An Alternative Structure for the Abstract Factory Software Design Pattern

Stuart Maclean
sdm@ecs.soton.ac.uk

Declarative Systems and Software Engineering Group
Technical Report DSSE-TR-97-7
December 1997

Department of Electronics and Computer Science
University of Southampton
Highfield, Southampton SO17 1BJ, United Kingdom

An Alternative Structure for the Abstract Factory Software Design Pattern

Stuart Maclean

December 1997

Abstract

The abstract factory design pattern is a well known and widely applied design in the construction of object-oriented software. An amendment to the abstract factory is presented based on the "envelope/letter" idiom. This results in factory products being instantiated in application code in a manner apparently independent of the factory pattern. Application code can thus forget about the factory, and concentrate on using the product objects. Effort in library design thus benefits application structure, always a worthy end result in program design. For C++ users, the resulting pattern makes factory-generated objects more amenable to language features including operator overloading and memory management.

1 Introduction

The "factory" is a widely used abstraction in object-oriented programming. It is modeled on a real factory, which produces items, be they widgets, cars, circuit boards, etc. We can group these classes of item under the generic name of "products". We map these concepts into the software domain, realising the abstractions factory and product. A factory is an object which makes (creates) product objects. The operations supported by the factory are those which create different products from the portfolio or "product suite" offered by the factory. Messages sent to a factory object thus result in new product objects being created and made available for use. A car factory object might produce door panels, wheels, bonnets, etc.

The intent of the abstract factory design pattern [3] is to provide an interface for creating families of related or dependent objects without specifying their concrete classes. The motivation behind this design is thus to loosely couple the factory from its clients. An important consequence of this is that a single change in client (i.e. application) code yields a whole new factory product suite. The change is in the concrete (dynamic) type of the factory itself; product types do not have to be changed. Since product objects comprise the vast majority of objects related to use of the factory pattern, this is clearly a win regarding issues such as portability and flexibility.

The structure of the abstract factory pattern is depicted in Figure 1. Solid lines between rectangles (classes) represent object references (associations); dotted lines represent the "creates" relationship. The base components of the pattern are the abstract factory and abstract products, denoted A and B in this case. To use the pattern, a designer creates a concrete[1] factory as a subclass of the abstract factory. In the figure, two such classes have been designed: `ConcreteFactory1` and `ConcreteFactory2`. To accompany the new factory type, a concrete class is required for each product in the product suite. `ConcreteFactory1` would thus be designed in tandem with product types `ProductA1`, `ProductB1` and so on for other products. The different factories share a common interface, that designated by the

[1]Concrete meaning instantiable, i.e. non-abstract.

