simpleOOP
Integrating OOP principles into gforth

Klaus.Schleisiek @ send.de

Preface

Unfortunately, I can not attend this years Forth conference, which for the first time is held in the Netherlands together with our Dutch friends. Nevertheless, I have written this paper, because I think that the topic is a burning Forth issue.

I hope that my attempts to provide a very concise and forthish OO wordset will be of value for a workshop on Forth and OO programming principles, without which contemporary design patterns in the software engineering community can not be understood.

For me, an embedded systems specialist, the primary benefit of OO is for debugging. What used to be hacked tools to observe a system and change code while it is running, is now a systematic approach thanks to dynamic late binding. In addition, polymorphism is the proper tool to produce interface code for minutely different peripherals. And above all, object specific methods are a simple and systematic tool to realize operator overloading, getting rid of the plethora of hyphenated wordnames that each basically do the same but only apply under specific circumstances. The latter is one of the reasons why todays programmers do not touch Forth. Too many words to learn.

1 Introduction

At the last euro4th conference Stephen Pelc raised the question that object oriented programming principles are not yet part of the Forth standard. Instead, there is a multitude of OOP-packages, which load on top of Forth and "convert" the underlying Forth system into an OOP system, which is more or less oriented towards a Forth syntax. According to Stephen, these packages take between 1000 and 2000 lines of code.

This inspired me to port the OOP extension, which I had realised in the MicroCore cross-compiler, to gforth, because I was convinced that this could be done in considerably less than 1000 lines of code. As it turned out, my understanding of OOP was quite shallow at the time. Thanks to Bernd Paysan's and Andrew Hayley's efforts, my understanding of OOP was deepend and in the end, I can present a package that integrates mainstream OOP functionality (late binding, single inheritance, polymorphism and proxies) into gforth - all in 420 lines of code, because the simpleOOP package makes use of existing Forth mechanisms whenever possible.

At present the code is system dependent and loads on gforth 0.62, because of the way the interpreter and compiler have to be redefined, and the word name manipulations necessary. It can be easily ported to gforth 0.71. I believe that the concepts used can be expressed in Standard Forth, but I am not an expert in it and therefore, I would need to collaborate with one if needed.

This work is based on ideas put forward by Manfred Mahlow for the first time. His concepts revolved around the so called "prelude concept", which would simplify class context setting without the need to make object defining words immediate and state smart. But using the prelude concept requires a modification of Forth's headers, which requires system dependent modifications and recompilation of the Forth system. Therefore, I have taken an alternative approach to compiling methods.

The Forth/Class context search mechanism was already present in my cross-compiler. Thanks to Andrew Haley's insistence I found an efficient late-binding mechanism, which is embarrassingly simple and as efficient as you can possibly get. Its runtime overhead is one additional Forth branch compared to static :-definitions. And Bernd Paysan did not stop to argue that I couldn't possibly do without vTables for proper polymorphism. As it turns out, Bernd was both right and wrong: simpleOOP does not have classical vTables; instead, the wordlist associated with each class constitutes the vTable for that class. The functionality is identical, but the implementation is radically different from classical OOP implementations.

Instead of creating new words for OOP handling, I tried to overload existing words to behave appropriately in a Class context. Therefore, there is e.g. no special defining word for methods. : just behaves differently when we are compiling into a class wordlist. This helps to minimize the number of new words needed for simpleOOP and therefore, the resulting code looks very "forthish".

Finally I had to learn that my implementation is so radically different to everything that has been done before that hardly anybody is able to understand these 420 lines of code. This is why I am writing this paper.

2 Searching the dictionary

The first mechanism that has to be realised for an OOP system is a way to compile methods of a specific class. In essence, looking up word names is what the "outer interpreter" does in every Forth system. We do not have to invent anything new, but to adapt existing mechanisms.

But what is needed for an OOP system?

Let us start with the way a new Class is defined. **`Class`** is a defining word, which is based on **`Vocabulary`**, adding three more fields: an attribute field, a field that specifies the physical size of an object, and a field that points to an inherited superclass, and eventually to Classroot. **`Classroot`** is akin to the **`Root`** vocabulary, which we find in systems with a vocabulary stack. **`Classroot`** holds all words, which are always needed in a class context.

Because it is based on **`Vocabulary`**, a **`Class`** has a wordlist of its methods and the third additional field links to the wordlist of inherited methods and eventually to **`Classroot`**.

Therefore, all mechanisms in Forth, which create new wordlist entries, will work without modification. Only when the method of a class is searched, the search order is different from the search order of e.g. a vocabulary stack system. This "class find" is realised by **`search-methods`**.

