# Essence Pattern

Andy Carlson

4 July 1998

## Abstract:

Many classes, particularly persistent ones, require that a certain subset of their attributes be valid before a given instance can be considered valid. How can this be guaranteed in component based or distributed environments where the client which creates the instances is outside our design control?

Traditional solutions involve long lists of parameters to be supplied by the client at creation time or allowing an invalid object to be created and using a separate validation step. This paper examines situations in which these solutions are not desirable or acceptable and presents an alternative solution which can be more generally applied.

Copyright © 1998 AT&T. Permission is granted to make copies for PLoP 98

## Pattern Name: *Essence*

### *Aliases:* none

### *Context*

You are building an information system in which objects have a known set of properties (attributes and/or relationships) each of which can have a meaningful value or not (e.g. can be in an uninitialized state, absent altogether, have a null value or refer to a *Null Object* [WOOLF97]). For some classes there is a subset of properties which must always have a meaningful value (compulsory properties). You are considering how to support creation of instances of these classes by clients which may not be under your design control (for example CORBA [OMG] or ActiveX clients).

### *Problem*

Many classes have 'compulsory' properties which must be specified in order for an instance to be in a valid state, for example, before having attributes used as keys in a dictionary or database index. We would therefore like to force the clients of such classes to specify the compulsory properties when creating new instances. Sometimes we also need to ensure that these properties cannot be altered at any time after creation.

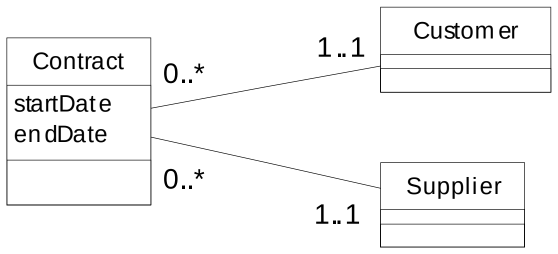

**Figure 1** Contract class with optional and compulsory information.

For example, consider model in Figure 1. When a Contract is created, both a Supplier and a Customer must be specified and these cannot be changed after creation. Start time must also be specified at the moment of creation although this may later be delayed or brought forward. End time is completely optional as the contract may be 'open ended'.

Typical solutions to this problem are:-

1. Avoid the issue at object creation time by using a zero argument constructor and providing public 'set' methods for all properties. Pick some later moment at which a separate validation step can be performed.
2. If the implementation environment permits it, use only constructors (or Factory objects) which require all compulsory properties to be specified.
3. If the compulsory properties are (or are derived from) a relationship to another (pre-existing) object, force the client to ask the related object perform the creation.

Solution 1 is not really solving the problem at all, it does not allow the creation process to provide feedback to the client on whether the object is legal (unless we rely on the client to call a validation method the object, which is placing even more faith in the client). It also assumes that we know when the object should be valid so that we can check it before we try to use it. This is not always possible (for example, when writing persistent objects to an ODMG93 [ODMG] database there is no 'save' operation which is identifiable to the application code). It also requires us to open up all properties to change at any time, which is unacceptable in the case of the Customer and Supplier relationships.

Solution 2 is better but it can result in cumbersome constructor parameter lists. This can be a particular problem when several different permutations of parameters are valid. We may, for example, wish to allow clients to specify customers and suppliers either by name or by passing an object reference. It also forces us into questionable techniques like throwing exceptions from constructors if we want to reject any invalid values.

Solution 3 is perfectly acceptable but is only applicable in a limited set of circumstances, mainly where the related object can in some way be considered as the 'owner' of the object being created. In the example, we would be forced to choose either the Customer or Supplier to be the creator of the contract, neither of which may be obvious or desirable.