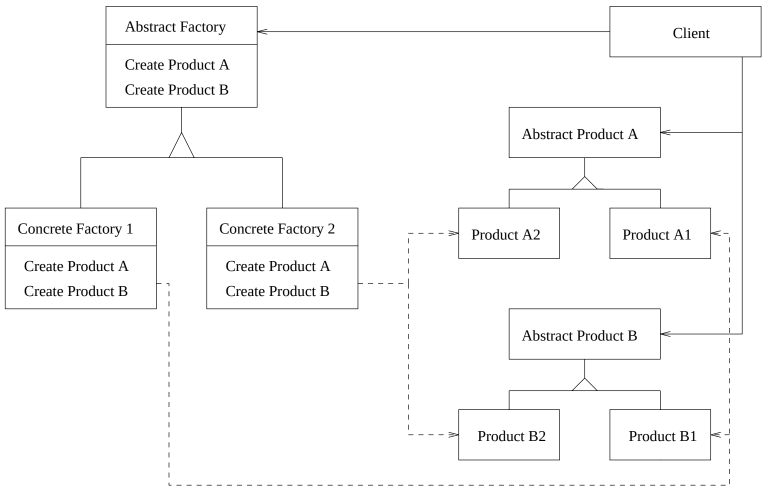

Figure 1: Abstract Factory Structure

AbstractFactory base. In the implementation of the operations, the concrete factories create and make available their particular product types. Client code references these products in terms of abstract product types only. Figure 2 shows this in practice; though the actual product suite is that emanating from ConcreteFactory1, product object use is through abstract product references (pointers in C++) only.

```
abstractFactoryUse()
{
  // no concrete product types visible

  AbstractFactory* absfac = new ConcreteFactory1;

  AbstractProductA* a1 = absfac->CreateProductA();
  a1->ProductAFirstOperation();

  AbstractProductB* b1 = absfac->CreateProductB();
  b1->ProductBOFirstOperation();

  AbstractProductA* a2 = absfac->CreateProductA();
  a2->ProductASecondOperation();
}
```

Figure 2: C++ Usage of the Abstract Factory Pattern

The practical contribution of this paper relates to integration of the envelope/letter idiom from [2] into the abstract factory pattern just described. This permits applications to use factory product objects without having to explicitly refer to the factory at all. This is achieved by shifting the concrete product instantiation call from application to library code. A consequence of this for C++ users is that the factory products become objects rather than pointers-to-objects. Firstly, this aids integration with operator overloading — this feature is clumsy to code if factory products are accessed via pointers. Furthermore, the technique improves memory management. In C++ [8], static object storage is handled well but heap storage is not (pointers are fragile in this respect).

The next section shows the abstract factory pattern in use; a threads factory example is introduced.

Usage of the pattern is discussed. Section 3 summarises Coplien's envelope/letter idiom. The abstract factory and envelope/letter patterns are combined in Section 4. Though factory product inner details become more complex, application code becomes simpler and more robust. The merits of the new architecture are discussed in Section 5. Conclusions are drawn in Section 6.

2 The Abstract Factory Pattern in Use

An example application area benefiting from the abstract factory pattern is that of a threads library — a collection of classes which encapsulates a so-called "threads package". Example threads packages include Solaris Threads and LinuxThreads. We are not concerned with the exact properties of these packages, e.g. lightweight vs heavyweight, only that we can identify certain well known and much needed components, e.g semaphores and monitors [9].

Our abstract threads factory provides an interface for creating thread-related objects. These might include thread objects themselves, plus condition variables, mutexes and semaphores. Figure 3 shows the abstract factory pattern applied to the threads package domain — concrete factories are given for `SolarisThreadsFactory` and `LinuxThreadsFactory`. Here, three product types are available: thread, mutex and semaphore.

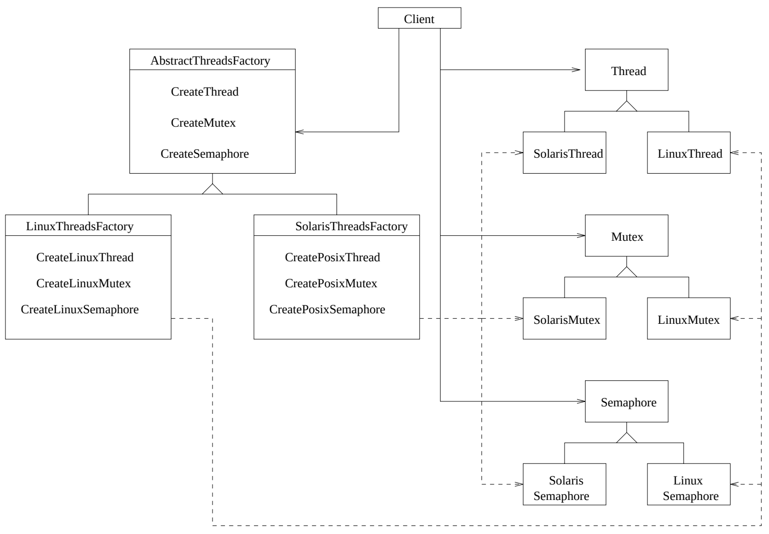

Figure 3: Abstract Factory Pattern Applied to Threads Package

Based on this class structure, application of the abstract factory pattern leads to code as typified in Figure 4. If the application is ported to a different platform, only the dynamic type of pointer `tf` needs to be altered, simply by instantiating a different factory type, e.g. `LinuxThreadsFactory`.

