Encapsulated Context Pattern

Formerly known as *Encapsulate Context*

(c) Allan Kelly - allan@allankelly.net

http://www.allankelly.net

Abstract

A system contains data that must be generally available to divergent parts of the system but we wish to avoid using long parameter lists to functions or global data. Therefore, we place the necessary data in a Context Object and pass this object from function to function.

Audience

Encapsulated Context is principally written for software developers designing and writing programs. The pattern was originally written for C++ developers, however examples have been reported from other languages such as Java and Smalltalk. It is believed that users of any language will find the pattern useful, although C++ developers may find the pattern of particular interest.

By exploring the pattern in depth this paper offers a rigorous explanation of where the pattern occurs, the forces and the consequences of using the pattern. For reference purposes a summary section has been included at the end of the paper. Experienced developers may prefer to read the summary first before reading the entire paper.

Example

In traditional structured programming, global data is minimised by use of function call parameters. This tradition has continued, with some modifications in object-oriented programming. For example:

```
void ProcessMarketTrade( MarketMessage& msg,
                         MarketDataStore& store) {
    if (msg.Trade() == Sell)
        store.Sell(msg.Commodity(), msg.Price(),msg.Quantity());
    else
        store.Buy(msg.Commodity(), msg.Price(), msg.Quantity());
} // ProcessMarketTrade
```

We now decide that any trade which results in a negative quantity should result in an error message, hence the function `Sell` must have access to the log manager, consequently a handle must be passed down. The code becomes:

```
void ProcessMarketTrade( MarketMessage& msg,
                         MarketDataStore& store,
                         LogManager* log) {
  if (msg.Trade() == Sell)
      store.Sell( msg.Commodity(), msg.Price(), msg.Quantity(), log);
  ... as before ...
```

Such changes have a habit of reoccurring, so, when we add a transaction history the code changes again:

```
void ProcessMarketTrade( MarketMessage& msg,
                        MarketDataStore& store,
                        LogManager* log,
                        TransactionHistory& history) {
    if (msg.Trade() == Sell)
        store.Sell(msg.Commodity(), msg.Price(), msg.Quantity(),
                   log, history);
    ... and so on ...
```

Several problems are clearly apparent. First the parameter list is growing with a negative effect on comprehensibility, even though the additional code is trivial it increases the bulk. Secondly, we are breaking encapsulation. Initially `Sell` was an encapsulated function, by adding more and more parameters its inner workings are being exposed.

More ominously, we have a ripple effect running through interface and implementation code. The function that calls `ProcessMarketTrade` must itself have access to `LogManager` and `TransactionHistory`, and in turn, the function that calls that function, and so on. Even though these functions will only act as pass-throughs for the handles they are affected.

Less obvious is the capacity for redundant code to enter the system. If at some future date we dispense with the transaction history then removal impacts at least three different functions. To be sure, the temptation would be to disable the code while leaving it in place, hence we simply make it an anonymous parameter in `Sell`:

```
void MarketStore::Sell( Commodity& c, Price& p, Quantity& q,
                        LogManager* log, TransactionHistory&) {
  ....
```

In choosing not to delete the history in full we are storing up complications for future refactorings, we are also half-way to implementing the *Poltergeist* anti-pattern (Brown, 1998).

These problems are exacerbated when a dependency inversion design is adopted. We may decide to recast our market message processing as a *Command* pattern (Gamma et al., 1994):

```
class MarketMessageCommand {
public:
    virtual void Action(MarketDataStore&, LogManager*) = 0;
    ....
};

class Buy : public MarketMessageCommand {
public:
    virtual void Action(MarketDataStore&, LogManager*);
    ....
};

class Sell : public MarketMessageCommand {
public:
    virtual void Action(MarketDataStore&, LogManager*);
    ....
};
```

To ensure substitutability each `MarketMessageCommand` must implement `Action` with the same signature as the abstract base class. Consequently commands such as `Buy` are complicated with parameters which are unused. Worse, the potential for ripple effects is magnified across all objects in the hierarchy. If the exchange introduces a programmatic way of signalling transition point in the trading day with an enumeration such as:

```
enum TradingDay {
    Closed, PreOpen, Open, Settlement, Suspended
};
```

A new market message is needed to handle this, but so too is a state variable:

```
class TradingDayChange : public MarketMessageCommand {
public:
    virtual void Action(MarketDataStore&, LogManager*,
                        TradingDay& activity);
    ....
};
```

Since our new message can change the state activity a new parameter is needed, to maintain a common signature this parameter must be added to `MarketMessageCommand` and all derived classes. Again, we are increasing the length of the parameter list, introducing a ripple effect and adding complexity. Our main loop may look like:

```
int main() {
    MarketDataStore marketData;
    LogManager *log(LogFactory());
    TradingDay exchangeStatus(Closed):
    MessageSource source;
    while (true) {
        auto_ptr<MarketMessageCommand> w(source.NextMessage());
        w->Action(marketData, log, exchangeStatus);
    }
    delete log;
    return 0;
}
```

Faced with the problem of adding yet more parameters we may be tempted to consider global variables. After all, an exchange is open or closed, there is only one instance of such a flag surely? A tempting solution, the exchange status is a simple variable, initialisation is not a significant problem, and being stack based a memory leak is a non-issue.

However, for `LogManager` a global variable is decidedly less tempting. The example above strictly controls the use of `log` through scope and parameter passing, were the same variable global it could potentially be accessed before creation, e.g. the `MarketDataStore` constructor may choose to log a message.

