Appeared in Proceedings of EuroPLoP, Irsee, July 1996

Recipes to Reuse

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

Abstract

We propose to use objects as closures for behavior parameterization. In contrast to reuse by inheritance, they realize reuse by composition. Closures allow black-box behavior parameterization with encapsulated components, promote function reuse, allow calculations on demand, represent first-class behavior, i.e., feature protocol-, undo-, and persistence mechanisms, and can represent "business-transactions". In this paper, we present closures as the object-oriented design pattern Recipe. In contrast to the Command pattern, Recipe establishes a useful collaboration with iterators. We show in particular how to use generic recipes with iterators in order to allow multi-dispatching operations on heterogeneous data structures.

1 Introduction

Smalltalk programmers use closures every day. They are used to parameterize behavior. For instance, the action to be performed during an iteration on a collection's elements is passed to the iteration method as a closure.

This paper explains the concept of closures and makes it accessible to any object-oriented language by describing it as the design pattern [13] Recipe.

In Smalltalk, closures are called blocks [14]. A block allows to *inject* behavior into an object. When a block is received and subsequently executed by an object, we can think of it as executing a new method in the object. There are vital differences, though: First, in contrast to a constant method, we can even inject freshly created blocks. This allows us to exchange and extend object behavior at run time. Second, encapsulation is fully preserved. So, blocks can use the public object interface only, but both object and block are protected against implementation changes.

Recent Smalltalk implementations implement blocks as *lexical* closures. What does this mean? A closure may refer to variables which are neither declared as parameters nor as local variables. These are called free variables. A lexical closure binds the values of these variables in its creation environment. What does this static binding buy us? It enables us to not only inject behavior into objects, but also data! Hence, the closure code can operate on two data spaces while these remain completely isolated from each other.

Up to now, we described closures with the same functionality as available in functional languages [12]. There is more to object-oriented closures: Imperative closures can produce side-effects and carry state. As a result, they can be used for parameterization of side-effects and to accumulate results internally.

The pattern description in section 2 will give applications for the remarkable properties listed above. While we use behavior refinement with inheritance as a negative pattern to motivate Recipes in section 2.3, parameterization is clearly not the only problem solved by Recipes. We illustrate further applicabilities and consequences with examples of their own. Section 3 summarizes Recipes as a powerful abstraction for software reuse.

2 Design Pattern: Recipe

2.1 Intent

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

2.2 Also Known As

Lexical Closure [5], Functor [7], Agent [16], Agent-Object [23], Functionoid [6], Functoid [25], Function-Object [37, 24].

2.3 Motivation

Behavior parameterization occurs in almost every program. Iterators are a good example. Consider a collection of books. We might be interested in

- whether a given title is in the collection,
- a list of all books written by one author,
- the date of the oldest book, etc.

All these operations need to traverse the collection structure. In fact, all operations can use the same traversal algorithm. The operations differ, however, in the test they apply to books and the sort of result they produce. Thus we decompose the operations into a common traversal algorithm and a set of functions that perform tests and produce results.

Concerning the traversal algorithm, we may provide an abstraction called *external* iterator. It will yield the elements of the collection one by one so we have to write a loop in order to access all elements. Yet, we expect to use the traversal algorithm many times, which will result in many explicitly written loops. Moreover, writing the loops is error-prone, since it is easy to use an incorrect exit condition or to forget to step to the next element (see also "Write a Loop Once" [28] and the discussion in the Iterator pattern [13]).

Consequently, we provide one more level of abstraction with an *internal* iterator. Given a function, it will apply the function to all elements in the collection.

Now, how do we compose an internal iterator and a function to be applied? We also want the books to be sortable according to various criteria, like author, title, and date and we want to choose the criterion at run time. In order to avoid switch statements we may use dynamic binding of iterators. The actual iteration method of an `ITERATOR` class is a Template Method [13], which depends on an abstract function method. The implementation for the abstract function method, and thus the specific function to be applied to the elements, is given in descendants of `ITERATOR` [27, 30, 28]. Composing traversal algorithm and functions then works through dynamic binding of the abstract function method. Selecting one of several functions corresponds to selecting one of the existing `ITERATOR` subclasses.

