Java Idioms: Code Blocks and Control Flow

Arno Haase
Arno Haase Consulting
Arno.Haase@Haase-Consulting.com
Arno.Haase@acm.org

1 Abstract

It is sometimes desirable to treat a piece of code as an object in order to parameterize an operation, thus abstracting and encapsulating control flow and making it reusable. This paper introduces two idioms that use anonymous local inheritance to implement this behavior.

It begins by explaining the concept of code blocks as objects that represent pieces of code and know the context of the method they are created in. Based on this concept, the idioms are presented: ANONYMOUS COMMAND using delegation to achieve flexibility and ANONYMOUS TEMPLATE METHOD employing inheritance.

The paper concludes with a practical discussion of how to use the idioms and when to choose which of them.

2 Code Blocks

At their core many design patterns are about introducing flexibility at a specific point in order to hide implementation details and provide a design that limits the scope of changes to the application. Typically they require the creation of new types to introduce this flexibility, one type for each concrete context in which the pattern is used.

Creating a full-grown type, however, introduces additional complexity and adds conscious effort to the implementation. So, by removing complexity and duplication, new complexity is added, and the advantages of the situation with the pattern must be weighed against those of the one without it.

For example, polymorphism can in principle be used instead of `switch` statements [Fowler99]. This can significantly reduce the complexity of the using code because it encapsulates the selection of behavior. However, this comes at a price: for every different case a type needs to be defined. The effort and complexity associated with the creation of types makes this approach inapplicable for sufficiently small scopes: obviously, most `if` statements are not candidates for replacement with polymorphism.

The point of equilibrium, the minimum complexity to make a pattern applicable, depends strongly on the effort and complexity associated with creating a new type. For many situations, it is sufficient to have an object that just contains a sequence of statements and allows them to be passed around and executed.

For example, code that uses JDBC to access a relational database must first acquire a database connection and a statement object, then access the database and finally release the resources in an exception safe way. The acquisition and release of the database resources is the same for most code accessing the database, so it can be factored out. But then the specific code operating on the database needs for example to be objectified and passed as a parameter to the abstracted algorithm.

The specific database operation is often only a single Java statement, for example the execution of an `UPDATE`. Still, a new type needs to be created, even if it contains only this single Java statement. Obviously in this case the overhead incurred by creating a new type is significant compared to the contents of the type.

So in order for patterns to be helpful in reducing complexity in a small context, it is necessary to have a way of creating objects that fulfils the following requirements:

- *local*. In the source code the type definition must be close to the code using it so that a human needs no context switches to understand the code.
- *aware of the surrounding scope*. If an object is created with the sole purpose of containing a sequence of statements, then there must be an easy way for these statements to communicate with the surrounding code, accessing local variables and being able to return data without much overhead.
- *easy to create*. There must not be much typing overhead associated with the definition of a type.

This kind of local, cheap objects containing just a sequence of statements is the concept of code blocks.

Code blocks are an abstract concept that is not fully supported in the Java programming language. Anonymous local classes, however, have many of their desirable properties.

- *local*. First of all, they are local in the sense that a type definition can be embedded in code so that a human reader can see it together with the using code without the need for a context switch.
- *aware of surrounding scope*. Anonymous local classes are also to some degree aware of the surrounding scope: code in an anonymous local class can access the `final` local variables of the containing method though the non-`final` variables are invisible to it.

The creation of anonymous local classes is, however, associated with almost as much overhead as the creation of named types, requiring a class body as well as full method declaration. Still, anonymous local classes are the best support that Java has for the concept, and therefore in this paper a method overridden in an anonymous local class will be called a code block.

3 The Idioms

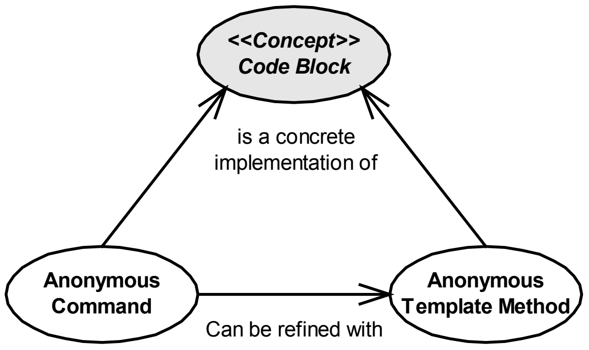
The object-oriented paradigm – and Java as one of its representatives – supports two fundamental ways of reusing code, delegation and inheritance. This paper introduces two idioms for abstracting control flow that utilize these ways of reuse: ANONYMOUS COMMAND using delegation and ANONYMOUS TEMPLATE METHOD using inheritance.

They both solve the same problem, but their different implementation make them applicable in slightly different situations which will be described below. This is reflected identical problem statements with different forces.

The following table summarizes the problem they solve and the solutions they offer.

Idiom	Problem	Solution
ANONYMOUS COMMAND	How do you make the common part of a sequence of statements reusable if it includes parts that are specific for the local context in which it is executed? The specific and the common code should be decoupled to allow the logging of operations or undo functionality.	Provide a class that implements an algorithm which delegates specific variant activities to command objects that are passed as parameters. To execute this algorithm in a local context, create anonymous local command classes for these variant aspects and pass them to the algorithm.
ANONYMOUS TEMPLATE METHOD	How do you make the common part of a sequence of statements reusable if it includes parts that are specific for the local context in which it is executed? The specific and the common code should be coupled to enable a refinement hierarchy of common code sequences where one uses and supplements the other.	Provide a superclass implementing the control flow and calling hook methods for the specific parts. To execute this control structure in a local context, derive an anonymous local class from the superclass, overriding some or all of the hooks.

