, so the only two types that are really created are class and array.

Appendix 2 - Preview pictures

The following pictures demonstrate typical usage of the tool. First a class is selected, then objects are edited, and lastly the object is serialized. These pictures can also be found in the source repository [18].







Bibliography

1. https://web.archive.org/web/20150405013606/http://isocpp.org/wiki/faq/serialization#serialize-overview
2. https://bitbucket.org/asomov/snakeyaml-engine/src/master/
3. https://yaml.org/spec/1.2/spec.html#id2759572
4. http://javascript-coder.com/tutorials/re-introduction-to-ajax.phtml
5. http://www.cowtowncoder.com/blog/archives/2012/03/entry_466.html
6. https://web.archive.org/web/20170831000434/wiki.fasterxml.com/JacksonFeatureObjectIdentity
7. https://www.baeldung.com/Jackson-bidirectional-relationships-and-infinite-recursion
8. http://www.jsonschema2pojo.org/
9. Even though the json format is used it is just and example Jackson support most common formats with existing modules
10. https://github.com/kh498/HandcraftObjectsTool#using-a-custom-object-mapper
11. https://datatracker.ietf.org/doc/draft-handrews-json-schema-validation/
12. https://www.baeldung.com/javax-validation
13. http://www.cowtowncoder.com/blog/archives/2009/01/entry_131.html
14. https://stackoverflow.com/questions/16600750/difference-between-class-and-type
15. https://docs.oracle.com/javase/specs/jls/se8/html/jls-4.html
16. https://github.com/google/gson
17. https://github.com/kh498/HandcraftObjectsTool#list-of-features
18. https://github.com/kh498/HandcraftObjectsTool
19. https://github.com/kh498/HandcraftObjectsTool/releases