Implementation of Object-Oriented Programming Design Patterns in *Mathematica*

Anton Antonov
antononcube@gmail.com
MathematicaForPrediction project at GitHub
MathematicaForPrediction blog at WordPress.com
October 2015
February 2016
March 2016
Version 0.6

This is a preliminary, not fully developed version of the document.

Introduction

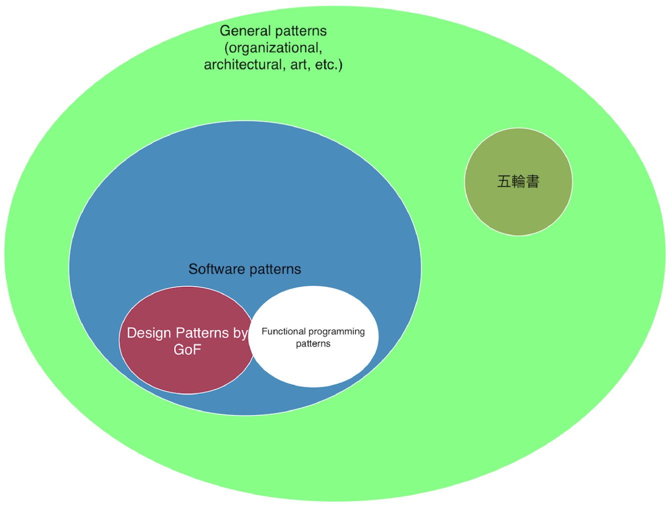
This document presents a particular style of programming in *Mathematica* that allows the application of the Object-Oriented Programming (OOP) paradigm (to a point). The approach does not require the use of preliminary implementations, packages, or extra code. Using the OOP paradigm is achieved by following a specific programming style and conventions with the standard programming constructs of *Mathematica*'s programming language (Wolfram Language).

We are going to illustrate the application of OOP with the so called Design Patterns that were introduced and became popular by the book [5].

The book [5] describes Design Patterns as software architectural solutions obtained from analysis of successful software products. We can say that the Object-oriented paradigm came to maturity with Design Patterns. Design Patterns help overcome limitations of programming languages, give higher level abstractions for program design, and provide design transformation guidance. Because of extensive documentation and examples, Design Patterns help knowledge transfer and communication between developers.

It is beyond the scope of this document to give an OOP introduction to Design Patterns. For detailed description of the patterns implemented and discussed we are going to refer to [5] and Wikipedia. (Wikipedia has entries for all of the design patterns considered in this document.)

The following Venn-like diagram shows the large context of patterns. This document is for the patterns in the dark red area ("Design Patterns by GoF").


General patterns (organizational, architectural, art, etc.)
五輪書
Software patterns
Design Patterns by GoF
Functional programming patterns

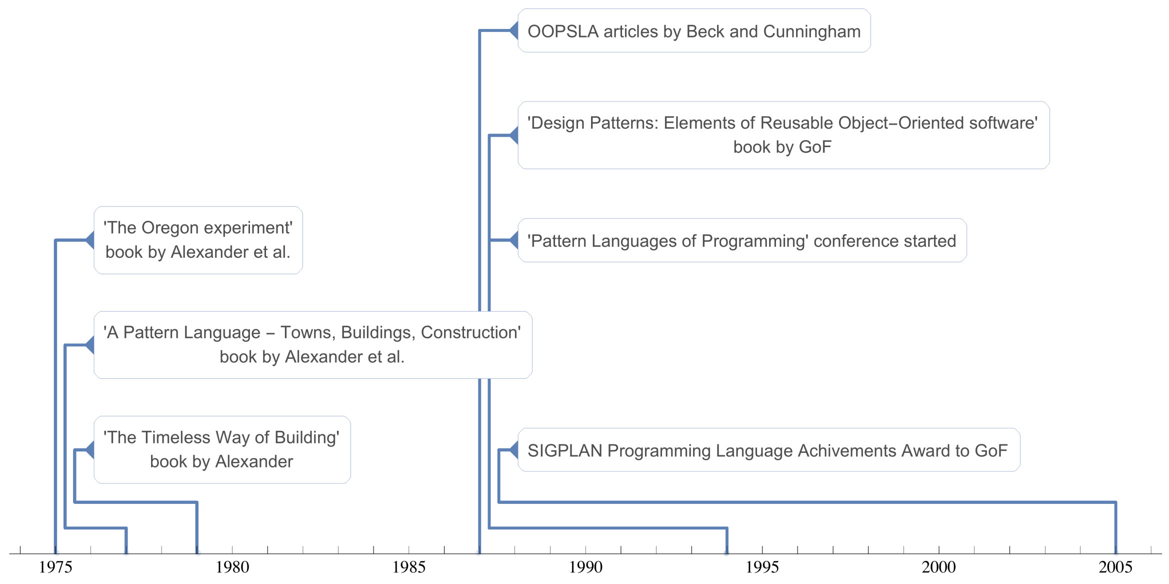
■ Design patterns history

In 1975 the architect Christopher Alexander et al., [2], introduced a pattern language for architectural solutions. Alexander's book "The Timeless Way of Building", [3], is the more fundamental one -- some sort of philosophical prequel of [1] and [2].

The computer scientists Kent Beck and Ward Cunningham started applying these ideas in software architecture in 1987 and presented results in the conference Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) -- see [4].

In 1995 the book by Gamma et al. "Design Patterns: Elements of Reusable Object-Oriented Software" provided a catalog of well researched and documented design patterns and that brought design patterns and pattern languages great popularity and acceptance. In 2005 ACM SIGPLAN awarded that year's Programming Language Achievements Award to the authors. The authors are of often references ad the "Gang of Four" (GoF).

Out[7]=



.

■ Design Patterns structure

The OOP Design Patterns in [5] are classified in three groups: creational, structural, and behavioral. The application of the structural and behavioral patterns in a design brings the need for the application creational patterns. In this document we are mostly interested in the structural and behavioral patterns, since in *Mathematica* creation of objects is much more immediate than in languages like C++ and Java.

The OOP Design Patterns are presented in a certain standard form. The form has four essential elements, [5], pattern name, problems for which the pattern is applicable, solution description, consequences of applying the pattern. The book [5] presents each pattern in the following form:

1. Pattern name and classification
2. Intent
3. Also known as
4. Motivation
5. Applicability
6. Structure
7. Participants
8. Collaborations
9. Consequences

10. Implementation
11. Sample code
12. Known uses
13. Related patterns

All enumerated parts are considered important.

(Because [5] is also structured as a manual it can be very difficult to read at first if the reader has little exposure to OOP.)

We are not going to follow the template given above. For each considered design pattern we are going to cite the intent from [5], provide code in *Mathematica,* and discuss applications inside *Mathematica* or with *Mathematica*.

■ Narration

It is accepted in technical and scientific writings to use plural voice or make neutral objective-like statements. Since Design Patterns were (and are) discovered in successful software that narrative style seems applicable. Understanding and applying Design Patterns, though, rely on extensive experience of software systems programming, maintaining, and utilization. Trying to apply them within a functional and rule based programming language like *Mathematica* inevitably requires a more personal perspective of programming and software goals.

Because of these observations I am going to use first person narration where objectivity cannot be fully guarantied, where there are multiple choices for the design solution, or where the design justification is based on past experience.

It seems appropriate at this point to discuss my past programming experiences.

■ Personal experiences with Design Patterns

This section is mostly to provide some background of why want to discuss the implementation of Design Patterns in *Mathematica* and why I think they are useful to apply in large development projects.

■ Large scale air-pollution modeling

In 2001 I obtained my Ph.D. in applied mathematics for writing an OOP framework for the numerical simulation of large scale air pollution (and writing related articles and thesis). Air pollution simulations over continents like Europe are grand challenge problems that encompass several scientific sub-cultures: computational fluid dynamics, computational chemistry, computational geometry, and parallel programming. I did not want to just a produce an OOP framework for addressing this problem -- I wanted to produce the best OOP framework for the set of adopted methods to solve these kind of problems.

One way to design such a framework is to use Design Patterns and did so using C++. Because I wanted to bring sound arguments during my Ph.D. defense that I derived on of the best possible designs, I had to use some formal method for judging the designs made with Design Patterns. I introduced the relational algebra of the Database theory into the OOP Design Patterns, and I was able to come up with some sort of proof why the framework written is designed well. More practically this was proven by developing, running, and obtaining results with different numerical methods for the air-pollution problems.