Now we have two different search strategies: A Forth search order and a Class search order. Using the two variables **`LastClass`** and **`ClassContext`** we can switch between these two search strategies: The first parameter of every object's data field holds a pointer to "its" class (see the definition of **`Object`** and **`set-class`**). When an object is executed or compiled, it sets **`LastClass`** to point to its class definition, and it sets **`ClassContext`** on. The outer interpreter, which has to be redefined, will look at **`ClassContext`** and either execute **`search-methods`** or **`find-name`** depending on whether it is on or off. Prior to searching, **`ClassContext`** will be set off again. See **`oop-compiler`** and **`oop-interpreter`** and how **`[`** and **`]`** are patched to modify the outer interpreter for the entire system.

Because the class context has to be set, each object and an object's data fields (**`Attribute`**, **`Polymorphic`**, or **`Proxy`**) must be immediate, and therefore, the DOES>-part of these defining words have to be state smart. A more elegant implementation would use Manfred Mahlow's "prelude concept", which realises the necessary context setting without having to make the object or its subfields immediate.

3 Late binding mechanism

Late binding is a mechanism that associates a specific semantic to a certain method even after the method may have been compiled into some word. This implies that a certain method may be defined and redefined several times and that the last (re)definition is to be applied whenever the method is executed directly or indirectly.

In simpleOOP, the word header of a method in a class wordlist is only layed down once. Whenever it is redefined, the same "old" word header is reused and a branch to its latest definition is compiled. This means that the xt of a certain method always remains the same and can be compiled statically. To make this happen, **`:`** has to be redefined considerably to do different things depending on whether we compile into the Forth context, or into a class context. In the latter case, we may be defining a method for the first time, or we may redefine a method, or we may redefine a polymorphic method (see below). See: **`new-method`**, **`redefine-method`**, **`redefine-polymorphic?`**

As a result of this implementation, the run time efficiency of compiled code is very good. Each method is compiled using a static xt the same way the Forth compiler has been working all along. The price we pay for late binding is just one additional branch on the Forth code level.

To make things even more complicated, when we redefine a method, we may want to compile its current semantic into the new redefinition. To this end, the following two requirements must be met:

1. The branch from the word header to the current semantic of the method will be compiled by **;** rather than **:**. This way the semantic of the previous version of the method is available while redefining it. See: **;** This is similar to the way redefinitions of Forth words are handled in many Forth systems using variable **Last** to make the new word invisible in the wordlist until ; is executed.
2. The destination of the branch to the semantic definition of a method must be preceeded by a "do-dolon" code field. Otherwise, an obsolete semantic definition of a method could not be embedded in its redefinition by compiling its xt.

For the latter mechanism, see **redefine** in **oop-compiler.**

4 vTables

Proper OOP can not possibly be done without using vTables, right? As it turns out, an OO system can be implemented without using vTables, but the functionality, which traditionally has been implemented using those tables, needs to be there.

Why did I choose a different approach? vTables do have one disadvantage: They need to be allocated at a certain point while compiling and that limits the number of methods, which can be associated with a class of objects. Instead I wanted to keep everything in the spirit of Forth, namely extensibility. As a result, additional methods can be defined even after instantiating objects, which is useful for debugging e.g. a life real-time system or a system with a complex state that is non trivial to build up.

Functionality	**Classical implementation**	**simpleOOP**
locate method xt	method index into table	static method xt
late binding mechanism	actual method xt fetch from table	branch to actual method
inheritance	subclass inherits superclass's table as copy	subclass inherits all method definitions by copying the superclass's method words into the subclass's wordlist see: **copy-methods**

It is possible to look at the simpleOOP implementation in "vTable" terms: The index into the table relates to the name lookup in the class wordlist, the actual "late bound" xt of the method is located by a branch to the current version, and copying a superclass's vTable into a subclass relates to copying all names of the superclass's wordlist into the subclass wordlist.

5 Polymorphism

I may not be using the term properly, but this is what I understand under polymorphism: A polymorphic method may have a subclass or even an object specific implementation. This reminds of deferred words in Forth, and could be called deferred methods. This means that the actual xt of a polymorphic method must be determined during runtime and can not be determined during compile time. Again, this is similar to Forth's deferred words.

In a classical vTable implementation, polymorphic methods are realised by making an object specific copy of the table and modifying specific table entries aka methods. In the case of subclasses, each subclass inherits a copy of its superclass's vTable and therefore, a subclass may modify certain methods without affecting the same method in the superclass.

simpleOOP uses an explicit polymorphism mechanism, which may be called "polymorphism on demand" by explicitly using defining word **`Polymorphic`**. Each polymorphic method uses a data field in each object of that class and this is why polymorphic methods can no longer be defined after instantiating an object of that class. The actual code for a polymorphic method always begins with a branch instruction to the actual code and this branch instruction is always located at the end of a polymorphic definition. The **`Polymorphic`** definition itself will be initialized to point to code that will produce a "not initialized" error message.

Compared to the vTable approach, the "deferred word approach" must be explicitly initialised when an object is instantiated. This is done by **`set-polymorphics`** when an object is instantiated. All words in the class's wordlist as well as all inherited wordlists are scanned for polymorphic definitions and will be set in the associated object's field if the field was not yet initialized.

