The Lambda Pattern

Dorin Sandu, Dwight Deugo
School of Computer Science, Carleton University
1125 Colonel By Drive, Ottawa, Canada K1S 5B6
{sandu, deugo}@scs.carleton.ca

Abstract The need often arises to write simple, possibly one-shot behaviors that do not seem to belong in the interface of any class. The behavior can be assigned a name and encapsulated in a method in a specific class, but the given name would convey little information beyond what can be inferred by directly looking at the code. In such cases it is better to express the behavior as lambda functions, unnamed behaviors expressed as blocks in Smalltalk and inner classes in Java.

Problem **How do you express simple, possibly one-shot behaviors that do not seem to belong in the interface of any class?**

Context Although the need for simple, one-shot behaviors can arise in other types of programming (such as functional and imperative procedural programming) this pattern applies to object-oriented programming and is particularly useful for use with languages like Smalltalk and Java. Use the lambda pattern under one or more of the following circumstances:

- *You need to express some small, self-evident behavior.* You could try to assign the behavior an Intention Revealing Selector [3] name, and encapsulate it in a method in a specific class, but the name would convey little additional information beyond what can be inferred by directly inspecting the code.
- *You need to save some context for later execution*. Behaviors operate on some local context. This context along with the behavior that operate on it have to be saved and evaluated a later time. This can happen under the following conditions:
  - — *You need to encapsulate how certain objects interact*. The way objects interact changes frequently so the objects have to be expressed such that, in order to minimize coupling, they do not reference each other explicitly.

— *You need to configure an object for later access*. The configuration is a one-shot deal and there is little chance that the code will be reused. The configuration code does not really belong in the interface of the caller or callee, but may require full access to the state and behavior of the former.

Additionally, the following may apply:

- *You need to adapt an interface*. The client object cannot collaborate with the object that provides the required behavior due to an interface mismatch. The caller object cannot be modified to the interface of the callee because it has been designed as a reusable component that expects a well-defined interface from the objects it interacts with. In other words, you are in a position to make use of the Adapter [1] pattern and the adapter behavior is not worth its own class or method.
- *You need to implement a series of algorithms.* The algorithms can be written in a couple of lines and have identical interfaces so they can be used interchangeably. The algorithms do not require access beyond their local context except for the input parameters. In other words, you are in a position to make use of the Strategy [1] pattern and the strategy behavior is not worth its own class or method.

Forces

- *Objects should be as simple as possible, but not simpler.* An object is defined as a collection of related behaviors working on the same data. This promotes modularity and information hiding, qualities that tend to dissipate when the object has either too much functionality, too much state, or too much of both.
- *Classes should have an optimum number of methods.* Behaviors in the public interface are usually implemented as Composed Methods [3]. They are coded in terms of operations at the same level of abstraction, which can be found in the protected/private interface of the same class. Splitting complex methods this way improves the readability of the code but, at the same time, increases its complexity.

  A possible solution to express a simple, one-shot behavior, would be to encapsulate the behavior in a private method in some class. If many such behaviors have to be written, the number of private helper methods will outnumber the rest of the methods, and therefore increase the complexity of the class.
- *Systems should have an optimum number of classes.* Classes in a system represent either concepts in the problem domain, or are used as helpers to glue the problem domain classes to platform-specific frameworks. The number of helper classes has to be minimized in order to increase the communicability of the system.

Another possible solution to express a simple, one-shot behavior, would be to create a class to store the behavior. This, however, introduces new helper classes, and therefore increases the complexity of the system.

- *Classes, methods, and variables should have meaningful names*. Good names provide insight into the purpose and design of a system, they reveal its inner workings, and communicate themes and variations of the present abstractions. Class names should convey the purpose of the class in the system, while its subclasses names should convey the difference from the base class (see Simple Superclass Name and Qualified Subclass Name [3]). Methods should be given Intention Revealing Selectors [3], according to which the name describes what the method is trying to accomplish. Similarly, variables should be given Role Suggesting Instance Variable Names or Role Suggesting Temporary Variable Names [3] such that the name reflects the role the variable plays in the computation.
- *Abstractions are sometimes self-evident in certain contexts*. For many abstractions, any one name can only hint at how the abstraction works or how it might be useful. Full understanding comes when the abstraction is used in a specific context. In some cases, there is little benefit to name the abstraction; its meaning is self-evident from the context in which is used.
- *Behaviors need access to local and context state*. Behaviors are defined as methods in some class. They have access to local state stored in temporary variables, and to context state stored in the instance/class variables of the class and its superclasses. Small, self-evident behaviors also need this kind of access if they are to replace full-fledged methods.