In my Ph.D. thesis I showed how to prove that Design Patterns provide robust code construction through the theory Relational Databases, [13].

(One of the key concepts and goals in OOP is reuse. In OOP when developing software we do not want changes in one functional part to bring domino effect changes in other parts. Design Patterns help with that. Relational Databases were developed with similar goals in mind -- we want the data changes to be confined to only one relevant place. We can view OOP code as a database, the signatures of the functions being the identifiers and the function bodies being the data. If we bring that code-database into a third normal form we are going to achieve the stability in respect to changes desired in OOP. We can interpret the individual Design Patterns as bringing the code into such third normal forms for the satisfaction of different types of anticipated code changes.)

While working at Wolfram Research Inc. I re-implemented that framework in order to demonstrate the capabilities of grid*Mathematica*. (And that demo became quite popular and presented in different conferences in 2006 and 2007.) The re-implementation in top-level *Mathematica* code was done using Design Patterns, [6].

■ Numerical integration

After joining Wolfram Research, Inc. I designed, developed, and documented the (large) numerical integration framework of NIntegrate. The implementation was done using both C and *Mathematica* and I used Design Patterns with both languages.

■ Other applications

In the last 9 years I have worked in the field of machine learning and data mining. I have used Design Patterns to implement digital media search engines, recommendation engines, and a broadcast scheduler. I use Design Patterns when I program larger projects in *Mathematica* or R and I always use Design Patterns when I program in Java and C++.

▪ Small example of Design Patterns in *Mathematica*

After I answered the question “Import and Plot Git Commit History” in *Mathematica* StackExchange, I realized that the problem and desired functionalities presented there are both complex enough and simple enough to make a good example of object-oriented implementation in *Mathematica* using Design Patterns. The package GitHubPlots.m, [7], provides are “standard” implementation, and the package GitHubDataObjects.m, [8], provides an implementation with Design Patterns.

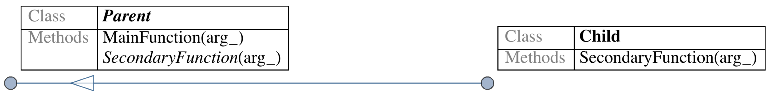
A separate document explaining that implementation is ....

UML diagrams

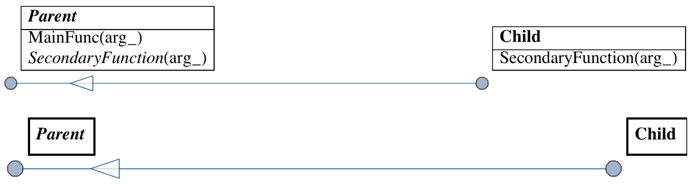
The Unified Modeling Language (UML) diagrams used in this document are more basic than the ones used in [5], but provide (I hope) a sufficient visual aid. They are generated over the *Mathematica* implementation code of the Design Patterns. See the package UMLDiagramGeneration.m, [10].

In UML abstract classes and operations (methods) are given in slanted font (italic). Inheritance is depicted with a line with unfilled triangle pointing to the parent class.

Here is an example:



Here are simplified versions of chart above:



The package UMLDiagramGeneration.m can be loaded with the command:

```
Import[
 "https://raw.githubusercontent.com/antononcube/MathematicaForPrediction/master/Misc/
   UMLDiagramGeneration.m"]
```

Preliminary concepts

Here is a list of notions I assume the readers of this document to be familiar with:

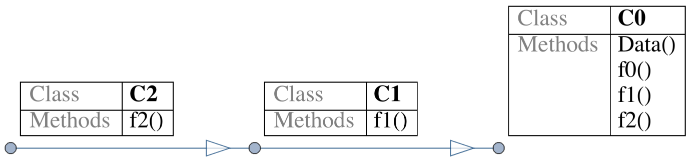
1. S-expressions,
2. Closures,
3. Dynamic and lexical scoping,
4. Inheritance & Delegation
5. DownValues, SubValues.

Class inheritance

Inheritance is the fundamental implementation to understand. This section shows delegation to a more abstract class. The section “Template Method” shows how to make the ancestor delegation even more useful.

Consider the following example. We have three classes C0, C1, and C2. The class C1 inherits C0, C2 inherits C1. We write this as C0←C1←C2.

■ UML diagram



■ Implementation

The basic idea is to use symbols` sub-values.

```
Clear[C0, C1, C2]

C0[d_]["Data"[]] := d;
C0[{a_, b_, c_, ___}]["f0"[]] := a;
C0[{a_, b_, c_, ___}]["f1"[]] := a + b;
C0[{a_, b_, c_, ___}]["f2"[]] := a + b + c;

C1[d_][s_] := C0[d][s]
C1[{a_, b_, c_}]["f1"[]] := a * b;

C2[d_][s_] := C1[d][s]
C2[{a_, b_, c_}]["f2"[]] := a^b;
```

■ Experiments

Here we define a C2 object:

```
obj = C2[{a, b, c}]

C2[{a, b, c}]
```

This invokes C1' s definition

```
obj["f1"[]]

a b
```

This invokes C2's definition:

```
obj["f2"[]]
```

a^b

Finally, this invokes C0’s definition:

```
obj["f0"[]]

a
```

Let us look into more detail of what is happening. We have made an object of the class C2, meaning we have an expression with head `C2` and some data inside the expression:

```
C2[{123, z2, z3}]

C2[{123, z2, z3}]
```

There are sub-value rules for `C2`:

```
SubValues[C2]

{HoldPattern[C2[{a_, b_, c_}][f2[]]] :→ a^b, HoldPattern[C2[d_][s_]] :→ C1[d][s]}
```

The first `C2` sub-value rule is specialized, it specifies the S-expression with head `C2` and one argument, a list of arbitrary three elements, would give a result combining two of them when applied to `"f2"[]`. The second sub-value rule is more general, an expression with head `C2` and one element that when applied over `s_` is directed to change its head to `C1` and use `C1`’s sub-values. Because *Mathematica* gives precedence to more specialized patterns we effectively have delegation of behavior to a more abstract entity.

Here is one more example:

```
C0[{1343, 1, 3}]["f1"[]]
```

```
1344
```

```
C2[{1343, 1, 3}]["f1"[]]
```

```
1343
```

It is instructive to look at the evaluation trace of `C2` and `C1` expressions called over “f1”. For example:

```
C2[{1343, 1, 3}]["f1"[]] // Trace
```

```
{C2[{1343, 1, 3}][f1[]], C1[{1343, 1, 3}][f1[]], 1343 × 1, 1343}
```

Static class inheritance

The inheritance implementation in the previous section was achieved through dynamic resolution of the symbols sub-value rules in order to invoke the corresponding implementation for a given signature. We can implement static inheritance by using direct manipulation of sub-values.

The idea of the static inheritance is simple: (i) we take the sub-values of the parent and assign them to the child, and (ii) we change one or several of child sub-values to correspond to the desired changes of behavior.

Note that the classes have the same definitions of the “delegated” function members. Using static inheritance we have saved copying and assigning code.

■ UML diagram

Class	**C0**
Methods	Data[] f0[] f1[] f2[]

Class	**C1**
Methods	Data[] f0[] f2[] f1[]

Class	**C2**
Methods	Data[] f0[] f1[] f2[]

■ Implementation

```
Clear[C0, C1, C2]
```

```
C0[d_]["Data"[]] := d;
C0[{a_, b_, c_, ___}]["f0"[]] := a;
C0[{a_, b_, c_, ___}]["f1"[]] := a + b;
C0[{a_, b_, c_, ___}]["f2"[]] := a + b + c;
```

Static inheritance C0 •→ C1:

```
SubValues[C1] = ReplaceRepeated[SubValues[C0], C0 -> C1];
```

```
pos = Position[First /@ SubValues[C1], "f1"[___], ∞][[1, 1]];
SubValues[C1] = Drop[SubValues[C1], {pos}];
C1[{a_, b_, c_, ___}]["f1"[]] := a * b;
```

Static inheritance C1•→ C2:

```
SubValues[C2] = ReplaceRepeated[SubValues[C1], C1 -> C2];
```

```
pos = Position[First /@ SubValues[C2], "f2"[___], ∞][[1, 1]];
SubValues[C2] = Drop[SubValues[C2], {pos}];
C2[{a_, b_, c_, ___}]["f2"[]] := a^b;
```

`pos` is for the positions of the overridden methods.

