COORDINATOR

BUILDING ROBUST FRAMEWORKS WITH LESS INHERITANCE

michael@rhizomorphic.org

ABSTRACT

Object oriented languages allow one to model solutions in a variety of ways, from packaging state and algorithm together in pure objects to treating state and algorithm them as separate objects and everything in-between. As Cockburn puts it, „object technology is fundamentally a program-packaging technology that permits a continuum of function-only and function-plus-data packages“ [Cockburn98]. *What is “right“ in a particular case depends on the prevailing context, constraints and user preference.*

Inheritance is often used in solutions because object oriented languages directly support it. However experience has shown that inheritance may have been oversold and that such solutions suffer from a variety of ailments. Other solutions, based on composition are more complex to assemble and maintain and must use home grown mechanisms for survival but they are often more responsive to change. Intransigent monolithic inheritance structures are broken down and re-glued in a less stringent way. Function and data are separated and dynamically loosely re-coupled to provide extra flexibility and robustness (usability over time).

The framework presented highlights some of the issues involved. Instead of inheritance, recursive algorithmic decomposition and indirection are substituted to gain flexibility, to influence runtime behaviour and to achieve robustness in the face of change. These mechanisms which are resolved at runtime provide great freedoms and can be seen as a move away from the strait-jacket that compiled, static inheritance based languages often impose. But they are appropriate only in given situations, suffering their own kinds of problems: they are not a universal panacea. Surprisingly enough these mechanisms have their heritage in pre-Object solutions. Booch observes that these traditional techniques have their place in „bringing order to chaos“[1] [Booch94].

PATTERN TYPE

Object Structural Behavioural

DERIVATION

Composite, Flyweight, Strategy and Adapter [Gamma+95].

[1] „bringing order to chaos“ was surely the role of religion in other times. And in those times religious authority determined the meta rules for architecture.

OTHER READING

Reactor[Schmidt95] and ACT[Pyarali+97] for the EventHandler running example.
Composition arguments in [Szperski98] and in [Gamma+95,2].

CONTEXT

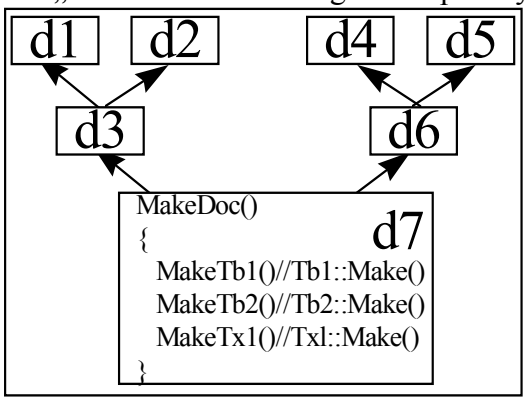
We are often faced with the need to provide a number of complex services which are similar in some ways. We generally seek to master complexity and factor out similarities in one way or another. Inheritance can be used, so can composition or any combination of the two.

EXAMPLE

1. **Report designer**

A report designer is used to design documents composed recursively of elements such as text blocks, tables and so forth. This document structure is edited with a WYSIWYG design interface. At runtime this structure is brought together with the actual information to be printed (held in business objects). The design GUI and printer module both use recursive descent to display or print out the document. Dynamic behaviour is desired. For example, one might design a particular kind of table but its actual size (over one or more pages) is governed by the amount of data to be presented. One might want to vary the document structure in some other way at runtime.
It is sensible to compose document building logic in a way analogous to the desired document, but this logic could also be modelled with inheritance. A cross section of sample Inheritance/Composition scenarios is shown below:

Figure 1 shows the extreme case of document building logic modelled as a monolithic object. All „d“ classes contain logic to explicitly build some elements of the complete document.



Where no object composition is used, we cannot use polymorphism to invoke document element Make()s because these exist only as separate classes not as separate objects.

Explicit construction calls only, throughout the heritage structure. Multiple inheritance alleviates some of the heavy binding a single inheritance solution would bring. Selected „d“ classes can be „mixed in“ to a number of documents.

Figure 1: Inheritance alone

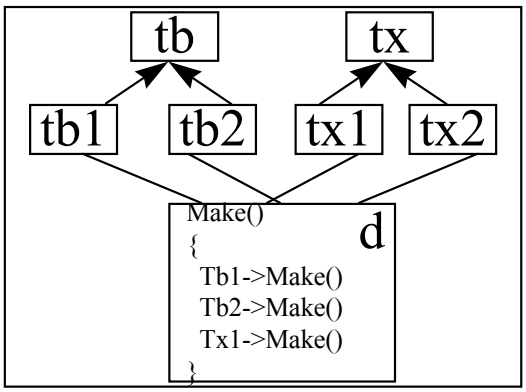


Figure 2: Inheritance/Composition mix

With an inheritance/composition mix (Figure 2) we can use polymorphism to invoke document element creation (all document elements inherit from a standard glyph class - not shown). The explicit serial sequence of hard coded document building calls is not generic and makes it difficult to model e.g. tables inside tables, and impossible to alter document building logic at runtime. Each different document must have its own class (subclass explosion).

The Make routines of Document Elements (e.g. tables) contain direct references to data in Domain Objects. So while 2 tables may be logically identical, their differing content or access to data leads to them being modelled as separate classes each with their own explicit Make (subclass explosion).

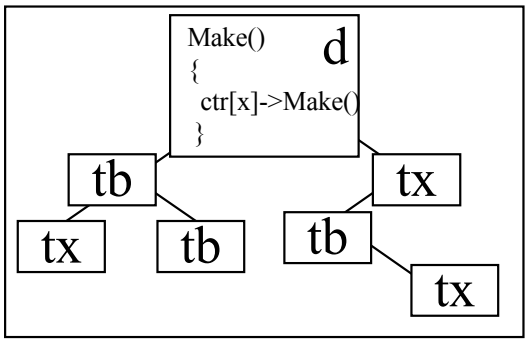


Figure 3: Composition alone

With most complete composition („inheritance on its side“) (Figure 3) the Make routine becomes generic (the same for all elements). It recursively calls Make on its members. Knowledge of document structure is thus distributed amongst the document elements. Tables inside tables can be modelled without a problem. What is more this structure can be dynamically changed. Subclass explosion is avoided by configuration. If you want a different document just plug together different logic building blocks.

To be pluggable, all document elements inherit from a standard class (not shown).We find we no longer need to model tables, text blocks in their own heritage hierarchies. The standard class with its polymorphic interface and composition suffices.

2. **Extensible EventHandlers**

In a message oriented middleware based on the Reactor pattern [Schmidt95] we are faced with the task of building a large number of EventHandlers. We want to be able to construct Event Handlers with the greatest of ease and efficiency. An Extensible EventHandler providing a service such as Customer Entry *can* model this service as a composite of the sub-services, ‘Check Permissions’, ‘Inflate incoming Object constellation’, ‘Check for existing customer’, ‘Create new Customer’ and these sub-services in turn may be assembled from finer grained elements. This algorithmic part, what is to be done, is brought together with the data to be acted upon, held in/via an ACT[Pyarali+97] object, at run time. EventHandlers have an added complexity: they must be able to reconvene services suspended by asynchronous calls to sub-service providers. In this paper the Coordinator is such an Extensible EventHandler.