The following diagram shows the idioms and their relationships.



After the idioms are described, a detailed example is used to discuss their applicability in practice.

4 ANONYMOUS COMMAND

Thumbnail

Problem	Solution
How do you make the common part of a sequence of statements reusable if it includes parts that are specific for the local context in which it is executed? The specific and the common code should be decoupled to the allow logging of operations or undo functionality.	Provide a class that implements an algorithm which delegates specific variant activities to command objects that are passed as parameters. To execute this algorithm in a local context, create anonymous local command classes for these variant aspects and pass them to the algorithm.

Example

The acquisition and release of resources before and after an operation is a common control flow. For a transaction, the result could look something like this simplified code fragment:

```
// Example of a sequence of operations that is executed
// around the specific code
try {
  Transaction transaction = new Transaction ();

  .... // perform operations in the transaction context

  // no exception occurred, so everything is OK
  transaction.commit ();
}
catch (Exception exc) {
  transaction.rollback ();
}
```

The code providing the context in which the specific operation must be executed is common to all usages of a transaction. It can be moved into a method of its own that takes the specific part as a parameter, making it reusable:

```
/** contains the sequence of statement common to all transactions*/
public class SimpleTransaction {
  /** is called to actually execute a transaction. The parameter
    * contains the code that is executed in the transaction.*/
  public void execute (TxOperation operation) {
    try {
      Transaction transaction = new Transaction ();
      // execute the code block in the transactional context
      operation.perform (transaction);

      transaction.commit ();
    }
    catch (Exception exc) {
      transaction.rollback ();
    }
  }
}

/** used as code block for SimpleTransaction */
public interface TxOperation {
  public void perform (Transaction tx) throws Exception;
}
```

Client code can then implement the `TxOperation` interface as an anonymous local class and pass the specific code to the transaction in this object.

```
// client code executing a transation using anonymous local
// inheritance
new SimpleTransaction ().execute (new TxOperation () {
  public void perform (Transaction tx) throws Exception {
    .... // the actual operations for the transaction go here
  }
});
```

The resulting code is more focused and less cluttered because the details of transaction handling are hidden. It is also makes it easier to avoid introducing errors or handling transactions inconsistently because all clients share the implementation of the actual transaction handling.

Problem

In software systems there tend to be common sequences of statements that are used in several places, like acquiring a resource, performing an operation on it and then releasing it again.

This sort of commonality makes it desirable to factor out and reuse the common part. Otherwise these common code sequences are difficult to change consistently and error prone to implement, even if the common parts are simple. Rather they all share the "bad smell" that is attached to code duplication in general.

In spite of this, parts of the sequence are often very specific for a local context, such as the actual operation on a resource or object, so that it is difficult to extract the common code into an abstraction of its own. For example, the context sensitive part of the code sequence often needs to be located close to its calling client so as to improve human readability. Moreover, the specific code often needs to communicate with its client, for example to access local client data or to throw exceptions in a specific to the particular client.

So how do you abstract a common sequence of statements in a reusable way so that using code can provide part of the sequence individually?

The following forces influence the situation:

- Since common sequences of statements frequently occur at a fairly small granularity a solution should require little typing overhead in order to be feasible for short common code parts.
- Often there is the same specific code in several places. This makes it desirable to reuse an implementation of specific code.
- The common part can belong to another component than the client, making it highly desirable to decouple the client from the common part.
- The common part may want to provide additional functionality that needs to keep track of the local parts, like keeping a history, performing logging or providing a cascaded undo operation.

Solution

Create a class containing the recurring control flow, the algorithm. Also provide an interface – the command – that captures the abstraction of the code block needed to parameterize the control flow.

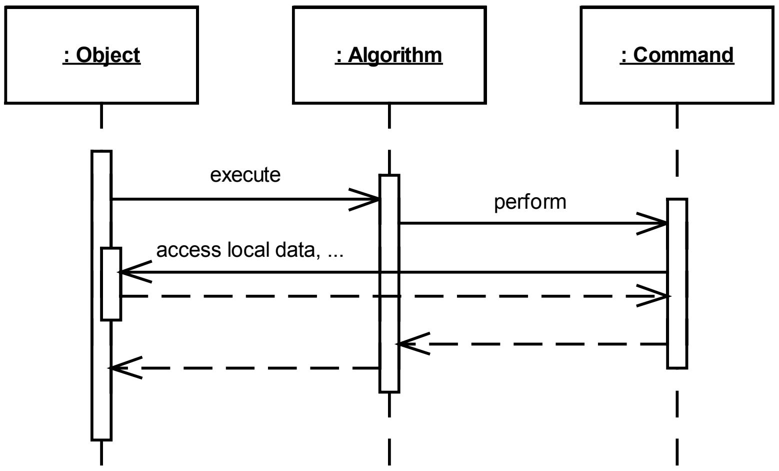
```
/** represents a common sequence of statements that needs to be
  * parameterized with an instance of Command in order to be
  * executed.*/
public class Algorithm {

  /** contains the actual common code. The parameter contains
    * the specific part of the code that needs to be supplied
    * by a client. */
  public void execute (final Command cmd) {
    .... // initialization that must always be performed goes here

    cmd.perform(); // call the code that was passed in

    .... // common code again
  }

  /** needs to be implemented by clients to provide the algorithm
    * with specific code. */
  public static interface Command {
    public void perform ();
  }
}
```