■ Experiments

Here we define a C2 object:

```
obj = C2[{a, b, c}];
```

Functions borrowed from C0:

```
obj["Data"[]]
obj["f0"[]]
```

```
{a, b, c}
```

```
a
```

Functions borrowed from C1:

```
obj["f1"[]]
```

```
a b
```

And finally genuine the C2 function:

```
obj["f2"[]]
```

a^b

It is instructional to see the definitions of the symbols defined above.

```
? C0
```

```
Global`C0
```

```
C0[d_][Data[]] := d

C0[{a_, b_, c_, ___}][f0[]] := a

C0[{a_, b_, c_, ___}][f1[]] := a + b

C0[{a_, b_, c_, ___}][f2[]] := a + b + c
```

Template Method

Function invocation by delegation through inheritance is a good first step but by itself is not enough to employ OOP in a powerful way. The next step is to provide an invariant algorithm in the abstract class and delegate the concrete operations of that invariant algorithm to the descendants. This is nicely demonstrated with the design pattern Template Method which is a fundamental micro-architectural solution in OOP.

Here is the “intent” of Template Method from [5]:

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm’s structure.

■ Motivational example

Mathematica has the function `Inner` that perfectly demonstrates the usefulness of Template-Method-like functionality code breakdown.

Here is an algorithm “skeleton” made with `Inner`:

```
in1 = Inner[op1, {a, b, c}, {x, y, z}, op2]
```

```
op2[op1[a, x], op1[b, y], op1[c, z]]
```

We can easily derive different algorithms by replacing the symbols `op1` and `op2` with concrete functions:

```
Inner[Times, {a, b, c}, {x, y, z}, Norm@*List]
```

$\sqrt{\text{Abs}[a\,x]^2 + \text{Abs}[b\,y]^2 + \text{Abs}[c\,z]^2}$

Alternatively, we can consider varying the data arguments of `Inner`:

```
in2 = Inner[op1, Mat @@@ Map[Array[#, {2, 1}] &, {a, b}],
  Mat @@@ Map[Array[#, {1, 2}] &, {x, y}], op2]
```

```
op2[op1[Mat[{a[1, 1]}, {a[2, 1]}], Mat[{x[1, 1], x[1, 2]}]],
 op1[Mat[{b[1, 1]}, {b[2, 1]}], Mat[{y[1, 1], y[1, 2]}]]]
```

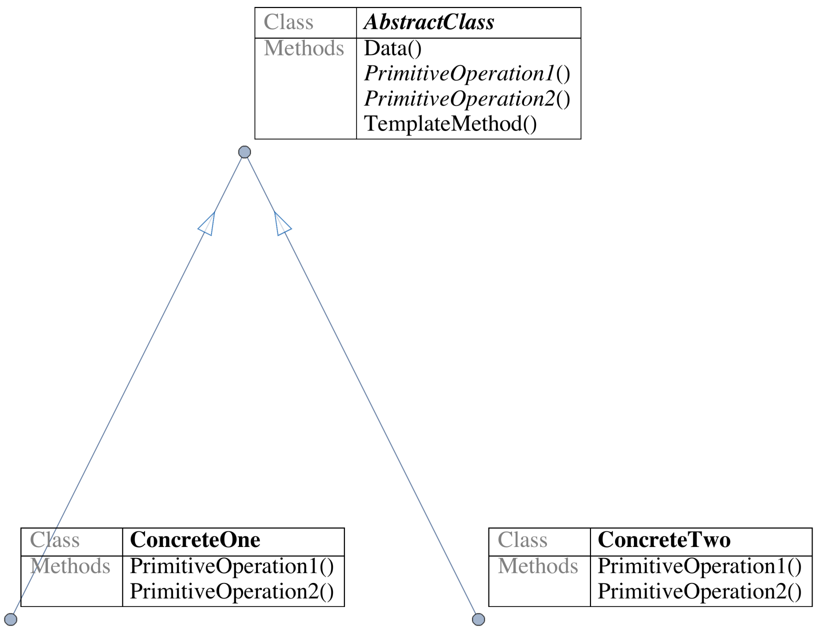
These concrete implementations of op1 and op2 provide different functionalities depending on the data types:

```
Block[{op1, op2},
 op1[x_?AtomQ, y_?AtomQ] := x ** y;
 op1[x_Mat, y_Mat] := (List @@ x).(List @@ y);
 op2[args___] := Plus[args];
 op2[args : (_List ...)] := MatrixForm[Plus[args]];
 Grid[{{"in1:", in1}, {"in2:", in2}}, Dividers → All]
]
```

in1:	a ** x + b ** y + c ** z
in2:	$\begin{pmatrix} a[1,1]\,x[1,1]+b[1,1]\,y[1,1] & a[1,1]\,x[1,2]+b[1,1]\,y[1,2] \\ a[2,1]\,x[1,1]+b[2,1]\,y[1,1] & a[2,1]\,x[1,2]+b[2,1]\,y[1,2] \end{pmatrix}$

Note that for _Mat objects the code above returns the result in MatrixForm.

UML diagram



Implementation

For the implementation of Template Method we basically combine sub-values with Block's dynamic scoping.

```
Clear[AbstractClass, ConcreteOne, ConcreteTwo];
```

```
CLASSHEAD = AbstractClass;
AbstractClass[d_]["Data"[]] := d;
AbstractClass[d_]["PrimitiveOperation1"[]] := d[[1]];
AbstractClass[d_]["PrimitiveOperation2"[]] := d[[2]];
AbstractClass[d_]["TemplateMethod"[]] :=
 CLASSHEAD[d]["PrimitiveOperation1"[]] + CLASSHEAD[d]["PrimitiveOperation2"[]]
```

```
ConcreteOne[d_][s_] := Block[{CLASSHEAD = ConcreteOne}, AbstractClass[d][s]]
ConcreteOne[d_]["PrimitiveOperation1"[]] := d[[1]];
ConcreteOne[d_]["PrimitiveOperation2"[]] := d[[1]] * d[[2]];
```

```
ConcreteTwo[d_][s_] := Block[{CLASSHEAD = ConcreteTwo}, AbstractClass[d][s]]
ConcreteTwo[d_]["PrimitiveOperation1"[]] := d[[1]];
ConcreteTwo[d_]["PrimitiveOperation2"[]] := d[[3]]^d[[2]];
```

■ Experiments

```
t = AbstractClass[{a, b, c}]
t["Data"[]]
t["TemplateMethod"[]]
```

```
AbstractClass[{a, b, c}]

{a, b, c}

a + b
```

```
c1 = ConcreteOne[{a, b, c}]
c1["Data"[]]
c1["TemplateMethod"[]]
```

```
ConcreteOne[{a, b, c}]

{a, b, c}

a + a b
```

```
c2 = ConcreteTwo[{a, b, c}]
c2["Data"[]]
c2["TemplateMethod"[]]
```

```
ConcreteTwo[{a, b, c}]

{a, b, c}
```

$a + c^b$

It is instructive to see the trace of the invocations.

This call uses a method defined in the abstract class:

```
c1["Data"[]] // Trace // ColumnForm
```

```
{c1, ConcreteOne[{a, b, c}]}
ConcreteOne[{a, b, c}][Data[]]
Block[{CLASSHEAD = ConcreteOne}, AbstractClass[{a, b, c}][Data[]]]
{CLASSHEAD = ConcreteOne, ConcreteOne}
{AbstractClass[{a, b, c}][Data[]], {a, b, c}}
{a, b, c}
```

This trace shows the call for the abstract algorithm (template method) for a concrete class object that invokes the primitive operation defined in that concrete class.

```
c1["TemplateMethod"[]] // Trace
```

```
{{c1, ConcreteOne[{a, b, c}]}, ConcreteOne[{a, b, c}][TemplateMethod[]],
 Block[{CLASSHEAD = ConcreteOne}, AbstractClass[{a, b, c}][TemplateMethod[]]],
 {CLASSHEAD = ConcreteOne, ConcreteOne},
 {AbstractClass[{a, b, c}][TemplateMethod[]], CLASSHEAD[{a, b, c}][PrimitiveOperation1[]] +
   CLASSHEAD[{a, b, c}][PrimitiveOperation2[]],
  {{{CLASSHEAD, ConcreteOne}, ConcreteOne[{a, b, c}]}},
   ConcreteOne[{a, b, c}][PrimitiveOperation1[]], {a, b, c}⟦1⟧, a},
  {{{CLASSHEAD, ConcreteOne}, ConcreteOne[{a, b, c}]}},
   ConcreteOne[{a, b, c}][PrimitiveOperation2[]], {a, b, c}⟦1⟧ {a, b, c}⟦2⟧,
   {{a, b, c}⟦1⟧, a}, {{a, b, c}⟦2⟧, b}, a b}, a + a b}, a + a b}
```