In this case, however, the application of an object-oriented design, using inheritance and dynamic binding, has some severe drawbacks:

- *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 [29], 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 uniformly send a message to a variable of type abstract-`ITERATOR` and 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 namespace. Implementation changes to either of the classes can easily invalidate the other. An interface between super- and subclasses, as the private parts in C^{++} [11], 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. An iteration used to count 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. Pre- and postconditions [31] can help to enforce behavioral identity between iteration schemes, but problems like the above are hard to cover and checking contracts at run time boils down to testing, as opposed to rigorous proofing.

- *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.

- *Obligatory Source-code.* Some languages enforce iteration classes to be available as source code. Otherwise it is not possible to derive subclasses. Hence, vendors do not have the option to sell pre-compiled library code only.

We can get rid of all these disadvantages if we sacrifice the ability to adapt iteration schemes for particular functions. We accomplish this by using higher-order functions, well-known from functional programming [4] and as blocks from Smalltalk. Instead of subclassing the iterator, we pass a function to be applied to the elements. We pass a Recipe that describes what to do with the elements. Instead of an "is-a", we establish a "uses-a" relationship. Actually, we call this special usage of a parameter for behavior parameterization "takes-a" [24]. Since we usually can not pass methods as functions and many object-oriented languages (e.g., C^{++}, Eiffel) do not feature blocks, we pass an object representing the function [5, 7, 24]. The object's interface makes it possible to receive arguments and to return a result. As the object represents a way of doing things we call it Recipe. Technically Recipes are closures, i.e., they can store any parameters or variables from their creation environment in order to use them even beyond the existence of environments.

Why is it useful to pass closures instead of plain functions, i.e., what is the use of being able to remember the value of free variables? Remember that we also want to sort books according to titles. We certainly need a function that compares the titles of two books. Now we can partially apply this function to one book we look for. We thus create a Recipe that only takes one more book and produces a certain result, depending on whether or not the titles match. Given this function and an iterator, we can realize the first library operation mentioned at the beginning of this section.

This is quite pleasing, since we reused a sorting predicate for a totally different purpose. The iteration does not have to worry about the number of parameters of recipes. For instance, in order to collect a list of books whose dates are in a given interval, we can use a function with three parameters. The first two specify the interval, the third is a book, and the function checks whether the book's date is in the interval. When the interval data is supplied in advance, we can use the resulting recipe for a standard library iteration.

2.4 Applicability

Recipes are free functions. They are not members of a particular data abstraction. They resemble so-called *design-objects* [30], like iterator objects, event handler, and Commands, which similarly constitute entities of their own right. The separation of functions from data can be beneficial in several ways:

- *Parameterization.* Recipes are a good candidate whenever general behavior can be adapted to special behavior. Use Recipe if one of the following aspects is desirable:

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

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

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

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

- *Uniform invocation.* Imposing a Recipe's interface on related operations makes it possible to uniformly invoke them. Instead of switching to different method names (e.g., editor commands), we invoke `evaluate` on an abstract Recipe and rely on dynamic binding [31, 13]. Consequently, when we add new operations, we do not need to change the event handler. If specific evaluation names (e.g., execute, evaluate, solve) are considered important, then they can be provided as aliases.

- *First-Class methods.* Recipes make methods amenable to persistent command logging, command histories for undoing, network distribution of commands, etc. Like Commands, Recipes can provide an `undo` method, which will use information in the Recipe's state to undo operations [31, 13]. A perfect candidate for first-class methods are so-called "business transactions" [7]. 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"* [7]. When transactions do not naturally fit into existing data abstractions, Recipes can lift them to first-class status while providing a uniform interface.