Then, in each context anonymously derive a local class from this interface and pass it to the object.

```
// provide the algorithm with specific code and execute it
new Algorithm().execute (new Algorithm.Command () {
  public void perform () {
    // this is where the specific code is provided
    ....
  }
});
```

This hides the details of the algorithm and provides a higher-level abstraction for the using code. It helps to focus on the intent of the code rather than the details of how that is achieved because the common algorithm is represented only by the class implementing it, providing an idiomatic context for the local code.

The following diagram shows the dynamics of the idiom, in particular explicitly displaying the code block's access to the surrounding method's state.



The algorithm's `execute` method executes the common code sequence, calling the command for the specific part. The command can then access local variables of the surrounding method to receive or return data. When the command is finished executing, the algorithm executes the second part of the common code.

The concept of encapsulating code in an object and passing it to another part of the system for execution is also at the core of the COMMAND pattern as described in [GHJV95]. The use of code blocks to implement the commands, however, makes ANONYMOUS COMMAND applicable for factoring out common sequences of statements.

Implementation

There are a lot of issues to consider when implementing ANONYMOUS COMMAND.

Passing data into the code block. The method signature of the code block reflects the structure of the underlying algorithm; e.g. a code block for an ATOMIC ITERATOR is typically declared to receive a `java.lang.Object` as its sole parameter. But in order to execute the operation on an element, the code block sometimes needs to access additional data. Since anonymous local classes can access all final variables in the scope of the surrounding method, store all such data that must be accessible from the code block in `final` local variables of the surrounding method:

```
// must be final to be visible for the anonymous local class
final ActionEvent evt = new ActionEvent (....);

// apply the operation to all listeners
new AtomicIterator (allListeners).doForEach (
  new Operation () {
    public void executeOn (Object obj) {
      ((ActionListener) obj).actionPerformed (evt);
    }
  });
```

Return values. The surrounding method often needs to know the results of the execution of an anonymous command – for example if it performs a read operation – making it necessary for that data to be passed from the code block to the surrounding method. There are several ways this can be achieved.

- *Return data using a return value*. If returning a value fits conceptually with the algorithm performed on the code block, then the introduction of a return value into the code block's signature is a good way of returning data because it makes the flow of data explicit. It means that the algorithm must take care of the returned results, possibly collecting them, and finally returning them itself.
- *Collecting Parameter*. A variation of this that works if a code block is executed several times is to use the COLLECTING PARAMETER pattern, passing a collection as a parameter to each code block and letting it add its results.
- *Save data in an attribute*. This works only in the special case that there is a corresponding attribute in the surrounding class. Accessing the surrounding object's state can lead to unpleasant subtleties in a multithreaded context, though. Nonetheless, if it does fit it is an easy and good solution.
- *Store data in a* `final` *holder*. If none of the above approaches work well – and that is the usual case – a `final` variable in the surrounding method can be used to hold the results and make them accessible outside the code block. A `final` variable cannot be changed, so an additional level of indirection is needed, making use of the HOLDER pattern. This approach is the most flexible but also the most cumbersome. An implementation looks like this:

```
Collection coll;
....

// the holder must be final to be accessible from the code block
final IntHolder holder = new IntHolder ();

new AtomicIterator (coll).doForEach (new Operation () {
  public void executeOn (Object obj) {
    // store the result in the holder
    holder.value += (obj.toString()).length();
  }
});

System.out.println ("Summarized length of strings: " +
  holder.value);
```

with a class IntHolder

```
/** provides an additional level of indirection to store an
  * int value */
public class IntHolder {
  public int value;
}
```

Exceptions. It is part of Java's specification that (checked) exceptions must be either caught or declared, which usually is a strength of the language because it makes exception handling more transparent. For code blocks, however, this introduces a complication: it must be decided early which exceptions an algorithm and its command interface declare to throw.

This reduces the scope in which an algorithm is usable. Consider an ATOMIC ITERATOR: if the operation is declared to throw nothing, the ATOMIC ITERATOR cannot be used in a transactional context where an exception triggers a rollback operation; if the operation is declared to throw an exception, the implementation is impractical to use in the general case.

There is no single simple solution to this dilemma. There are, however, several alternatives with specific strengths and weaknesses.

- If a checked exception is supposed to be thrown by a code block and propagated by the algorithm then this exception needs to be declared both in the command interface and the algorithm's `execute` method. If the exception is conceptually related to the algorithm then this can work well.

  Often, however, the exception is unrelated to the algorithm and specific to the context of the surrounding method. If this is the case, the only option is to duplicate the algorithm for every combination of checked exceptions that needs to be propagated; this reduces the benefits of ANONYMOUS COMMAND because it introduces new code duplication. Still, if there are only few different sets of exceptions that are needed then this is an option.

  Declaring the algorithm to throw a generic `Exception` does not work well because this forces all clients to handle generic exceptions, whether they can actually occur in the context or not.