■ Other concrete examples

■ Template Method version of the motivational example

Using the motivational example let us implement a class hierarchy adhering to the Template Method that provides an invariant algorithm and different specializations of its concrete operations.

```
CLASSHEAD = AbstractInner;
AbstractInner[d___]["TemplateMethod"[x_, y_]] :=
  Inner[CLASSHEAD[d]["op1"[#1, #2]] &, x, y, CLASSHEAD[d]["op2"[##]] &];
AtomInner[d___][s_] := Block[{CLASSHEAD = AtomInner}, AbstractInner[d][s]];
AtomInner[d___]["op1"[x_, y_]] := NonCommutativeMultiply[x, y];
AtomInner[d___]["op2"[args___]] := Plus[args];
MatInner[d___][s_] := Block[{CLASSHEAD = MatInner}, AbstractInner[d][s]];
MatInner[d___]["op1"[x_, y_]] := Dot[List @@ x, List @@ y];
MatInner[d___]["op2"[args___]] := MatrixForm[Plus[args]];
```

```
obj1 = AtomInner["Atoms"];
obj1["TemplateMethod"[{a, b}, {x, y}]]
```

```
a ** x + b ** y
```

```
obj2 = MatInner["Mat objects"];
obj2["TemplateMethod"[
  {Mat[{a1, b1}, {a2, b2}], Mat[{A1, B1}, {A2, B2}]},
  {Mat[{x}, {y}], Mat[{X}, {Y}]}]
]
```

$$\begin{pmatrix} a1\ x + A1\ X + b1\ y + B1\ Y \\ a2\ x + A2\ X + b2\ y + B2\ Y \end{pmatrix}$$

Note that there is the question of the right selection of a Template Method descendant for a given pair of argument objects. There are creational Design Patterns that address these type of problems (Abstract Factory, Factory Method); see [5]. In *Mathematica* we can easily address that problem with pattern matching.

■ GitHubData

The functionality of the main GitHub data class in [5] `GitHubData` is modified with the descendant object `GitHubDataMessageHyperlinks`
 so the ticks of the plots have hyperlinks to the github.com pages corresponding to the commits. The template method algorithm is the method “Plot”, the primitive operations are “ParseData”, “TickLabels”, and “DatePoints”. The class `GitHubDataMessageHyperlinks` provides its own implementation of “TickLabels” (making the ticks click-able).

Strategy

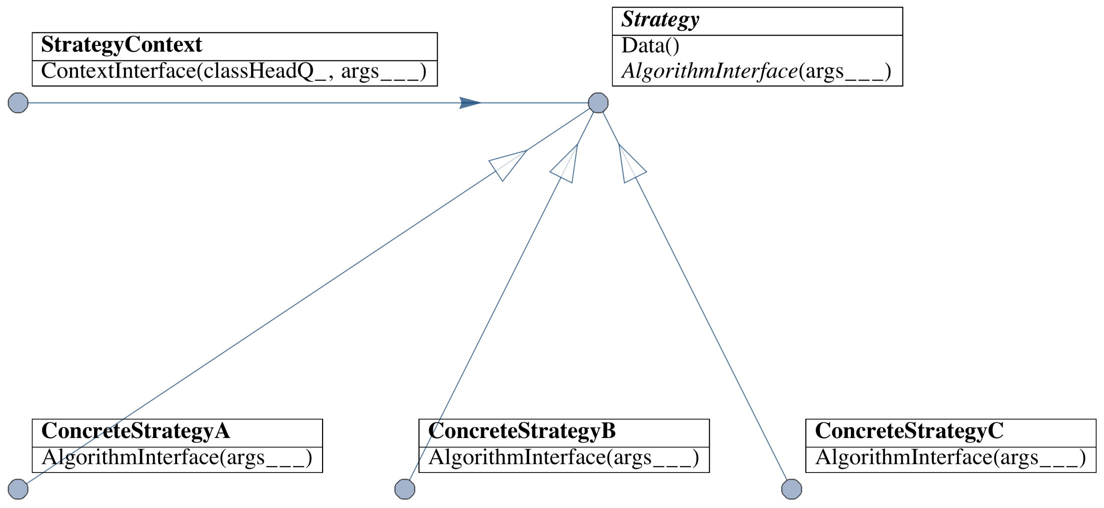
Here is intent of the design pattern Strategy from [5]:

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Generally Strategy’s structure is not needed in *Mathematica* because of the functional programming paradigm support and pattern matching of function signatures. Conceptually Strategy is applied all the time, e.g. second order functions or writing functions with arguments being other functions.

Nevertheless, we are going to give an implementation of Strategy that might be useful when prototyping in *Mathematica* and considering moving to production languages like C++ or Java. The implementation is essentially similar to the one given in the section “Inheritance”. Here we are going to endow it with additional code to facilitate the use of the class hierarchy within several Design Patterns.

▪ UML diagram


StrategyContext
ContextInterface(classHeadQ_, args___)
Strategy
Data()
AlgorithmInterface(args___)
ConcreteStrategyA
AlgorithmInterface(args___)
ConcreteStrategyB
AlgorithmInterface(args___)
ConcreteStrategyC
AlgorithmInterface(args___)

▪ Implementation

```
Clear[Strategy, StrategyContext, "ConcreteStrategy*"]
```

```
CLASSHEAD = Strategy;
Strategy[d__]["Data"[]] := d
Strategy[d__]["AlgorithmInterface"[args___]] := None;
ConcreteStrategyA[d__][s_] := Block[{CLASSHEAD = ConcreteStrategyA}, Strategy[d][s]];
ConcreteStrategyA[d__]["AlgorithmInterface"[args___]] :=
  (StringJoin @@ Riffle[Map[ToString, {args}], ","]);
ConcreteStrategyB[d__][s_] := Block[{CLASSHEAD = ConcreteStrategyB}, Strategy[d][s]];
ConcreteStrategyB[d__]["AlgorithmInterface"[args___]] := Length /@ {args};
ConcreteStrategyC[d__][s_] := Block[{CLASSHEAD = ConcreteStrategyC}, Strategy[d][s]];
ConcreteStrategyC[d__]["AlgorithmInterface"[args___]] :=
  Thread[{args} → Range[Length[{args}]]];
```

```
StrategyContext[sobj_]["ContextInterface"[classHeadQ_, args___]] := Print[
   If[classHeadQ, ToString[Head[sobj]] <> " : ", ""], sobj["AlgorithmInterface"[args]]];
```

▪ Experiments

Here we create an object of StrategyContext using a concrete strategy object:

```
obj1 = StrategyContext[ConcreteStrategyA[Unique[]]]
```

```
StrategyContext[ConcreteStrategyA[$32]]
```

```
obj1["ContextInterface"[False, 1, b, Range[3]]]
```

```
1,b,{1, 2, 3}
```

Here we simply replace one concrete strategy with another concrete strategy:

```
obj2 = ReplacePart[obj1, 1 → ConcreteStrategyC[Unique[]]];
obj2["ContextInterface"[True, 1, b, Range[3]]]
```

```
ConcreteStrategyC : {1 → 1, b → 2, {1, 2, 3} → 3}
```

We can do something more elaborated and create a `StrategyContext` object initialized with the abstract class `Strategy` and change its behavior by changing the value of the symbol `Strategy`. This way we are going to emulate the so called substitution principle in OOP.

```
obj3 = StrategyContext[Strategy[Unique[]]];
Block[{Strategy = #}, obj3["ContextInterface"[True, a, b, c]]] & /@
 ToExpression /@ Names["ConcreteStrategy*"]
```

```
ConcreteStrategyA : a,b,c

ConcreteStrategyB : {0, 0, 0}

ConcreteStrategyC : {a → 1, b → 2, c → 3}

{Null, Null, Null}
```

Uses in *Mathematica*

As it was mentioned above all second order functions (`Map`, `Fold`, `Nest`, `FixedPoint`, etc.) provide good examples of the functionality achieved by Strategy. Further, very often the algorithms specified by the `Method` option in different functions can be seen -- at least conceptually -- as playing within the Strategy design pattern.