Solution

Write the behaviors as lambda functions.

The notion of lambda functions comes from Lambda Calculus [4][5], a language developed in the 1930s by Alonzo Church to help with the formalization of programming languages and programming in general. Lambda functions are used to introduce abstractions into a system, where each abstraction is defined by some state (bound and/or free variables) and some behavior (sequences of expressions that may recursively include other lambda functions). These abstractions can be evaluated in either applicative order, where the occurrences of the variables in the function's body are replaced by the value of the argument expressions, or in normal order, where the variables are replaced by the unevaluated argument expressions. In the context of the application, the functions are considered first class objects, in other words functions can be passed as arguments to other functions, used for return values, or assigned to variables.

Lambda functions can be named by assigning them to a variable and using the variable as a placeholder for the entire function. The decision whether to

name the function or not stems from how the behavior is to be used. One names a function only when that function can be reused in other parts of the application or in future applications; when the behavior is used only once, it is not necessary to assign it a name. Since, by definition, lambda functions are nameless sequences of actions declared and used in the context where are needed, they are particularly suited to express simple one-shot behaviors.

Table 1: Lambda-Function equivalents in different programming languages.

Language	Lambda-Function equivalent
Smalltalk	blocks
Java	anonymous inner classes
Lisp/Scheme	lambda-functions
C/C++	function pointers

Although primarily used as a model for functional languages and functional programming, variants of lambda functions have been introduced in object-oriented languages, most notably Smalltalk, and recently Java, as shown in Table 1. Smalltalk supports such functions via blocks, a way to represent deferred sequences of actions. These actions are compiled by the compiler into executable objects stored in the body of the method where they are defined. Block objects have full access to the context of the method, can be assigned to variables, or can be passed as arguments to other methods. The code inside blocks is evaluated at a later time, when requested to do so by sending the messages *value*, *value:*, *value:value:*, or *valueWithArguments:*.

Java can simulate lambda functions through anonymous inner classes. As opposed to a top-level class which has to be defined in the context of a package, an inner class can be defined in the context of a class or a method. A further refinement of the inner class, the anonymous inner class, can only be defined in the context of an expression. Inner classes can be declared as normal classes, i.e., they can have a name and be instantiated many times in the context where they have been defined, but anonymous inner classes are only instantiated once, in the expression where they have been defined. Therefore, anonymous inner classes are either declared as right-hand sides in assignment expressions or as arguments in method calls, in much the same fashion as blocks are declared in Smalltalk.

Although both Smalltalk blocks and Java inner classes are first class objects, provide deferred code execution, and can be used as language specific lambda functions, they are not entirely similar. They differ in two aspects: interface signature granularity and return behavior. Smalltalk blocks provide the equivalent of a one method interface whereas a Java inner class allows the

definition of more than one method. A hardcoded return in a Smalltalk block causes execution to jump in the context of the method where the block is defined, whereas in Java, an explicit return in the method of an inner class resumes execution in the caller of that method.

Some languages offer a very restrictive implementation of lambda functions. In C and C++, lambda functions can be approximated by function pointers. However, the use of function pointers voids the benefits of static type checking done by the compiler, and still forces the user to represent simple one-shot behavior in a named function, removed from the context where it makes sense.

Example

One recurring problem in the design of good container classes is how to provide a way to sort the elements of the container in some specified order without revealing the underlying representation of the container, and without having to create subclasses specialized by the sorting algorithm. One common approach is to use a single container object that can delegate sorting behavior to a sort-policy object which is provided by the client. The main difference is how the sort-policy objects are implemented. While some libraries choose to implement them as distinct classes, some implement them using the lambda function equivalent of the chosen implementation language.