- This problem can be avoided by restricting code blocks to throwing subclasses of `RuntimeException`. This means that the compiler cannot ensure that all exceptions are taken care of but it separates the algorithm from the concern of propagating exceptions that are unrelated to it.

  This approach makes the use of ANONYMOUS COMMAND more convenient. If the system usually uses checked exceptions to indicate certain conditions, however, this causes a mismatch if such a condition occurs inside a code block.

- As a compromise, EXCEPTION TUNNELING can be used: the code block is declared to throw a generic exception which the algorithm wraps in an unchecked exception. This allows client code to ignore exception handling if it is not needed and to unwrap – and then handle or rethrow – the actual exception as best fits its needs.

Simplicity. For the use of code blocks to parameterize algorithms to work in practice, it is essential to keep their use as simple as possible. Providing unneeded flexibility that complicates the code using an algorithm adds complexity and therefore reduces the applicability.

Default Implementations. Common commands can be provided in the form of default implementations. These can be complete – with or without configurability in the constructor – or partial, requiring one or more methods to be overridden. The latter case is an application of ANONYMOUS TEMPLATE METHOD, described below. If a default implementation is stateless, it is even possible to make a single instance globally accessible, further simplifying its use.

Stepwise Refinement. Client code using ANONYMOUS COMMAND needs to introduce some syntactic overhead, but that is at a fairly manageable level.

The idiom can be used in a nested fashion to factor out common code within commands, but doing so naively forces client code to explicitly represent the entire hierarchy of abstractions. The following example from the domain of relational database access shows client code, leaving out the exception handling:

```
// BAD CODE: This Code is overly complicated. Alternatives follow

// ConnectionHandler contains the common code to handle database
// connections, StatementCommand is a default command implementation
// that creates a database statement and provides this to a nested
// command.
new ConnectionHandler().execute (
  new StatementCommand (new StatementHandlingCommand ()
  {
    public void doForStatement (Statement stmt) {
      stmt.executeUpdate ("DELETE * FROM employee;");
    }
});
```

This code is complicated to understand because all nested abstractions are explicitly present.

There are two alternatives. Firstly, the `StatementCommand` could be implemented using ANONYMOUS TEMPLATE METHOD so that the second level of abstraction is hidden to the client. Secondly, a `StatementHandler` class could be written that takes a `StatementHandlingCommand` and uses a `ConnectionHandler` to perform its operation.

Constructor parameter. Some flexibility can be achieved by passing a parameter to the constructor, based on which a decision about the algorithm is made. This reduces the flexibility, but it also reduces the external complexity.

Static execute method. The instantiation of the algorithm can be avoided by providing a `static execute` method. This slightly improves the performance because one less object needs to be created but reduces the flexibility because the algorithm is forced to be stateless.

Several code blocks. If an algorithm needs several code blocks, they can be grouped into one command interface or split into several; this decision should be made taking into account cohesion and reusability considerations.

Scope of the command interface. The command interface can be either an inner type of the algorithm or a top level type of its own. The first option explicitly connects the two types, whereas reuse of default implementations across several algorithms favors the latter.

Callbacks. The code block has full access to the surrounding object. But care needs to be taken when the surrounding object is actually accessed from the code block because there is no implicit guarantee that the containing method has left the object in a consistent, valid state before executing the code block. Accessing the object state from a code block is a callback operation by nature, sharing some of the subtle issues of multithreading.

Editor macros. ANONYMOUS COMMAND is very useful for providing functionality that has system-wide use or is even reusable as part of a library, for example executing code in a transactional context. In such a situation an editor macro makes the use of ANONYMOUS COMMAND much more convenient because much of the typing overhead (instantiating the algorithm and the interface, writing the method signature, calling the `execute` method) is identical for all places where ANONYMOUS COMMAND is used.

Making an editor macro available together with the algorithm can significantly improve the acceptance of the idiom in a team.

Consequences

The ANONYMOUS COMMAND idiom has the following benefits:

- The resulting code is written at a higher level of abstraction, allowing for code that is easier for human readers to understand.
- Code duplication is avoided, allowing changes and fixes to be applied at a single place and reducing the potential for erroneous copy and paste programming.
- Not only algorithms but also code blocks can be reused, even across several algorithms. If one algorithm takes several code blocks, defaults for these can be used independently of one another. Several clients can share specific code at the class or even at the object level.
- The communication of the code block with the surrounding code becomes more explicit. This is often helpful for assessing the quality of the general design and pointing out opportunities for refactoring.

- The specific local code provided by clients is strongly decoupled from the common code because they are separate objects and share no code. This allows the common part to reside in a separate component with a clean and simple interface.
- Several or all clients can share the same handler object. Actually any multiplicity of handlers is possible, for example using the SINGLETON pattern or a pool. This is particularly helpful if the handler is expensive to create or represents a limited resource. The other extreme of having one handler per local context, however, makes sense for handler objects that are cheap to create. The essential point is that ANONYMOUS COMMAND cleanly decouples local context and handler so that any multiplicity can be implemented.
- Commands are passed to the handler as separate objects, so the handler can store them in a straightforward way in order to provide additional functionality. Usually this only makes sense if there is one global handler instance for all clients so that there is one place to keep the commands. Examples of such functionality are the keeping of a history, performing logging or providing cascaded undo functionality.

