DIVISION OF COMPUTER SCIENCE

The Fusion Method
A second generation object-oriented software development method

Jens Diedrichsen

Technical Report No.231

September 1995

The Fusion Method
A second generation object-oriented software development method

An evaluation by

Jens Diedrichsen

email : jens@hursley.ibm.com

Acknowledgement

I would like to thank Carol Britton, Wilf Nichols and Mick Wood for their advise during the work on this Technical Report. Mick also contributed greatly to the section that compares the Fusion method with Rumbaugh's OMT.

Table of Contents

1. Introduction

An indication of the attractiveness of the object-oriented idea can be obtained by looking at the hype around this popular approach to software engineering which was, and in many cases still is, created by both the academic and the commercial computing related media. Another indicator is the number of *gurus* who have emerged to advocate and praise their own method as being the panacea for mastering object-oriented technology. It is virtually impossible to work through all of the methods which promise to deliver procedures to capture object-oriented analysis, design, programming or combinations of them. Ivar Jacobson referred during a panel discussion at OOPSLA'94 [Mon94] to the Object Management Group's (OMG) special interest group on analysis and design (SIGAD) which described at that time 26 different object-oriented methods. It is therefore very refreshing to find a method that does not claim to invent novel techniques, but instead advertises itself as a combination or a fusion of the best parts of the existing methods. The authors of the Fusion method [Cole94] call it a *"second generation object-oriented software development method"*. They see Fusion as a method which *"integrates and extends existing approaches"*. A number of object-oriented methods such as *Object Modelling Technique* [Rumb91], *Booch* [Booch91], *Objectory* [Jaco92] and the *Class Responsibility Collaborator* approach [BeCu89] are explicitly mentioned as being the main ancestors of Fusion.

This report investigates the validity of the authors' claim to have taken the *best* of the existing methods. Supposedly interesting or helpful features of some selected methods are evaluated and compared with their Fusion counterpart, if existent. The report also considers lists of requirements for object-oriented methods, as they have been drawn up by Ian Walker [Walk92] and others, to see how the Fusion method scores against those abstract ideals.

The reader of this report should have a sound understanding of object-oriented concepts and terminology. Knowledge of various object-oriented development methods is helpful but not essential for following the discussions presented in the report.

The remainder of this report starts with a brief summary of the Fusion method as it has been advocated in [Cole94]. The following chapter compares the Fusion method with its ancestor methods and with some of the methods to which the authors have not referred explicitly. After that the Fusion method is checked against the previously mentioned lists of requirements for object-oriented design methods. The report ends with some concluding comments on the usefulness of the Fusion method and on the success of the authors of the Fusion method with regard to the aims they express in [Cole94].

2. The Fusion Process

Fusion promises to help its users through the entire development life cycle of object-oriented software. During this process it provides a variety of models or documents that are created in the individual stages and which are then carried forward to help construct the models of the next stage.

Fusion also provides a number of mechanisms that help the developer to improve the overall quality of the system to be created. It supplies a number of checks that help to detect inconsistencies among the different models. It suggests a series of heuristics that give guidance in the individual stages about the *do's* and *do not's* in object-oriented software engineering.

The Fusion Process on the whole is clearly partitioned into an analysis phase, a design phase and an implementation phase. Fusion provides a task list for each phase, so that the user is encouraged to create appropriate output during the phases and to test the results before continuing with the next phase. This very structured approach with its well defined phases, completion tests and deliverables bears some resemblance to the traditional waterfall model. The Fusion Process Summary in the appendix of the book [Cole94] provides an overview diagram on the method which seems to support the waterfall method analogy. Although the Fusion process is clearly more elaborate than its early ancestor, the main flow of control is still dominantly directed

2.1 Analysis

The Fusion process starts with the analysis stage. It assumes that a requirements specification is produced by the customer.

The analysis phase captures the intended behaviour of a system. It concentrates on *what* a system does as opposed to *how* it does it. The authors describe the analysis phase as *viewing the system from the user's perspective rather than that of the machine*. This means that the main emphasis is on the externally visible behaviour of the system that is to be built.

2.1.1 Developing the Object Model

The object model is used to describe the static structure of the information in the system. It captures the concepts that exist in the problem domain and the relationships between them. These relationships can be based on generalisation (i.e. *kind-of* or *is-a* relationships), on aggregation (i.e. *part-of* or *has-a* relationships) or any other logical relationship (e.g. representing an action such as *Professor*-**teaches**-*Course*).