Other concrete examples

The plotting functionality of the main GitHub data class in [5] `GitHubData` given by the method "Plot" is changed through the Strategy design pattern. The data member "PlotFunction" can be set to one of the functions GHDDateListPlot, GHDBarPlot, or GHDGraphics3D. In this way different plots are derived for the representation of the ingested GitHub data.

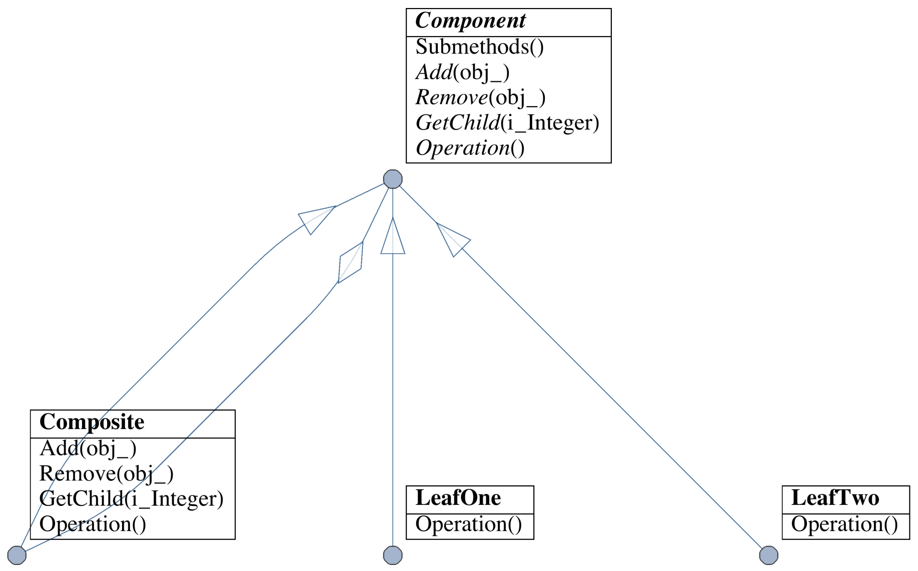
Composite

The design pattern Composite allows the treatment of multiple objects as one. More formally Composite's intent is, [5]:

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

A typical example is moving one or several desktop icons by mouse selection and dragging.

UML diagram



Implementation

```
Clear[Component, LeafOne, LeafTwo, Composite];

CLASSHEAD = Component;
Component[d_, sm_]["Submethods"[]] := sm;
Component[{a_, b_, c_}, sm_]["Add"[obj_]] := CLASSHEAD[{a, b, c}, sm]["Add"[obj]];
Component[{a_, b_, c_}, sm_]["Remove"[obj_]] := CLASSHEAD[{a, b, c}, sm]["Remove"[obj]];
Component[{a_, b_, c_}, sm_]["GetChild"[i_Integer]] :=
 CLASSHEAD[{a, b, c}, sm]["GetChild"[i]];
Component[d_, sm_]["Operation"[]] := Print["error"]
```

```
LeafOne[d_, sm_][s_] := Block[{CLASSHEAD = LeafOne}, Component[d, sm][s]]
LeafOne[{a_, b_, c_}, sm_]["Operation"[]] := a;

LeafTwo[d_, sm_][s_] := Block[{CLASSHEAD = LeafTwo}, Component[d, sm][s]]
LeafTwo[{a_, b_, c_}, sm_]["Operation"[]] := a^b^c;

Composite[d_, sm_][s_] := Block[{CLASSHEAD = Composite}, Component[d, sm][s]]
Composite[{a_, b_, c_}, sm_]["Add"[obj_]] := Composite[{a, b, c}, Append[sm, obj]];
Composite[{a_, b_, c_}, sm_]["Remove"[obj_]] :=
 Composite[{a, b, c}, Select[sm, #1 =!= obj &]];
Composite[{a_, b_, c_}, sm_]["GetChild"[i_Integer]] := sm[[i]];
Composite[{a_, b_, c_}, sm_]["Operation"[]] := Map[#1["Operation"[]] &, sm];
```

■ Experiments

First we create concrete non-composite Component objects (leafs).

```
leaf1 = LeafOne[{a, b, c}, {{3}, {4}}]
leaf1["Submethods"[]]
leaf1["Operation"[]]

LeafOne[{a, b, c}, {{3}, {4}}]

{{3}, {4}}

a

leaf2 = LeafTwo[{a, b, c}, {{5}, {6}}]
leaf2["Submethods"[]]
leaf2["Operation"[]]

LeafTwo[{a, b, c}, {{5}, {6}}]

{{5}, {6}}
```

a^{b^c}

Next we create a composite object and add the leaf objects to it.

```
comp = Composite[{a, b, c}, {}]["Add"[leaf1]]
comp = comp["Add"[leaf2]]
comp["Operation"[]]

Composite[{a, b, c}, {LeafOne[{a, b, c}, {{3}, {4}}]}]

Composite[{a, b, c}, {LeafOne[{a, b, c}, {{3}, {4}}], LeafTwo[{a, b, c}, {{5}, {6}}]}]
```

$\{a, a^{b^c}\}$

Here is code creating and operating a second composite object that uses the first composite object:

```
comp1 = Composite[{a, b, c}, {}]["Add"[leaf1]]
comp1 = comp1["Add"[comp]]
comp1["Operation"[]]

Composite[{a, b, c}, {LeafOne[{a, b, c}, {{3}, {4}}]}]

Composite[{a, b, c}, {LeafOne[{a, b, c}, {{3}, {4}}],
  Composite[{a, b, c}, {LeafOne[{a, b, c}, {{3}, {4}}], LeafTwo[{a, b, c}, {{5}, {6}}]}]}]
```

$\{a, \{a, a^{b^c}\}\}$

■ Other concrete examples

The `GitHubData` object utilizes Composite to visualize the commit histories of a collection of GitHub repositories.

Decorator

The design pattern Decorator can be seen as special case of Composite. More formally, Decorator’s intent is, [5]:

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub-classing for extending functionality.

A typical example is adding scroll-bars to screen outputs.

■ Motivational example

Here is a core algorithm that takes randomly a specified number of words from a text:

```
CoreAlg[n_Integer] := CoreAlg["Hamlet", n];
CoreAlg[title_String, n_Integer] :=
  RandomSample[StringSplit@ExampleData[{"Text", title}], n];
```

`CoreAlg` might give a very large output, so we want to confine its output in a small window and use scroll-bars to examine the content:

```
Pane[CoreAlg[2000],
 ImageSize → {400, 150}, Scrollbars → {True, True}]
```

```
{love, to, That, tyrannically, death,, wrinkled;,
 cannoneer, the, some, this, As, Priam,, now,, I,
 well, a, you, and, A, his, 'Twill, father, a, them,
 My, head, I, a, say, he, Queen., To, unknowing, we,
 Ham., or, Nothing., my, my, a, that,, late,, As,
 coming:, of, us,, Or, play,--and, gave, those,
 in, I, cuffs, That, the, is, the, beauty, they,
 I, the, is, pay, round, each, a, in, had, lord!,
 poet, not, Horatio, where, silence, answerest,
```