ANONYMOUS COMMAND has liabilities as well:

- The use of ANONYMOUS COMMAND introduces additional complexity to the communication of the code block with the surrounding code. This is especially true if complex data needs to be returned or specific exceptions thrown from the code block.
- Once the command interface has been introduced and is widely used, it cannot be changed without affecting many parts of the system.
- The command code is embedded in the client code, making it difficult to test and debug separately. There is also no separate hierarchy of command classes that can be verified a priori to make sure all commands behave correctly. That makes it easier for a command to be intentionally or accidentally implemented in such a way that it corrupts the state of the command handler or compromises system security. Such a faulty command can have non-local consequences that are hard to reproduce and track to their source.
- ANONYMOUS COMMAND cannot be used across process boundaries. The common part is cleanly separated from the client code so that a command object could in principle be passed from a client in one process to a handler in another. This distribution, however, would prevent the command from accessing the local context of the calling method. If the handler is in another process than the client, only the classical COMMAND pattern can be used.
- Creating anonymous local classes requires syntactic overhead, which limits this idiom's applicability for very short code sequences; the syntactic overhead has a tendency to hide the actual content of the code block. As the idiomatic use of ANONYMOUS COMMAND becomes habitual, however, the syntactic overhead becomes less distracting, and its obfuscating effect diminishes.
- Using ANONYMOUS COMMAND causes a slight performance penalty because it

requires the creation of one additional object for each command that is passed to the algorithm.

Known Uses

The nature of this idiom is so fundamental that it is used in most large Java systems. For example, the Java core library uses it to implement the security API. The interface `java.security.PrivilegedAction` is used to pass operations to the `doPrivileged` methods of `java.security.AccessController`. Client code often uses anonymous local classes to implement the operations.

There are also two other idioms that are often implemented using ANONYMOUS COMMAND:

- ATOMIC ITERATOR provides a method that applies an operation to all elements of a collection in an atomic way. The operation to be applied is often implemented as an ANONYMOUS COMMAND. The JGL collection library is implemented in this way.
- EXECUTE AROUND METHOD executes an operation in a specific context, performing operations before and after it, for example acquiring and releasing a resource that is needed by the operation. This idiom is also often implemented to accept the operation as an ANONYMOUS COMMAND.

5 ANONYMOUS TEMPLATE METHOD

Thumbnail

Problem	Solution
How do you make the common part of a sequence of statements reusable if it includes parts that are specific for the local context in which it is executed? The specific and the common code should be coupled to enable a refinement hierarchy of common code sequences where one uses and supplements the other.	Provide a superclass implementing the control flow and calling hook methods for the specific parts. To execute this control structure in a local context, derive an anonymous local class from the superclass, overriding some or all of the hooks.

Example

The transaction example from ANONYMOUS COMMAND can also be implemented using inheritance.

The following code fragment shows typical transaction handling as it is often needed throughout an application:

```
try {
  Transaction transaction = new Transaction ();
  .... // perform operations in the transaction context

  // no exception occurred, so everything is OK
  transaction.commit ();
}
catch (Exception exc) {
  transaction.rollback ();
}
```

Most of this code sequence is common to all transactions. This common part can be moved into a superclass that looks like this:

```
/** implements a template method that manages the transaction and
  * calls a hook method to execute the actual code.*/
public abstract class SimpleTransaction2 {
  public final void execute () {
    try {
      Transaction transaction = new Transaction ();

      // call the hook method in which subclasses
      // provide specific code
      perform (transaction);

      transaction.commit ();
    }
    catch (Exception exc) {
      transaction.rollback ();
    }
  }

  // override with the transactional application code
  protected abstract void perform (Transaction tx);
}
```

This superclass allows the execution of code in a transactional context by creating an anonymous local subclass and overriding the hook method.

```
// override the hook method in an anonymous local class
new SimpleTransaction2() {

  // this method is called as a hook from the execute method
  public void perform (Transaction tx) {
    ....
  }
}.execute ();
```

This hides the details of transaction handling, and only the local code that is executed inside the transaction remains.

Problem

In software systems there tend to be common sequences of statements that are used in several places, like acquiring a resource, performing an operation on it and then releasing it again.

This sort of commonality makes it desirable to factor out and reuse the common part. Otherwise these common code sequences are difficult to change consistently and error prone to implement, even if the common parts are simple. Rather, they all share the "bad smell" that is attached to code duplication in general.

In spite of this, parts of the sequence are often very specific for a local context, such as the actual operation on a resource or object, so that it is difficult to extract the common code into an abstraction of its own. For example, the context sensitive part of the code sequence often needs to be located close to its calling client so as to improve human readability. Moreover, the specific code often needs to communicate with its client, for example to access local client data or to throw exceptions in a specific to the particular client.

So how do you abstract a common sequence of statements in a reusable way so that using code can provide part of the sequence individually?

The following forces influence the situation:

- Since common sequences of statements frequently occur at a fairly small granularity a solution should require little typing overhead in order to be feasible for short common code parts.
- Often there is the same specific code in several places. This makes it desirable to reuse an implementation of specific code.
- Often there are points in the sequence for which some clients use a default implementations while other clients need to provide specific code. These different needs are best served by allowing clients to ignore some points of variability entirely and execute the default behavior implicitly. This couples the common code and the clients.
- Often there is a hierarchy of reusable common code sequences, one using and supplementing the other in order to provide more refined functionality. An example of this is the access of a relational database with JDBC: the most general common code sequence just acquires and releases a database connection and performs exception handling, but in many places another code sequence that also acquires and releases a `Statement` object is needed as well. Such a hierarchy blurs the distinction between the responsibilities of the common part and the client code by coupling the two.