The notation used for the object model is derived from the extended entity relationship model. Labelled boxes and arcs are used to present classes and relationships. The names in the boxes are used as unique identifiers. This makes it possible to spread details such as attributes of a particular class over multiple boxes in one or more diagrams. A comprehensive picture of a class can be obtained by taking the union of all boxes within one object model that refer to the same class.

Relationships can be annotated with cardinality constraints. Properties which cannot be expressed via cardinalities are expressed through invariants which are simply textual annotations to the object model.

Specialised relationships can be derived from primitive ones. However, the authors suggest that only primitive relationships should be used in the object model. Derived relationships may be entered into the data dictionary.

Complex diagrams can be broken down into several sub diagrams. The complete object model is then the union of all sub diagrams.

The naming of the object model is somehow unfortunate, since the object model shows classes, not objects, as the authors admit in Chapter 2. The entire concept is a bit confusing in the book, as the authors continue to explain that "*in the object model a relationship is used to model the idea of association or correspondence between the objects belonging to classes*". My interpretation of the object/class model problem is that the authors see the analysis phase primarily concerned with classes. Hence, the objects of interest in the analysis phase are in fact classes in the traditional interpretation. This makes the usage of the term *"object model"* a bit more comprehensible, but it does not make it any better.

2.1.2 Determining the System Interface

The interface of a system is described as *the set of system operations to which it can respond and the events that it can output* [Cole94, p.45]. Scenarios of usage are recommended as a technique to describe a system interface. Scenarios can be depicted in timeline diagrams where the temporal ordering of system operations and events are shown. For each system operation multiple scenarios may be produced to explore alternative communication paths between external agents and the system. Every system operation is entered into the data dictionary.

System Object Model

A system object model is produced by using the information captured in the object model and the system interface. The system object model is a refinement of the object model. It separates the classes and relationships that belong to the system from the ones that belong to the environment.

2.1.3 Developing an Interface Model

Life-Cycle Model

The life-cycle model defines the allowable sequences of system operations and events. It shows how the system communicates with its environment. The notation used for this model is based on extensions to regular expressions. The life-cycle model is created by generalising the scenarios that were developed earlier.

Operational Model

In the operational model a schema is created for each system operation describing the operation in a textual format. It contains the operation name, an informal natural language description, the classes with which it has to communicate and the preconditions and postconditions for the operations.

The operational model is developed by extracting information from the life-cycle model and combining it with the invariants on the system object model. During this process it can become necessary to revisit the system object model and to update the data dictionary if additional information has been acquired.

2.1.4 Checking the Analysis Models

The Fusion method provides guidelines to check the completeness and the consistency of the analysis models. This consists of checking the completeness against the requirements document, a consistency check regarding the overlapping areas in the various models, and semantic consistency checks which capture the output of events, the preservation of invariants and also include desk-checks of scenarios.

2.2 Design

The design phase defines *how* the specified behaviour is obtained from the implementation-oriented components. *The output of design is a collection of interacting objects that realise the operation model* [Cole94, p. 62].

2.2.1 Object Interaction Graphs

For each system operation an object interaction graph is created showing how the operation is implemented, i.e. which objects are required to contribute in order to achieve the operation. The schema from the operation model is used as a starting point to identify the objects. One of the objects involved gets assigned to be responsible for the operation. This object is called the controller whereas the others are known as the collaborators. After the message passing procedures between the different objects have been decided, the object interaction graph can actually be drawn.

The notation for the graph consists of labelled boxes and arcs. The boxes denote individual objects or groups of objects, and the arcs represent messages that are sent between them. The arcs can obtain numerical annotations to indicate the order in which the messages are sent. These numbers are also used to reference the messages from within the data dictionary entry that associates each object interaction graph.

Fusion provides two checks for the object interaction graph to ensure consistency with the analysis models. First, each of the classes in the system object model must appear in at least one object interaction graph. Second, the functional effect of each object interaction graph must satisfy the conditions specified in the operation model.

2.2.2 Visibility Graphs

A visibility graph defines the reference structure of classes in the system. For this, the object interaction graphs are analysed. For each message in those graphs it is necessary to establish the required visibility reference between the sender and the receiver of the message. Fusion provides a number of different types of visibility information, such as reference lifetime, server visibility, server binding and reference mutability. After analysing and annotating all the messages in the object interaction graph, a visibility graph is drawn for each class showing all the references and their types that are required by the class.

