Appeared in C++ Report, Vol 9 #9, pages 32–42, October 1997

The Function Object Pattern

Thomas Kühne (kuehne@isa.informatik.th-darmstadt.de)
Department of Computer Science, TU Darmstadt
Magdalenenstr. 11c, D-64289 Darmstadt

1 Introduction

The Function Object pattern can be regarded as a variation to the Command pattern that takes parameters and returns a result. The pattern describes how to build encapsulated components for behavior parameterization, function reuse, calculation on demand, representing "business-transactions", and first-class behavior, e.g., protocol-, undo-, and persistence mechanisms. In contrast to Command, Function Object establishes a useful collaboration with iterators. We show in particular how to use generic function objects with iterators in order to allow multi-dispatching operations on heterogeneous data structures.

Function Object belongs to the class of design patterns that describe how to compensate for missing language features. For instance, Convenience Methods [7] shows how to emulate default parameters and Visitor [5] is less needed in languages with multi-dispatch. Function Object describes how to achieve the benefits that Smalltalk programmers gain from blocks (and functional programmers gain from higher-order functions) in object-oriented languages without such a feature. Very similar to the Command pattern [5] Function Object objectifies behavior and thus opens up many useful applications (see section 2.4) which go even beyond the potential of Smalltalk blocks (see section 2.9.4). As Command and Strategy [5] Function Object defines behavior that is not tied to a particular data abstraction. So-called free functions usually do not occur during domain analysis. They resemble so-called *design-objects* [12], like iterator objects, and event handler.

Technically commands and function objects can be called closures. A closure is a function that on creation is able to capture variables values from its environment. In case of Command the environment variables (e.g., the text for an editor command) are passed as constructor arguments. In addition, a function object accepts further arguments after creation from any client it was passed to and with all benefits of partial parameterization (see section 2.3).

The following section not only explains a design technique but deliberately uses many examples to illustrate the various usages of Function Object. Section 3 then attempts to explain Function Object's implications for software reuse by enumerating the abstract concepts involved.

2 Pattern: Function Object

2.1 Intent

Encapsulate a function with an object. This is useful for parameterization of algorithms, partial parameterization of functions, delayed calculation, lifting methods to first-class citizens, and for separating functions from data.

2.2 Also Known As

Lexical Closure [2], Functor [4], Agent [6], Agent-Object [8], Functionoid [3], Functoid [10], Function-Object [14].

2.3 Motivation

Behavior parameterization occurs in almost every program. Iterators are a good example. The iteration algorithm is constant whereas an iteration action or function varies. A special kind of iteration is the `has` (member test) function of an collection. Consider a collection of books. Member testing should cover testing for a particular book title, book author, book type, etc.

2.3.1 Problem

One way to do this is to use an *external* iterator. Then the varying predicate (compare title, compare author, etc.) can be combined with the traversal algorithm by placing the predicate in an explicit loop that advances the external iterator one by one. As a result, the number of explicit loops corresponds to the number of member test predicates.

However, there are good reasons to write such a loop only once (see, e.g., the discussion in the Iterator pattern [5]). Consequently, we use an *internal* iterator. Given a predicate, it returns true if any of the books fits the predicate. Here is how the predicate is "given" to the internal iterator conventionally:

The actual member test method is a Template Method [5], which depends on an abstract predicate. The implementation for the abstract predicate, and thus the specific member test operation, is given in descendants [12]. Selection of the member tests is done by selecting the appropriate descendant. So, traversal algorithm and functions in general are combined through dynamic binding of the abstract function method. Note that this forces us to place the member test method outside the collection of books (e.g., at iteration objects) since we do not want to create book collection subclasses but member test variations only. Further disadvantages aligned with the above application of an object-oriented design, using inheritance and dynamic binding are:

Static combination. All possible combinations of iteration schemes and functions are fixed at compile time. Neither is it possible to create a new function at run time.

Combinatorial explosion. Sometimes it is useful to select not just one, but a combination of functions or tests and functions. With subclassing, it is not feasible to provide any independent combination, since it leads to an exponentially growing number of subclasses.

Subclass proliferation. Each new function demands a new subclass of `ITERATOR`. The name space for classes is cluttered by many concrete `ITERATOR` subclasses. We may combine all functions in one subclass using repeated inheritance, but this only makes things worse. First, it is non-local design to lump all functions in one class. Second, we have to apply heavy renaming for iteration schemes and functions in the subclass; any combination of iteration scheme and function must be given a distinct name. Third, we lose the ability to use dynamic binding for the selection of a function. Since all functions belong to one class, we no longer can use concrete `ITERATOR` instances to select the actual combination of iteration and function.