We would then be forced into the position of trying to enforce creation before use. This is known to be problematic and the best known solution (access through a function) suffers from known issues in multi-threaded systems. Further, the same problems occur in reverse when cleanly ending the program.

While we may be able to survive one or two such global variables we quickly find the number increasing, first the exchange status, then the log manager, what of our transaction history? Have we loaded any DLL plug-ins? Better have a global list of their handles. As we add more global variables it becomes harder to reason about the initialisation sequence for each – particularly important when one makes use of another. It is also more difficult to reason about the internal state of the program because it is dispersed with no central point of reference.

Even with the best will in the world the old issues of globals still exist. Judicious use of namespaces, and careful coding may afford us the luxury of a few globals but the old issues have not gone away, merely repositioned or hidden for a while.

The solutions so far suggested do nothing to improve either the testability of our system or the transfer of components to follow-on projects. Suppose we wish to use our `MarketMessageCommand` in a market simulator. Long parameter lists, and global variables force us to implement plumbing around the hierarchy so we can use the commands.

Likewise, if we wish to write a test harness for our hierarchy, or force test data through the system we must implement the necessary plumbing to support the classes.

Each additional parameter or global variables makes the classes and methods more specific and less of a commodity. Without such specifics, the `MarketMessageCommand` hierarchy implements generic, run-time polymorphic handling of messages. Longer parameter lists increase coupling, tying classes closer to the environment, shorter interfaces are more loosely coupled and result in a more general the class.

The nub of the problem is the ever-expanding parameter list. At first this appears simply unsightly, however, as we can see, the need pass more and more parameters is a real issue.

Problem

Access to common data is important to many systems. Many systems contain data which must be generally available to divergent parts of the system, e.g. configuration data, run-time handles and in-memory application data.

However, we wish to avoid using global data - such data is normally regarded as poor engineering practice. Traditionally the problem is addressed by passing such data as function call parameters but over time parameter list become longer. Long parameter lists themselves have an adverse effect on maintainability and on object substitutability.

While access to such data is a common requirement neither of the two common techniques are without problems. Access to the data is not as trivial as it first appears, and as any system grows the drawbacks of each solution become greater.

Forces

There are several forces that any solution to this problem must accommodate for it to be widely applicable.

1. Substitutability

   Software designs based on common interfaces, with object substitutability – either run-time polymorphic or compile-time polymorphic – are restricted in the parameters that can be easily passed to an object because all objects must conform to a common interface with common function signatures to ensure commonality of access - i.e. the Liskov Substitution Principle - LSP (Liskov, 1988, Martin, 1996b).

   However, were all data is supplied to objects and function via call parameters, if any object requires additional data it must be passed via a call parameter, to keep

LSP all similar objects must also accept this parameter even if they have no functional requirement for it.

For an object, changing any function-method call signature, whether by addition, revision or removal breaks LSP. The object in question can no longer be substituted for other similar objects. The compiler should refuse to compile the resulting program. Typically we must either change every class in the same hierarchy to match the new signature, change every call to the function-method, or both.

Having broken LSP we are forced to restore LSP by changing other parts of the system. This creates ripple effects through the code base. A good solution to the overall problem would ensure that LSP is not broken, and consequently, ripple effects within the code base are minimised.

2. Encapsulation

   Good software practice values encapsulation, however, traditional solutions threaten encapsulation:

   - Over-long parameter lists to function calls reduce encapsulation because the parameters suggest the internal workings to developers.
   - Global variables break encapsulation by definition. They are considered poor programming practice, leading to side-effects and increased coupling.
   - Within C++ systems there are additional problems associated with instantiation and destruction - particularly in multi-threaded developments. Although C++ namespaces allow better management of globals they do not resolve instantiation and coupling problems.

   A good solution would preserve encapsulation thereby minimising side effects and coupling.

3. Coupling to the environment

   The parameters passed to a function, or method, define the state of the system external to the object in question. An object receiving a method call knows its own state (even if this is stateless), what it does not know is the state of the rest of the system, i.e. the context in which it is called. If global data is used it becomes harder to reason about the state of the system at the point of call.

   Likewise, a simple function maintains little or no state between calls, the external state is everything, the result of the function call depends on the context in which it is called.

   The more tightly coupled an object is to its environment the more difficult it is to use the object in a different setting. Opportunities for using the object in a different environment, e.g. within a test harness, or re-used in a different system, are much reduced. At the same time, the amount of consideration developers must pay the object's environment is increased. Thereby, reducing readability, understandibility and maintainability.

   A solution that minimised coupling would do much to improve understandibility, maintainability and improve the opportunities for alternative uses.

4. Avoid data copying

One solution to the global v. parameter conflict would be to retain a copy of such data in individual objects. Unfortunately, this is not always practical, especially when the system has a large number of small objects and/or objects exist in difference execution threads.

Reasons for not copying pieces of data may include, but are not limited to:

- Data may be changing rapidly, e.g. equity market prices, and needs to be available in several different locations in the program
- Data and operations on the data may overwhelm the class, e.g. a simple command class used in a *Command* pattern may only have one significant method, to additionally store data, handles, and accessors would rob the class of its simplicity.
- Overhead of a copy operation both in terms of time and memory used – this is particularly so if the data is seldom accessed, e.g. command line options.
- Data may be singleton in nature, or encounter problems when copied, e.g. a handle to a log file may be easily copied but we do not wish to store multiple copies of the handle to prevent dangling pointers (or references) when the file is closed. However, use of the *Singleton* (Gamma, 1995) pattern may not be appropriate.