3 The Handle/Body and Letter/Envelope Idioms

The envelope/letter idiom [2] describes a relationship between classes intended to achieve greater flexibility than that offered by compile-time binding of objects to classes. This strong binding is featured in the "strongly-typed" object-oriented languages, e.g. C++ and Java [1]. Envelope/letter object pairs provide more polymorphism and dynamic type support than can be achieved with inheritance and dynamically dispatched (i.e. virtual) functions alone. The resulting programming style is more akin to

```
threadsApp()
{
  ThreadsFactory* tf = new SolarisThreadsFactory;

  Thread* t1 = tf->makeThread( funcPtr, funcArgs );
  t1->start();

  Mutex* m1 = tf->makeMutex();
  m1->obtain();

  Semaphore* s1 = tf->makeSemaphore();
  s1->signal();

  Thread* t2;
  t2->start();     // <-- using as yet undefined object!

  delete s1;
  delete m1;
  delete t1;
  delete tf;
}
```

Figure 4: Application Code Using the Abstract Factory Pattern

Smalltalk [4], where object-to-name bindings are dynamic in nature and compile-time type checking is eliminated.

The envelope/letter idiom is derived from the more general case of the handle/body pattern [2]. Both comprise a pair of objects, one of which only is visible by clients, and use delegation between the two to access the contained (invisible) object. The handle/body idiom is briefly reviewed here for completeness.

The handle/body class idiom is a technique useful in adding some extra behaviour to an existing, generally stable, abstraction. An example is adding reference counting capabilities to `String` objects to save on character copying operations. The body class maintains the "stringness", i.e. the string-related intelligence, while the handle class manages the reference counting facility. The general procedure for turning an existing abstraction, say `String`, into a body class to be handled is via a sequence of simple operations (from Coplien):

1. rename `String` to `StringRep`.
2. add a reference count field to `StringRep`.
3. create a new `String` class, the handle class, which contains a reference (i.e. pointer in C++) to a `StringRep`.
4. arrange for the handle class to delegate most of its operations, i.e. those not concerned with sharing, to the body class[2].

Figure 5 shows the relationship between the handle and body classes when using the idiom to perform reference counting on strings.

[2]Actually, this last point is non-trivial. Criticisms of its impact on reuse, along with a proposed solution, are given in [5].

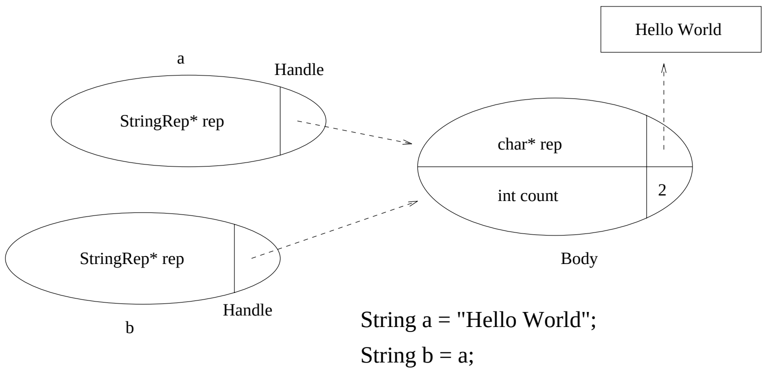

Figure 5: Handle and Body Classes for Reference Counting Strings

The envelope/letter idiom extends the handle/idiom further in that the two classes — the envelope class and the letter class — are related by inheritance, with the letter derived from the envelope. Application code deals exclusively with envelope objects. The intelligence is held in the letter objects, as is the case with body objects in the handle/class pattern. Letters are both contained and referenced from envelopes — we effectively achieve delegated polymorphism. Figure 6 shows the diagrammatic representation of the relationships involved in applying a `Number` abstraction to the envelope/letter idiom. The envelopes are objects `a` and `b`; application code accesses the objects only. The letters are instantiations are `Complex` and `RealNumber`, classes derived from the base `Number` class. When the envelope objects are created, the letters are also created, though this fact is hidden to the user.