- *Monolithic Algorithms.* When a data structure is stable, but the operations on it often change, it is not a good idea to use the standard object-oriented method to distribute the operations over the object types involved. For instance, each change or addition of an algorithm on abstract syntax trees (such as `typeCheck`, `compile`) demands a change in all node-object types. In addition, it is not possible to exchange the operations at run time.

  If we turn the algorithm into a Recipe, we must use a generic Recipe (see subsection *Multi-dispatch*) to dispatch on the node types, but in analogy to the Strategy pattern [13],

  - the algorithm logic is localized,
  - Recipe's state can accumulate results, and
  - we can dynamically choose an algorithm.

- *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 Recipes 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 Recipe. The main transformation is to replace the temporary variables with Recipe 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 Recipe attributes. The main computation method (e.g., `evaluate`) simply puts the pieces together, itself being as clear as documentation [3].

- *Call-by-need Semantics.* One aspect of call-by-need is to calculate a result only once, no matter how many times the calculation is requested. Recipes can do this, but class methods can also perform this using a technique called memoization. Class methods, however, can not realize the second aspect of call-by-need which is to postpone a calculation until the result is actually needed. If the result is never needed, this pays off in run time efficiency. A Recipe represents a calculation that is performed only if someone is in need of the result.

  Note that lazy evaluation enables infinite data structures and supports modularization by decoupling data generation from data consumption [17].

- *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 [18], where only the type of the receiver objects is taken into account, 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. As a result, the classes involved are unnecessarily coupled.

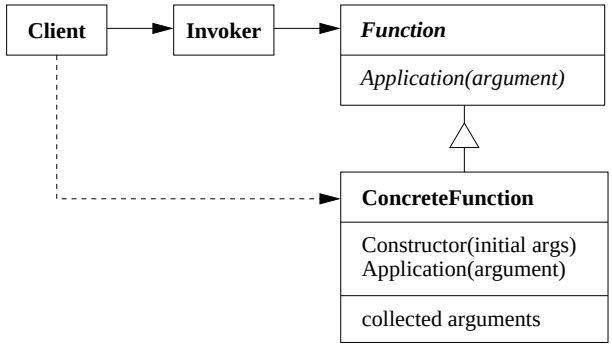
  A generic[1] Recipe removes the dispatching code from the argument types and concentrates it in one place. It uses run time type identification to select the correct code for a given combination of argument types. As such, it is not simply an *overloaded* Recipe, which would statically resolve the types.

  Unfortunately, the necessary switch statements on argument types are sensitive to the introduction of new types[2]. Yet, in the case of single-dispatch simulation, new dispatching methods (e.g., `add_complex`) are necessary as well.

  Generic Recipes may use coercions to reduce the possible number of combinations [7], and employ partial parameterization to avoid nested type switches: Upon receipt of an argument, the generic Recipe uses one type switch statement to create a corresponding new generic Recipe that will handle the rest of the arguments.

[1] Named after CLOS' [10] generic functions.

2.5 Structure



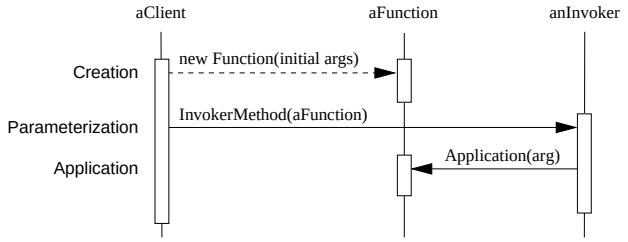
2.6 Participants

- **Recipe**
  - declares an interface for application.
- **ConcreteRecipe** (e.g., "printBook")
  - implements a procedure or function.
  - carries state for free variables and results.
- **Client** (e.g., Application)
  - creates a ConcreteRecipe.
  - possibly applies it to arguments.
  - calls an invoker method, passing ConcreteRecipe.
- **Invoker** (e.g., Iterator)
  - applies a ConcreteRecipe to more arguments or simply evaluates it.