Since these potential solutions are unavailable they represent forces in their own right. Further, as modern systems frequently end up with a large number of small objects these problems are increasing.

Solution

Provide a Context container that collects data together and encapsulates common data used throughout the system.

For example:

```
class Context {
    LogManager*        log_;
    ComandLineOptions  cmdOpts_;
    ApplicationData*   store_;
....
};
```

Rather than supply multiple parameters, we supply a Context object. The object acts as a container for program state data, a central repository for widely used data within the system. The Context object provides few, if any, functions itself. The object is passed, or more likely a reference is passed, to functions when they are called - utilising the “parameterize from above” paradigm.

There are typically three types of data found in a context class:

- *Configuration data,* e.g. command line options.
- *Application data,* e.g. market data.
- *Transient run-time data,* e.g. handle to log manager.

The example given here uses one context class for simplicity. While the simplicity of a single context has a lot to recommend it, without careful attention the class may become a kitchen-sink, overwhelmed with any, and all, data in a system. When this happens we start to see the emergence of a *Blob* anti-pattern (Brown et al., 1998).

To counter the drift towards Blob we can split the class into two or more discrete classes, e.g. one for system data and handles with a second for application data.

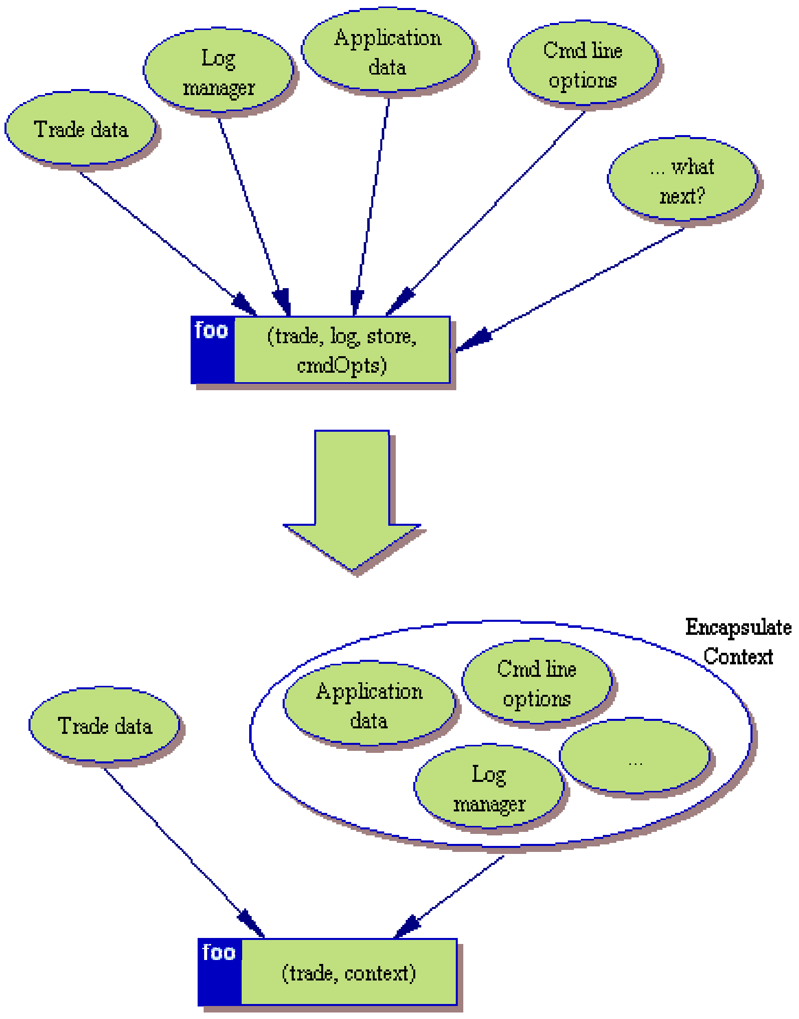

Figure 1 Solution places context data in a single container

Specifically, we can distinguish three types of split:

- *Temporal:* data is separated on the basis of its lifespan, data which is short lived is kept separate from data which exists for long periods. . It is better not to mix transient data with persistent data lest expired data remains in the container.
- *Horizontal:* separating reference data from value data, usually needed when one application becomes large itself, inflating the size of the context.
- *Vertical:* separating the context class into a small hierarchy, usually needed when the same context is needed in a family of programs. This allows for specialisation through inheritance to provide each family member with a specialised Context object and common code to be shared across the family.

Such splits will mitigate the Blob tendencies but also detract from the pattern simplicity. Splitting the context class should also help improve compile times, since we can assume that although some functions will need to be passed all the fragments of the original context, many will require fewer fragments thus reducing dependencies.

However, while it may be desirable to split the Context class for a variety of reasons this can be taken too far. The use of many fine-grained Context objects may return us over long parameter lists.