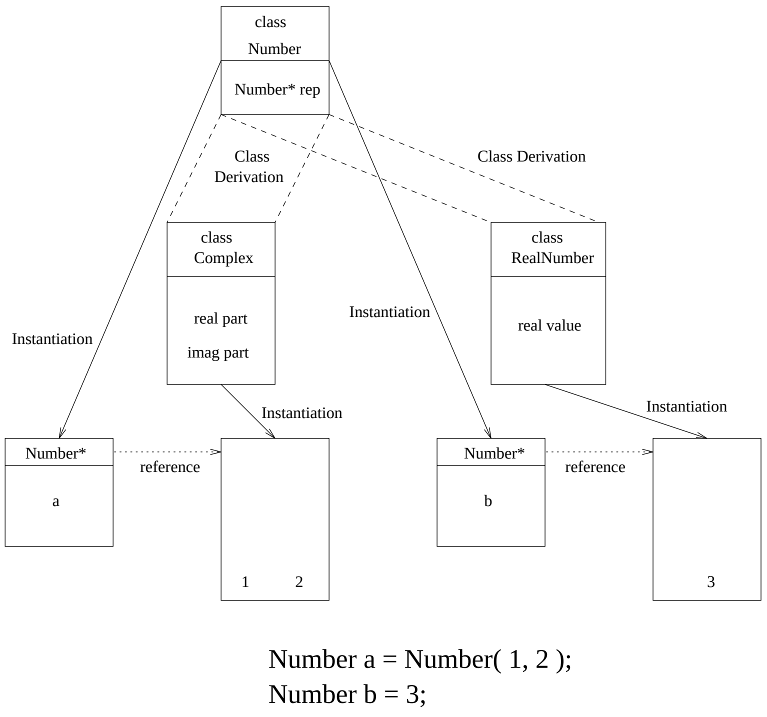

Figure 6: Structure of the Envelope/Letter Idiom

Figure 7 shows the C++ implementation of the idiom as applied to the `Number` domain described above. The `Number` constructor taking `DummyClass` as parameter is required to avert infinite constructor looping.

```
struct DummyClass {};

class Number {
public:
  Number() { rep = new RealNumber(0.0); }
  Number( double d ) { rep = new RealNumber(d); }
  Number( double r, double i ) {
    rep = new Complex( r, i );
  }
protected:
  DummyClass() { rep = 0; }
private:
  Number* rep;
};

class RealNumber : public Number {
public:
  RealNumber( double r ) : Number(DummyClass()) {
    value = r;
  }
private:
  double value;
};

class Complex : public Number {
public:
  Complex( double r, double i ) : Number(DummyClass()) {
    rpart = r; ipart = i;
  }
private:
  double rpart, ipart;
};

UseNumbers()
{
  Number a = Number( 1, 2 );
  Number b = 3;
}
```

Figure 7: Envelope/Letter Idiom Class Design and Use in C++

Applications thus use `Number` objects for all number types, e.g. `Complex`, `RealNumber`, etc. As operations are performed on these `Number` objects, the runtime type of the enclosed letter object may change. The envelope's behaviour thus appears to change throughout its lifetime. The simplification of user code to manipulate one abstraction only (class `Number`) rather than a family different number types improves that code. An important thing to note is that the envelopes are *not* pointers. Thus application code manipulates objects rather than pointers. Contrast this with the abstract factory pattern usage from Figure 4, where pointers, and by consequence and memory management operations, are prevalent.