PROBLEM

In changing systems, interface syntax and inheritance coupling is prone to breakage.

Object oriented languages provide an inheritance based approach to the solution of problems.

Gamma et al. [Gamma+95,2] and others have discussed the merits of inheritance and composition and the consensus is summed up in the second principle of Object Oriented design „favour composition over class inheritance“.
Although Object Orientation was supposed to be change friendly the Pandora's box of inheritance undermines this goal. Inheritance breaks encapsulation affecting reuse and, because of coupling (fragile base classes), inheritance based results are unwieldy and not change friendly, „unmanageable monsters“ [Gamma+95,2] result. Subclass explosion is another unwanted effect.
And still because it is seemingly provided *for free* a great amount of time is spent fashioning inheritance based solutions in such a way that they might survive change and offer up some kind of reuse.

Is there an easier way of building non-degradable systems?

FORCES

[numbered for comparison with resolution]

1. Class inheritance frameworks provided by mainstream object oriented languages suffer from problems associated with flexibility, longevity and subclass explosion. An alternative is needed.
2. The alternative must be extensible and usable in a variety of contexts. It must in some sense be a configurable meta-solution as inheritance is.
3. Dynamic extensibility - configuration must be changeable at runtime (otherwise inheritance suffices). I.e. Hot swapping of services.
4. Interfaces must survive change. Since we seek to provide solutions in all manner of contexts we cannot not realistically provide a fixed set of dummy behaviours to be filled in. We need to be able to make *any* number of services available - a kind of poor-man's reflection.
5. Fragile interface compilation (lengthy recompiles on interface change) in C++ leads to a hesitancy to alter interfaces unless absolutely necessary, hindering free flowing design increments.
6. Inheritance encourages reuse, we should provide fine grain reuse mechanisms too.
7. Can we exceed inheritance mechanisms and provide low level (code) reuse too?
8. Ease of use. The solution should not be markedly more complex to apply than inheritance.
9. Performance. Any solution should not be overly resource hungry.

SOLUTION

In areas were change leads to breakage adopt mechanisms that permit a greater degree of freedom.

Instead of using inheritance to realise a service adopt a composition based approach. Prevent ad hoc composed solutions by providing active framework mechanisms to encourage and control object composition in a uniform way. Place these active mechanisms inside a Coordinator object.

The framework is composed of state and algorithm now preserved in separate objects (Figure 4). Once separated they may vary independently (particularly important in asynchronous EventHandlers see Figure 12). Wholesale separation is not a requirement. Typically we separate non-core service/routing logic from fully-fledged business objects.

Internally the Coordinator acts as glue between algorithm and state, externally it provides an

unchanging (unbreakable) interface so that the service(s) it provides may be ascertained and invoked.

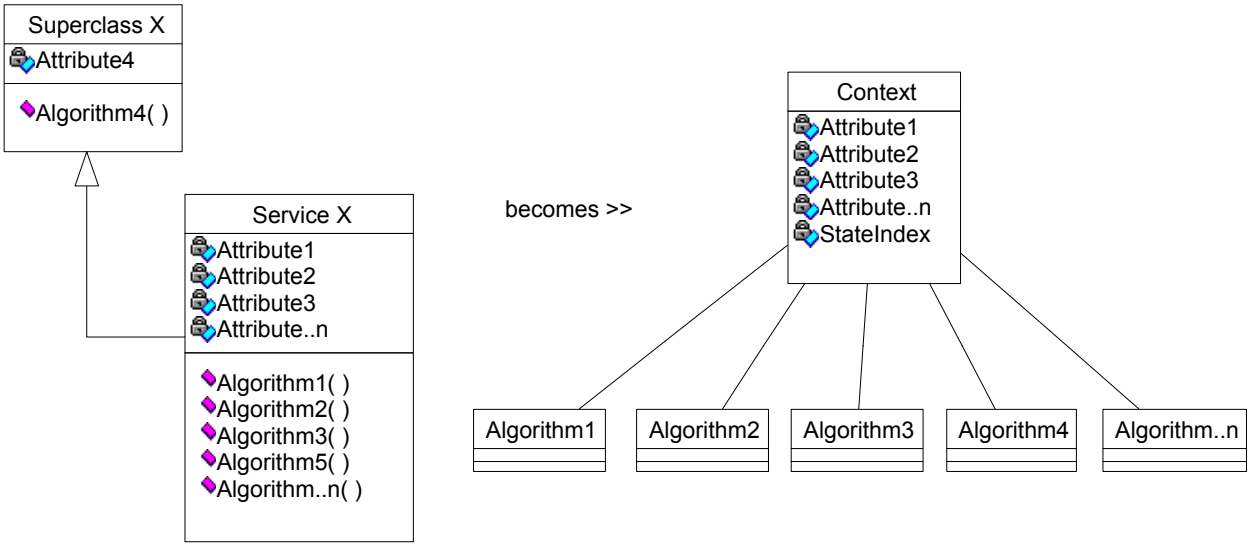


Figure 4: Separating context from algorithm

A Coordinator object is configured by plugging in algorithms which together combine to provide the complete service(s). Algorithms are themselves configured in a similar manner. Active framework policy itself is also an algorithm pluggin. Plugged in algorithms represent framework Hotspots [Pree95].

Participant interfaces never change syntactically: message content and its interpretation allow *any* behaviour to be modelled and changed - i.e. semantic change.

By modelling Algorithm as a composite the framework directly supports composition and reuse in a standard way. Algorithmic objects, instantiated *many times* appear in many algorithms hierarchies.
Further, low-level reuse is encouraged by modelling these algorithms as flyweights [Gamma+95]. Algorithmic objects, instantiated *only once* can appear in many algorithm hierarchies.

Use of interface indirection, generic algorithms and indirect access to state in a composed environment lacking inheritance coupling ensures robustness. By refusing to be specific in this way we can cope with change.

Use of builders and factories is made to make the job of the application developer as easy has possible.

APPLICABILITY

Use this framework in situations where change is likely and separation seems appropriate.
The flexibility of the framework compared to inheritance based solutions makes it particularly suitable to prototype building. Algorithm pluggins instead of subclassing and unchanging interfaces allow an active system to be up and running in record time from the roughest of designs leading to quick concept proofs.
One might modify prototypes built with the unchanging interfaces and use explicit interfaces for increased type safety once these have been fully established and matured.

Exhaustive use of these escape mechanisms may mean that the use of other more appropriate

languages or environments which provide the desired features built-in is indicated. Dynamic interfaces and dynamic inheritance are becoming available.

GEOMETRY

The composed objects, Reactor, Coordinator and Algorithms themselves, are each strong recursive centres of design. The diagram (Figure 5) shows only part of a system. A Reactor is configured by four Coordinators. Each Coordinator is in turn configured by Algorithms some of which themselves are composites.

This snowflake of centres can be nested in all dimensions. Algorithms can lead to Reactors which in turn are composed etc.