Thus, any implementation of *Encapsulated Context* pattern should consider the following issues:

- *Is a single Context class the best answer?* The initially simplicity of a single Context may lead to difficulties as anti-patterns emerge.
- *What is the life expectancy of the data?* Bundling short-lived or rapidly changing data together with constant data may lead to confusion or inaccuracies.
- *Is there a family of programs under development?* Is there benefit from creating vertical hierarchy of Context facilitating technology transfers between programs?
- *Are we creating problems by mixing reference and value data in the same context?* Could this data be split horizontally between several Context objects?
- *Are we in danger of creating too many, fine-grained, Context classes?*

These issues must be addressed together as the answers to each question influences the others.

Resolution

Applying this solution to the example given at the start of this paper we get:

```
// MarketContext.hpp
class LogManager;
class CommandLineOptions;
class MarketDataStore;

class MarketContext {
    LogManager*          log_;
    CommandLineOptions   opts_;
    MarketDataStore*     marketData_;
public:
    MarketContext( LogManager*, CommandLineOptions&, MarketDataStore*);
    LogManager* Log();
    MarketDataStore* MarketData();
    CommandLineOptions& CmdOptions() const;
};
```

With this context class the presence or absence, of a `TransastionLog` is abstracted to a detail about `MarketContext`.

The class should take a minimal role in the lifetime of enclosed classes, it is better to present these as ready constructed to the class. This removes life-cycle issues from the domain of the context class, and, because enclosed classes are often just references or pointers, the .hpp interface file should only need forward declarations thereby reducing potential ripple effect.

(The decision on whether to use pointers or references to object is outside the scope of this paper.)

Continuing this example the body of the program is refactored::

```
class MarketMessageCommand {
public:
    virtual void Action(MarketContext&) = 0;
    ....
};

int main() {
    LogManager* log(LogFactory());
    CmdLineOptions options(argc, argv);
    MarketDataStore marketData;
    MarketContext context(log, options, &marketData);
    MessageSource source;
    while (true) {
        auto_ptr<MarketMessageCommand> w(source.NextMessage());
        w->Action(context);
    }
    return 0;
}
```

The context provides access to data which otherwise may be made *Singleton*, global or both, for example the `LogManager`.

In this example the Context object is passed to the `Action` method, an alternative would be to pass the Context to the `MarketMessageCommand` constructor and store a reference. This would allow `Action` to be parameterless at the cost of adding state to the class. Further, by renaming `Action` to `operator()` the class acquires the characteristics of a *function object* (Stroustrup, 1997, p.515) or *functor*. For example:

```
class MarketMessageCommand {
    MarketContext& context_;
public:
    MarketMessageCommand(MarketContext& mc) : context_(mc) {}
    virtual void operator()();
    ....
};
```

While this potentially increases the design's flexibility more attention must be given to lifetime management of the Context object in this case.

Variations

- *Provide parent's `this` pointer*

  The passing of `this` pointers to *worker objects* can be seen as a variation on this theme, in effect the calling object is itself acting as a context object for the worker objects. (One consequence of using Context classes is that the need to pass `this` is usually reduced.)

- *Provide forwarding functions to encapsulated data*

  Rather than expose an entire member class the `MarketContext` class could implement forwarding methods, for example, the `CmdOptions` member could be replaced with:

  ```
  class MarketContext {
      ...
      bool IsVerbose() const { return opts_.IsVerbose(); };
      ... and other forwarding functions ...
  };
  ```

  However, it is best to keep the class as lightweight as possible, to this end, the class exposes the key objects encapsulated rather than implement pass through calls onto the underlying data. It is the underlying class that decides what to expose rather than the context class. Further, although such forwarding functions may be convenient they contribute the tendency for the context class to become a Blob (Brown, 1998) so are best avoided.

Consequences

As a result of the pattern, several of the forces detailed above are resolved or balanced:

1. Substitutability

   Parameters passed to a function call can be restricted to Context objects containing system state data and parameters which specifically refer to the function call task in hand, e.g. market trades. Functions signitures are free of the clutter which can make them fragile - there is no longer a need for every class method in the hierarchy to accept every parameter ever needed.

2. Encapsulation

   The Context object effectively compacts the parameter list on a function call signature, thereby abstracting state variables and promoting encapsulation of the function. In addition there is a reduction in ripple effect as function signatures become more stable.

   Having relieved the problems of passing parameter to a function the attractions of global data are reduced. Indeed, the Context object provides a natural home for data with characteristics of global variables.

3. Coupling to the environment

   The Context classes is encapsulated through its own, well-known, common, interface. This allows the solution to be applied to compile-time and run-time polymorphic designs, using either template metaprogramming or v-table dispatch techniques.

   By providing several context classes data is encapsulated along temporal, horizontal or vertical lines further reducing coupling. It is difficult to eliminate all coupling because some classes will always need other classes, to be sure, choosing the granularity of the coupling is a design issue.

   Additionally, by separating the classes implementing algorithms, from the plumbing which supplies the data the classes themselves are less coupled and more like commodities, making transfer to other developments easier.

4. Avoid data copying

   Since the Context class contains common data with little overhead there is no need to copy the data in local objects.

   There may be multiple references to the Context object in the system, particularly if multiple threads are being used. Hence some care must be taken to avoid dangling references to Context objects.

In addition there are other beneficial consequences:

5. Reasoning

   State data that needs to be shared or retained is factored, objects are left with either transient data or completely stateless. By centralising the core data within a system we have made it easier to reason about the system. We can halt the program and look in one place to see what state the program is in rather than having to look in multiple places.

6. Instantiation

   Instantiation issues are simplified because objects must be created before being placed in the context and are subsequently only accessed through the context. Destruction issues are similarly handled because all access is via the context. The life-span of the context can be clearly defined at a high level.