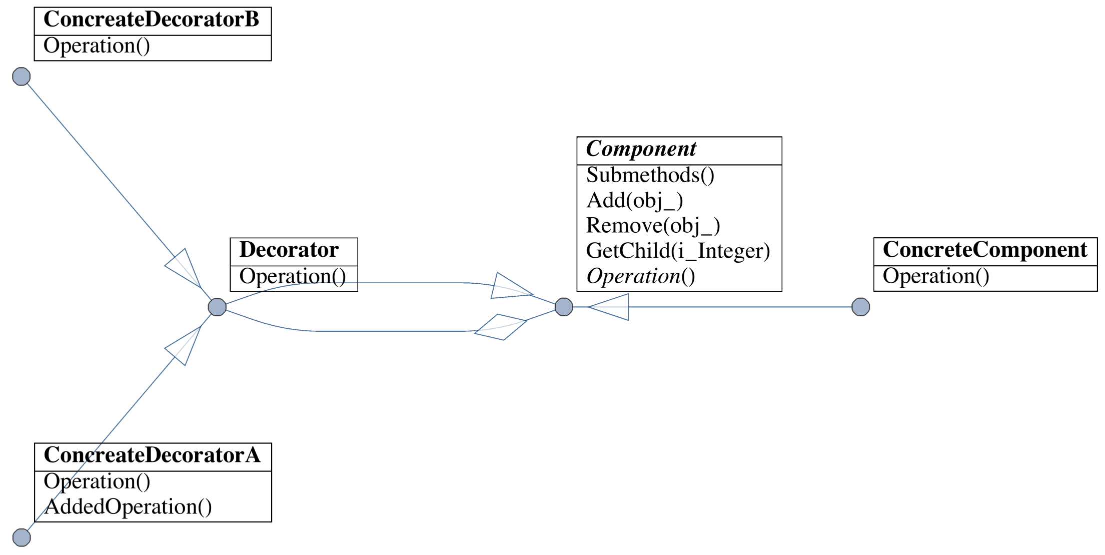
Further, we might want to show in red certain special words:

```
Pane[CoreAlg[2000],
 ImageSize -> {400, 150}, Scrollbars -> {True, True}] /.
 Map[# -> Style[#, Red] &, {"Ham.", "Hamlet", "King", "Queen"}]
```

```
{Clown., name, freely;, So, is, now, not, Ham., poison,
 understand, king, Revisit'st, my, celestial, am,
 this, upon, Claudius,, more, not, his, you, for,
 at, Norman, definement, applaud, my, thy, calls, in,
 bleed, name, to, arrest,--O,, As, north-north-west:,
 in, make, true, This, no, easy, lion's, our,
 somever, the, save, his, And, ever, have, the, dear,
 authorities., How, time, shall, dead!, it, flat,,
 a, a, And, we, or, this, retrograde, gentleman,
```

The two additional operations, imposing a window with scroll-bars and coloring of selected words, can be seen as operations that decorate the core algorithm.

■ UML diagram



■ Implementation

```
Clear[Component, ConcreteComponent, ConcreateDecoratorA, ConcreateDecoratorB, Decorator];
```

```
SELFHEAD = Component;
Component[d_]["Operation"[]] := Print["error"]
```

```
Decorator[d_, component_][s_] := Block[{SELFHEAD = Component}, component[s]]
Decorator[d_, component_]["Operation"[]] := (Print["Decorator:", Head[component]];
   component["Operation"[]]);
```

```
ConcreteComponent[d_][s_] := Block[{SELFHEAD = ConcreteComponent}, Component[d][s]]
ConcreteComponent[d_]["Operation"[]] := Total[Flatten@{d}];
```

```
ConcreateDecoratorA[d_, component_][s_] :=
 Block[{SELFHEAD = ConcreateDecoratorA}, Decorator[d, component][s]]
ConcreateDecoratorA[d_, component_]["Operation"[]] :=
  Decorator[d, component]["Operation"[]]^d[[1]];
ConcreateDecoratorA[d_, component_]["AddedOperation"[]] :=
  Decorator[d, component]["Operation"[]]^2;
```

```
ConcreateDecoratorB[d_, component_][s_] :=
 Block[{SELFHEAD = ConcreateDecoratorB}, Decorator[d, component][s]]
ConcreateDecoratorB[d_, component_]["Operation"[]] :=
  1 / Decorator[d, component]["Operation"[]];
```

■ Experiments

First we create concrete components.

```
cc = ConcreteComponent[{a, b, c}];
cc["Operation"[]]
```

```
a + b + c
```

```
cdA = ConcreateDecoratorA[{h}, cc];
cdA["Operation"[]]
cdA["AddedOperation"[]]
```

```
Decorator:ConcreteComponent
```

$(a+b+c)^h$

```
Decorator:ConcreteComponent
```

$(a+b+c)^2$

Next we create concrete decorators that wrap around the component objects.

```
cdB = ConcreateDecoratorB[{}, cc];
cdB["Operation"[]]
```

```
Decorator:ConcreteComponent
```

$\frac{1}{a+b+c}$

```
cdAB = ConcreateDecoratorB[cdA[[1]], cdA];
cdAB["Operation"[]]
```

```
Decorator:ConcreateDecoratorA
```

```
Decorator:ConcreteComponent
```

$(a+b+c)^{-h}$

```
cdAABA = ConcreateDecoratorA[{f},
   ConcreateDecoratorA[{g}, ConcreateDecoratorB[{}, ConcreateDecoratorA[{h}, cc]]]];
cdAABA["Operation"[]]
```

```
Decorator:ConcreateDecoratorA
```

```
Decorator:ConcreateDecoratorB
```

```
Decorator:ConcreateDecoratorA
```

```
Decorator:ConcreteComponent
```

$\left(\left((a+b+c)^{-h}\right)^g\right)^f$

■ Other concrete examples

In [6] Decorator was used to implement a Single Instruction Multiple Data (SIMD) parallel algorithm for handling the simulations.

`GitHubData` objects use Decorator for changing of the font styles and the colors of graphs and charts. See [8].

Observer

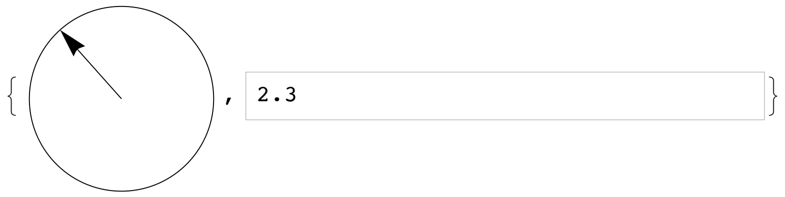
The design pattern Observer is fully implemented and supported in *Mathematica* through the functionalities of `Dynamic`, `Manipulate`, and related functions. Here for didactic purposes we show an implementation that predates the introduction of `Dynamic` (in version 6.0).

The design pattern Observer is also known as Model-View-Controller (MVC).

■ The MVC functionality within Mathematica

Here is a slightly modified version an example from the function page for `Dynamic`:

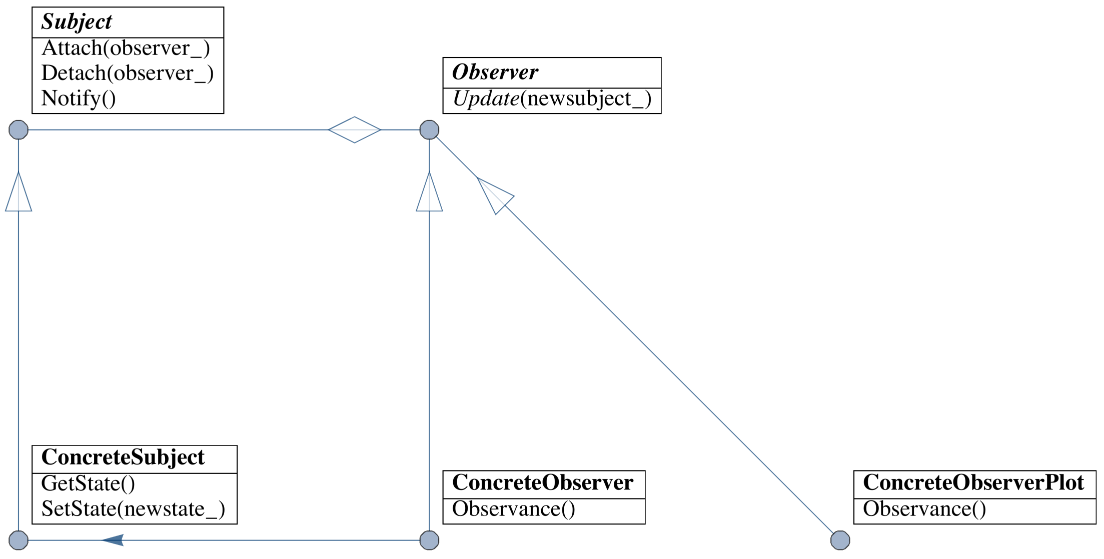
```
DynamicModule[{angle = 0, p = {1, 0}}, {LocatorPane[
   Dynamic[p, (angle = ArcTan @@ #; p = {Cos[Dynamic[angle]], Sin[Dynamic[angle]]}) &],
   Graphics[{Circle[], Arrowheads[0.15], Arrow[Dynamic[{{0, 0}, p}]]},
    ImageSize -> Tiny], Appearance -> None], InputField[Dynamic[angle], Number]}]
```



The example above allows to change the value in the input field by moving the arrow, and change arrow's direction by changing the input value. (This MVC behavior is similar in spirit to the one in the motivational example for Observer in [5].)

▪ UML diagram

For Observer it is beneficial to see not just an structural UML diagram but also an UML interaction sequence diagram, see[5]. Here is shown just a structural one.