Awkward Reuse. Reusing the functions for other iteration schemes or different purposes is practically impossible if they are defined in `ITERATOR` subclasses. The solution is to extract the functions in classes of their own. But now multiple inheritance is necessary in order to inherit from `ITERATOR` **and** to inherit from a particular function. At least multiple tests or functions can be "mixed-in", but scope resolution is needed, and each function combination results in a combinator subclass.

Poor encapsulation. Composing an iteration scheme and a function with inheritance joins the name spaces of both. In fact, the multiple inheritance solution causes iterator, function, and combinator class to share the same name-space. Implementation changes to either of the classes can easily invalidate the other. An interface between super- and subclasses, as the private parts in C^{++}, alleviates the problem considerably.

Unrestricted flexibility. Creating a designated class for the combination of an iteration scheme and a function opens up the possibility of overriding the iteration scheme for particular actions. Explicitly counting the elements in a collection could be replaced by just returning the value of an attribute `count`. Unfortunately, this advantage for the designer of a library is a disadvantage for the user of a library. The user may rely on properties of the original iteration scheme. If the iteration function not only counts the elements, but in addition produces some side-effect, the side-effects will not be executed in the optimized version described above.

Identity changes. In order to change the iteration function a different iterator instance must be used. While one would seldom need to rely on an unchanging iterator instance, this property is inhibiting in other settings of parameterization. For instance, it might be mandatory to keep the same

instance of a cook, while being able to process different recipes.

2.3.2 Solution

The best way to get rid of the above disadvantages is to objectify predicates with the Function Object pattern. Combining traversal algorithm and function with Function Object works through literally "giving" an objectified function to an internal iterator. In our example, the member test method accepts a test predicate to be used during iteration:

```
bool has(Function<Book, bool>& pred) {
 for (int i=0; i<count; i++)
   if (pred(books[i]))
     return true;
 return false;
};
```

Here, the collection of books simply consists of an array of books that is traversed with an integer loop. The predicate variable `pred` is passed by reference in order to allow dynamic binding. Note how C++ allows to use a nice function application syntax for passing the current book to the predicate. The common interface for all function objects is:

```
template <class In, class Out>
class Function {
public:
 virtual Out operator() (In arg)=0;
};
```

The `()`-operator is defined to take a generic argument type `In` and to return a generic result type `Out`. The member test method from above instantiates these to `Book` and `bool` respectively. The application operator is declared `virtual` since derived `Function` classes will define concrete functions.

A predicate to check for a bible instance is:

```
class IsBible : public
       Function<Book, bool> {
public:
 virtual bool operator() (Book b) {
   return b.type == bible;
 }
};
```

Now member testing looks like:

```
containsBible = library.has(IsBible());
```

Checking for a book with a certain title can be achieved by passing a predicate that receives the book title through its constructor:

```
library.has(CheckTitle("Moby Dick"));
```

Yet, there is another exciting way to achieve the same thing. Although the member test method expects a predicate with one book parameter only we can make a two argument compare function fit by passing a book with the title to be looked for in advance:

```
library.has(TitleCompare()(mobyBook));
```

`TitleCompare()` creates a predicate with two parameters. Passing `mobyBook` results in a predicate with one parameter that perfectly fits as an argument to the `has` method. Thanks to the generic type parameters of function objects the compiler can check for correct function application and will reject wrong uses concerning type or number of arguments.

While returning a predicate as a result works in the particular case above, a more sophisticated scheme has to applied in C++ for the general case. See section 2.9.1 for further details.

Finally, we may combine multiple search criteria like title comparison and date checking by composing predicates with a composite function object:

```
library.has(And()(pred1)(pred2));
```

`And` takes a variable number of predicates as arguments, applies each of them to the book it receives, and returns `true` if all predicates hold.

2.4 Applicability

- *Parameterization.* Function objects are a good candidate whenever general behavior can be adapted to special behavior:

  Dynamics. In addition to run time selection of existing function objects, new function objects can also be created at run time. A user may dynamically compose a multi-media function object from text-, graphic-, and sound-producing function objects.

Orthogonality. Having more than one behavior parameter creates the problem of handling all possible combinations of the individual cases. Function objects can freely be mixed without interfering and without combinator classes.

Reuse. Function objects can be used by any adaptable algorithm that knows their interface. Even if the algorithm was not designed to supply a function with necessary arguments, it is often possible to supply them to the function in advance. Consider an error reporter, parameterized by output format functions, only intended for generating text messages. We can upgrade the reporter to create a graphical alert-box by passing a function object that already received information about box-size, colors, etc.

Identity. When the behavior of an object should change while keeping its identity, function objects can be used as behavior parameters to the object. In contrast, encoding behavior in subclasses calls for something like Smalltalk's "`become:`" in order to achieve the same effect.