The notation for visibility graphs consists of labelled client boxes and server boxes which are connected by visibility arrows. A client box represents the class that requires access whereas a server box is used for the class which is accessed.

Fusion suggests three checks to verify the consistency and completeness of the created model. First, the consistency with the analysis models is reviewed by checking that a path of visibility exists for each relationship in the object model. Second, the mutual consistency within the set of visibility graphs must be ensured. This relates mainly to exclusive and shared targets. Third, all message passing defined in the object interaction graph must be captured in the visibility graph.

2.2.3 Class Descriptions

The authors regard the class descriptions as the specification from which coding begins. They are constructed by extracting information from the system object model, the object interaction graphs and the visibility graphs. A class description contains details of each attribute and each method of a class. It also lists the super classes of a class where applicable.

The checks for the correctness of the class descriptions suggest visiting each type of information captured in the description and verifying that all the details kept in the earlier models are recorded in the class description.

2.2.4 Inheritance Graphs

Inheritance relationships between classes are identified by revisiting the previously created models. The authors recommend starting with looking at the system object model to find the generalisation and specialisation structures. In addition, common functionality can also be discovered by analysing the object interaction graphs and the class descriptions. Visibility graphs should be inspected to find common reference structures which could possibly become the basis for new inheritance relationships.

The notation used for inheritance graphs is the same as the object model notation. Classes are represented by boxes which contain the class name and class attributes. Arcs between the class boxes show inheritance relationships. The arcs are annotated with a triangle which points towards the super class. Solid triangles indicate the existence of an abstract super class.

The inheritance graphs are checked against the system object model, the object interaction graphs, the visibility graphs and the class descriptions. Hereby it can become necessary to update the class descriptions in order to incorporate new details that may have evolved during the creation of the inheritance graphs.

2.3 Implementation

The Fusion method provides a number of guidelines to support the developer in the process of translating the design documents into the actual source code. The authors point out that although Fusion is not directed to any particular programming language, the examples used in the book [Cole94] are directed towards C++ and Eiffel.

2.3.1 Coding

The life-cycles are translated into (non deterministic) state machines and implemented in a conventional fashion. This implementation serves as a framework for the rest of the system.

The implementation of the class descriptions is very straightforward, since these descriptions are already expressed in a pseudo-code format. Only the gap to the preferred implementation language needs to be bridged.

Finally the concepts kept in the data dictionary have to be evaluated in the light of implementation details. This includes such things as predicates, functions, assertions, types and events, some of which were only introduced to make analysis concepts clearer and can be ignored in the implementation.

2.3.2 Performance

It is recommended that performance issues accompany the entire implementation process. Alternative designs should be evaluated with performance issues in mind. The authors give a few recommendations regarding topics such as inline methods, references versus values, the impact of dynamic binding, and memory issues.

2.3.3 Review

The authors of Fusion suggest *inspections* and *testing* as the major reviewing techniques. It is recommended to carry out these tests while producing the code. It should not be left to the end.

2.4 Critique on the book

The Fusion method and related topics are presented and discussed in [Cole94]. The book starts with a brief evaluation of some of the current problems in software development such as poor predictability, low-quality programs, high-maintenance costs and duplication of effort. The authors do not provide an in-depth discussion of these topics. They merely try to focus the reader's mind on the general process of software development. This is followed by a brief overview on the benefits of development methods in general and how methods can support the development of object-oriented software in particular. Some of the advantages and disadvantages of the object-oriented approach to software development are also introduced. This cannot be regarded as anything more than a reminder to the reader, since the entire introductory chapter is presented on less than ten pages of text which also include a brief overview of the Fusion method.

The core (chapter 2-5) of the book describes the different stages in the Fusion process including the documents and deliverables, the tests and the heuristics that should be applied during each stage. For the design phase a separate section is provided which deals with *Principles of good design*. This sub chapter briefly presents ideas and common-sense facts. The reader is referred to other publications for more details. It certainly gives the less experienced developers some practical guidelines to start with. Sanjiv Gossain points out in [Goss94] that the design chapter in the Fusion book ignores a few interesting questions such as how to determine the roles and responsibilities of objects. He feels that the chapter is "*a little short on detail and breadth given its importance*".

The chapter which deals with the implementation contains a section that attempts to describe how to translate a state transition diagram into source code. Again the authors merely scratch the surface of the topic, but at least they give a practical example for such a translation. They also include source code to demonstrate how a state machine could be implemented in C++.