## Forces

- **Object Integrity and Future Proofing:** If clients are not under your design control (or not designed yet), you cannot rely on them to specify valid properties or validate the object after creation. Sooner or later a client will attempt to create an invalid object.
- **Property Immutability:** There are many situations where immutable properties are required, either because of design decisions or implementation constraints (e.g. indexing mechanisms).
- **Negative Aspects of Encapsulation:** In some solution domains (e.g. some persistence frameworks) the application code has no control over when other parts of the system expect the object to be 'valid'.
- **Language Limitations:** The syntax of many common object oriented languages is limited in the area of supporting different permutations of constructor parameters without needing to specify every possibility separately.
- **Flexibility vs Complexity:** If every permutation of constructor parameters is specified, a cluttered interface may result.
- **Elegance and Safety:** Throwing exceptions from constructors is a problematic implementation technique and should be avoided. It may not be allowed in some implementation environments.
- **Interface Coherence:** Classes are easier to understand and use if the methods for manipulating and validating their properties are part of the same class.

## Solution

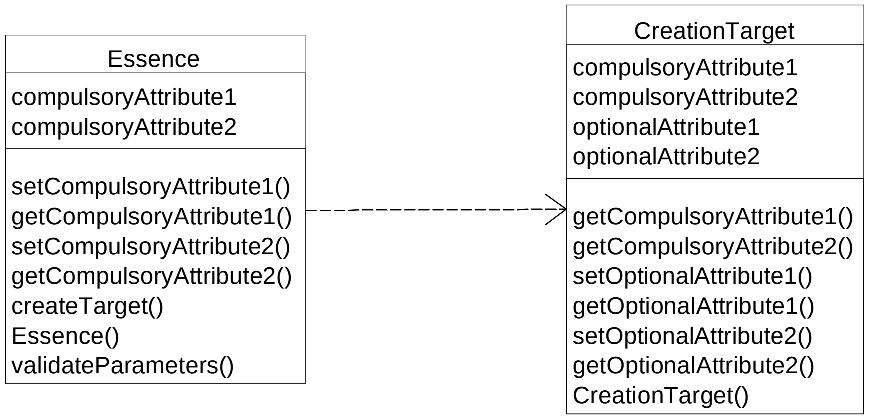

**Figure 2** Structure of the *Essence* pattern

Use a separate object (the *Essence* object) to receive the compulsory properties of the object being created (the CreationTarget) as shown in Figure 2. Creation of the *Essence* object should be unrestricted. Assign a separate method for each property in the *Essence* class and have the method store the parameter value in the *Essence*. For properties which should not be changed after object creation, provide 'set' methods for these only in the *Essence* class.

Force clients to obtain new instances of the CreationTarget class via a method of the *Essence* class. Have this method validate the properties supplied and then either create the new object or report a failure to the client. When creating the CreationTarget the *Essence* should pass a reference to itself to the CreationTarget constructor to allow it to retrieve the parameter values and initialize its own properties.