- *Business transactions.* Often the *functions* are the stable concepts of a system and represent good maintainance spots, in order to cope with changing functionality. Instead of being a well-defined operation on one single object, transactions are *"an orchestration of objects working together toward a common goal"* [4]. When transactions do not naturally fit into existing data abstractions, Function Object can lift them to first-class status.

- *Monolithic Algorithms.* Just like Strategy [5] Function Object can be used to define localized algorithms for data structures. See section 2.13 for a comparison of Command, State, Strategy, and Function Object.

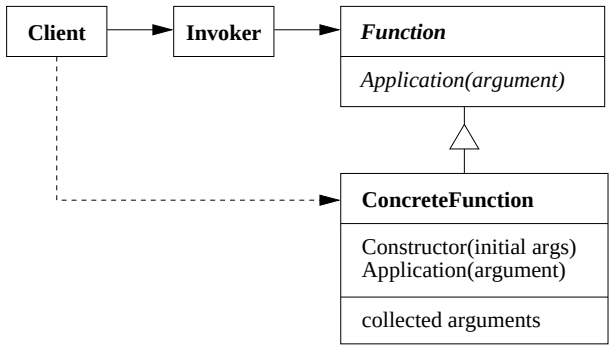
- *Small interfaces.* When an object potentially supports many extrinsic operations (e.g., CAD-objects may support different viewing methods, cost calculations, etc.), but its interface preferably should contain the basic, intrinsic functions only (e.g., geometric data), then the functionality can be implemented in function objects that take the object as an argument.

- *Method simplification.* If a large method, containing many lines of code, can not be split into smaller, more simple methods, because the code heavily uses temporary variables for communication, then the method can be transformed into a function object. The main transformation is to replace the temporary variables with function object attributes. As a result, the method can be split up into more manageable sub-methods, without passing parameters between inter-method invocations, since communication still can take place via function object attributes. The main computation method simply puts the pieces together, itself being as clear as documentation [1].

- *Delayed Calculation.* Function objects postpone the calculation of their result until it is actually needed. When a function object is passed as a parameter but the receiving method does not make use of it, it will not be evaluated. If the result is never needed, this pays off in run time efficiency. Function Object effectively supports lazy evaluation and thus can be used to implement infinite data structures and supports modularization by decoupling data generation from data consumption.

Do not use Function unless you have a concrete reason. There is a time and space penalty in creating and calling a function object, instead of just invoking a method. Also, there is an initial effort to write an extra function class. Functions with many parameters require as many class definitions in order to exploit partial parameterization. Finally, sometimes *unrestricted flexibility* as mentioned in section 2.3.1 is clearly desirable. Functions that vary with the implementation of their argument should be members of this abstraction and not free function objects. Also, see the remarks concerning *flexibility* and *efficiency* in section 2.8.

2.5 Structure

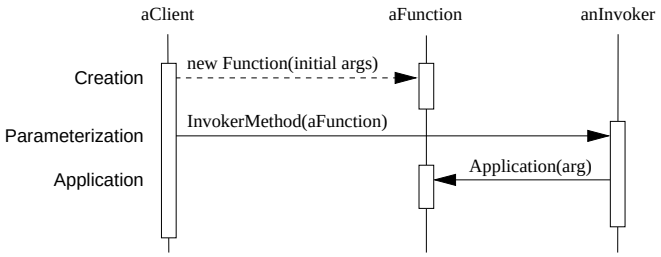


2.6 Participants

- **Function**
  - declares an interface for function application.
- **ConcreteFunction** (e.g., `isBible`)
  - implements a function.
  - collects argument values until evaluation.
- **Client**
  - creates a ConcreteFunction.
  - possibly applies it to arguments.
  - calls Invoker with a ConcreteFunction.
- **Invoker** (e.g., Iterator)
  - applies ConcreteFunction to arguments
  - returns a final result to its Client.

2.7 Collaborations

- A client initializes a Function.
- An invoker takes the Function as a parameter.
- The invoker applies the Function to arguments.
- The client receives a result from the invoker.