7. Uncluttered code

   Pass-through code and long parameter lists have been minimised, and the potential for future redundant code has been reduced – it is easier to add and remove elements from the Context class. (This may entail a recompile of the whole system when the interface to the Context class is changed but recompilation should be well-defined procedure.)

8. Synchronisation point

   The Context class can provide a useful place to add mutexes for multi-threaded systems. In multi-threaded environments the Context object can hold all shared data, acting as a gatekeeper with mutex control. This is reminiscent of the *Monitor Object* pattern (Schmidt et al., 2000) with the same potential for bottlenecks if lock access is not carefully considered.

   Bottlenecks may be avoided if the data is either immutable (e.g. command line options which do not change), or data elements manager their own locking (e.g. a log manager which implements its own synchronisation) and application data is absent.

However, there are several less desirable consequences:

9. *Blob* tendencies

   As already mentioned, care must be taken as systems develop that a context class does not become a *Blob*. Already in the example given we see the mixing of value data and reference data. Without vigilance context classes may grow to encompass far more data and functionality than is strictly necessary.

   Invariably, the context class ends up touching most aspects of the system. It is therefore best-placed low down the dependency hierarchy of classes – although this can lead to its own dependency inversion problems and small changes necessitate a major recompile of the system.

   Once this happens we are in danger of implementing the *Blob* anti-pattern.

   Fortunately, change to the Context class tends to by additive in nature so seldom break other parts of the system, still, the friction of change is increased. One way to minimise this is to ensure that no operations are placed inside the context class. A second technique is to use multiple Context classes as described above, however, introducing too many Context classes will introduce some of the original problems we sought to revolve.

10. Hidden Globals

Blind use of Context classes can give rise to an abuse knows as "Hide Forbidden Globals" (Green, 2001). This is characterised by a kitchen-sink approach to the Context class were every second variable is listed. Typically we see Context members which are referenced in only a few points within the system, usually such data would be better embedded in specific classes rather than placed in Context.

11. Dominant sibling

    Program families may share a common root Context class, which they embellish through inheritance. In this model the context underpins the common code of the family. If one family member becomes dominant there will be pressure to enhance the common root to facilitate the dominant member. This has a negative effect on the other family members which start to see the common root as a Blob, forcing upon them additional dependencies and complications they do not need.

    In the program family we find elements of functional overlap, e.g. a market trading system and a market simulation system. Both may use the `MarketMessageCommmand` and hence rely on the `MarketContext` class as above. As one program, say the simulation, becomes more important and bigger objects start to appear in the command hierarchy which are specific to the one application, eventually, one of these will require some data which is not available in the context class. For immediate simplicity we are tempted to add this into the context. Unfortunately, the trading system now has this data even though it is never used. If continued, over time, the trading system will be inhibited by a Context class which is obscured with unused functions.

    More confusing too are the results if the trading system now develops its own specialist message commands, and makes demands for specific fields on the context class.

    This is normally an indication that the Context class should be split vertically. We may choose to create a hierarchy of three classes: a common base class, a derived class with simulator enhancements and second derived class with the trading system enhancements.

    At this point we may compile different versions accepting either a `SimulatorContext` or a `TradingContext`, or we may choose to down-cast the provided context – assuming that the simulator message classes will only ever be passed a `SimulatorContext` by way of a `MarketContext` handle.

Known uses

- *Chutney Technologies Apptimizer (C++)*

  Apptimizer uses a single Context object to store handles to important system objects, e.g., `Configuration`, `CachedData`, `ConnectionServer`, etc. These system objects are accessed by polymorphic command objects, which receive the Context as a parameter to their `execute()` method.

- *Reuters Liffe Connect data router (C++)*

  This system uses two context objects, split horizontally. The first encapsulated system data, log manager handles, a configuration cache, COM parameters, while the second holds application data exclusively.

- *Jiffy XML database server (C++)*

  The Jiffy server has three context objects split along temporal lines. One Context object exists for the length of the program run, this encapsulates process wide context, items such as: log manager handle, command line options and the database store index. A second Context class is used to represent data associated with connections. Each TCP connection is assigned a session context to hold items such as the user id for the connection. Finally, the underlying database from Sleepycat uses it's own database-context object to maintain state between database calls.

  In this case, the database-context objects are short lived, each one is limited to function call scope (although it will be passed to several underlying functions in turn). A session context lives for the duration of the TCP connection, while the process context is created shortly after the application starts running and is destroyed at the end of the program run.

- *Enterprise Java Beans*

  Enterprise Java makes use of Session Beans and Context Beans that encapsulate program state information. Although the objective of Java Beans is to implement component based transaction programming the most of the underlying forces are the same, namely: substitutability of different *beans*, encapsulation of context from server to client and clearly defined coupling.

  However, the fourth force, *avoid data copying*, is absent. In the distributed environment for which Java beans is designed data copying is essential.

Related patterns

- *Command, Chain of Responsibility and Objects for States.*

  Although the *Command* pattern is cited here the same principles apply to any design based on the dependency inversion principle using class hierarchies, e.g. *Chain of Responsibility* (Gamma et al., 1994), *Objects for States* (Henney, 2002), etc. Each of these the hierarchy provides the algorithm while the Context object(s) provide the data.

- *Singleton*

  Encapsulated Context may be a useful alternative to *Singleton* (Gamma et al., 1994) in many program designs.