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

2.7 Collaborations

- A client creates a Recipe. Free variables are bound in the client's environment.
- An invoker takes the Recipe as a parameter.
- The invoker applies the Recipe to arguments.
- The invoker gets a result from the Recipe and both invoker and client may optionally request the Recipe for further (accumulated) results.



2.8 Consequences

- *Abstraction.* Recipes abstract from function pointers and in particular from pointers to class methods. Instead of the C++code: `aFilter.*(aFilter.current)(t)`, we can write `aFilter(t)` [7].
- *Simplicity.* Recipes do not introduce inheritance relationships and do 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.* As Macro-Commands [13], Recipes can be dynamically composed to form a sequence of actions. Unlike Macro-Commands, Recipes may form a calculation pipeline by forwarding intermediate results to the next processing Recipe. A composite Recipe can also apply several component Recipes in parallel, producing multiple results at once.
- *Encapsulation.* As Recipes establish client relationships only, they are protected from implementation changes to algorithms that use them. Likewise, the implementation of Recipes can change without invalidating the algorithms. Hence, Recipes allow so-called *black-box reuse* [22] and help to advance reuse by inheritance to reuse by composition [21].
- *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 section 2.3).
- *Flexibility.*
  - A statement like `iterator.map(c)` 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 Recipe `c` 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 Recipe that will discriminate between data structure variations.
- *Separation.* Partial parameterization allows to separate concerns between arguments.
  - The ability to carry data (e.g., alert box size) enables Recipes to operate on data from outside an (error-reporter) algorithm as well as (error text) data from inside the algorithm. The data from outside the algorithm can be local to the algorithm caller. There is no need to communicate via global data. Since Recipes can combine local data spaces, they allow decoupling data spaces from each other while still supporting communication.

  - Generic Recipes separate type dispatching code from data. Even different aspects of dispatching can be separated. For instance, a state transition function, applied to a state, will yield a function that maps inputs to new states (the principle of pattern State [13]). Type and/or value of a state can be used to decide between resulting mappings.

- *Reuse.* Recipe's impact on reuse is twofold:

  First, adaptable algorithms become more reusable because they do not need to know about additional parameters for Recipes.

  Second, Recipes are multi-purpose:

  - Recipes are not bound to a particular adaptable algorithm, e.g., comparing book-titles is useful for sorting and for membership testing in collections.
  - Since Recipes are so easy to compose, it is feasible to form useful composite Recipes. For instance, we may combine a test- with an action-Recipe, in order to perform conditional actions on elements of a collection. Then, we do not necessarily need to provide special iterations as "`do_if`". [30].
  - One Recipe with n parameters actually represents n Recipes and one value. The first Recipe has n parameters. The second, created by applying the first to an argument, has $n-1$ parameters, and so on, until the last Recipe 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 [26]:

    $$\begin{aligned} &\mathit{Grav\_law}\ m_1\ r\ m_2 &&= \tfrac{G\ m_1\ m_2}{r^2} \\ &\mathit{force}_{earth} &&= \mathit{Grav\_law}\ \mathit{mass}_{earth} \\ &\mathit{force}_{surface} &&= \mathit{force}_{earth}\ \mathit{radius}_{earth} \\ &\mathit{force}_{my} &&= \mathit{force}_{surface}\ \mathit{mass}_{my} \end{aligned}$$

- *Iteration.* Recipes suggest the use of internal, rather than external, iterators. Internal iterators avoid explicit state and avoid 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 [13]. However, we simply propose to extend an iterator to accept not just one, but n data structures. A *transfold*[3]-method may access the first, second, etc., elements of all data structures simultaneously. When used to compare data structures, it can stop the iteration as soon as a mismatch has been found.