2.8 Consequences

- *Abstraction.* Function objects abstract from function pointers and in particular from pointers to methods. Instead of the C++ code: `aFilter.*(aFilter.current)(t)`, we can write `aFilter(t)` [4].
- *Simplicity.* The use of function objects does not introduce inheritance relationships and does not create spurious combinator classes.
- *Explicitness.* The code `cook.prepare(fish)` is easy to understand. When recipes are wired into `COOK` subclasses, `cook.prepare` depends on the actual `cook` type. Clever variable names (e.g., `fish_cook.prepare`) often are not an option, e.g., `cook.prepare(fish)`, followed by `cook.prepare(desert)`.
- *Compositionality.* In analogy to Macro-Command [5], function objects can be dynamically composed to form a sequence of functions by forwarding intermediate results to successing function objects. Composite function objects may also apply several component function objects in parallel. Variants differ in the way they produce a final output from the single results.
- *Uniform invocation.* Imposing a function object's interface on related operations allows to uniformly invoke them. Instead of switching to different method names (e.g., `compTitle`, `compAuthor`), we evaluate an abstract function object and rely on dynamic binding [5]. Consequently, we can add new operations, without changing the caller (e.g., event handler).
- *Encapsulation.* As function objects establish client relationships only, they are protected from implementation changes to algorithms that use them. Likewise, the implementation of function objects can change without invalidating the algorithms. Hence, Function Object allows *black-box reuse* and helps to advance reuse by inheritance to reuse by composition.
- *Security.* A client of an adaptable algorithm can be sure not to change the algorithm semantics. It is impossible to be given an optimized version which does not fully comply to the original semantics (see 2.3.1 *Unrestricted flexibility*).
- *Flexibility.*
  + A statement like `iterator.do(f)` is polymorphic in three ways:
    1. `iterator` may internally reference any data structure that conforms to a particular interface.
    2. The actual instance of `iterator` determines the iteration strategy (e.g., pre- or post-order traversal on trees).
    3. The actual instance of function object `f` determines the iteration function.

– It is not possible to automatically optimize algorithms for specific functions. Nevertheless, one more level of indirection can explicitly construct combinations of functions and optimized algorithms.

– When a function has been extracted from a data structure, it is no longer possible to simply redefine it in future derivations. One way to account for this is to make the extracted function a generic Function Object (see section 2.11) that will discriminate between data structure variations.

- *Reuse.* Adaptable algorithms become more reusable because they do not need to know about additional parameters for functions.

  Moreover, function objects are multi-purpose:

  + Functions are not bound to a particular adaptable algorithm, e.g., comparing booktitles is useful for sorting and for membership testing in collections.

  + One function with n parameters actually represents n functions and one value. The first function has n parameters. The second, created by applying the first to an argument, has $n-1$ parameters, and so on, until the last function is applied to an argument and produces the result.

    An example from physics shows the useful functions which can be created from the gravitational force function:

    $$\begin{aligned} GravityLaw\ m_1\ r\ m_2 &= \frac{G\, m_1\, m_2}{r^2} \\ force_{earth} &= GravityLaw\ mass_{earth} \\ force_{surface} &= force_{earth}\ radius_{earth} \\ force_{my} &= force_{surface}\ mass_{my} \end{aligned}$$

- *Iteration.* Function Object suggests the use of internal, rather than external, iterators. Internal iterators avoid explicit state and reoccurring explicit control loops. Often external iterators are promoted to be more flexible. It is said to be practically impossible to compare two data structures with an internal iterator [5]. However, we simply propose to extend an iterator to accept not just one, but n data structures. A *transfold*[1]-method may access the first, second, etc., elements of all data structures simultaneously.

  Function Object allows to make iteration a method of data structures since it does not demand for subclassing the data structure. This facilitates the use of iterators and allows to redefine iteration algorithms for special data structures. Moreover, the data structure (e.g., `DICTIONARY`) then does not need to export methods (e.g., `first`, `next`) in order to allow iterators to access its elements.

- *Efficiency.*

  + A function object may calculate partial results from arguments and pass these to a result function. Hence, the partial result is computed only once, no matter how many times the resulting function object will be applied to different arguments in the future, e.g., $force_{surface} = force_{earth}\ costly\_calc$, and then $force_{my} = force_{surface}\ mass_{my}$; $force_{your} = force_{surface}\ mass_{your}$.

  – Passing client parameters to function objects can be more inefficient than to, e.g., directly access internal attributes of an iterator superclass. In principle this could be tackled by compiler optimizations.

  – Care should be taken not to unnecessarily keep references to unevaluated calculations, i.e., function objects. Otherwise, the occupied space can not be reclaimed.

  – Finally, function objects access the public interface of their servers only. This represents positive decoupling, but can be more inefficient than unrestricted access. However, selective export (Eiffel) or *friends* (C^{++}), allow to trade in efficiency for safety.

2.9 Implementation

- *Call-by-value.* Function objects must copy their arguments. Otherwise, their behavior will depend on side-effects on their arguments. In general, this will produce unpredictable results. In some cases, however, it may be desirable. The function object then plays the role of a future variable, which is passed to an invoker before all data needed to compute the result is available. Long after the function object has been passed to the invoker, it can be supplied with

[1] Its functional definition shall be: $transfold\ f\ a\ g \equiv (foldr\ f\ a) \circ (map\ g) \circ transpose$

the necessary data by producing side-effects on arguments.