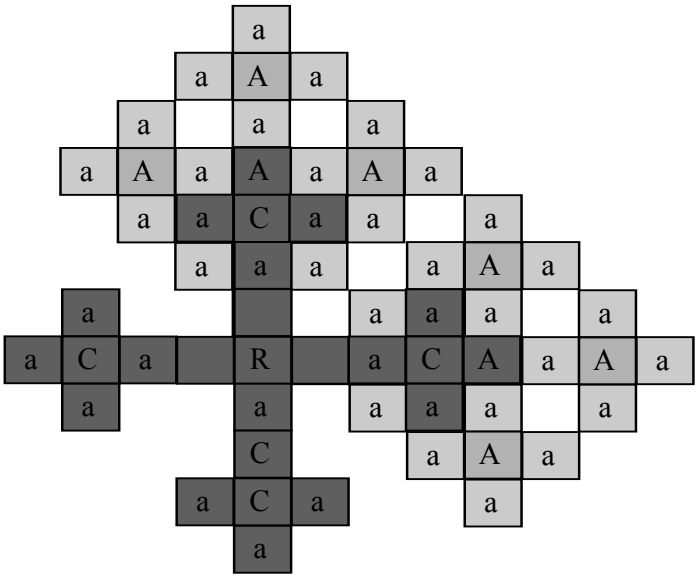


Figure 5: '3d' snowflake fractal of design centres

A = Composed Algorithm
a = leaf Algorithm
C = Coordinator
R = Reactor

STRUCTURE AND PARTICIPANTS

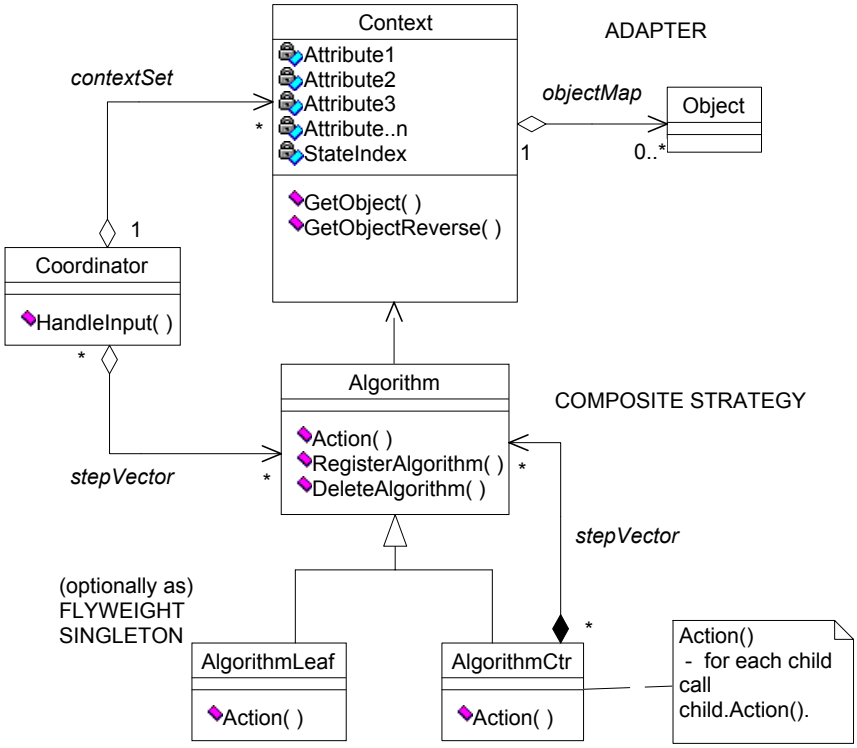


Figure 6: Structure - Class diagram - framework policies shown in text boxes

[Aggregations are given STL like names to indicate how they might be implemented. Note

that the inheritance shown is for polymorphism, not for reuse]

Figure 6 illustrates the following participants in the Coordinator pattern:

Coordinator Internally the Coordinator acts as glue between algorithm and state, externally it provides unchanging (unbreakable) interface so that the services it provides may be ascertained and invoked.

Framework build time An instance of Coordinator is created and subsequently configured to realise the services it is to provide. Configuration takes place by registering Algorithms with the Coordinator via a call to

```
RegisterAlgorithm(anAlgorithm:Algorithm*,
                        anAlgorithmType : int ) : void
```

The AlgorithmType is optionally used to group strategies together in certain ways. Coordinators are never subclassed, they are configured by algorithms.

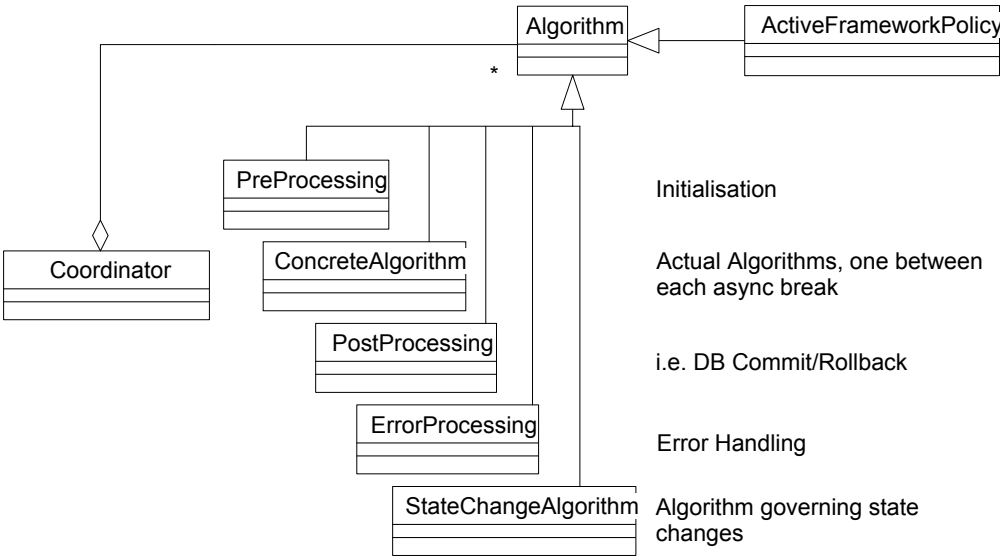


Figure 7: Coordinator - Typical EventHandler algorithm plugins

Framework run time An incoming service request is handled by the unchanging interface `HandleInput( aMessage:Message* ).`
The message passed is interpreted at runtime. If the service requested in the message is not provided by this Coordinator the request is suitably terminated. If the service is recognised the relevant algorithm is now invoked (this dispatching responsibility can be hived off to a Reactor like object) by the Coordinator.
The unchanging interface
`Action( aCoord:Coordinator*, anACT:Context*=0 ):int`
is called on one or other of the Algorithms held in stepVector.

Note that the exact actions of the Coordinator on message receipt are customisable too. This active framework policy is itself also an algorithm plugged in at runtime which governs how an incoming message is interpreted and which algorithms are invoked in which order as a result, and which algorithms to invoke in exceptional circumstances. This behaviour, critical to

the survival of the framework, is provided the framework developer.