4 Pattern Combination

The primary contribution of this paper is now presented. We *combine* the abstract factory pattern together with the envelope/letter pattern. We achieve an architecture in which the factory product objects are envelopes, delegating operations to letter objects which are also products. The notion of "abstract" product is eliminated. It is replaced by an envelope which is implemented so as to delegate all messages to its letter. The internal letter objects correspond to the concrete products in the basic abstract factory pattern. The pattern does not utilise the part of the envelope/letter idiom which suggests that the type of the letter can change based on the history of the enclosing envelope; in our pattern, the letter's type is fixed at its, and the envelope's, construction time.

The new combined pattern draws upon the envelope/letter idiom to construct the concrete product. However there is a slight difference between the two. In the original envelope/letter approach, the particular letter type created depended on the parameters passed to the envelope constructor. In the new case, the envelopes obtain the concrete products from another source — the abstract factory. There is a *uses* relationship [7] between all envelopes and the abstract pattern. We use a variation on the singleton pattern [3] to access the (unique) factory object.

From an application viewpoint, the described pattern combination produces the following results:

- the ease of maintenance and porting promoted by the abstract factory is carried over to the new pattern.
- save for factory object instantiation, no references are made to the factory. Letter (concrete product) object creation is localised, being within the corresponding envelope constructor.
- factory products are values (objects) rather than references (pointers). Client usage is therefore simplified both at the simple syntactic level and with regard to envelope management.
- possible inlining of messages to the envelope remove any performance cost introduced by the delegation from envelope to letter.

4.1 Class Design With Pattern Combination

Figure 8 shows a typical factory product, a `Thread`, written in C++ and conforming to the new combined pattern architecture. This class is the envelope for concrete thread objects such as `SolarisThread` or `LinuxThread`.

The primary constructor for `Thread` requests creation of a specialised thread object from the threads factory (the sole instance is accessed via the static `Instance` operation [6]). This new object then becomes the letter inside the new `Thread` envelope — this is the assignment to `rep` at line `***`.

All thread operations are delegated from the envelope to the letter, hence the forwarding of e.g `start()` to `rep->start()`. Use of inlining eliminates the expense of the extra function call. Inlining *is* possible since although the thread operations are virtual, all client messages are to thread *values*, not *references*. Such calls are statically compiled, hence the scope for inlining.

The destruction of a thread envelope (`Thread::~Thread`) takes care of freeing the associated letter. Care must be taken to ensure that when this same destructor is called on the letter itself (since it derives from the envelope, producing a destructor chain), the `rep` field inside the letter is left alone. This is set to zero during the letter construction, explaining the check for non-zero in `Thread::~Thread`.

Space overhead is simply that due to the pointer in the envelope which refers to the letter. However, any shared attributes of different concrete thread objects placed in the `Thread` base will also be unused in the envelope.

```
class Thread {
public:
  Thread( void* (*)(void*), void* = 0 );
  virtual ~Thread();
protected:
  Thread( DummyClass ) { rep = NULL; }
public:
  virtual int id() const { return rep->id(); }
  virtual void start() { rep->start(); }
  virtual int getPriority() const { return rep->getPriority(); }
  virtual void setPriority(int i) { rep->setPriority(i); }
  virtual int wait() { return rep->wait(); }
private:
  Thread* rep;
};

Thread::Thread( void* (*f)(void*), void* a )
{
  rep = ThreadsFactory::Instance()->makeThread( f, a ); // ***
}

Thread::~Thread()
{
  if( rep )
    delete rep;
}
```

Figure 8: Envelope/Letter and Abstract Factory Designs Combined

Subclasses of Thread represent the range of concrete Thread products created by different threads factories. Figure 9 shows the "combined pattern" implementation of a concrete SolarisThread type. The majority of the code (naturally) deals with the underlying Solaris API for thread manipulation. Two pattern points are relevant however. Firstly, the constructor has to call its superclass constructor which has signature Thread::Thread(DummyClass); this avoids the infinite loop and ensures the rep field in this instance is zeroed. Secondly, all operations have private access only. This precludes direct instantiation of the class — since the class structure is tightly integrated into the design pattern, this is the proper choice. The factory which would produce SolarisThread objects — Solaris_ThreadsFactory – is granted permission to create SolarisThreads via the friend mechanism. Though the regular operations are private, calls to these *can* be dispatched from the ancestor Thread envelope object.