- *Observer*

  Encapsulated Context may be contrasted with *Observer* (Gamma et al., 1994). Like the Subject in *Observer* the Context class is a central repository of data. Like *Observer* there is a many to one relationship. However, the critical difference lies in the updating mechanics.

  The subject in *Observer* knows its observers, when it is updated it will update all its observers. This satisfies the motivation for the pattern that seeks to keep two, or more, objects consistent. Thus, when one *Observer* changes, and hence changes the Subject the other Observers must be informed. In effect, Subject is an active participant in the execution of the program.

In *Encapsulated Context* there is no requirement on the Context class to inform its clients that something has changed. Indeed, it doesn't know who its clients are so it cannot inform them. *Encapsulated Context* keeps the various objects consistent by centralising the data. It is essentially passive during execution.

While there is obvious transformation for turning a Context object into a Subject, and hence *Encapsulated Context* into an *Observer* pattern, and vice versa, there are fundamentally different motivation and forces underlying two patterns.

- *Monitor*

  As noted above (Consequences section), in multi-threaded systems mutex control can be added to *Encapsulated Context* to assist with synchronisation issues. In this the pattern is acting like Schmidt's *Monitor Object* (2000). While this can provide a simple way to synchronise access to resources it is not without a cost.

  Firstly, by using the context class as a monitor introduces pressure to perform more processing within the monitor class. This contributes to the *Blob* tendencies already described.

  Secondly, the consequences encountered by *Monitor Object* are introduced into the design. Specifically, the liabilities associated with *Monitor Object* need to be recognised, i.e. limited scalability, complicated extensibility semantics, inheritance anomaly and nested monitor lockout.

  Readers are strongly advised to read Schmidt before using *Encapsulated Context* as a synchronisation point.

- *Arguments Object*

  This pattern shares much in common with Nobel's *Arguments Object* pattern (Nobel, 1997). The key difference is that Nobel suggests the pattern as code level pattern for reducing the number of parameters passed to a function, while *Encapsulated Context* advocates using the same paradigm as a high level feature to wrap the state of the system.

- *Introduce Parameter Object*

  Both *Encapsulated Context* and *Arguments Object* pattern resemble Fowler's *Introduce Parameter Object* refactoring pattern (Fowler, 2000). However, Fowler introduces this as only a refactoring pattern without discussion of the issues involved in grouping data or alternative solutions. It is possible to view Fowlers pattern as an application of either *Encapsulated Context* or *Arguments Object* when refactoring code.

- *Parameter Block*

  Some of the motivations of *Encapsulated Context* are shared with *Parameter Block* (Patow and Lyardet, 2003). Both aim to provide a consistent interface through which, diverse parts of a system may access parameters. The focus of *Parameter Block* is internal mechanisms of the context object and how this object may support a dynamic set of parameters at run-time. In contrast, *Encapsulated Context* focuses on parameter passing at compile-time. *Parameter Block* considers a parameter block which stored the various parameters, this has clear parallels with the context class in *Encapsulated Context*. The two patterns do not exclude one another, and under the right circumstances may be complementary.

Discussion

Separating data

At first glance *Encapsulated Context* may seem counter to the principles of object-orientation, this is not so. Instead we are separating the data into that which (a) truly belongs to a given object (e.g. market price and quantity) and (b) that which is owned the system as a whole. There is a casual similarity with the separation of algorithm and container used by the Standard Template Library.

Instantiation issues

While this paper notes the instantiation problems associated with global objects it does not provide an in-depth discussion or offer detailed solutions. To do so is beyond the scope of this pattern. However, it is suggested that some of these problems can be alleviated by application of *Encapsulated Context* pattern.

Testing

With designs based on *Encapsulated Context* we may arrange for artificially configured context objects to used for testing. For example, a test harness could create a context object and populate with data to simulate a scenario we wished to test, the test can then be run without to see how the system behaves in these conditions.

Extending this ideas we can imagine two versions of the `MarketContext` class, one of which validates all inputs and one that is optimised for speed. Alternatively, a context class could load test data to create a specific test scenario, or dump their “state” to file at the end of a test – or in the event of program failure.

Aspect oriented programming

Aspect oriented programming may provide an alternative means to resolving some of the forces which produce this pattern. The data within the Context class certainly seems to cross-cut the systems concerned. The logger functionality is both a core example for both Aspect documentation and this pattern. Since C++ does not currently support Aspects, nor are they a standard part of Java they cannot be regarded as a common solution to this problem and forces.

The main difference appears to centre on the method of passing the context object to the function. This pattern assumes that the context object is passed by way of a function parameter, however, beyond this assumption the concept of bundling the context into one object is still applicable. The key difference is the mechanism for accessing the context object.

Pre and Post conditions

By their nature, context objects represent the state of the system. This makes them very good places to make uses of pre and post conditions to validate system state. Indeed, developers using context objects should be encouraged to use pre and post conditions.

Use of such pre and post conditions is regarded by many ad good programming practice. Used as comments these can help developers reason about the state of the system, used as compiler enforced checks (e.g. macros in C++, conditionals in C#) the system can perform a degree of self validation as well as helping programmers reason.

Pre and post conditions could be placed within the context objects "getter and setter" functions to validate the state of the object, or used by functions accepting context objects to ensure the program is in a suitable state for the function.