Event Handler Active Framework policy example

In the EventHandler, Coordinator behaviour is modelled not to implement one or more services but instead to implement the various stages of a single multi-step service. These steps are the logic between various asynchronous calls to sub-services. Responses from such sub-services are routed to their originating Coordinator and the original service is reconvened with the next logical step.
A Reactor interrogates the incoming message and ensures it is passed to the correct Coordinator. The Coordinator then checks to see whether it recognises the incoming ACT(also in the message). If the ACT is not recognised, an error has occurred. If it is recognised, the `stateIndex` is incremented (actually a configurable state change algorithm is called) and control is passed to the `stepVector[ stateIndex ]` algorithm, together with this ACT which governs access to state.

Typical EventHandler framework policy called from `HandleInput(...)` is sketched below:

1	if new message	invoke pre-processing , create ACT/Context Object, set stateIndex to 1
2	if reply to outstanding asynchronous request	invoke StateChangeAlgorithm - typically increment stateIndex
3	<no condition>	invoke stepVector[stateIndex] *(switch to the relevant step of this service)*
4	if error arisen	invoke ErrorProcessing
5	if last ConcreteAlgorithm	invoke PostProcessing

Table 1: Coordinator - Typical EventHandler active framework policy

Algorithm

Algorithm is a Strategy composite which means it may be assembled of many like parts.

Framework build time

Assembly takes place by registering the algorithms via
`RegisterAlgorithm(anAlgorithm:Algorithm* ):void`
See also AlgorithmCtr below. All assembly is initially governed by the central Builder and Factory. Algorithms are never subclassed, they are configured by other algorithms.

Framework run time

An incoming request to the unchanging interface
`Action( aCoord:Coordinator*, anACT:Context*=0 ):int`
is passed recursively down the algorithm tree. Exceptional situations are handled by exceptions, errors are handled by having Action return non-zero leading to automatic unwinding of the stack.
The second parameter, `Context*`, allows access to certain special state information (such as stateIndex) held directly in the Context object and allows indirect navigation to extrinsic state.

Algorithm Ctr

AlgorithmCtr is the node object in the Algorithm hierarchy, the *glue* between all Algorithm parts. This link information is decoupled from the Algorithm leaves because it represents extrinsic state. Algorithms are kept „dumb“ to permit reuse.

Immutable algorithms which have no extrinsic state may be placed in common algorithm pools to be freely shared amongst many algorithm hierarchies. Such reuse can not be modelled with inheritance.

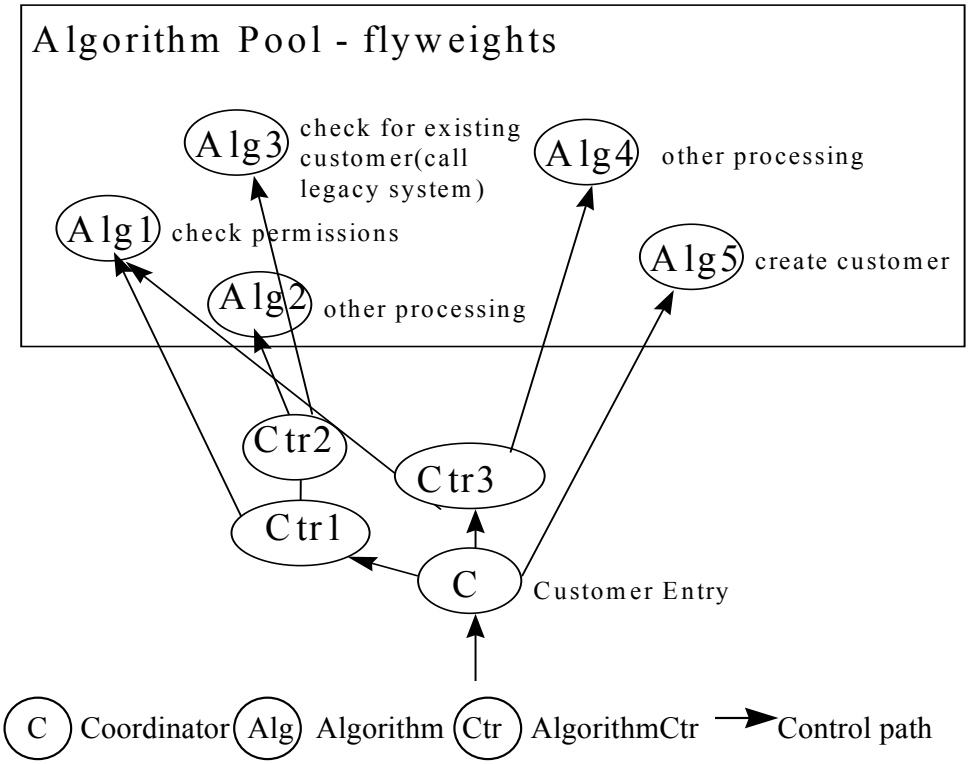


Figure 8: Pools of code reusable algorithms - typical EventHandler example

Figure 8 shows a Coordinator with three services, two of which are composed (Ctr1 and Ctr3).
AlgorithmCtrs do not appear in algorithm pools, they are held with each Coordinator. AlgorithmCtrs are never sub-classed, they are composed at runtime to contain the relevant Algorithms.
Algorithms should be built cooperatively with input from all application developers involved in implementing a particular group of services. A registry of Algorithms would be desirable to avoid repeated coding and garner full reuse benefits.
Algorithm flyweight pools may have system-wide or sub-system scope.

Framework build time Algorithms or AlgorithmCtrs are registered with the current AlgorithmCtr via `RegisterAlgorithm(anAlgorithm:Algorithm* ):void`

Framework run time `Action( aCoord:Coordinator*, anACT:Context*=0 ):int` simply recursively calls Action(..) on each container member.

Context Holds control information and allows the dynamic addition of and provides generic access to data of *any* type using name/value pairs. E.g. an STL multimap or a Java Hashtable holding a key and objects of type Object. All objects to be stored must have this Object wrapper (Adapter Pattern) or a basic Object class must be mixed in to their heritage. The Context never knows

what kind of objects it is actually dealing with, it merely stores and retrieves them. GetData... methods are parameterised to return the desired object. This kind of wrapped generic data access is reminiscent of variant records.

Generic access is an issue as soon as low level reuse is practised since all extrinsic state must then be removed from algorithm. Anytime a flyweight algorithm „X" is reused anywhere in a algorithm hierarchy it will access its extrinsic state in the *same* way. We can make no assumptions about the order in which data is stored or the order in which sharable algorithms may be used.

In the EventHandler the *address* of the Context object can be the ACT.

Context has three roles:

1. Indirect parameter passing

Parameters are usually passed between objects directly but all Action() calls have the same signature. The passing of any kind of data is made possible by „hiding" the actual parameters behind the Context object which is passed instead.

2. Separation, access to state.

In certain cases (low level reuse) we separate all extrinsic state from logic. Such state is accessible via the context object.

3, Preservation.

In the EventHandler example, each Coordinator has many *different* context objects. These, used alongside the algorithms, allow an EventHandler to manage asynchronous breaks.

Builder and Factory

(not shown but well known)
Classic use of builder and abstract factory classes are made to initially configure the system. The use of several nested loops is made to automatically build the framework from the outside in.
E.g. Reactor, made up of Coordinators, made up of Algorithms
At each level products are delivered by the factory until it has no more.
Alternatively incoming messages can be used to configure the system on the fly, requests being used to order services from the factories. The services thus built could be subject to a „least recently used" unloading scheme.