Lisp and Scheme, both functional languages using Lambda Calculus as their underlying model, use a lambda function to directly specify the sorting criteria for a list, as shown in the following example, which returns (1 2 3 4) as the sorted list:

```
(sort (lambda (a b) (< a b)) '(4 2 1 3))
```

In Smalltalk, the same result can be accomplished by using blocks:

```
#(4 2 1 3) asSortedCollection: [:a :b | a < b]
```

In Java, the same result can be achieved by using an anonymous inner class that implements the *Comparator* interface. The comparator interface provides protocol for comparing two objects and returns a negative integer, zero, or a positive integer if the first object is less than, equal to, or greater than the second. The anonymous inner class can then be passed to a sort method along with the collection that needs sorting:

```
Vector collection = new Vector();
collection.add(new Integer(4));
collection.add(new Integer(2));
collection.add(new Integer(1));
collection.add(new Integer(3));

Comparator comparator = new Comparator() {
   public int compare(Object source, Object target) {
      int sourceInt = ((Integer)source).intValue();
```

```
      int targetInt = ((Integer)target).intValue();
         return(sourceInt<targetInt ? -1
            : (sourceInt==targetInt ? 0 : 1));
   }}

Collections.sort(collection, comparator);
```

As you can see from the above example, a sorting criterion can be specified inline, as an argument to the sort method. The service provider is the container object which can be configured with a sort criterion by its clients. This configuration is done without extra classes or methods, in a way that keeps the design simple and understandable.

Resulting Context

- By expressing behavior with blocks in Smalltalk and anonymous inner classes in Java instead of creating helper methods and classes, the system as a whole becomes more understandable and maintainable. This is mainly because the system complexity is reduced by minimizing the number of classes and methods. However, lambdas should not be seen as a substitute for helper methods and classes; lambdas should only be used to express behavior that is self-evident in the context where it is needed.
- The use of the lambdas may not simplify the system, because code written in lambda function format can be difficult to read. Such code use should be limited to those functions that are very small (no more than a method or two) and whose use is well-understood. If the meaning of the code in a lambda function is not self-evident, then it maybe needs to be refactored, an opportunity to create either new classes or methods.

  For example, in the Java version of the sorted collection presented above, we can clean up the code by moving the comparison logic into the class of the argument objects of the *compare* method, in this case Integer:

```
...
// Source in client class.
Comparator comparator = new Comparator() {
   public int compare(Object source, Object target) {
      return ((Integer)source).compareTo((Integer)target);
   }}
Collections.sort(collection, comparator);
...
// Source in Integer class.
public int compareTo(Integer target) {
   int sourceInt = this.value;
   int targetInt = target.value;
   return(sourceInt<targetInt ? -1
      : (sourceInt==targetInt ? 0 : 1));
}
...
```

This, however, will not work if the source and target objects are of different classes. In this case, a different approach, such as double dispatching [3][1], must be used.

- When switching between lambda functions and classes, a Java application needs not be modified extensively because the interface of top-level, inner, or anonymous classes is the same to all clients. However, in Smalltalk, since blocks can only be evaluated using a fixed protocol (*value*, *value:*, *value:value:*, or *valueWithArguments:*), two possibilities exist:
  - — The block protocol can be added to the new class. The methods *value*, *value:*, *value:value:*, or *valueWithArguments:* are added to the new class to implement the desired behavior directly or to call the appropriate methods. This is better in the short run, while prototyping, because it avoids changing all the methods that used to evaluate the block.
  - — The methods that evaluate the blocks can be modified to use the protocol of the new class. This is better in the long run because the implementation of these methods can be more easily grasped, specially if the methods in the new class are given Intention Revealing Selector [3] names.
- In Smalltalk, blocks cannot inherit from other blocks whereas, in Java, inner classes can subclass any class in the current scope.
- In Java, all variables and parameters accessed from within an inner class must be declared final because of potential synchronization problems.

Rationale Lambda functions resolve the forces mentioned above as follows:

- *Objects should be as simple as possible, but not simpler.* Lambda functions are defined as first class objects, a collection of related behaviors working on some data. The encapsulated state and behavior promote modularity and information hiding, as does any other object in the system.
- *Classes should have an optimum number of methods.* By encapsulating state and behavior in lambda functions, the number of helper methods otherwise required to express the same functionality is reduced, and the complexity of the class that would have had to store the methods is kept to a minimum.
- *Systems should have an optimum number of classes.* By encapsulating state and behavior in lambda functions, the number of helper classes otherwise required to express the same functionality is reduced and the complexity of the system that would have had to store the classes is kept to a minimum.

- *Classes, methods, and variables should have meaningful names*. The behavior expressed with lambda functions is meant to be self-evident and this is accomplished by keeping the code short and assigning meaningful names to the classes, methods, and variables referenced within the lambda.
- *Abstractions are sometimes self-evident in certain contexts.* Lambda functions are nameless sequences of actions whose intent is better understood by looking at the encapsulated state and behavior directly, in the context where is needed, rather than at some name that describes it.
- *Behaviors need access to local and context state*. Lambda functions have full access to the behavior and state of the context where they are defined. For example, in Smalltalk, a block has access to a local context (block temporary variables), the method context in which is defined (method temporary variables, method arguments), the class context (instance variables, class variables, instance methods), and the global context (global variables).