Use of such conditions to check state of the system is common practice formal methods systems, e.g. VDM (Jones, 1986) and Z (Wordsworth, 1992). Such languages specify a "state" for the system before and after and operation - the *program state* in VDM parlance. Further research is need on whether *Encapsulated Context* pattern can be useful in development of formal methods based systems.

Value data or reference data

The solution section above notes that care should be taken when reference and value type data is mixed within a single Context object. Such mixing may be a signal that refactoring may be required, and that the Context object should be split horizontally.

However, Context objects observed in actual system frequently mix these data types. While this may indicate poor design it also reflects the fact that Context objects may be required to group various types of data with different reference characteristics. This fact may also indicate that the pattern has been introduced to a system as the result of refactoring and that other parts of the system have not been refactored yet.

Further patterns

There is more that could be said about *Encapsulated Context*, most likely this is one of several patterns in a sequence. At EuroPLoP 2005 Kevlin Henney presented several patterns that follow from *Encapsulated Context.* These are *Encapsulated Context Object, Decoupled Context Interface, Role-Partitioned Context* and *Role-Specific Context Object*. These are available from his website (http://www.curbralan.com/) and will be included in the conference proceedings in due course.

Genesis of a pattern language - further research

Many of the issues raised in the discussion section suggest further variations of this pattern beyond those outlined already. It is also possible to see how, taken together, *Arguments Object, Introduce Parameter Object, Singleton, Parameter Block* and *Encapsulated Context* may represent part of an entire pattern language. We may tentatively label this pattern language *Context objects.*

For example, *Singleton* could be redefined as an example of *Encapsulated Context* were there is only one instance of the Context object, and the object is accessed via a global variable instead of via parameter passing.

There are four groupings within which to consider variation within the *Context objects* pattern language:

- *Access mechanism*

  Function parameter passing is used in *Encapsulated Context* to make the Context object accessible. In contrast, *Singleton* uses a global access point. Thread local storage has been suggested as an alternative access mechanism for multi-threaded systems. A further access mechanism, were available, is the Point Cut provided by AspectJ and other aspect oriented languages.

- *Context lifetime*

  While *Singletons* are generally instantiated for the lifetime of a program run, Nobel's *Arguments Objects* are more ephemeral, being created and destroyed in a short space of time. By extending the consideration of the temporal aspects - described above for *Encapsulated Context* - more pattern variations are possible.

- *Cardinality of context*

  Related to the discussion of lifetime is the issue of cardinality of Context objects. Obviously in cases such as *Argument Object* it is of little importance whether one or one hundred Context objects co-exist. However, in some cases it may be important to limit the number of Context objects within a system, for example, we may wish to limit each thread to one instance of an object, or limit a whole program to one Context object corresponding to the mouse state.

- *Internal implementation*

  Encapsulated Context assumes a fixed internal state were data elements are hard coded and fixed at compile time. In contrast *Parameter Block* allows the content of the Context to change at run time. As already noted both patterns share other similarities and thus may belong to a common language. In this case, the internal representation of data can have a significant effect on system design.

The creation of a *Context objects* pattern language is beyond the scope of this paper. However, it is clear that such a language could unify existing patterns and probably help identify more patterns.

The author looks forward to hearing about such a project and is more than willing to participate in such an endeavour.

More examples

The examples presented are given in C++ although it is expected that the pattern is generally applicable to all languages. The author looks forward to hearing of implementations in Java and C# especially.

Summary

In any non-trivial system there will be a number of data elements that are widely used throughout the program, e.g. log manager and the application data model. Typically these will be classes in their own right and accessed through handles (references or pointers.) Since global data is regard as poor practice it is likely that these handles will be passed by way of function call parameters. However, this technique can soon lead to long parameter lists which are not only difficult to understand but tend to make the program more fragile.

Therefore, we create a context class that encapsulates these data element and pass a handle to this object to the diverse functions.

While similar techniques has been suggested by others (e.g. Nobel, 1997, Fowler, 2000) this pattern discusses the forces and consequences when applied system wide.

This can bring considerable benefits to a design but if used recklessly can result in a number of known anti-patterns.

Rather than use a single context class it may be appropriate to design a system with several. These are divided along temporal, horizontal or vertical lines to ensure that each is consistent and promotes good design.

Acknowledgements

This pattern was the result of a conversation on the ACCU-General mailing list entitled: "overload 49 and state" with signification contributions from Kevlin Henney and Josh Walker, running from 18th June 2002. I am grateful to Kevlin for acting as initial pattern shepherd and Josh for reviewing the results and providing an additional example. The paper was further shepherded by Frank Buschmann in April 2003 for submission to EuroPLoP. Again, I am most grateful to Frank for his time and interest.

I am also most grateful to all in *Workshop D* at EuroPLoP 2003 for their many varied and useful comments concerning the pattern, their support and their suggestions for improvement.

In addition I am grateful to the two anonymous reviews who reviewed this paper in preparation for *Pattern Languages of Program Design volume 5* "PLoPD5".