Another threads factory product type, the Mutex, is tailored to the new pattern in Figure 10. The locked attribute is present in the envelope class Mutex since it is a common property of all concrete (letter) mutex classes, so has been propagated up to the common superclass.

The behaviour of factory objects in the combined pattern is unchanged from the original abstract factory pattern. Figure 11 shows a Solaris implementation of the threads factory in C++. Conforming to the interface specified by the abstract ThreadsFactory class, the factory creates and returns its own product suite: SolarisThread, SolarisMutex and SolarisSemaphore.

```
class Solaris_ThreadsFactory;

class Solaris_Thread : public Thread {
  friend class Solaris_ThreadsFactory;
private:
  Solaris_Thread( void* (*)(void*), void* = 0 );
  ~Solaris_Thread() {}
  unsigned id() const { return tid; }
  int getPriority() const;
  void setPriority( int );
  void start();
  int wait();
private:
  thread_t tid;
  void* (*startfunc)(void*);
  void* arg;
};

Solaris_Thread::Solaris_Thread( void* (*f)(void*), void* a ) :
     Thread( DummyClass() )
{
  int result;
  tid = 0;
  startfunc = f;
  arg = a;
  if( ::thr_create( NULL, 0, startfunc, arg, THR_SUSPENDED, &tid ) )
    ... error ...
}

void Solaris_Thread::start()
{
  ::thr_continue( tid );
}

int Solaris_Thread::wait()
{
  thread_t departed; int stat;
  ::thr_join( (thread_t)0, &departed, (void**) &stat );
  return stat;
}

int Solaris_Thread::getPriority() const
{
  int pri;
  ::thr_getprio( tid, &pri );
  return pri;
}

void Solaris_Thread::setPriority( int p )
{
  ::thr_setprio( tid, p );
}
```

Figure 9: A Concrete Thread Product In Letter Form

```
class Mutex {
public:
  Mutex();
  virtual ~Mutex() { if( rep ) delete rep; }
protected:
  Mutex( DummyClass ) { rep = 0; locked = false; }
public:
  virtual void lock() { rep->lock() locked = true; }
  virtual bool trylock() { return rep->trylock(); }
  virtual void unlock() { rep->unlock(); locked = false; }
private:
  Mutex* rep;
  bool locked;
};

Mutex::Mutex()
{
  rep = ThreadsFactory::Instance()->makeMutex();
  locked = false;
}
```

Figure 10: Mutex Class Tailored to Combined Pattern

```
class Solaris_ThreadsFactory : public ThreadsFactory {
public:
  Solaris_ThreadsFactory() : ThreadsFactory() {}
  ~Solaris_ThreadsFactory() {}
  Solaris_Thread* makeThread( void* (*f)(void*), void* a ) const {
    return new SolarisThread( f , a );
  }
  Solaris_Mutex* makeMutex() const {
    return new SolarisMutex;
  }
  Solaris_Semaphore* makeSemaphore() const {
    return new SolarisSemaphore;
  }
};
```

Figure 11: Concrete Threads Factory Behaviour

4.2 Application Structure With Pattern Combination

Complexity in the class design results in simplified structure at the application level. Example application code making use of the class designs described is shown in Figure 12. Factory instantiation uses the "employer" pattern [5], a variation on the singleton. In this example the target platform is Linux, hence the installation of the `Linux_ThreadsFactory` as the concrete factory object. Product objects such as thread and mutex are simple to instantiate and use, much more so than with the original abstract factory (e.g. in Figure 4). In fact the notion of factory has largely been eliminated.

```
Linux_ThreadsFactory fac;   // <-- instantiates concrete factory

threadsApp()
{
  Thread t( func, args );   // <-- constructor auto-builds
                            //     LinuxThread object in letter
  t.start();

  Mutex m;                  // <-- constructor auto-builds
                            //     LinuxMutex object in letter
  m.lock();

  // t and m destructors called
}
```