Related Patterns

The following patterns from [1] can be implemented using the lambda pattern:

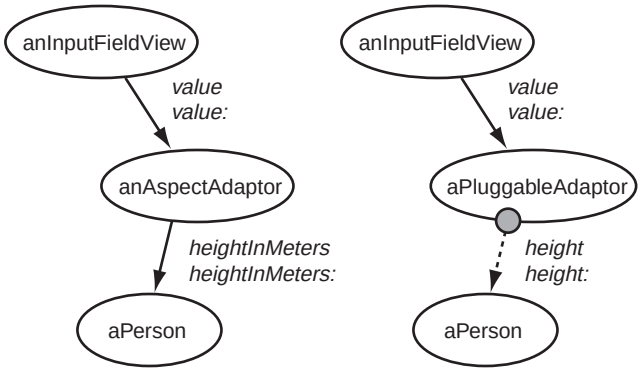
- **Adapter**. The *Adapter* pattern is used to convert the interface of a class to another interface that client objects expect. In this case, rather than implement an Adapter class, the interface adaptation can be done using a lambda function. In Java, interfaces with more than one method can be easily adapted using anonymous inner classes containing multiple methods. In Smalltalk, however, this can only be accomplished by using a block for each of the method to be adapted (for example, the *PluggableAdaptor* example in Section “Smalltalk Model View Controller” on page 9 provides blocks for getting, setting, and updating the model).
- **Bridge**. The *Bridge* pattern is used to decouple an abstraction from its implementation so the two can vary independently. The implementation of the abstraction, as described in [1], is provided in concrete classes but can also be provided as lambda functions.
- **State**. The *State* pattern permits an object to alter its behavior when its internal state changes in such a way that the object appears to change its class. This is accomplished by having the object keep track of a current state object that can be substituted with different other state objects. Rather than have a hierarchy of state classes, a single state object can be used. This state object can be configured with lambda functions that will execute on state transitions (see Section “Smalltalk HotDraw Tools” on page 10 for an example).

- **Strategy**. The *Strategy* pattern is used to define a series of interchangeable algorithms. These algorithms can be implemented as lambda functions rather than full classes. For example, the criterion for sorting collections from Section “Example” on page 5 has been represented as a lambda function rather than as a class.

Known Uses

Smalltalk Model View Controller

The Model View Controller framework in VisualWorks Smalltalk uses adapters (variations on the Adapter Pattern [1] which are presented in [2]) to provide a level of indirection between the view, controller and the model. Two of the most used adapters are the ProtocolAdaptor and PluggableAdaptor which convert the messages *value* and *value:* sent by the view/controller pair to the interface of the model. The AspectAdaptor, a concrete subclass of ProtocolAdaptor, translates *value* and *value:* into protocol understood by the model, whereas the PluggableAdaptor converts the messages into arbitrary actions defined by blocks.



To illustrate this concept, consider the issue of unit conversion. Suppose, due to internationalization constraints, that the height of a person object needs to be converted in the user interface from inches to meters. The person object provides the protocol *height* and *height:* to access and modify the height value in inches only. One possible solution is to implement the methods *heightInMeters* and *heightInMeters:* that perform the conversion and then call the *height* and *height:* methods. Then, the person object can be adapted to the input field via an instance of AspectAdaptor as follows:

```
anInputFieldView model: (
   AspectAdaptor
      accessWith: #heightInMeters
      assignWith: #heightInMeters:)
```

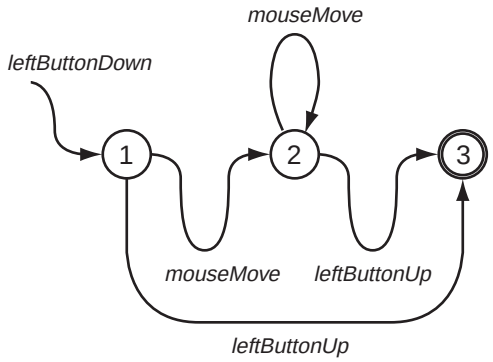
However, this solution complicates the interface of the person object by adding the methods necessary to do the conversion. Furthermore these methods do not belong in the application model, they are an artifact of the user inter-

face. A better solution would be to express *heightInMeters* and *heightInMeters:* as lambda functions using an instance of PluggableAdaptor. This way, the conversion itself is done in blocks, with the blocks being defined in the context of the class that creates and links the user interface to the domain model:

```
anInputFieldView model: (
   PluggableAdaptor new
      getBlock: [:model | model height * 0.3]
      putBlock: [:model :value | model height: (value / 0.3)]
      updateBlock: [:model :aspect :param | "do nothing"])
```