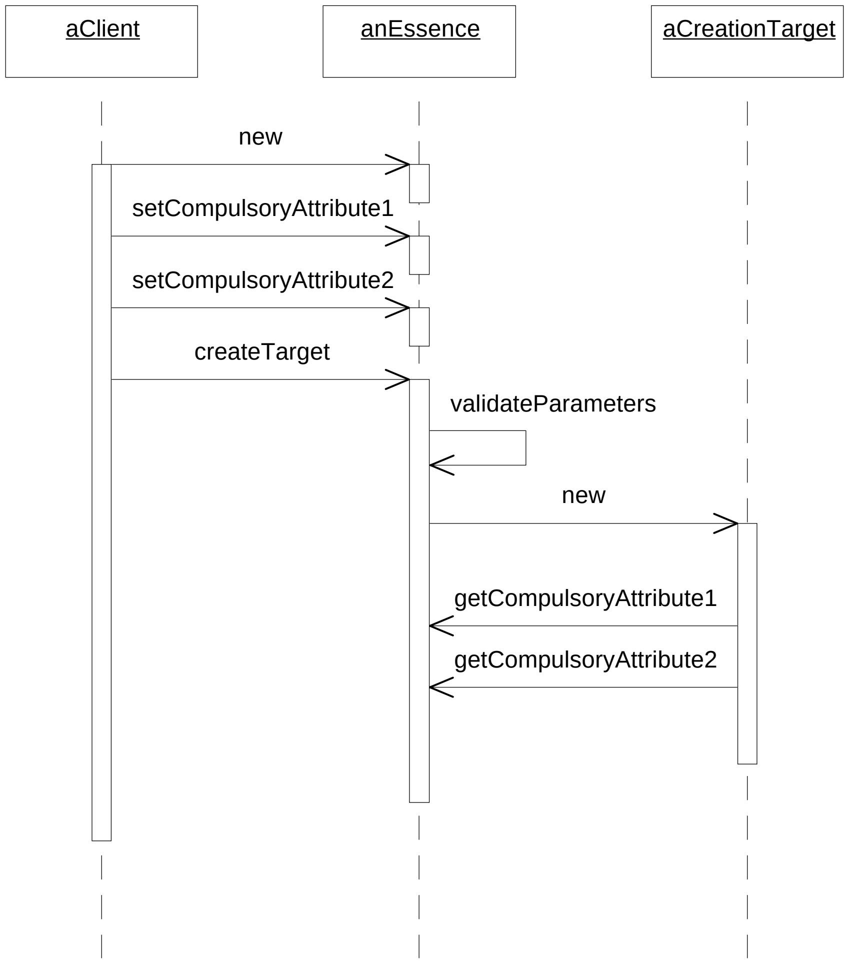

**Figure 3** A client creating a CreationTarget using the *Essence* pattern

From the client's perspective, he must first create an instance of an *Essence* object as shown in Figure 3. No extra information is required to do this. He must then call methods on the *Essence* object to specify each of the compulsory parameters. Once this has been done, the client can call the createTarget method. At this stage, the *Essence* must validate the supplied parameters and either reject them or create a new CreationTarget as requested, passing itself to the new object's constructor. The values can then be retrieved by the CreationTarget from the *Essence*. Finally, the fully constructed CreationTarget is passed back to the client.

As an optional addition, the client can call the validateParameters method of the *Essence* object one or more times during specification of parameters, for example to provide an interactive user with feedback as to whether the object's fields have been completed satisfactorily.

To return to our original example, the methods for the ContractEssence class might be as shown in Figure 4. Note that we are now able to support alternative ways to specify customer and supplier without needing to include all possible permutations in the interface design.

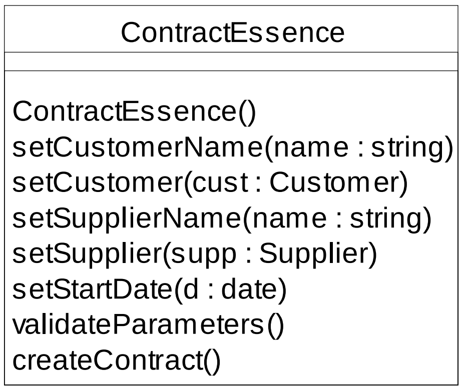
ContractEssence
ContractEssence()
setCustomerName(name : string)
setCustomer(cust : Customer)
setSupplierName(name : string)
setSupplier(supp : Supplier)
setStartDate(d : date)
validateParameters()
createContract()

**Figure 4** ContractEssence interface

## *Consequences*

- ✓ Clients are forced to supply valid property values
- ✓ The object is never created unless and until valid properties values are supplied by the client.
- ✓ The CreationTarget object is valid at the end of its constructor's execution.
- ✓ There is no need to provide 'set' methods for properties which must not be changed after creation.
- ✓ Using a method per parameter allows more flexibility in allowing different permutations of parameters without needing to decide on these before the interface design is fixed.
- ✓ If multiple instances of the same CreationTarget class are being created with parameters which differ only slightly from each other, a single *Essence* can be used to create several targets, saving the client from repeated specification of identical property values.
- ✓ The *Essence* object can also be used as a *Builder* [GAMMA94] if the desired end result is a number of related objects rather than a single object.
- ✗ Adding the *Essence* class introduces another class to the design, possibly causing confusion to client developers who must now decide which class to call when setting property values.
- ✗ Responsibility for validation of attribute values for CreationTarget is in two places instead of one.
- ✗ This pattern requires a circular dependency between the *Essence* class and the CreationTarget class.

## *Implementation*

The main problem to be solved in implementing this pattern is how to prevent clients from directly creating instances of the CreationTarget object but still allow the *Essence* objects to do so.