Later on, a polymorphic method may be set to a different xt using **`bind`**, which is similar to **`is`** for deferred words.

6 Proxy

`Proxy` is a defining word, which reserves a field in the object of the current class as a placeholder for an object of a specific class. A **`Proxy`** can be thought of as a deferred object embedded in another object. When the object that contains a proxy is instantiated, the proxy does not have to be assigned nor initialised yet. Instead, an xt may be assigned to a **`Proxy`** using **`bind`**. This xt does not necessarily have to be an object, it could as well be a piece of code that initializes the proxy with a concrete object. See the definition of **`init`** in **test.fs**.

7 Glossary

7.1 Forth words

`Class ( <name> -- )`

Defining word. Based on Forth vocabularies, it holds a wordlist for the class's methods and keeps tabs of attribute sealed/unsealed, the size of an object, and a pointer to the parent class, which may have been inherited. By default, the parent class points to the ClassRoot wordlist, which is therefore inherited by every class (see below).

When **`<name>`** is executed later on, it sets the class context. As with Forth vocabularies, **`definitions`** can be used to add to its wordlist.

`ClassRoot ( -- )`

A wordlist of fundamental OOP words, which are accessible in every class.

`Polymorphic ( -- )`

Used in the form

`Polymorphic <name>`

to define polymorphic methods. At first, <**`name>`** will be associated with the "un-initialized method" error handler. Later on, its specific behaviour in its own class or in a sub-class can be defined by just defining a colon definition of the same name. An object specific method can be assigned using **`bind`**.

`Oop ( -- )`

a Forth vocabulary that holds all primitives, which are needed to implement simpleOOP.

7.2 Root words

`classes ( -- )`

Lists all defined classes

`methods ( -- )`

Lists the methods of the most recently executed class and its inherited sub-classes.

7.3 ClassRoot words

`Object ( <name> -- )`

Used in the form **`<classname> Object <name>`**. Creates object **`<name>`** of class **`<classname>`**.
`<name>` is a state smart word. When interpreted, it returns its object data field address and sets the class context.
When compiled, it compiles its data field address as a Literal.

`Attribute ( <name> -- )`

Create attribute **`<name>`** in the current class as an embedded object of the most recent class as an immediate word. Used in the form **`<classname> Attribute <name>`** in object definitions before the class is sealed.

<name> is a state smart word.
When interpreted, it adds its data field offset to the object address on the stack.
When compiled, it compiles the offset as a Literal followed by + as operator.

Proxy (<name> --)
Create proxy **<name>** of the recent class as an immediate word. Used in the form **<classname> Proxy <name>** in object definitions before the class is sealed. It reserves a field in class's objects for an execution token, which will be initialized with the "un-initialized reference" error. When **<name>** is executed later on, it executes code, which can be assigned to an object using **bind**. It resembles **defer** for objects.
<name> is a state smart word.
When interpreted, it executes the assigned code.
When compiled, it compiles the offset of **<name>**'s field in the object as a Literal followed by a word that fetches and executes the assigned code.

bind (xt obj <proxyname> --), I
bind assignes execution token **xt** to a proxy or a polymorphic method, which has defined for object **obj**.
bind is a state smart word. When interpreted, it stores **xt** in the proxy / polymorphic method field of **obj**.
When compiled, it compiles the offset of **<proxyname>** in **obj** as a Literal followed by instruction **do-proxy** that assignes **xt** to the proxy or polymorphic method when executed later on.

units (u1 -- u2)
Used to compute the size u2 of allocating space needed for u1 object data fields of the recent class. Used e.g. in the form **<classname> units allot**.

allot (u --)
allocate u bytes space in objects of the current class.

size (-- bytes)
universal method that returns the size of the data field of the recent class's objects.

' (<name> -- xt)
returns the **xt** of **<name>** in the class context.

addr (obj -- addr), I
Used in the form **<objectname> addr** as a universal method to switch back into the Forth context leaving **obj**'s data field address on the stack. **<objectname>** may consist of the name of an object followed by a sequence of attribute names.

definitions (--)
used in the form **<classname> definitions** in order to make **<classname>** the compilation class.

.. (obj -- addr), I
Synonym for **addr**. Used while debugging in order to escape the class context.

`see` **`( <name> -- )`**
de-compiles <name>.

`order` **`( -- )`**
displays the actual class context.

`methods` **`( -- )`**
used in the form **`<classname> methods`** to list the methods of class **`<classname>`** and its inherited sub-classes.

`words` **`( -- )`**
used in the form **`<classname> words`** to list the methods of class **`<classname>`**.

8 The Code

At **`www.forth-ev.de/repos/simpleOOP/`** the code can be found. For downloading, go to the **`archiv/`** to get a zipped version.

File	Content
oop.fs	simpleOOP loading on top of gforth 0.62
test.fs	some test suites to test Proxy, Polymorphic, bind, late binding
demo.fs	another test suite to demonstrate polymorphism and late binding