▪ Implementation

```
ClearAll[Subject, ConcreteSubject, Observer, ConcreteObserver, ConcreteObserverPlot];

SUBJECTHEAD = Subject;
(*SetAttributes[Subject,HoldAll]*)
Subject[id_Symbol]["GetObservers"[]] := SUBJECTHEAD[id]["Observers"];
Subject[id_Symbol]["Attach"[observer_]] :=
  (SUBJECTHEAD[id]["Observers"] = Append[SUBJECTHEAD[id]["Observers"], observer]);
Subject[id_Symbol]["Detach"[observer_]] :=
  (SUBJECTHEAD[id]["Observers"] = DeleteCases[SUBJECTHEAD[id]["Observers"], observer]);
Subject[id_Symbol]["Notify"[]] := Through[(SUBJECTHEAD[id]["Observers"])["Update"[]]];

ConcreteSubject[d___][s_] := Block[{SUBJECTHEAD = ConcreteSubject}, Subject[d][s]];
ConcreteSubject[id_Symbol, observers_] :=
  Block[{}, ConcreteSubject[id]["Observers"] = observers;
   ConcreteSubject[id]];
ConcreteSubject[id_Symbol]["Data"] := {};
ConcreteSubject[id_Symbol]["GetState"[]] := ConcreteSubject[id]["Data"];
ConcreteSubject[id_Symbol]["SetState"[newstate_]] :=
 (ConcreteSubject[id]["Data"] = newstate);
```

```
OBSERVERHEAD = Observer;
Observer[id_Symbol]["SetSubject"[newsubject_]] :=
  (OBSERVERHEAD[id]["Subject"] = newsubject);
Observer[id_Symbol]["GetSubject"[]] := OBSERVERHEAD[id]["Subject"];

ConcreteObserver[id_Symbol, subject_] :=
  Block[{}, ConcreteObserver[id]["Subject"] = subject;
   ConcreteObserver[id]];
ConcreteObserver[d___][s_] := Block[{OBSERVERHEAD = ConcreteObserver}, Observer[d][s]];
ConcreteObserver[id_Symbol]["Update"[]] := Grid[{{"observer id", "output"},
    {id, ConcreteObserver[id]["GetSubject"[]]["GetState"[]]}}, Dividers → All];

ConcreteObserverPlot[id_Symbol, subject_] :=
  Block[{}, ConcreteObserverPlot[id]["Subject"] = subject;
   ConcreteObserverPlot[id]];
ConcreteObserverPlot[d___][s_] :=
  Block[{OBSERVERHEAD = ConcreteObserverPlot}, Observer[d][s]];
ConcreteObserverPlot[id_Symbol]["Update"[]] :=
  Grid[{{"observer id", "output"}, {id, Apply[Plot,
      ConcreteObserverPlot[id]["GetSubject"[]]["GetState"[]]]}}, Dividers → All];
```

■ Experiments

First let us we create a concrete subject to be observed.

```
ClearAll[csubj]
csubj = ConcreteSubject[Unique[], {}]

ConcreteSubject[$173]
```

The signature of the creating function is such that it has the head of the concrete subject class `ConcreteSubject` and it takes two arguments, an identifier symbol and a list of observers (that can be empty). The creator function returns a `ConcreteSubject` object.

Next we are going to set a state for the concrete subject.

```
csubj["SetState"[{Sin[x], {x, 0, π}}]];
csubj["GetState"[]]

{Sin[x], {x, 0, π}}
```

Let us create two concrete Observer objects, one for printing the state of the subject, the other for plotting the state of the subject.

```
ClearAll[ob, obp];
ob = ConcreteObserver[Unique[], csubj];
obp = ConcreteObserverPlot[Unique[], csubj];
```

What the creator functions of the observers do parallels what the creator for concrete subject objects does. The signature of each creating function is such that it has the head of the concrete observer class and it takes two arguments, an identifier symbol and a list of subjects (that can be empty). The creator functions return the corresponding concrete observer objects.

Next we attach the observers to the subject.

```
csubj["Attach"[ob]];
csubj["Attach"[obp]];

csubj["GetObservers"[]]

{ConcreteObserver[$174], ConcreteObserverPlot[$175]}
```

This command retrieves the current state of the subject:

```
csubj["GetState"[]]

{Sin[x], {x, 0, π}}
```

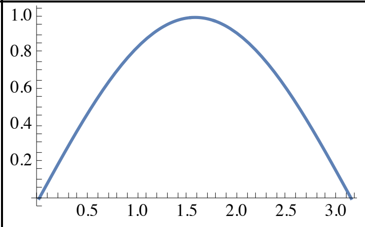
The following command notifies the observers attached to the subject about its state. The result is a list of the observers views on the subject's state.

```
csubj["Notify"[]]
```

{

observer id	output
$174	{Sin[x], {x, 0, π}}

,

observer id	output
$175	

}

Let us change the state of the subject to something else.

```
csubj["SetState"[{Cos[x], {x, 0, 2 π}}]]
```

```
{Cos[x], {x, 0, 2 π}}
```

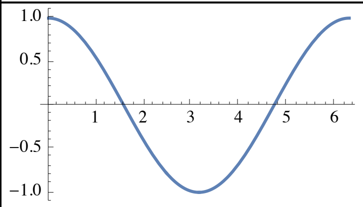
And let us notify the attached observers to react to the state change.

```
csubj["Notify"[]]
```

{

observer id	output
$174	{Cos[x], {x, 0, 2 π}}

,

observer id	output
$175	

}

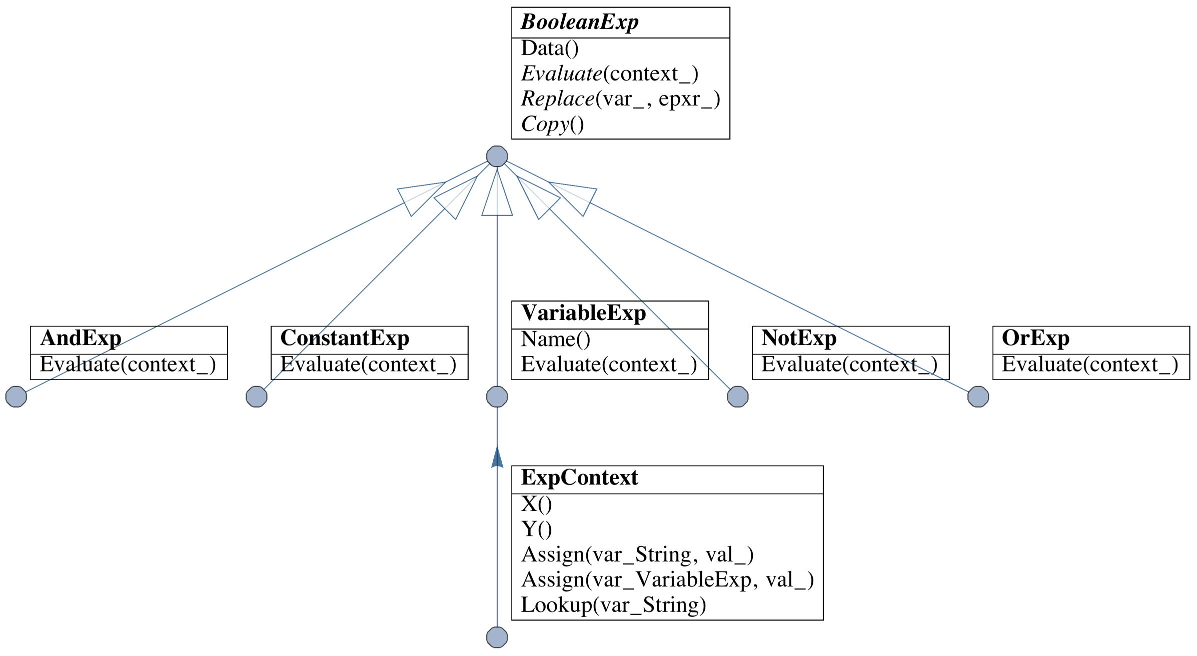
Interpreter

The design pattern Interpreter prescribes how to implement interpretation of (small and context free) languages. The intent given in [5] is:

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

Interpreter is a powerful design pattern that allows to approach the solution of re-occurring problems using Domain Specific Languages (DSL's).

■ UML diagram



■ Backus-Naur Form (BNF)

The implementation -- and explanation -- of the design pattern Interpreter is based on the Backus-Naur Form (BNF). For each BNF rule we program a class that provides an interpretation for it. Interpreter traverses a tree of linked objects corresponding to the BNF rules.