Chapter 6 presents a small but detailed case study, and chapter 7 challenges the reader with a series of exercises. Together those chapters can provide the user with a good impression of how

software development can be done by using the Fusion method. It is unfortunate that the authors do not provide any example solutions to the exercises.

Since Fusion claims to be a second generation method a separate chapter is provided to review and compare the methods from which Fusion originated. The authors manage to give a concise overview of the methods involved and point out their advantages and disadvantages.

Another very useful part of the book is the comprehensive set of appendices. Appendix A presents the entire Fusion process including the steps, heuristics and checks of each stage. Appendix B contains a summary of the notation, and Appendix C holds a reference manual for the Fusion process. A lot of information is duplicated throughout the book, but this helps the reader to find particular details of the method more quick. The last two appendices give an example of a life-cycle translation algorithm, and a list of Fusion services.

However, the few inconsistencies in the book are rather confusing. As an example, Figure 2.12 and Figure 2.16 do not match. The accompanying text explains how aggregation provides a means of levelling, but during that process important information seems to get lost. It is also annoying that the index in the first print of the book is out of step by a few pages with the actual layout. According to the authors this problem has been fixed in later prints of the book, as reported in the Fusion Interest Group on the internet.

3. Fusion and its ancestors

The Fusion development method is greatly influenced by four major object-oriented methods which are OMT [Rumb91], Booch [Booch91], Objectory [Jaco92] and CRC [BeCu89]. However, the authors of Fusion certainly used their experiences with other methods too while shaping Fusion. A comparison of the Fusion method with its ancestors is provided by the authors in chapter 8 of their book [Cole94]. This chapter provides some additional comments and personal opinions on the relationship between Fusion and other methods.

3.1 Comparison with OMT

There are at least two areas in which a comparison of Fusion with OMT might be instructive. Firstly, Fusion explicitly acknowledges a debt to OMT in certain respects, notably in the analysis phase, and most obviously in the object model. The somewhat subtle differences here illustrate the extent to which agreement is lacking on what a method should provide.

More generally, Fusion's claim to be *second generation*, and therefore to supersede OMT and similar methods, invites a critical comparison of the two methods over the whole development process.

3.1.1 Comparison of the Object Models

This section compares the object model used in Fusion with that of OMT [Rumb91], which according to [Cole94] provided the main inspiration for the Fusion analysis phase. Although these two object models are described there as similar "apart from relatively minor notational differences", it is instructive to compare them in some detail, particularly as regards what was adopted from OMT and what was not.

classes and relationships

On "minor notational differences", we note that Rumbaugh and his co-authors use the term *association* in preference to *relationship*, as the latter has database connotations. The methods provide similar notations for diagramming classes and relationships. In addition, OMT provides for *instance diagrams*, in which object instances are related by *links*, which appears to be a useful distinction; Fusion gets by without any equivalent.

As noted earlier, Fusion classes do not have a method interface during analysis; this is not the case with OMT classes, although identification of methods generally takes place later, during dynamic and functional modelling. Nevertheless, OMT objects do exhibit dynamic behaviour during its analysis phase.

Rumbaugh et al are very wary of ternary and higher-arity relationships, and provide a mechanism (qualification) for reducing certain types of ternary relation to a binary one. There is also the concept of a *candidate key* for a relationship, which can be used as an unambiguous alternative to cardinalities in the case of ternary (etc.) relationships.

By contrast, the Fusion authors, although paying lip service to the difficulties associated with ternary relations, provide plenty of examples of these; figure 3.13 for example has two 4-ary relations, and it is not at all clear why this aspect of the ECO storage depot has been modelled like this. Unfortunately, this last comment could be applied to many of the case study examples presented in [Cole94].

In Fusion, a relationship between (say) two classes is a set of tuples, a subset of the Cartesian product of the two classes, corresponding to a mathematician's understanding of a *relation*. A relationship is total in one of its classes if all objects in that class must participate; Fusion provides a *total marker* notation to indicate this situation.

A number of additional, more advanced, concepts with associated notations are described in [Rumb91] which have no counterpart in Fusion. These include metadata (class attributes, methods, etc.), derived objects, attributes and links, and homomorphisms.