COLLABORATIONS

The Collaboration and Implementation sections include references to a Reactor which is always present in the EventHandler example but not necessarily present in all solutions. A Reactor is an initial dispatcher which routes messages for a particular sub-system from a request queue to the relevant Coordinator.

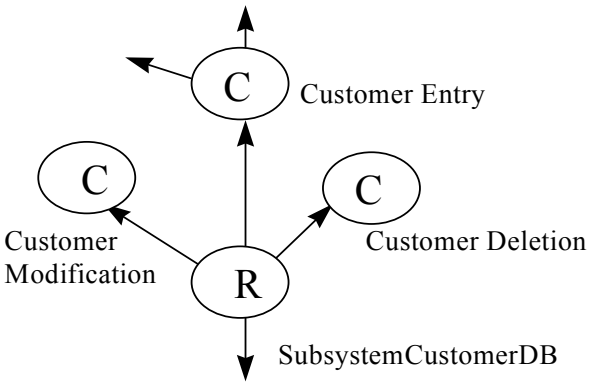


Figure 9: Reactor dispatch to relevant Coordinator (see also Figure 8)

As seen in the Structure and Participants section there are two phases; „build time“ when the framework is assembled and „runtime“ when control is passed to the assembled framework (see also [Marquadt98]).

Framework build time:

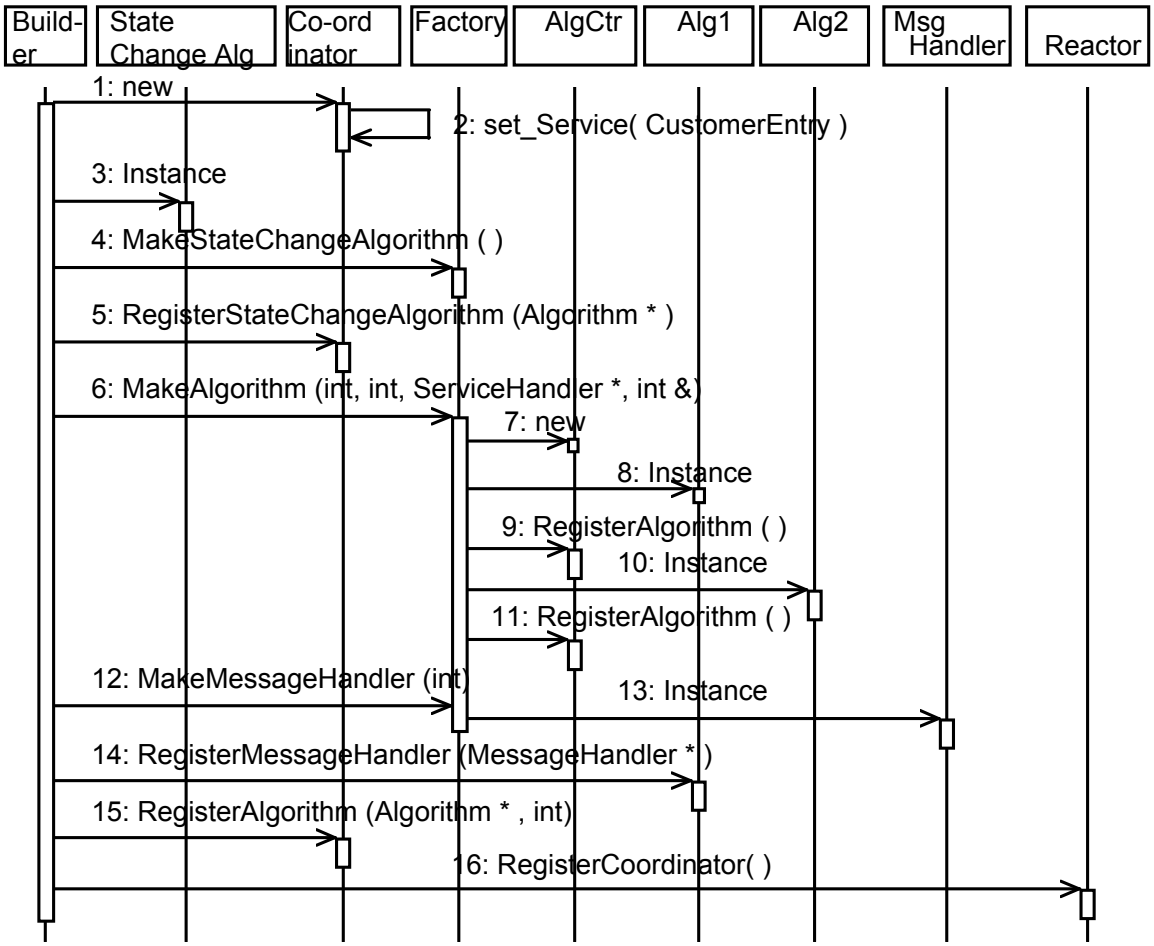


Figure 10 Builder - Framework/Coordinator assembly, EventHandler example with Reactor

Figure 10 shows creation of Coordinator for a Reactor. Steps 6 to 15 make up the inner Algorithm creation loop which is repeated until the factory delivers no further products for

the current Coordinator. Coordinators are created in the outer loop, steps 1-5 and 16.
Because everything bar the algorithms themselves are configured and not subclassed building is carried out by code that never changes.

Peculiar to the EventHandler example:

- each Algorithm may have an attached MessageHandler which inflates incoming messages into objects using the relevant schema. This handler could just as well be an algorithm too (steps 12-14).
- Each Coordinator has its own ChangeStateAlgorithm (generally the default) which governs how the EventHandler switches to which state following the receipt of a response to an outstanding sub-service request
- Each Coordinator is registered with the Reactor (Reactors are built in a loop around the Coordinator assembly shown above) at the end of the outer loop (step 16).

Framework runtime:

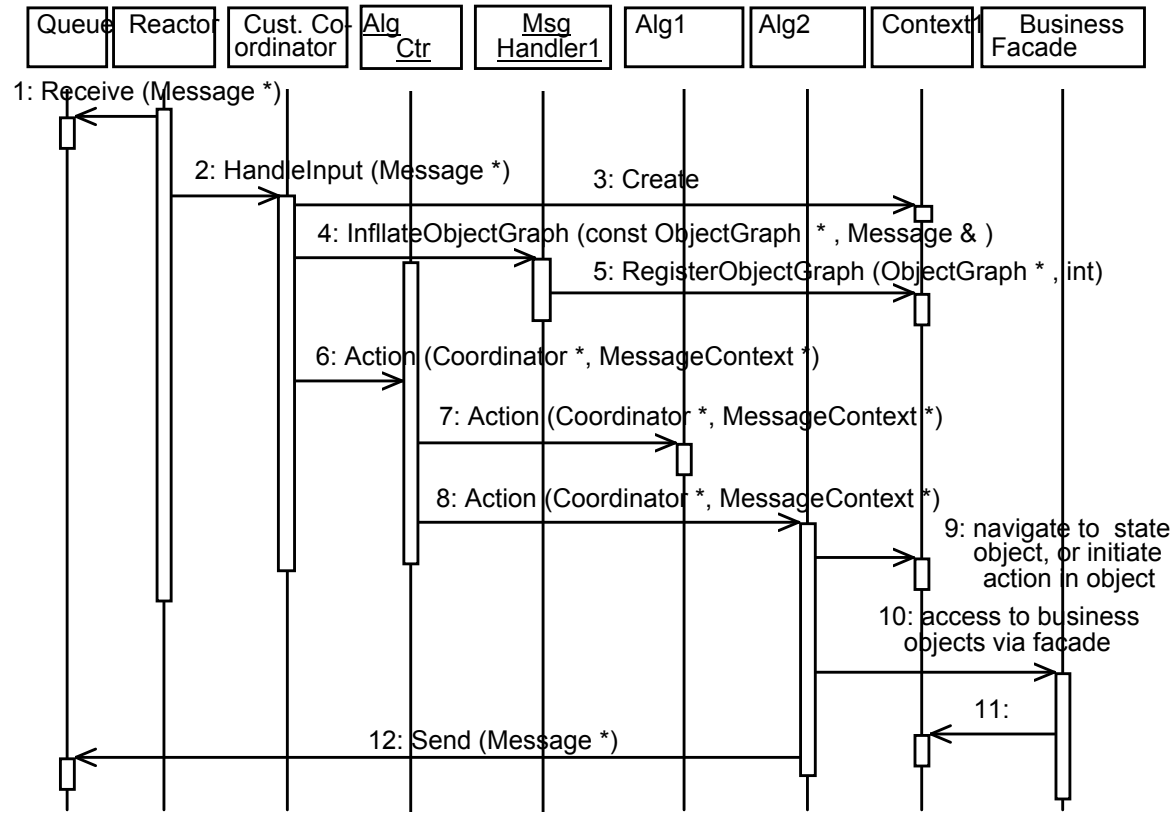


Figure 11 Framework runtime, EventHandler (Coordinator) with Reactor, Queue ..

Figure 11 shows typical Coordinator behaviour in the context of Reactor/EventHandler, but Coordinator behaviour always proceeds along these lines.
In step 2 a service request is made of the Coordinator.
Steps 6 to 12 show this service being carried out with a recursive `Action(...)` call to the algorithm structure concerned. Steps 9 and 10 shows how from Algorithm 2 we access state (context) and invoke further actions in or extract data from business objects.
Peculiar to the EventHandler example:

- in step 3 the incoming message is found to have no ACT so one(here called MessageContext) is created to govern future processing.
- the attached MessageHandler inflates the incoming message and places the result in the object container of the context object.

- In step 12 a request to a sub-service or a response to a client is sent. The message to be placed in the queue is assembled by the algorithmic code.

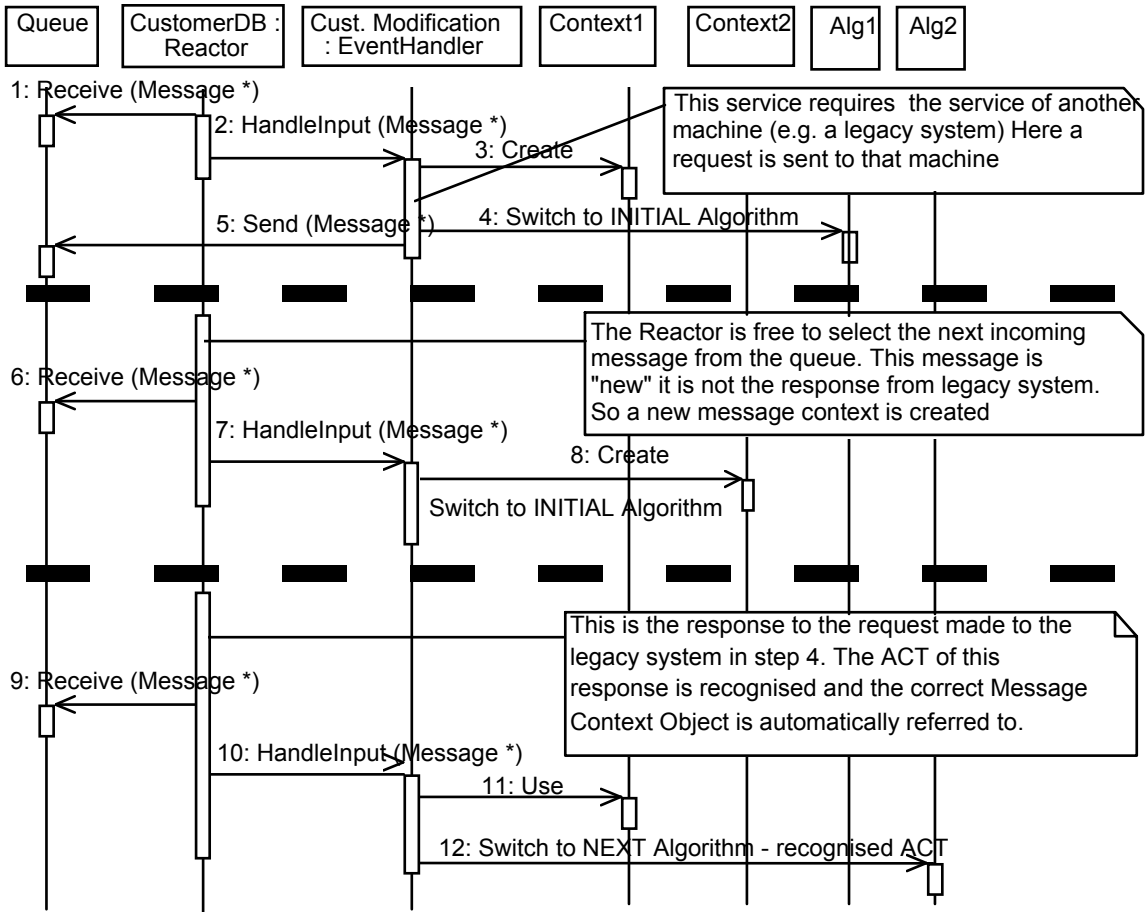


Figure 12: Services do not block, Algorithms in the Event Handler are reused directly

- - - - - - - - =Asynchronous message break

Figure 12 shows how in EventHandlers it is necessary for Algorithm and Context to vary independently. Context is stored between message breaks and Algorithm proceeds to work with a different Context (Step 5). In steps 9-10 the previous context is brought together with a new Algorithm.

ROLES

- **Framework developer -** Leverages this framework into the design, configures the interfaces and determines one or more algorithm pluggins to provide framework policy. This policy should be as active as possible to relieve the application developer from repeated error prone programming and to provide stability. Ideally the framework has default behaviour. Provides builders and factories to produce specific products for framework assembly which the application developer modifies to configure a specific application.
- **Application developer** (framework user) - Uses the framework to realise the end application. Configure the framework with algorithms of his/her own. Does not program the framework policy algorithm pluggin.