Here is the BNF of a DSL for handling Boolean expressions of a certain type:

```
BooleanExp::=VariableExp|Constant|OrExp|AndExp|NotExp|'(' BooleanExp')'
AndExp::=BooleanExp'and' BooleanExp
OrExp::=BooleanExp'or' BooleanExp
NotExp::='not' BooleanExp
Constant::='true'|'false'
VariableExp::='A'|'B'|...|'X'|'Y'|'Z'
```

■ Implementation

The implementation follows the BNF in the previous section.

```
Clear[BooleanExp, VariableExp, ConstantExp, AndExp, OrExp, NotExp]

SELFHEAD = BooleanExp;
BooleanExp[d___]["Data"[]] := d;
BooleanExp[d___]["Evaluate"[context_]] := Null;
BooleanExp[d___]["Replace"[var_, epxr_]] := Null;
BooleanExp[d___]["Copy"[]] := Null;

VariableExp[d___][s_] := Block[{SELFHEAD = VariableExp}, BooleanExp[d][s]]
VariableExp[{var_String, rest___}]["Name"[]] := var;
VariableExp[{var_String, rest___}]["Evaluate"[context_]] := context["Lookup"[var]];

ConstantExp[d___][s_] := Block[{SELFHEAD = ConstantExp}, BooleanExp[d][s]]
ConstantExp[{val_, rest___}]["Evaluate"[context_]] := If[val === "true", True, False];

AndExp[d___][s_] := Block[{SELFHEAD = AndExp}, BooleanExp[d][s]]
AndExp[{operand1_, operand2_, rest___}]["Evaluate"[context_]] :=
  operand1["Evaluate"[context]] && operand2["Evaluate"[context]];
```

```
OrExp[d___][s_] := Block[{SELFHEAD = OrExp}, BooleanExp[d][s]]
OrExp[{operand1_, operand2_, rest___}]["Evaluate"[context_]] :=
  operand1["Evaluate"[context]] || operand2["Evaluate"[context]];

NotExp[d___][s_] := Block[{SELFHEAD = NotExp}, BooleanExp[d][s]]
NotExp[{operand1_, rest___}]["Evaluate"[context_]] :=
  Not[operand1["Evaluate"[context]]];

Clear[ExpContext]
ExpContext[d_]["Assign"[var_String, val_]] := ExpContext[d][var[]] = val;
ExpContext[d_]["Assign"[var_VariableExp, val_]] :=
 Module[{name = var["Name"[]]}, ExpContext[d][name[]] = val];
ExpContext[d_]["Lookup"[var_String]] := ExpContext[d][var[]]
```

Dummy client class for the UML diagram.

```
Clear[Client]
Client[___]["Operation"[context_ExpContext, expr_BooleanExp, ___]] :=
 expr["Evaluate"[context]]
Client[{context_ExpContext, expr_}]["Evaluate"] := expr["Evaluate"[context]]
```

▪ Experiments

```
(true and x) or (y and (not x))
```

```
tr = ConstantExp[{"true"}];
fl = ConstantExp[{"false"}];
x = VariableExp[{"X"}];
y = VariableExp[{"Y"}];
orexp = OrExp[{AndExp[{tr, x}], AndExp[{y, NotExp[{x}]}]}];

Clear[cont]
cont = ExpContext["first"];
cont["Assign"[x, False]]
cont["Assign"[y, False]]

False

False

orexp["Evaluate"[cont]]

False

texp = OrExp[{y, NotExp[{x}]}];

texp["Evaluate"[cont]]

True
```

▪ Other concrete examples

I designed and implemented `NIntegrate`'s `Method` option using the Interpreter design pattern. (That design was also applied to `NSum`.) See [9] for brief details.

▪ Extension

Interpreter is probably my favorite design pattern and I have extended it with the package FunctionalParsers.m, [11]. With that package we can overcome one of the serious limitations of Interpreter, its applicability to simple grammars, [5]. A sufficiently complex and concise application of the package FunctionalParsers.m is given in my blog post "Natural language processing with functional parsers", [12].

For example here is the Extended BNF (EBNF) of a simple grammar for expressing food desires:

```
ebnfCode = "
<lovefood> = <subject> , <loveverb> , <object-spec> ;
<loveverb> = ( 'love' | 'crave' | 'demand' ) <@ LoveType ;
<object-spec> = ( <object-list> | <object>
    | <objects> | 'Range[2,100]' , <objects> ) <@ LoveObjects ;
<subject> = 'i' | 'we' | 'you' <@ Who ;
<object> = 'sushi' | [ 'a' ] , 'chocolate'
    | 'milk' | [ 'an' , 'ice' ] , 'cream' | 'a' , 'tangerine' ;
<objects> = 'sushi' | 'chocolates' | 'milks' | 'ice' , 'creams' | 'tangerines' ;
<object-list> = ( <object> | <objects> ) , { 'and' ⊳ ( <object> | <objects> ) } ; ";
```

This command generates the grammar parsers:

```
GenerateParsersFromEBNF[ToTokens@ebnfCode];
```

These commands show the parsing of different sentences of the grammar defined above:

```
sentences = {"I love sushi", "We demand 2 ice creams", "You crave chocolate and milk"};
Magnify[#, 0.8] &@
 ParsingTestTable[pLOVEFOOD, ToLowerCase@sentences, "Layout" → "Vertical"]
```

```
1  command: i love sushi
    parsed: {Who[i], {LoveType[love], LoveObjects[{sushi, {}}]}}
  residual: {}
2  command: we demand 2 ice creams
    parsed: {Who[we], {LoveType[demand], LoveObjects[{2, {ice, creams}}]}}
  residual: {}
3  command: you crave chocolate and milk
    parsed: {Who[you], {LoveType[crave], LoveObjects[{{{}, chocolate}, {milk}}]}}
  residual: {}
```

At this point it is easy enough to develop interpreters for the parsed sentences.

Future plans

1. Implement the generation of UML object interaction sequence diagrams using Design Patterns implementations (in *Mathematica*).

2. Implement code generation based on specified Design Patterns. It is especially interesting the code generation to be specified by natural language sentences describing the system architecture.

3. Investigate discovering design patterns within existing code.

References

[1] Christopher Alexander et al., *The Oregon Experiment,* Oxford University Press, 1975.

[2] Christopher Alexander et al., *A Pattern Language - Towns, Buildings, Construction,* Oxford University Press, 1977.

[3] Christopher Alexander, *The Timeless Way of Building*, Oxford University Press, 1979.

[4] Kent Beck, Ward Cunningham, “Using Pattern Languages for Object-Oriented Programms”, OOPSLA-87, (1987).

[5] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[6] Anton Antonov, Air pollution modeling with grid*Mathematica*, Wofram Technology Conference 2006. URL: http://library.wolfram.-com/infocenter/Conferences/6532/ .

[7] Anton Antonov, GitHubPlots, *Mathematica* package, MathematicaForPrediction at GitHub, 2015. URL: https://github.com/antononcube/-MathematicaForPrediction/blob/master/Misc/GitHubPlots.m .

[8] Anton Antonov, GitHubObjects, *Mathematica* package, MathematicaForPrediction at GitHub, 2015. URL: https://github.com/antonon-cube/MathematicaForPrediction/blob/master/Misc/GitHubDataObjects.m .

[9] Anton Antonov, Object Oriented Design Patterns, Wofram Technology Conference 2015. URL: http://www.wolfram.com/broad-cast/video.php?c=400&v=1470 .

[10] Anton Antonov, UML Diagram Generation, *Mathematica* package, MathematicaForPrediction at GitHub, 2016. URL: https://github.-com/antononcube/MathematicaForPrediction/blob/master/Misc/UMLDiagramGeneration.m.

[11] Anton Antonov, Functional parsers, *Mathematica* package, MathematicaForPrediction at GitHub, 2014. URL: https://github.-com/antononcube/MathematicaForPrediction/blob/master/FunctionalParsers.m .

[12] Anton Antonov, "Natural language processing with functional parsers", (2014), MathematicaForPrediction at WordPress blog. URL: https://mathematicaforprediction.wordpress.com/2014/02/13/natural-language-processing-with-functional-parsers/ .

[13] Anton Antonov, Object-oriented framework for large scale air pollution models, 2001. Ph.D. thesis, Informatics and Mathematical Modelling, Technical University of Denmark, DTU.