Solution

Create an abstract superclass containing a public method that contains the common code – the template method – calling hook methods where configurable behavior is needed.

```
public abstract class DemoTemplate {

  public final void execute () {
    // this method contains the common sequence of statements in
    // a reusable way. At points where variability is needed hook
    // methods are called so that subclasses can determine the
    // behavior.
    ....
    doSpecific1 ();
    ....
    doSpecific2 ();
    ....
  }

  protected abstract void doSpecific1 ();
  protected abstract void doSpecific2 ();
}
```

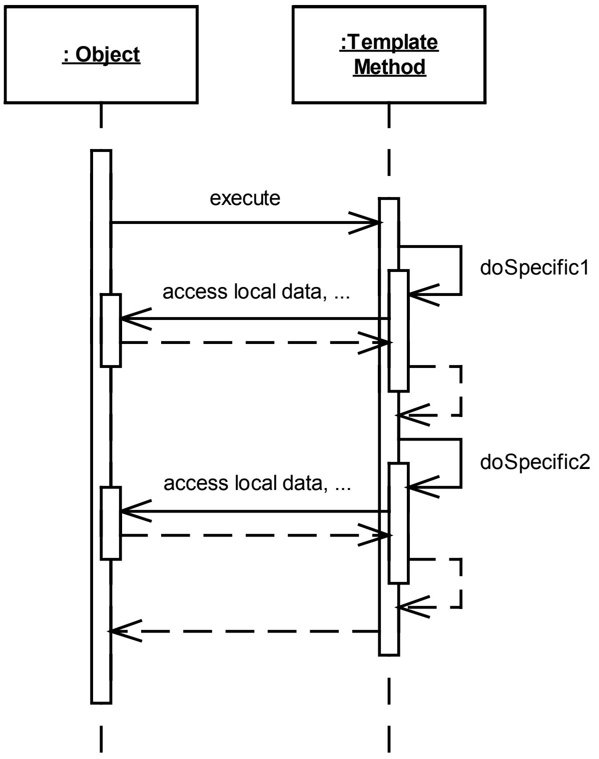
In each place where the common behavior is needed, derive an anonymous class from this superclass, overriding the hooks, and call the public method on its instance.

```
new DemoTemplate() {
  protected void doSpecific1 () {
    // The body of this method is an actual code block that is
    // passed by the client to the operation in the execute method
    // of the superclass
    ....
  }
  protected void doSpecific2 () {
    // The body of this method is another code block
    ....
  }
}.execute ();
```

This idiomatic kind of anonymous local inheritance provides a higher-level abstraction for code using it because it contains the common part of the code sequence in a class and gives it a name that can be used to talk about it.

The anonymous local inheritance introduces some syntactic overhead in the code using the abstracted control flow, but this is unlikely to be significantly bigger than the amount of code that is abstracted into the superclass.

The following sequence diagram illustrates the order of events, explicitly showing the possible access of the anonymously overridden methods to the containing object's state.



The template method ("`execute`") executes the common part of the code. When it reaches a point where clients can provide specific code, it calls one of the hook methods that the client could have overridden. Since the overridden hook method is part of an anonymous local class, it can access local variables of the containing method.

ANONYMOUS TEMPLATE METHOD shares the structure with the TEMPLATE METHOD pattern as described in [GHJV95]: a superclass implements an algorithm that calls hook methods at points for variability, and subclasses provide specific behavior by overriding the hook methods.

The context of ANONYMOUS TEMPLATE METHOD is, however, much more specific, and the anonymous local inheritance introduces the callbacks into the surrounding method to the dynamics of the idiom.

Implementation

There are a lot of issues to consider when implementing ANONYMOUS TEMPLATE METHOD. Some of them are related to anonymous local inheritance and are therefore shared with ANONYMOUS COMMAND, but others are specific for ANONYMOUS TEMPLATE METHOD.

Passing data into the code block. The method signature of the code block reflects the structure of the underlying algorithm; for example, a code block for an Atomic Iterator is typically declared to receive a `java.lang.Object` as sole parameter. But in order to execute the operation on an element, the code block sometimes needs to access additional data. This can be achieved by storing the data in `final` local variables as described in more detail for ANONYMOUS COMMAND.

Return values. The surrounding method often needs to know the results of the execution of an anonymous command – for example if it performs a read operation – making it necessary for data to be passed from the code block to the surrounding method. This can be achieved by using a return value – implementing the COLLECTING PARAMETER pattern – or by storing data in an attribute or a `final` holder variable. These alternatives are described in more detail for ANONYMOUS COMMAND.

Exceptions. Java requires checked exceptions to be either caught or declared which in the general case makes it difficult to pass them from a code block to the surrounding method. There are three approaches that are discussed in more detail for Anonymous Command: either the algorithm is duplicated for every combination of checked exceptions that is needed or code blocks are restricted to throw only unchecked exceptions or EXCEPTION TUNNELING is used as a compromise.