  Since Recipes do not rely on inheritance, it is possible to make iteration a method of data structures. 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 Recipe may calculate partial results from arguments and pass these to a result Recipe. Hence, the partial result is computed only once, no matter how many times the resulting Recipe will be applied to different arguments in the future, e.g., $\mathit{force}_{surface} = \mathit{force}_{earth}\ \mathit{costly\_calc}$, and then $\mathit{force}_{my} = \mathit{force}_{surface}\ \mathit{mass}_{my}$; $\mathit{force}_{your} = \mathit{force}_{surface}\ \mathit{mass}_{your}$.
  - Passing client parameters to Recipes 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 [35].
  - An overhead exists in creating and calling a Recipe, instead of just invoking a method. This suggests only to use Recipes when truly needed.
  - Care should be taken not to unnecessarily keep references to unevaluated calculations, i.e., Recipes. Otherwise, the occupied space can not be reclaimed.
  - Finally, Recipes 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.

[3] Its functional definition shall be: $\mathit{transfold}\ f\ a\ g \equiv (\mathit{foldr}\ f\ a) \circ (\mathit{map}\ g) \circ \mathit{trans}$

2.9 Implementation

- C++allows to overload the `"()"` operator, which gives a nice syntax for Recipe application [7]. Eiffel offers the infix operator `"@"`.

- In order to provide true static binding, Recipes must copy their arguments. Otherwise, their behavior will depend on side-effects on their arguments. In some cases, however, this may be desirable. The Recipe then plays the role of a future variable, which is passed to a client before all data needed to compute the result is available. Long after the Recipe has been passed to the client, it can be supplied with the necessary arguments by producing side-effects on bound values.

- How shall the initially free variables of a Recipe be bound? When closures are emulated with objects, the free variables can not be bound implicitly at the place of creation as usual. One way out is to forbid free variables and demand that they be parameters. This results in a uniform treatment of parameters and free variables. Alternatively, one can tie the binding of the free variables to the creation of the Recipe (e.g., C++constructor or Eiffel `creation` method). This saves implementing the intermediate classes, which may result from partial application of free variables. Of course, both variants can coexist.

  Note that explicit binding is equivalent to implicit binding. In fact, explicit binding does not force the Recipe to use the same names for free variables, as the environment prescribes. Ergo, a Recipe can be used in various environments without the need to make the free variables names match the environment.

- In addition to standard application, Recipes 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 $Grav\_law\ m_1\ m_2\ r = \frac{G\,m_1\,m_2}{r^2}$, (note the different order of parameters) we can not define: $force_{surface} = Grav\_law\ mass_{earth}\ radius_{earth}$.
  With keyword parameters we can write:
  `f:=Grav_law.m1(e-mass).r(e-radius);`

- Commonly a function is evaluated after it has received its last argument. Yet, Recipe application and evaluation can be separated by corresponding methods `apply` and `evaluate`. As a result, the client, invoking `evaluate`, does not have to know about the last argument. In addition, 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 [38]).

- Recipes do not break the encapsulation of the objects they are passed to. If a particular Recipe is closely coupled to a specific object type, then declaring the Recipe as the object's friend, i.e., using selective export, allows efficient access to internal data nevertheless. As a result the object can keep its public interface to other clients to a minimum.

- Two extremes to implement partial parameterization exist. The less verbose is to always keep the same instance of Recipe and assign incoming arguments to corresponding internal attributes. This will cause trouble when the same Recipe is used by several clients. As the application of a Recipe 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 Recipe produces a new instance of a different type, then static typing is enabled again and no unwanted sharing of Recipe state can occur. Unfortunately, this forces us to write at least $n-1$ classes for a Recipe with n parameters.

- Generic Recipes must use run time type identification to deduce the type of actual arguments. There are many ways to account for this: `type-case` statement (Sather [33]), `typeid` (C++, 3.0 specification), reverse assignment attempt (Eiffel [31]).

- How shall we accumulate the results of an iteration? A functional programmer would implement all of the three library operations (see section 2.3) as a *fold* over a list of books.

The result type of *fold* is determined by the function supplied. It will return whatever the function returns (e.g., Boolean, list of Books, Date of book).

Imperative Recipes, can alternatively accumulate the result in their local state. Thus, we can replace functional *folds* with imperative *maps* (i.e., standard iterators) and the results will be available by request to the Recipe. Composite Recipes especially are suited to calculate more than one result during a traversal.