- *Initial arguments.* Real closures (e.g., Smalltalk blocks) implicitly bind variables, which are not declared as parameters, in their creation environment. This is not possible with function objects. One way out is to treat initial arguments and standard arguments uniformly: Initial arguments are passed as arguments to the function object in its creation environment. Alternatively, it is possible to pass initial arguments through the function object constructor (see `CheckTitle` in section 2.3.2). This saves the intermediate classes needed to implement the partial application to initial arguments. Of course, both variants do not exclude each other. Note that the implicit variable binding of closures forces them to use the the same variables names as its creation environment. In contrast, a function object can be used in various environments without the need to make initial argument names match the environment.

- *Delayed calculation.* Commonly a function is evaluated after it has received its last argument. Yet, function object application and evaluation can be separated by corresponding methods for application and evaluation. As a result, the client, triggering evaluation, does not have to know about the last argument. Also, the supplier of the last argument does not need to enforce the calculation of the result, which is crucial for lazy evaluation. In order to enable automatic evaluation on full argument supply while still supporting the above separation, it is possible to evaluate on the last parameter and allow a kind of dummy parameter (called `unit` in ML).

- *Partial parameterization.* Two extremes to implement partial parameterization exist. The less verbose is to always keep the same instance of function object and assign incoming arguments to corresponding internal attributes. This will cause trouble when the same function object is used by several clients. As the application of a function object does not create a new instance, the clients will get confused at the shared state. Furthermore, static typing becomes impossible. If each application of a function object produces a new instance of a different type, then static typing is enabled again and no unwanted sharing of function object state can occur. Unfortunately, this forces us to write at least $n-1$ classes for a function object with n parameters.

2.9.1 Implementation in C^{++}

Of the languages discussed in this section, C^{++} is the only one without garbage collection and requires more effort in order to support *upward-funargs*[2] and delayed calculations. Constructors and destructors only work for function objects with lifetimes determined by scoping rules. The Bridge pattern [5] can suitably be used to achieve both dynamic binding and value semantics with correct memory management [10].

2.9.2 Implementation in Eiffel

Eiffel suggests to use the infix operator `"@"` for application. In principle, Eiffel allows to use a non-generic function interface since the application method can be covariantly redefined. However, we recommend to use a generic interface as in C^{++} in order to avoid complications through the combination of polymorphism and covariant redefinition, i.e., *catcalls*.

2.9.3 Implementation in Java

Java neither features covariant redefinition of methods nor a genericity mechanism. This means that we either have many function interfaces, one for each function type, or we must explicitly downcast arguments in concrete functions and downcast results in client code respectively. The latter approach is not type safe and is further complicated by the fact that basic values must be explicitly converted into objects and vice versa. As a minor point, Java does not allow the nice `()`-syntax of C^{++}.

2.9.4 Implementation in Smalltalk

Implementation of Function Object is easy and despite the presence of blocks can still be reasonable: Function Object, in addition, allows to reference functions by (class) name, exploit inheritance between functions, and to apply Kent Beck's method simplification (see section 2.4). From the extensions to Function Object listed in the next section,

[2] Functions returned as results. These do not have a determined lifetime.

blocks also do not support keyword parameters and undoing of imperative function objects.

2.10 Sample Code

In combination with the already presented C++ code snippets this section provides a full running example using a function object to find books in a library.

First let us define a simple `book` interface:

```
class Book {
public:
  Book() : title("No Title") {};
  Book(string t) : title(t) {};
  string title;
};
```

A collection of such books is maintained in a (in this case very simple) library:

```
class Library {
public:
  Library() : count(0) {};

  void add(Book book) {
    books[count++]=book;
  }

  bool has(Function<Book, bool>& pred) {
    for (int i=0; i<count; i++)
      if (pred(books[i]))
       return true;

    return false;
  }

private:
  int count;
  Book books[10];
};
```

Class `Library` allows to `add` books and it features the `has` method that we previously encountered. Any predicate `pred` to be used with `has` derives from class `Function`. Here is the code for predicate `CheckTitle`:

```
class CheckTitle : public
      Function<Book, bool> {
public:
  CheckTitle(Book book) :
  _title(book.title) {}

  virtual bool operator() (Book arg) {
    return _title == arg.title;
  }

private:
  string _title;
};
```

`CheckTitle` receives a book during creation and memorizes its title. When applied to a `book` argument, the memorized title is compared with the book's title.