Simplicity. For the use of code blocks to parameterize algorithms to work in practice, it is essential to keep their use as simple as possible. Providing unneeded flexibility that complicates the code that uses an algorithm adds complexity and therefore reduces the applicability.

Use of qualifiers. The template method should be `final` to make explicit that it is not to be overridden. The superclass should be made `abstract, even` if all hook methods have default implementations.

Visibility. The template method should be the superclass' only `public` method. It is the only method that is supposed to be called by a client. The hook methods should be `protected`.

Default implementations. ANONYMOUS TEMPLATE METHOD makes it easy to make one or more code blocks optional by providing default implementations for the hook methods. This is a convenient way of providing additional points of flexibility without forcing every user into the increased complexity.

Stepwise refinement using inheritance. Sometimes there are several levels of abstractions for which stepwise refinement of the implementations is needed. This can be modeled well by using an inheritance hierarchy of ANONYMOUS TEMPLATE METHOD classes, each implementing some of its superclass' hooks and calling new hooks. Derived TEMPLATE METHOD classes can make their implementations of hook methods `final` to make explicit which hooks are supposed to be overridden on them.

Providing an ANONYMOUS TEMPLATE METHOD as a command. If Anonymous Command is used to provide a configurable algorithm, an ANONYMOUS TEMPLATE METHOD implementing the `Command` interface can be used to provide a variable reusable command. An example of this is contained in section 6 "The Idioms in Practice".

Constructor parameter. Some flexibility can be achieved by passing a parameter to the constructor which influences the algorithm's behavior. This reduces the flexibility, but it also reduces the external complexity.

Callbacks. The code block has full access to the surrounding object. But care needs to be taken when the surrounding object is actually accessed from the code block because there is no implicit guarantee that the containing method has left the object in a consistent, valid state before executing the code block. Accessing the object state from a code block is in the nature of a callback, sharing some of the subtle issues of multithreading.

Editor macros. Much of the syntactic overhead in a client using ANONYMOUS TEMPLATE METHOD is identical for every usage. Providing an editor macro together with the superclass makes it easier to use and can improve acceptance in a team.

Consequences

The ANONYMOUS TEMPLATE METHOD idiom has the following benefits:

- The resulting code is written at a higher level of abstraction, allowing for code that is easier to understand by human readers.
- Code duplication is avoided, allowing changes and fixes to be applied at a single place and reducing the potential for erroneous copy and paste programming.
- The communication of the code block with the surrounding code becomes more explicit. This is often helpful for assessing the quality of the general design and highlighting opportunities for refactoring.
- The superclass can provide a default implementation for hook methods, allowing clients to either provide their own implementations or entirely ignore the fact that there is a potential point of variability. This optional variability significantly reduces the complexity of the client code that uses the default implementation.
- A hierarchy of reusable code sequences where one uses and supplements the code of the other can be implemented using an inheritance hierarchy of ANONYMOUS TEMPLATE METHODs. This allows client code to use the abstraction which best fits its needs and ignore the hierarchy.

ANONYMOUS TEMPLATE METHOD also imposes some liabilities:

- The use of ANONYMOUS TEMPLATE METHOD introduces additional complexity to the communication of the code block with the surrounding code. This is especially true if complex data needs to be returned or specific exceptions thrown from the code block.
- Once the hook methods have been introduced and are widely used, their signature cannot be changed without affecting many parts of the system.
- The code block implementations are embedded in the client code, making them difficult to test and debug separately. That makes it easier for a code block to be intentionally or accidentally implemented in such a way that it corrupts the state of the command handler or compromises system security. Such a faulty command can have non-local consequences that are hard to reproduce and track to their source.

- The common code sequence and the local specific code provided by the client and the handler code reside in the same object, coupling the two; the specific code is not objectified but only methodified. This prevents the use of a global pool of handler objects.
- Creating anonymous local classes requires syntactic overhead, which limits this idiom's applicability for very short code sequences; the syntactic overhead has a tendency to hide the actual content of the code block. As the idiomatic use of ANONYMOUS TEMPLATE METHOD becomes habitual, however, the syntactic overhead becomes less distracting, and its obfuscating effect diminishes.
- Using ANONYMOUS TEMPLATE METHOD causes a slight performance penalty because it requires the creation of an object whenever it is used.

Known Uses

The nature of this idiom is so fundamental that it is used in most large Java systems. Specifically, two other idioms that are widely used are often implemented using it:

- ATOMIC ITERATOR provides a method that applies an operation to all elements of a collection in an atomic way. The operation to be applied is often provided as an ANONYMOUS TEMPLATE METHOD.
- EXECUTE AROUND METHOD executes an operation in a specific context, performing operations before and after it, for example acquiring and releasing a resource that is needed by the operation. Again, the idiom is often implemented to take the operation as an ANONYMOUS TEMPLATE METHOD.

Another frequent use of ANONYMOUS TEMPLATE METHOD is to create a parameterizable default command for ANONYMOUS COMMAND.

A hierarchy of ANONYMOUS TEMPLATE METHODs was used in the PDE application, a globally distributed business planning application. The server code that receives a call from a client needs to initialize several resources and to provide exception handling code. There are several kinds of calls that form a specialization hierarchy, one level using and supplementing the common code of the other. This hierarchy of calls is represented by an inheritance hierarchy of ANONYMOUS TEMPLATE METHODs.