If possible, however, the functional version is preferred since it works without side-effects. Furthermore, non-strict operators such as "`or`" allow to exit a *fold* if the result can be determined without looking at the whole structure.

- An abstract Recipe class must support the type interfaces of all descendants. Eiffel may covariantly redefine argument application, but C^{++}demands the abstract class to be a template. Otherwise, all parameters of a Recipe must have the same type. Läufer proposes a solution using the Bridge pattern [13], which also covers memory management in C^{++} [25].

- Languages without garbage collection are less suited to support *upward-funargs*[4] [1] and delayed calculations. Constructors and Destructors only work for Recipes with lifetimes determined by environment lifetimes.

2.10 Known Uses

Apart from the uncountable uses of Recipes in functional programming and e.g., Scheme [1], there are many truly object-oriented uses: Smalltalk [14] features blocks as Recipes with implicit binding of free variables. Sather provides Recipes as *bound routines* [33]. The Eiffel Booch Components use Recipes for searching, sorting, transforming and filtering of containers [16]. The Standard Template Library, which was adopted as part of the standard C^{++}library, uses Recipes to inline operations for arithmetic, logic, and comparison [37].

[4]Functions returned as results. These do not have a determined lifetime and must remember their environment.

2.11 Related Patterns

2.11.1 Categorization

- *Objectifier:* A Recipe, like Objectifier, does not represent a concrete object from the real world [39], though one can reasonably take business-transactions for real. Recipe 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" Recipe. The latter is a *uses*, not a *has-a* relationship.

- *Command:* A Recipe which does not take any arguments after creation and produces side-effects only boils down to the Command pattern [13]. One key aspect of Command is to decouple an invoker from a target object. Recipes typically do not delegate functionality. Rather then delegating behavior to server objects they implement it themselves. So, Recipes normally do not work with side-effects, but return their computation as an argument-application result. Nevertheless, Recipes 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 Recipes by supplying arguments.

- *Visitor:* Data structures need to know about Visitors because they have to provide an `Accept` method [13]. Sometimes this is undesirable because of the so-introduced mutual dependency between data structures and Visitors. When the data structure is not available as source code, it is even impossible to add the `Accept` method. A combination of Iterator and generic Recipe avoids these drawbacks, while providing the same functionality as Visitor:

  - It frees the data structures from needing to provide the operations themselves.
  - It differentiates between types in the data structure. The generic Recipe chooses the appropriate code for each combination of operation and element type.
  - It allows heterogeneous interfaces on the data elements. Once the generic Recipe has done the dispatch, the exact element

type is known and access to the full interface is possible.

- It concentrates operations at one place and provides a local state for them. Between invocations, Recipes can hold intermediate results, e.g., variable environments for a type-checking algorithm on abstract syntax nodes.

A generic Recipe can be realized as a type dispatcher, parameterized with a set of Recipes that actually perform an operation. This allows reuse of the dispatching part for various operations. Of course, the type switches are sensitive to the addition of new structure objects. However, if the structure is unstable, Visitor is not recommended either [13].

- *State/Strategy:* Recipe, State [13], and Strategy [13] 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. Recipes are explicitly chosen by the client. Strategies are chosen by the client, 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 [13]. Recipes avoid 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. Recipes take part in the "takes-a" relation. A Recipe and its receiver are only *temporarily* combined in order to accomplish a task.

2.11.2 Collaboration

- *Iterator:* Recipes work well in collaboration with Iterator [13], since they allow the use of data from inside (elements) and outside the collection (previous Recipe arguments). Gamma et al. do not identify a collaboration between Command and Iterator, which is due to the fact that Command does not take arguments.

- *Chain of Responsibility:* Pairs of test- (check responsibility) and action recipes can be put into a Chain of Responsibility in order to separate the determination of responsibilities from the execution of tasks. Recipe allows to replace the inheritance relationship between Links and Handlers [13] with object-composition.