Smalltalk HotDraw Tools

HotDraw is a framework that helps with the construction of drawing editors. Such editors usually provide a tool palette and a drawing area where the user can manipulate graphic figures in different ways based on the currently selected tool. Internally, tools make use of a finite state machine in which transitions are made based on the current mouse event and the figure under the mouse cursor. Transition into a new state causes the action block associated with that state to be evaluated.



For example, consider the implementation of the selection tool. Using this tool, the user can select or unselect figures by clicking in the drawing area, or move the currently selected figures by dragging the mouse with the left button pressed. In order to implement the finite state machine for the tool, as shown in the figure above, the developer constructs the state machine out of ToolState and TransitionTable objects, and provides a command block for each state, as follows:

- **State 1:** Compute the figure under the cursor. Add this figure to the set of currently selected figures if SHIFT is pressed, else make this figure be the current selection set. Also remember the current mouse cursor location *lastPoint*.

```
Tool states
   at: 'Selection Tool Select'
   put: (ToolState
      name: 'Selection Tool Select'
      command: [:tool :event |
         | drawing lastPoint figure |
         drawing := tool drawing.
         lastPoint := tool cursorPointFor: event.
         tool valueAt: #lastPoint put: lastPoint.
         figure := drawing figureAt: lastPoint.
         tool sensor shiftDown
            ifTrue: [drawing toggleSelection: figure]
            ifFalse: [(drawing isSelected: figure)
               ifFalse: [drawing selection: figure]]]).
```

- **State 2:** Compute the current mouse location *newPoint*. Move the set of currently selected figures by the amount of *newPoint* - *lastPoint*. Make *lastPoint* be the value of *newPoint*.

```
Tool states
   at: 'Selection Tool Move Figure'
   put: (ToolState
      name: 'Selection Tool Move Figure'
      command: [:tool :event |
         | delta newPoint |
         newPoint := tool cursorPointFor: event.
         delta := newPoint - (tool valueAt: #lastPoint).
         tool valueAt: #lastPoint put: newPoint.
         tool drawing selections
            do: [:each | each translateBy: delta]]).
```

- **State 3:** Do nothing.

```
Tool states
   at: 'End State'
   put: (EndToolState
      name: 'End State' command: [:tool :event | ]).
```

As seen from above, the entire tool behavior is realized without extending any of the HotDraw code. Tools are constructed from already available objects, which are configured with one-shot behaviors in the form of block objects, which are provided by the developer inline, in the context of the application that makes use of the HotDraw framework.

Java Abstract Windowing Toolkit

Java uses anonymous inner classes in the Abstract Windowing Toolkit and the Swing user interface frameworks. These frameworks provide a clear separation between the application and the user interface via an intermediary layer that links the two. This layer consists of callbacks that the application must register with the framework in order to respond to events in the user interface.

This layer is implemented with inner classes in order to avoid having to subclass the user interface classes for every application and having to handle events in huge case statements.

Consider, for example, the following code which shows a portion from the implementation of a search dialog:

```
public class SearchDialog {
   private Frame frame;
   private Button searchButton;
   ...
   private addSearchButton() {
      searchButton = new Button("Search"));
      searchButton.setActionListener(
         new ActionListener() {
            public void actionPerformed(ActionEvent e) {
               search();
            }
         });
      frame.add(searchButton);
   }
   ...
   private void search() {...};
}
```

The method addSearchButton() is called during the initialization of a SearchDialog instance. This method creates the "Search" button to which it binds an action listener, which happens to be an instance of an anonymous inner class. Now, whenever the user presses the button, actionPerformed() will be called by the framework to execute the search() method.

References

[1] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[2] Sherman R.Alpert, Kyle Brown, Bobby Woolf. The Design Patterns Smalltalk Companion, Addison-Wesley, 1998

[3] Kent Beck. Smalltalk Best Practice Patterns, Prentice-Hall, 1998.

[4] Greg Michaelson. An Introduction to Functional Programming through Lambda Calculus, Addison-Wesley, 1988.

[5] Alonzo Church. The Calculi of Lambda Conversion, Princeton University Press, 1941.