RESOLUTION OF FORCES

1. **Alternative to class inheritance** - Object composition is known to relieve certain inheritance problems. A Coordinator supports composition, provides external interface, mediates between state and algorithm and encapsulates framework policy. Configuration avoids subclass explosion.
2. **Extensible meta-solution** - Instead of subclassing, algorithms (including the framework policy itself) are plugged in to the Coordinator object at run time.
3. **Runtime configuration** - Plug in configuration is completely dynamic. E.g. by plugging a new algorithm into the framework at runtime we can perform a new database permissions check.
4. **Configurable meta interface** - The Coordinator uses syntactically unchanging message interfaces backed up by plug in interpreters. The desired service is ascertained from information in the message. The Coordinator either carries out this service or returns the request to be handled by another Coordinator.
5. **Fragile interface compilation** - This problem is circumvented because interfaces never change.
6. **Fine grain reuse mechanisms** - Algorithms themselves are composed of sub-Algorithms. This mechanism can be used to provide design reuse.
7. **Low level (code) reuse** - Removal of extrinsic state from Algorithms render their code directly reusable. Such algorithms must be able to access such state (accessible via the Context object) in a generic way.
8. **The solution must be easy to use** - Builders and factories are provided to automatically assemble all the different Coordinator objects at runtime. The application developer need only configure the builder and factory correctly, assembly from these parts is then automatic. The framework itself serves to reign in the increase in complexity caused by the proliferation of objects resulting from the composition approach.
9. **Performance** - The indirection mechanisms and navigation to objects in the composed solution (especially generic navigation to data) is costly in time terms. This can be mitigated by reducing the number of objects using the low level reuse mentioned in 7 above. Used optimally, this solution can completely avoid the unwanted baggage of repeated logic in unneeded subclasses (classes identical but for the data they access) and overweight objects resulting from ageing/sub-optimal inheritance. In other words overall code-footprint is markedly reduced positively affecting time performance.

CONSEQUENCES

The Coordinator is sadly not a cure all. It suffers from its own various ailments some of which (e.g. Stove pipe architectures) object orientation specifically solves.

+

- By adopting separation, low-level object algorithm reuse becomes possible. Swapping/thrashing inefficiencies due to the repeating of algorithmic code in many hundreds of algorithmically identical objects is avoided.
- Easy to get solutions up and running from minimal design. Complex design decisions resulting from inheritance complexity can be postponed or even completely by-passed.
- Does not suffer interface fatigue.
- Robust.
- Fewer subclasses because of configuration.

- Since we can mix together what we need at a particular time we are unlikely to be confronted with legacy inheritance structures offering hundreds of methods, only ten of which are still relevant.
- Extensibility: Any of the algorithms plugged into each Coordinator at framework build time can be varied at runtime. By mixing algorithms into new algorithm composites a certain extensibility without the need for recompilation is attained. With C++ one can place the relevant factories into DLLs to be dynamically swapped in and out of the running system. Other languages offer built-in support for varying degrees of extensibility.

-

- Reversion to stove-pipe architectures, global data traffic: Wherever separation is used, algorithm has the added overhead of needing to navigate dynamically to its context and this is obviously less efficient than if context were classically available. System complexity bemoaned elsewhere in inheritance based designs now re-emerges in this navigation.
- Indirection leads to slowness partly offset by the reduced code footprint and associated potential need to swap.
- More objects, partly offset by use of singleton algorithms.
- Reduced type safety. The generic interfaces enforce no type checking. The application developer can introduce some form of Run Time Type Information checking, but for some domains this will be „too late".
- Use of complex message content and simple unchanging interfaces makes it difficult to ascertain run time behaviour at a glance. A typical run time stack containing a number of nested Actions() is less informative than a stack with more traditional specific interfaces. So debugging is difficult. Debugging inheritance based solutions can be nightmarish too of course.

IMPLEMENTATION

The Framework developer rolls out a factory class (to be configured by the Application developer) and algorithms governing framework policy and StateChangeAlgorithm(s).

The Application developer must configure this framework. The delivery of the following products from the factory must be ensured:

- All the Algorithms that make up a service. These Algorithms generally hold logic to de/code (interpret) Messages and the Service logic itself.
- The ServiceEnums.h file.

The framework can be used immediately with just these products.

The following can be optionally supplied:

- One or more customised dispatch Algorithm(s) for the Reactor.
- One or more customised switch Algorithm(s) for the Coordinator.
- One or more customised timeout Algorithm(s) for the Coordinator.
- One or more customised StateChangeAlgorithm(s) overrides for the Coordinator.
- Other algorithms...

Development of Coordinator and Algorithms can generally proceed unilaterally. However there are some central locations where development must be coordinated.

Other issues to bear in mind when during design and implentation are; problems of killing the algorithm singletons, handling of system crashes, suspend/resume.

SOURCE CODE

Framework configuration:

Each Coordinator has a ServiceId used in dispatch to and checking of services at runtime. Messages will contain this Id when making a request of a service. The file **ServiceEnums.h** holds all service Ids:

```
// Note: 0 used for error condition in framework building
enum theCustomerServices { CustomerEntry = 1, CustomerDeletion,
                                    service3, etc. };
```

This file details all services provided by the/all Reactor(s). Many factories may conspire to build the complete framework so these service keys must be unique. The SubsystemBuilder should check key uniqueness.

Application developers configure the Factory class (delivered by the Framework developer) that is used to obtain developer products when building the Framework. Configuration proceeds by adding lines of code to the Factory class code in a set way. These lines serve to register Application developer products with the framework. Development of this class must be coordinated with the developers of other services and their Algorithms. Only the MakeAlgorithm method is affected:

```
Algorithm *
FactorySample1::MakeAlgorithm
(

      int numberCoordinator,
      int numberAlgorithm,
      Coordinator * aCoordinator

) //const
{

      switch (numberCOORDINATOR)
      {
      case 0:
            aCoordinator->set_theService( CustomerEntry );
           // set GENERAL pre/post and error strategies, may be
          // overidden
            aCoordinator->RegisterAlgorithm(
                PreAlgorithm::Instance(), PreEnum );
            aCoordinator->RegisterAlgorithm(
                PostAlgorithm::Instance(), PostEnum );
            aCoordinator->RegisterAlgorithm(
                TransactionErrorAlgorithm::Instance(), ErrorEnum );

            if (numberAlgorithm == 0)
            {
A...             result = SampleAlgorithm1::Instance(); //customise
            }
            else if (numberAlgorithm == 1)
            {
D...             // Be careful to avoid recursive loops.
                 // Here composite 2 container objects are created.
                 // If the same containers are used many times the
```

```
                // application developer may opt to
                  // build his/her own custom container classes
                // instead of building them dynamically here...

                  result = new AlgorithmCtr;
                  Algorithm * theAlgorithmCtr2 = new AlgorithmCtr;

                  // the container objects are filled with leaf
                // products or other containers
                  // the order of registration is the order in which
                // sub-transactions will be called.
                  result->RegisterAlgorithm(
                            SampleAlgorithm1::Instance() );
                  result->RegisterAlgorithm( theAlgorithmCtr2 );
                  result->RegisterAlgorithm(
                            EndAlgorithm1::Instance() );

                  theAlgorithmCtr2->RegisterAlgorithm(
                            SampleAlgorithm2::Instance() );
                  theAlgorithmCtr2->RegisterAlgorithm(
                            SampleAlgorithm3::Instance() );
            }
            else if (numberAlgorithm == 2)
            {
                  result = SampleAlgorithm3::Instance();
            }
            else if (numberAlgorithm == 3)
            {
                  result = EndAlgorithm1::Instance();
            }
            break;
      case 1:
            aCoordinator->set_theService( CustomerDeletion );

            if (numberAlgorithm == 0)
            {
B...               // if SampleAlgorithm1 has already been created it
                // is reused - singleton.
                  result = SampleAlgorithm1::Instance();
            }
            else if (numberAlgorithm == 1)
            {
                  result = new AlgorithmCtr;
                  Algorithm * theAlgorithmCtr2 = new AlgorithmCtr;

                  result->RegisterAlgorithm(
                              SampleAlgorithm1::Instance() );
                  result->RegisterAlgorithm( theAlgorithmCtr2 );
                  result->RegisterAlgorithm(
                              EndAlgorithm1::Instance() );

                  theAlgorithmCtr2->RegisterAlgorithm(
                              SampleAlgorithm2::Instance() );
                  theAlgorithmCtr2->RegisterAlgorithm(
                              SampleAlgorithm3::Instance() );
            }
            else if (numberAlgorithm == 2)
            {
                  result = SampleAlgorithm3::Instance();
            }
            else if (numberAlgorithm == 3)
            {
                  result = EndAlgorithm1::Instance();
            }
      break;
      case 2:
```

```
            aCoordinator->set_theService( Service3 );

            if (numberAlgorithm == 0)
            {
C...              result = SampleAlgorithm1::Instance();
            }
            break;
      default:
            cout << "ERROR ? Make Algorithm - Default " ;
            result = 0;
            break;
      }

return result;
}
```