2.11.3 Implementation

- *Composite:* One way to uniformly access standard and composed Recipes is to use the Composite pattern [13]. Composite Recipes forward argument-application to its component Recipes. A tuple-Composite applies all Recipes in parallel to the same argument and thus represents a multi-result tuple. A pipeline-Composite applies each Recipe to the result of its predecessor and thus forms a calculation pipeline.

- *Prototype:* Often it is useful to distribute the accumulated state of a Recipe to different clients. For instance, a Recipe used as 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 Recipes must maintain different pointers to the deleted text. Consequently, Prototype can be used to clone pre-configured Recipes which should not share their state any further.

- *Chain of Responsibility:* Instead of using a fixed switch statement for argument types, a generic Recipe can employ a Chain of Responsibility. Each member of the chain checks whether it can handle the actual argument type. This enables a highly dynamic and flexible exchange of the dispatch strategy.

3 Conclusion

The closure concept is a basic design technique that solves many problems. This makes its description as a single pattern difficult. Nevertheless, we choose to present all applicabilities and problems that can be solved, rather than restricting ourselves to a traditional problem, context, and solution triple. We used just one aspect of Recipes for their motivation and supplied illustrating examples for other aspects individually.

The power of closures makes it an ideal candidate for a language construct. Actually, it is one of only four abstractions supported by Smalltalk [2]. Likewise, Beta's [27] patterns are often used as genuine closures. Functional closures, i.e., higher-order functions, even allow to base an entire paradigm on them. Functional programming heavily relies on general list processing functions, which can be specialized for various purposes. Here, behavior parameterization ("takes-a") is a fundamental technique for software reuse. Functional programming can even outperform object-oriented programming in terms of reuse [15]. This particular result can partly be attributed to the heavy use of library functions for intermediate list data structures. This suggests that object-oriented programs can also increase metric reuse factors by using internal iterators. Our proposed *transfold* operator combined with respective Recipes yields many useful operations like *map*, *filter*, *sum*, *length*.

As presented here, Recipes support both *downward-funargs* and *upward-funargs* [1]. While the former constitutes behavior parameterization, the latter occurs at partial applications and delayed calculations. With respect to behavior parameterization, Recipes and inheritance are not redundant concepts [24]. We clearly documented the added run time flexibility and decoupling properties of Recipes.

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 [5]. We have seen, however, that explicit binding does not lose anything essential, but on the contrary decouples the Recipe from its creating environment.

Naturally, Recipe 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, Recipes:

- accept arguments, which, e.g., enables them to be used for iteration. However, the complete number of arguments is effectively hidden to adaptable algorithms, which allows for transparent behavior extensions.
- allow partial parameterization. One Recipe definition actually introduces as many Recipes as the number of its arguments. Run time partially parameterized Recipes can be regarded as dynamically created functions [9].
- capture data from their creation environment and previous arguments. Hence, data providers can be separated from each other. A Recipe 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 Recipes, 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 Recipes may hold state for algorithms or may accumulate results during iterations. Hence, Recipes can compute, e.g., traversal results without the need to modify the result type of the traversal algorithm.

In summary, Recipes hide the number of both arguments and results to clients. This can be viewed as an aid to modularization, just like classes in object-oriented design [34] or higher-order functions and lazy evaluation in functional programming [17]. 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 [36]. It is therefore a means to reach the goal of component oriented software [20, 32]. In combination, inheritance and Recipe allow for flexible prototyping as well as safe *black-box* composition.

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

should immediately communicate the concepts of environment capturing, partial parameterization, first-class methods, black-box composition, and so on. Also, the term generic Recipe is worth being adopted in a designer's vocabulary. We clearly pointed out the ability of a generic Recipe 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 Recipe, even abstract from the implementation paradigm. Indeed, Recipes re-introduce 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 [19]. Concerning the optimal balance between free functions and object-oriented decomposition, further research is necessary [8]. In any case, withstanding the temptation to implement parameterization with inheritance, but using higher-order functions 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.