6 Other Patterns

ANONYMOUS COMMAND and ANONYMOUS TEMPLATE METHOD are related to a number of other patterns and idioms in various ways.

They share the structure of COMMAND and TEMPLATE METHOD [GHJV95]. The ANONYMOUS versions described in this paper, however, set this structure in the context of anonymous local inheritance, coupling the code blocks to the surrounding method and introducing callbacks to their dynamics.

EXECUTE AROUND METHOD and ATOMIC ITERATOR (closely related to ENUMERATION METHOD [Beck97]) are idioms that rely on code blocks and are therefore implemented using one of the ANONYMOUS idioms described in this paper.

The COLLECTING PARAMETER pattern described in [Beck97] can be used to return data from a code block to the surrounding method, especially if the algorithm executes the code block repeatedly.

And finally EXCEPTION TUNNELING can be used to propagate a checked exception from a code block to the surrounding method without having to declare it.

The following table summarizes these patterns.

Name	Problem	Solution
COLLECTING PARAMETER [Beck97]	How do you return a collection that is the collaborative result of several method calls?	Add a parameter to all the methods that collects the result.
COMMAND [GHJV95]	How can you parameterize clients with different requests?	Encapsulate a request as an object and pass this object to a client.
ENUMERATION METHOD [Beck97]	How do you perform iteration over a collection in an atomic way?	Implement a method that executes a block for each element of the collection.
EXCEPTION TUNNELING	How do you propagate a checked exception without declaring it? This is particularly an issue if the exception needs to be propagated through code that is used in more than one context.	Wrap the checked exception in an unchecked exception and throw that. At a point where the checked exception can be meaningfully handled or propagated, catch the unchecked exception and rethrow or handle the inner exception.
EXECUTE AROUND METHOD [Beck97]	How do you abstract a pair of actions that need to be performed before and after a sequence of statements?	Pass a code block containing the sequence of statements to a method that performs the first action, then executes the block, and finally performs the second action.
TEMPLATE METHOD [GHJV95]	How can you provide a family of algorithms, allowing certain steps to be redefined?	Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.

7 Conclusion

This paper discussed the use of anonymous local inheritance to implement the concept of code blocks.

It introduced two idioms that reify a recurring sequence of statements and allow it to be parameterized using anonymous local inheritance, one using inheritance and one using delegation. Both ANONYMOUS COMMAND and ANONYMOUS TEMPLATE METHOD allow code to be written at a higher level of abstraction, reducing complexity and avoiding code duplication.

Both idioms solve the same problem, but they resolve different forces and have different benefits and liabilities.

Different abstractions. The abstractions created by the two are slightly different. ANONYMOUS COMMAND focuses more on the code block being passed to the algorithm whereas ANONYMOUS TEMPLATE METHOD emphasizes the parameterized algorithm. There is no functional relevance attached to this distinction but it can be used to help human readers understand the code.

Coupling. ANONYMOUS COMMAND uses a handler object that is distinct from the command objects that are passed to it by client code, resulting in high degree of decoupling between the client code and the handler. Commands and the handler have clearly distinct responsibilities. This separation allows the use of one global handler that can even store a history of executed objects to implement logging or multiple undo.

In ANONYMOUS TEMPLATE METHOD the common code sequence and the code blocks provided by clients reside in the same object, creating a stronger coupling between them. This tighter coupling makes it easier to implement a refinement hierarchy of code sequences where one uses and supplements the other. Such a hierarchy can be represented by an inheritance hierarchy of ANONYMOUS TEMPLATE METHODs very naturally.

As a system evolves, the two idioms are often used together. Typically, ANONYMOUS COMMAND is introduced first to provide a context for local client specific code. Then later it turns out that there are groups of commands that again have common code. This second level of commonality can conveniently be implemented by providing an ANONYMOUS TEMPLATE METHOD superclass that implements the command interface. This provides the benefits of decoupling at the highest level while allowing clients to choose from a hierarchy of partially implemented default commands.

Defaults. Both idioms allow default code blocks to be provided, but they do so in different ways. If an algorithm is to be parameterized in several places, the idioms have specific benefits and liabilities.

ANONYMOUS COMMAND allows default command implementations to be used for several algorithms as long as the command interface remains the same. It is also possible to provide several default commands for each point of variability and combine them independently. But it confronts every client with every point of variability, not allowing for implicit defaults.

ANONYMOUS TEMPLATE METHOD on the other hand allows a client to implicitly use a default implementation of a code block, not forcing it to address the point in any way. Several subclasses can be created to provide different defaults, but in the case of several hook methods the defaults cannot be combined independently.

8 Acknowledgements

I would like to thank my shepherd Frank Buschmann for the thorough and helpful feedback he gave. He helped me greatly to improve this paper. I would also like to thank Jan Hermanns, Markus Lauer, Kevlin Henney, Bernd Gründling and Pascal Costanza for the fruitful discussions and input they provided.

9 References

[Beck97] Kent Beck, *Smalltalk Best Practice Patterns*, Prentice Hall, 1997

[Fowler99] Martin Fowler, *Refactoring. Improving the Design of Existing Code*, Addison-Wesley, 1999

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995

[Henney99] Kevlin Henney, *Patterns inside out*, presented at *Application Development 1999*

[Henney00] Kevlin Henney, *C++ Patterns – Executing around Sequences*, presented at *EuroPloP 2000*