Figure 12: Using Enveloped Factory Product Objects

5 Merits

The new design pattern has a number of advantages but also some disadvantages. The advantages relate to application programs use of the factory products. The disadvantages arise from tight integration of the product abstraction with the pattern structure. This section provides a discussion of the combined patterns' merits.

5.1 Advantages

The biggest win using the additional envelope/letter concept is that of object lifetime management, i.e. object construction and destruction. With the value-based letter objects (`Thread`, `Mutex`, `Semaphore`), the following categories for error are eliminated:

- using an uninitialised factory product object (an example of which appears in Figure 4).
- introducing a resource (memory) leak by forgetting to apply the `delete` operator to a product object reference (pointer).
- unwanted sharing of product objects via pointer aliasing. Since products are objects, the class designer has more control over their use. Examples include defining the semantics of copy construction or assignment, discussed under the banner of "orthodox canonical class form" in [2].

In addition, the product object instantiation call is transferred from application code to the envelope constructor code, so localising the effect of any required changes in the factory's interface.

5.2 Disadvantages

To derive possible disadvantages of the combined pattern compared to the regular abstract factory pattern, consider the effort involved in adapting a set of existing classes C_1, C_2 $\ldots C_n$ to conform to the combined pattern described. Further denote the overall "functionality" of the task set by F. This task of adapting/re-engineering will be identified as beneficial when it is recognised that a target platform's native API implements the functionality F in a different way to that of the current system.

A typical example is the transferring of a GUI widget system ($F \equiv$ widget display capability) from say X-Windows to MS-Windows.

Under the abstract factory pattern, we require an abstract factory class, a concrete factory capable of producing objects of classes C_1, C_2 $\ldots C_n$ and an abstract class for each product type. These abstract product type are purely abstract in that no implementations are given for any of the supported operations.

Under the combined pattern, we again require the abstract and concrete factories, which would have behaviour identical to the case above. However, the abstract product types are no longer abstract. They become the (concrete) envelopes. As such, an implementation is required for every supported operation of every product type C_1, C_2 $\ldots C_n$. This can be a non-trivial coding effort, and at present would probably be performed through manual programmer effort.

6 Conclusions

In this paper we have discussed an integration of the envelope/letter idiom to the well-known abstract factory design pattern. This pattern is widely applicable, particularly if porting effort across platforms is to be minimised. A complication in factory product objects design, through the delegation mechanism between the envelope and letter object pair, reaps benefits for application programmers.

Drawing on experience through real use of the described threads factory products, the integration of the envelope/letter construct to the abstract factory pattern certainly produces a "friendly" or "non-intrusive" threads package interface for application programmers. This is especially true in C++ which has, by default, value semantics. The advantages of the architecture are reduced where reference semantics are the norm, e.g. in Java.

A useful addition to the work presented would be automatic derivation of the envelope class code for each product type, since this merely involves a request for letter object creation from the associated factory followed by direct delegation of every operation to the internal letter. This last point is an example of "context-oriented" design as suggested in [5].

References

[1] Ken Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley, 1996.

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

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

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

[5] Stuart Maclean. Contexts — Language Designs for Improved Reuse. in preparation.

[6] Stuart Maclean. On The Singleton Software Design Pattern. Technical Report DSSE-TR-97-4, Dept of Electronics and Computer Science, University of Southampton, September 1997.

[7] Robert C. Martin. *Designing Object-Oriented C++ Applications Using the Booch Method*. Prentice-Hall, 1995.

[8] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, second edition, 1991.

[9] Andrew S. Tanenbaum. *Operating Systems: Design and Implementation*. Prentice-Hall, 1987.