Principles and Patterns glossary

Pattern Name	**Description**
Arguments Object (Nobel, 1997)	"Large protocols [interfaces] are easy to use because they offer a large amount of behaviour to their clients. Unfortunately, they are often difficult or time consuming to implement, and for client programmers to learn. ... Therefore: make an arguments object to capture the common parts of the protocol."
Blob (Brown et al., 1998, p.73)	"The Blob is found in designs where one class monopolizes the processing, and other classes primarily encapsulate data. This AntiPattern is characterized by a class diagram composed of a single complex controller class surrounded by simple data classes, ... Architectures with the Blob have separated process from data; in other words they are procedural-style rather than object oriented architectures."
Chain of Responsibility (Gamma et al., 1994, p.223)	"Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it."
Command (Gamma et al., 1994, p. 233)	"Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations."
Dependency	"A. High level modules should not depend upon low level

Inversion (Martin, 1996a)	modules. Both should depend upon abstractions. B. Abstractions should not depend upon details. Details should depend upon abstractions.”
Hide Forbidden Globals (Green, 2001)	“Since global variables are "evil", define a structure to hold all the things you'd put in globals. Call it something clever like EverythingYoullEverNeed. Make all functions take a pointer to this structure (call it handle to confuse things more). This gives the impression that you're not using global variables, you're accessing everything through a "handle". “
Introduce Parameter Object (Fowler, 2000, p.295)	“Often you see a particular group of parameters hat tend to be passed together. Several methods may use this group, either on one class or in several classes. Such a group of classes is a data clump and can be replaced with an object that carried all the data. It is worthwhile to turn these parameters into objects and just to group the data together. This refactoring is useful because it reduces long parameter lists, and long parameter lists are hard to understand.”
Liskov Substitution Principle (Liskov, 1988)	“Functions that use pointers or references to base classes must be able to use objects of derived classed without knowing it.” (Martin, 1996b) When using class hierarchies as a means of data abstraction, sub-types must be able to fully substitute for the super-types.
Monitor Object (Schmidt et al., 2000, p.399)	Synchronises concurrent method execution to ensure that only one method at a time runs within an object. It also allows an object’s methods to cooperatively schedule their execution sequences.
Observer (Gamma et al., 1994, p.293)	“Define a one-to-many dependency between objects so that when one object changes state, all its dependants are notified and updated automatically.”
Objects for State (Henney, 2002)	“Allow an object to alter its behaviour significantly by delegating state-based behaviour to a separate object.”
Parameter Block (Patow and Lyardet, 2003)	“Open Arguments is used to create a generic interface for parameter passing, decoupling the API declaration of the procedures and functions from the type and number of the parameters they receive.” A parameter block is passed from function to function, the block contains a dynamic store (often a map) of parameter names and values.
Singleton (Gamma et al., 1994, p.127)	“Ensure a class only has one instance, and provide a global point of access to it.”

History

Date	**Event**
December 2005	Minor changes prior to inclusion in Hillside Europe's electronic archive: Increased font size and minor revisions to abstract.
November 2005	Pattern name changed from *Encapsulate Context* to *Encapsulated Context*. Revised name tells you *what you get* rather than *what you do.*
May 2005	Pattern revised for *Pattern Languages of Program Design Volume 5* (forthcoming) after anonymous peer review. Version for book has some changes to this version, mainly these are cuts. Additions have been incorporated into this version, the web-version will remain the complete pattern but alternative versions may appear elsewhere.
October 2004	Published in ACCU Overload (63) magazine - minor changes.
June 2004	Published in EuroPLoP proceedings (Henney and Schütz, 2003)
Autumn 2003	Revised following conference feedback
June 2003	Work shopped at EuroPLoP 2003
Spring 2003	Pre-conference shepherding by Frank Buschmann
August 2003	Pattern written with help from Kevlin Henney (shepherding) and Josh Walker (reviewing).
June 2003	Pattern proposed on ACCU-General mailing list

Bibliography

Brown, J. B., Malveau, R. C., McCormick, H. W. and Mowbray, T. J. 1998 *Anti-Patterns,* Wiley.

Fowler, M. 2000 *Refactoring,* Addison-Wesley.

Gamma, E., Helm, R., Johnson, R. and Vlassides, J. 1994 *Design Patterns,* Addison-Wesley.

Green, R. 2001 *How to write unmaintainable code*, http://www.web-hits.org/txt/codingunmaintainable.html,

Henney, K. 2002 *Objects for State*, http://www.curbralan.com,

Henney, K. and Schütz, D., 2003, *Proceedings of the 8th European Conference on Patterns Languages of Programs 2003*, EuroPLoP, Kloster Iresee, Germany, UVK Universitätsverlag Konstanz GmbH,

Jones, C. B. 1986 *Systematic Software Development using VDM*.

Liskov, B. 1988 Data abstraction and hierarchy, *SIGPLAN Notices,* 23, 17-34.

Martin, R. C. 1996a The Dependency Inversion Principle, *C++ Report***,** http://www.objectmentor.com/resources/articles/dip.pdf.

Martin, R. C. 1996b The Liskov Substitution Principle, *The C++ Report***,** http://www.objectmentor.com/resources/articles/lsp.pdf.

Nobel, J., 1997, *Arguments and Results*, Pattern Languages of Programming (PLoP) conference, Allerton Park, Monticello, Illinois, Washington University, http://citeseer.nj.nec.com/107777.html.

Patow, G. and Lyardet, F., 2003, *Parameter Block*, EuroPLoP 2003, Irsee, Germany, proceedings pending publication,

Schmidt, D., Stal, M., Rohnert, H. and Buschmann, F. 2000 *Monitor Object* In *Pattern-Oriented software architecture 3*Wiley, pp. 399-422.

Stroustrup, B. 1997 *The C++ Programming Language,* Addison-Wesley, Reading, MA.

Wordsworth, J. B. 1992 *Software Development with Z.*