In all „Make“code grey areas are those to be customised by the application developer to deliver the desired product. **No other lines should be altered.**
The Factory above produces three services; CustomerEntry and CustomerDeletion and Service3.
C.. shows an example of a Service with a single Algorithm.

Some algorithms are flyweight singletons. For example: SampleAlgorithm1 is referenced and used by both CustomerEntry and CustomerDeletion (highlighted as A.. and B.. above).
D... shows the assembly of a composite Algorithm from sub-components.

Framework build:
The products of the Factory are automatically hooked into the Framework at runtime by the SubsystemBuilder class. Each Subsystem is built by iterating generically through the supplied factory. Figure 10, shows how the framework is assembled by calls to the factory. As with most factories, product ownership is given to the framework. The developer should not delete these products.

Framework run:

```
main()
{
      // This call leads directly to creation of the Reactor. The
      // Reactor is never subclassed.
      // Call also leads to assignment of initial Algorithm to the
      // reactor
      Reactor::Instance()->set_theDispatchAlgorithm(
                              DispatchAlgorithmServer::Instance());

      // This call also leads to creation of all Coordinators
      // and their registration with the Reactor and to creation of
     // all Algorithms and registration with the Coordinator
      SubsystemBuilder::Instance( FactorySample1::Instance() );
      // many factories may be used to build the framework
      // In this way sensible factory groupings may emerge
      SubsystemBuilder::Instance( FactorySample2::Instance() );


      //pass control to the reactor
      Reactor::Instance()->Dispatch();
}
```

This main() shows how the Reactor is assigned its DispatchAlgorithm and the call of SubsystemBuilder with a Factory to build the Reactor/Coordinator/Algorithm constellations for this Subsystem. These two calls serve to completely assemble the Framework. The

subsequent Reactor Dispatch() call passes control to the freshly built framework. Because the Reactor, Coordinators and Algorithms are configured in turn by each other assembly is simply generic.

ACKNOWLEDGEMENTS

Thanks to Christa Schwanninger for her guidance and to the Europlop 98 workshop members for their insights, helpful comments and encouragement.

KNOWN USES

These mechanisms have been applied in a Reactor based Communcation Framework for a mission critical reservations system serving 30,000+ clients - used as sketched in the Extensible EventHandlers running example. Also used in a Report Designer/Renderer - used as sketched in the running example.
Workflow management products glue third party black boxes together across WANs. Data is passed from application to application in tokenised form, interpreters serve to convert the input/output data to the expected format for each application. These frameworks are powerful because they are non-specific.

RELATED PATTERNS

Overall the Coordinator patterns can be viewed as a generalisation of the Pipes and Filters pattern [Buschmann+] and [Meunier95].
The „Do-it-yourself reflection“ [Sommerlad98+] patterns to build flexible systems deal with issues similar to those presented here.
Algorithm is an example of the Strategy, Flyweight, Composite and Singleton patterns.
Issues involved in context separation are discussed in State Object [Dyson+97].
The use of three classes (a gateway (Coordinator), data and algorithm logic) is typical of algorithm solutions and can be observed for example in „Algorithm design by patterns“ [Galve-Francés+98].

REFERENCES

[Booch94] Booch, „Bringing order to chaos“. Object-Oriented Analysis and Design, pp 16-21.

[Buschmann+] Buschmann, Meunier, Rohnert, Sommerlad, Stal „Patterns of Software Architecture“

[Cockburn98] Cockburn, „What is not suited“, Surviving Object Oriented Projects, Addison-Wesley 1998 pp 36

[Dyson+97] Dyson, Anderson „State Patterns“. In Martin, Riehle, Buschmann (eds) *Pattern Languages of Program Design 3*, Addison Wesley 1997 pp -125,142

[Gamma+95,2] Inheritance v. Composition. In Gamma, Helm, Johnson, Vlissides, *Design Patterns:Elements of Reusable Object Oriented Software*, Addison Wesley 1995 pp -17,19

[Galve-Francés+98] Galve-Francés, García-Martín, Burgos-Ortíz, Sutil-Martín. *„An approach to algorithm design by patterns“.* Presented at the Europlop Conference, Kloster Irrsee 1998.

[Gamma+95] Gamma, Helm, Johnson, Vlissides, *Design Patterns:Elements of Reusable Object Oriented Software,* Addison Wesley 1994

[Marquadt98] Klaus Marquadt, The Advent Pattern in *„Physical Patterns“*. Presented at the Europlop Conference, Kloster Irrsee 1998.

[Meunier95] Regine Meunier, *„Pipes and Filters Pattern“,* In Coplien, Schmidt (eds) *Pattern Languages of Program Design 1*, Addison Wesley 1995 pp -427,440

[Pree95] Pree, Design Patterns for Object Oriented Software Development, Addison Wesley 1994, Chapter 4, Metapatterns.

[Pyarali+97] Pyarali, Harrison, Schmidt, „Asynchronous Completion Token“. In Martin, Riehle, Buschmann (eds) *Pattern Languages of Program Design 3*, Addison Wesley 1997 pp -245,260

[Schmidt95] Schmidt, „Reactor“. In Coplien, Schmidt (eds) *Pattern Languages of Program Design 1*
Addison Wesley 1995 pp -529,546

[Sommerlad98+] Sommerlad, Ruedi. „Do-it-yourself reflection“. Presented at the Europlop Conference, Kloster Irrsee 1998.

[Szperski98] Szperski „Connection oriented programming“. In Szperski *Component Software.*.Addison-Wesley/ACM 1998 pp-148