References

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, Cambridge, MA, London, 6. edition, 1987.

[2] Micheal Beaudouin-Lafon. *Object-oriented Languages*. Chapman and Hall, 1994.

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

[4] Richard Bird and Philip Wadler. *Introduction to Functional Programming*. C.A.R. Hoare Series. Prentice Hall International, 1988.

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

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

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

[8] James O. Coplien. Multi-paradigm design and implementation. Summer School on Object Orientation in Tampere, Finland, AT&T Bell Laboratories, Naperville, Illinois, USA, August 1995.

[9] Laurent Dami. *Software Composition: Towards an Integration of Functional and Object-Oriented Approaches*. PhD thesis, Uni. of Geneva, April 1994.

[10] Linda G. DeMichiel and Richard P. Gabriel. The common lisp object system: An overview. In J. Bezivin, J-M. Hullot, P. Cointe, and H. Lieberman, editors, *Proceedings ECOOP '87*, LNCS 276, pages 151–170. Springer Verlag, June 1987.

[11] M. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[12] Anthony J. Field and Peter G. Harrison. *Functional Programming*. Addison-Wesley, 1988.

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

[14] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, MA, 1983.

[15] R. Harrison, L. G. Samaraweera, M. R. Dobie, and P. H. Lewis. Comparing programming paradigms: An evaluation of functional and object-oriented progams. Technical Report SO171BJ, University of Southhampton, Dept. of Electronincs and Computer Science, UK, August 1994.

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

[17] John Hughes. Why functional programming matters. In David A. Turner, editor, *Research Topics in Functional Programming*, pages 17–42. Addison-Wesley, August 1987.

[18] Daniel H. H. Ingalls. A simple technique for handling multiple polymorphism. In *Proceedings OOPSLA '86*, pages 347–349, November 1986.

[19] Ivar Jacobson and et al. *Object-Oriented Software Engineering: A use case driven approach*. Addison-Wesley, 4. edition, 1994.

[20] Mehdi Jazayeri. Component programming – a fresh look at software components. In *Proceedings of the 5th European Software Engineering Conference, Sitges, Spain*, September 1995.

[21] Ralph E. Johnson. How to develop frameworks. In *ECOOP '94 Tutorial Documentation*, July 1994.

[22] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June 1988.

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

[24] Thomas Kühne. Parameterization versus inheritance. In C. Mingins and B. Meyer, editors, *Technology of Object-Oriented Languages and Systems: TOOLS 15*, pages 235–245, Prentice Hall, 1995.

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

[26] Gary T. Leavens. Fields in physics are like curried functions or Physics for functional programmers. Technical Report TR #94-06b, Department of Computer Science, Iowa State University, 229 Atanasoff Hall, May 1994.

[27] Ole L. Madsen, Kristen Nygaard, and Birger Möller-Pedersen. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley and ACM Press, 1993.

[28] Robert Martin. Discovering patterns in existing applications. In James O. Coplien and Douglas C. Schmidt, editors, *Pattern Languages of Program Design*, pages 365–393. Addison-Wesley, 1994.

[29] B. Meyer. *Eiffel the language*. Prentice Hall, 1992.

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

[31] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, NJ, 1988.

[32] Oscar Nierstrasz and Theo Dirk Meijler. Research directions in software composition. *ACM Computing Surveys*, 27(2):262–264, June 1995.

[33] Stephen M. Omohundro. The Sather 1.0 specification. Technical report, International Computer Science Institute, Berkeley, December 1994.

[34] D. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:1053–1058, 1972.

[35] Peter Sestoft. Replacing function parameters by global variables. Diku, University of Copenhagen, October 1988.

[36] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. *Proceedings OOPSLA '86*, 21(11):38–45, November 1986.

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

[38] Ake Wikström. *Functional Programming Using Standard ML*. International Series in Computer Science. Prentice Hall, 1987.

[39] 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.