Typical client code will declare some books and add them to the library:

```
Library library;
Book book1("The Time Machine");
Book book2("2001: A Space Odysee");

library.add(book1);
library.add(book2);

class Function<Book, bool>&
      searchTitle=CheckTitle(book1);

cout << "Is book1 in library? "
     << library.has(searchTitle)
     << endl;

cout << "Is book2 in library? "
     << library.has(CheckTitle(book2))
     << endl;

cout << "Is Moby Dick in library? "
     << library.has(CheckTitle(
                    Book("Moby Dick")))
     << endl;
```

The code above demonstrates

- how to declare and initialize a function object that is then passed via a variable to the `has` method.
- in-place creation of a function object with its constructor.
- in-place creation of both function object and its book argument.

2.11 Function Object variants

This section shortly touches upon important extensions to Function Object.

- *Keyword Parameters.* In addition to standard application, function objects may provide keyword parameters. Accordingly, parameters can be passed in any order. This makes sense, in order to create useful abstractions. If the definition of the gravitational force in the example of section 2.8 had been $GravityLaw\ m_1\ m_2\ r = \frac{G\ m_1\ m_2}{r^2}$, (note the different order of parameters) we can not define: $force_{surface} = GravityLaw\ mass_{earth}\ radius_{earth}$. With keyword parameters we can write: `f:=GravityLaw.m1(e-mass).r(e-radius);`

- *Imperative result.* It is possible to use the internal state of a function object to calculate one or multiple results (add accumulated results to ConcreteFunction in the structure diagram of section 2.5). For instance, one function object may count and simultaneously sum up the integers in a set during a single traversal. The set iteration client must request the results from the function object through an extended interface (add an arrow getResult from aClient to aFunction in the diagram of section 2.7). Note that imperative function objects may produce any result from a standard traversal algorithm. The latter does not need any adaption concerning type or whatsoever.

- *Procedure Object.* If we allow function objects to have side effects on their arguments we arrive at the Command pattern extended with parameters and result value. Procedure Object makes methods amenable to persistent command logging, command histories for undoing, network distribution of commands, etc. Like Command, Procedure Object may feature an `undo` method, which uses information in the procedure object's state to undo operations [5]. Note how easy it is to compose a procedure object, to be used as an iteration action, with a function object predicate that acts as a sentinel for the action. As a result, special iterations as "`do_if`" [12] can be replaced with a standard iteration.

- *Multi-dispatch.* Sometimes an operation depends on more than one argument type. For instance, adding two numbers works differently for various pairs of integers, reals, and complex numbers. Simulating multi-dispatch with standard single dispatch results in many additional methods (like `add_Integer`, `add_real`). The dispatching code is thus distributed over all involved classes. If, as in the above example, the operation must cope with a symmetric type relation (e.g., `real`+`int` & `int`+`real`), **each** class has to know all other argument types.

  A generic[3] function object removes the dispatching code from the argument types and concentrates it in one place. It uses run time type identification to select the correct code for a given combination of argument types. Note that nested type switches can be avoided with partial parameterization: Upon receipt of an argument, a generic function object uses one type switch statement to create a corresponding new generic function object that will handle the rest of the arguments. A full application of generic Function Object is presented in [9]. Unfortunately, the necessary switch statements on argument types are sensitive to the introduction of new types[4]. Yet, in the case of single-dispatch simulation, new dispatching methods (e.g., `add_complex`) are necessary as well.

  The goals of the Visitor pattern [5] can be achieved with a combination of generic Function Object and any iteration mechanism. A generic function object chooses the appropriate code for each combination of operation and element type. Once the generic Function Object has done the dispatch, the exact element type is known and access to the full interface is possible. Between invocations, function objects can hold intermediate results, e.g., variable environments for a type-checking algorithm on abstract syntax nodes.

  Note that a generic function object can be realized as a type dispatcher, parameterized with a set of function objects that actually perform an operation. This allows reuse of the dispatching part for various operations.

  Visiting data structures with Function Object is acyclic w.r.t. data dependencies [11] and does not force the visited classes to know about visitors.

[3] Named after CLOS' generic functions.

[4] A more flexible approach is to use dynamically extendible dictionaries that associate types with code.

2.12 Known Uses

Apart from the uncountable uses of Function Object in functional programming and Scheme, there are many truly object-oriented uses: Smalltalk features *blocks* as true closures with implicit binding of free variables. Sather provides *bound routines*. The Eiffel Booch Components uses Function Object for searching, sorting, transforming and filtering of containers [6]. The Standard Template Library, which was adopted as part of the standard C^{++} library, uses Function Object to inline operations for arithmetic, logic, and comparison [14].

2.13 Related Patterns

2.13.1 Categorization

- *Objectifier:* A function object, like Objectifier, does not represent a concrete object from the real world [15], though one can reasonably take business-transactions for real. Function Object is very similar to Objectifier, in that it objectifies behavior and takes parameters during initialization and call. Per contra, clients "have-an" Objectifier, while clients "take-a" function object. The latter is a *uses*, not a *has-a* relationship.