In C++ this can be achieved by making all of the CreationTarget constructors private and making the *Essence* class a friend of the CreationTarget class.

In Java, the CreationTarget constructors can be made protected. The *Essence* can be granted access to the constructors by placing the *Essence* and CreationTarget classes in the same package. This assumes that the clients are in a different package, otherwise they too would be able to use the protected constructors.

In distributed environments (e.g. CORBA) this job is typically much easier as the object creation process is often forced by the distribution architecture to go through some indirect mechanism (for example, a Factory Object). Clients are therefore unable to use constructors directly.

In environments which lack the ability to tailor method accessibility depending on client, some protection can be obtained by the requirement that an *Essence* is passed when creating the CreationTarget. In these situations the CreationTarget constructor should call the validateParameters method on the *Essence* class. In the event that a client creates an 'incomplete' *Essence* and then directly creates a CreationTarget (i.e. without going via the createTarget method on the *Essence*), the CreationTarget constructor at least has the opportunity to flag an error by whatever means are available.

Another question is how to report creation failure back to clients. This can be done either by returning a null value (or *Null Object*) or by throwing an exception.

The designer has three choices of location for validation. Validation of attribute values which cannot be changed after location should be placed in the *Essence* class. Validation of attributes which are not involved in the creation process should obviously be done by the CreationTarget. Parameters which can be changed after creation are more difficult to deal with as they require validation in both the *Essence* and CreationTarget. A class level method on CreationTarget is a possibility to avoid code duplication in this situation.

An enhancement which may be useful, depending on the nature of the client is to use Bjarne Stroustrup's [STROUSTRUP94] suggestion of having all of the 'set' methods on the *Essence* return a reference to the Essence. This allows the full set parameters to be specified by the client in a single line if desired, for example:-

```
(new WindowEssence()).textColor(black).font(courier).bckColor(grey)
```

Although it is no longer necessary, overloaded constructors can still be provided for common parameter combinations if desired.

## Known Uses

The AT&T Rialto system provides a component called the Rialto Repository. The Repository is implemented as a CORBA server which provides an interface on top of which data management tools can be built. Underlying this is an ODMG93 database and it is the responsibility of the CORBA server to ensure that only valid data reaches the database. The *Essence* pattern has been used in several situations to solve the problem of compulsory and/or immutable properties. For example:-

- Corporate Legal Entities must have a Company Name and at least a minimal set of contact information.
- Contracts must have both a supplier and a customer.
- Domains must have a domain name

## Related Patterns

This pattern was inspired by a section in Bjarne Stroustrup's 'Design and Evolution of C++' [STROUSTRUP94] discussing the case for an argument naming syntax in C++ and alternatives in light of the lack of one. The motivation for this pattern is, however, somewhat different from Bjarne's.

*Abstract Factory* [GAMMA94] is typically used in distributed environments to create new instances of objects and can be used as the source of *Essence* objects.

## References

[GAMMA94] E Gamma, R Helm, R Johnson, J Vlissides. *Design Patterns : Elements of Reusable Object Oriented Software*. Reading, MA: Addison Wesley, 1994.
[ODMG] R Cattell, D Barry (eds). *The Object Database Standard: ODMG 2.0*. San Fransisco, CA: Morgan Kaufmann, 1997.
[OMG] Object Management Group. *The Common Object Request Broker: Architecture and Specification*. 1995.
[WOOLF97] B Woolf. *Null Object* from *Pattern Languages of Program Design 3*. R Martin, D Riehle, F Buschmann (eds.). Reading, MA: Addison Wesley Longman, 1998.
[STROUSTRUP94] B Stroustrup. *The Design and Evolution of C++*. Reading, MA: Addison Wesley, 1994.

## *Acknowledgements*

I would like to thank my shepherd, Wolf Siberski for his helpful comments on this paper.

## *Trademarks*

Rialto is a trademark of AT&T.

**Keywords:** essence, compulsory property, compulsory parameter, object creation

Copyright © 1998 AT&T. Permission is granted to make copies for PLoP 98