- *Command:* A procedure object which does not take any arguments after creation and produces side-effects only boils down to the Command pattern [5]. One key aspect of Command is to decouple an invoker from a target object. Function objects typically do not delegate functionality. Rather than delegating behavior to server objects they implement it themselves. So, function objects normally do not work with side-effects, but return their computation as an argument-application result. Nevertheless, function objects also can be used for client/server separation, i.e., as Call-back functions. In addition to Command, invokers are then able to pass additional information to function objects by supplying arguments.

- *State/Strategy:* Function Object, State [5], and Strategy [5] are concerned with encapsulating behavior. A decision between them can be based on concerns such as:

  * Who is responsible for changing the variable part of an algorithm?

    The State pattern manages the change of variability autonomously. Function objects are explicitly chosen by the client. Strategies are chosen by the client also, but independently of operation requests.

  * Is it feasible to impose the same interface on all variations?

    If the available Strategies range from simple to complex, the abstract Strategy must support the maximum parameter interface [5]. Function Object avoids this by partial parameterization.

  * Does the combination of common and variable part constitute a useful concept?

    The State pattern conceptually represents a monolithic finite state machine, so the combination of standard- and state-dependent behavior makes sense indeed. Strategies are a *permanent* part of general behavior and thus provide default behavior. Here, the combination acts as a built-in bookkeeping for the selection of the variable part. Function Objects take part in the "takes-a" relation. A function object and its receiver are only *temporarily* combined in order to accomplish a task.

2.13.2 Collaboration

- *Iterator:* Function objects allow the use of data from inside (elements) and outside the collection (previous arguments). There is no collaboration between Command and Iterator, since Command does not take arguments.

- *Adapter:* Function Object extends the use of *Parameterized adapters* as described in the implementation section of the Adapter pattern [5] from Smalltalk to any object-oriented language.

- *Chain of Responsibility:* Pairs of test- (check responsibility) and action function objects can be put into a Chain of Responsibility in order to separate responsibility checks from the execution of tasks. Function Object allows to replace the inheritance relationship between Links and Handlers [5] with object-composition.

2.13.3 Implementation

- *Composite:* Standard and composed function objects can be uniformly accessed with the Com-

posite pattern [5]. A composite function forwards arguments to its component functions. A tuple-Composite applies all functions in parallel to the same argument and thus produces multiple results. Several reduce functions (e.g., `And` of section 2.3.2) may somehow fold all results into one output. A pipeline-Composite applies each function to the result of its predecessor and thus forms a calculation pipeline.

- *Prototype:* Often it is useful to distribute the accumulated state of a function object to different clients. For instance, a a command for deleting text can capture the information whether to ask for confirmation or not. However, when placed on a history list for undoing, different commands must maintain different pointers to the deleted text. Consequently, Prototype can be used to clone pre-configured function objects which should not share their state any further.

- *Chain of Responsibility:* Generic Function Object can employ a Chain of Responsibility for argument type discrimination. Chain members check whether they can handle the actual argument type. This enables a highly dynamic exchange of the dispatch strategy.

3 Conclusion

The Function Object (i.e., closure) concept is a basic design technique that solves many problems. This makes its description as a single pattern difficult. Nevertheless, we refrained from restricting ourselves to a traditional problem, context, and solution triple. We used just one aspect of Function Object for its motivation and supplied illustrating examples for other aspects individually.

While true closures bind their free variables implicitly, our object-oriented version requires explicit binding. Breuel shows how to still achieve implicit binding by nested class definitions [2]. We have seen, however, that explicit binding does not lose anything essential, but on the contrary decouples the function object from its creation context.

Naturally, Function Object shares many properties with Command and Strategy. They abstract from function pointers, support composition, allow to undo operations, and achieve client/server decoupling. In addition, function objects:

- accept arguments, which, e.g., enables them to be used for iteration. However, the complete number of parameters is effectively hidden to adaptable algorithms, which allows for transparent behavior extensions.

- allow partial parameterization. One function object definition actually introduces as many function objects as the number of its parameters. Run time partially parameterized function objects can be regarded as dynamically created functions.

- capture data from their creation environment and previous arguments. Hence, data providers can be separated from each other. A function object allows to combine **local** data from environments, even beyond their lifetime.

- may dispatch on argument values and/or types. Similar to the State pattern, input-discriminating switches can be distributed to individual generic function objects, rather then being nested at one place.

- can be composed sequentially as well as in parallel. Sequential composition establishes a calculation pipeline. Parallel composition calculates tuples of results, e.g., during a traversal.

- provide local state. Imperative function objects may hold state for algorithms or may accumulate results during iterations. Hence, function objects can compute, e.g., traversal results without the need to modify the result type of the traversal algorithm.

In summary, function objects hide the number of both parameters and results to clients. This can be viewed as an aid to modularization, just like classes in object-oriented design or higher-order functions and lazy evaluation in functional programming. Accordingly, aggregation ("has-a"), inheritance ("is-a"), and behavior parameterization ("takes-a") should be equally well-known to designers. "Takes-a" realizes object-composition, as opposed to breaking encapsulation with inheritance. It is therefore a means of reaching the goal of component oriented software. In combination, inheritance and Function Object allow for flexible prototyping as well as safe *black-box* composition.

As well as other patterns, Function Object can raise the level of design discussions. The term

Function Object should immediately communicate the concepts of environment capturing, partial parameterization, first-class methods, black-box composition, and so on. Also, the term generic Function Object is worth being adopted in a designer's vocabulary. We clearly pointed out the ability of generic Function Object to lift Iterator to the functionality of Visitor.

Another advantage of pattern-aided design is to work above the level of particular programming languages. Some patterns, including Function Object, even abstract from the implementation paradigm. Indeed, function objects reintroduce some flavor of structured analysis and design to object-orientation. This is definitely useful. While adding new objects to a system is caught by an object-oriented decomposition, adding functionality often is better handled by extending functional abstractions. The *control-objects* in Jacobson's "use-case driven approach" represent such points of functional extendibility. Concerning the optimal balance between free functions and object-oriented decomposition, further research is necessary. Anyway, withstanding the temptation to implement parameterization with inheritance but using function objects means introducing part of the functional paradigm into the object-oriented paradigm. No longer can we choose one technique from one paradigm only in order to solve a problem. We must carefully choose between paradigms first. While this may appear an extra complication to a novice, it is an essential enrichment to the expert.

It is well-known that one language's patterns are the other language's features. In general, patterns are thus useful as an indicator of insufficient programming language support.

In our point of view, Function Object belongs to the category of patterns that any implementation language should fully support. In contrast to more complex patterns (e.g., Mediator [5] or Bureaucracy [13]), which are clearly well above implementation level, Function Object works at a basic design level though still supporting higher level goals.

4 Acknowledgments

I would like to thank Erich Gamma, Robert Hirschfeld, Ralph Johnson, Jiarong Li, Reinhard Müller, Dirk Riehle, and Douglas C. Schmidt for their comments.

References

[1] Kent Beck. Method object. *Patterns mailing list Digest*, 96(26), April 1996.

[2] Thomas M. Breuel. Lexical closures for C++. In *C++ Conf. Proc.*, pages 293–304, October 1988.

[3] Derek Coleman and et al. *Object-oriented development: The Fusion Method*. Prentice Hall, 1994.

[4] James O. Coplien. *Advanced C++: Programming Styles and Idioms*. Addison-Wesley, 1992.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Object-Oriented Software Architecture*. Addison-Wesley, 1994.

[6] Aaron Hillegass. The design of the eiffel booch components. *Eiffel Outlook*, 3(3):20–21, December 1993.

[7] Robert Hirschfeld. Convenience methods. In *The 1^{st} Annual European Conference on Pattern Languages of Programming, EuroPLoP '96*, Kloster Irsee, Germany, July 1996.

[8] Thomas Kühne. Higher order objects in pure object-oriented languages. *ACM SIGPLAN Notices*, 29(7):15–20, July 1994.

[9] Thomas Kühne. The translator pattern — external functionality with homomorphic mappings. In *The 23^{rd} TOOLS conference USA '97*, St. Barbara, California, July 1997.

[10] Konstantin Läufer. A framework for higher-order functions in C++. In *Proc. Conf. Object-Oriented Technologies (COOTS)*, Monterey, CA, June 1995.

[11] Robert C. Martin. Acyclic visitor. In Robert C. Martin, Dirk Riehle, and Frank Buschmann, editors, *Pattern Languages of Program Design 3*, Reading, Massachusetts, 1997. Addison-Wesley.

[12] B. Meyer. *Reusable Software*. Prentice Hall, 1994.

[13] Dirk Riehle. Bureaucracy — a composite pattern. In Robert C. Martin, Dirk Riehle, and Frank Buschmann, editors, *Pattern Languages of Program Design 3*, Reading, Massachusetts, 1997. Addison-Wesley.

[14] A. Stepanov and M. Lee. The standard template library. ISO Programming Language C++ Project. Doc. No. X3J16/94-0095, WG21/NO482, May 1994.

[15] Walter Zimmer. Relationships between design patterns. In James O. Coplien and Douglas C. Schmidt, editors, *Pattern Languages of Program Design*, pages 345–364. Addison-Wesley, 1994.