Homomorphism is described by Rumbaugh as a mapping between two associations. He uses the example of a parts catalogue for an automobile. An item in the catalogue may contain other subitems. In comparison the corresponding real physical item is also composed of the corresponding physical subitems. Hence, "*the contains aggregation on catalogue items is a homomorphism of the contains aggregation on physical items*".

aggregation

More significant is the different approach taken by the two methods to aggregation; Rumbaugh regards this as a special *part-of* form of association (relationship), corresponding to physical containment, and therefore having additional properties such as transitivity [Rumb91] (page 57), plus propagation of some properties from the assembly to its components. There is even a special notation for the latter, plus heuristics for deciding when to use aggregation in preference to a simple association.

For the Fusion authors, aggregation is also "a mechanism for structuring the object model". A class may occur inside more than one aggregate class, as there is no requirement that aggregation necessarily corresponds to a *part-of* relation. As will be discussed below, aggregate classes are used in Fusion to provide for abstracting detail in a levelled presentation of an object model, there being no higher level *module* or *cluster*-type construct. It is arguably unfortunate that two distinct notions (aggregation as an assembly of components, and a mechanism for abstraction and structuring in large models) are muddled in this way.

We also note that, whereas OMT has a special notation for cases where an association is itself to be treated as a class (perhaps because it participates in another relationship), Fusion uses an aggregate class as a rather artificial box around a relationship to achieve the same effect.

generalisation and inheritance

On generalisation and inheritance, the two methods are very similar (despite contradictory notations for distinguishing subclasses which partition the superclass from those which do not). Both strongly advocate a strict adherence to subtype inheritance during Analysis.

packaging/clustering of entities

Other object-oriented methods provide a mechanism for clustering a set of closely associated classes and relationships into a single unit to provide higher levels of abstraction mechanism in large systems (e.g. *subjects* in Coad-Yourdon [Your94]). Here OMT provides a *module* construct as a logical component of a system model. Guidelines for developing modules are given in OMT, although there do not seem to be any special notations for documenting either the external interface of a single module, or the composition of the whole system in terms of modules. In particular, coupling between modules has apparently to be deduced by looking for occurrences of

the same class in different modules. The physical unit of decomposition is a *sheet*, which is effectively a diagram small enough to be comprehended. Normally a module would occupy several sheets.

Fusion allows for separate diagrams (corresponding to OMT sheets), and provides rules to cover cases where the same class and/or relationship occurs on more than one. As noted above, there is nothing corresponding to an OMT module, and aggregate classes are used for abstraction and levelling.

system boundary

Compared with earlier object-oriented methods, Fusion makes the distinction between the system and its environment very clear. During object modelling, problem domain classes and relationships are identified. A system object model is then derived from this, excluding classes which correspond to external agents with which the system interacts (the nature of their interaction being examined in the Interface Model). This is a useful distinction, as confusion over what is part of the system (i.e. therefore subject to the developers' decisions) and what is not (i.e. effectively therefore unchangeable), often recurs throughout modelling if not sorted out initially.

OMT does not seem to address this at all, although neither do other object-oriented methods.

data dictionary

In line with older *structured* approaches to development, Fusion advocates the use of a data dictionary to document the entities derived during Analysis (and Design), a *central repository of definitions of terms and concepts.* A detailed notation for this is given in the Reference Manual (although most of this is not illustrated in the body of the book), although the use of an existing alternative is also sanctioned. Presumably several of the notations provided by the OMT object model and absent from the Fusion one would be replaced in the latter by suitable data dictionary entries.

OMT also recommends the use of a data dictionary ([Rumb91], p156), but says very little about it.

constraints/invariants

In [Cole94] p16, an invariant is defined as *an assertion that some property must always hold.* An invariant could involve an arbitrarily complex restriction on objects, attributes and relationships in an object model. Invariants cannot therefore be represented by the Fusion object model, and so are documented in the data dictionary. OMT has the similar (if more general?) notion of a *constraint*, a *functional relationships between entities of an object model* ([Rumb91], p.73). Various annotations to the object model diagrams to indicate constraints are described, although it is hard to avoid the conclusion that such use would produce very detailed diagrams, and that use of a data dictionary might be preferable.

3.1.2 Semantics

Development methods which make extensive use of graphical notations are often criticised for inherent ambiguity and lack of clarity. The precise meanings of diagrams (and of the concepts they represent), and the resulting requirements for consistency between the different models produced by a method, are often left implicit. As well as the danger of confusion and misunderstanding among developers using the method, this lack of semantic clarity also complicates the task of producing tool support.

Fusion goes further than OMT (and other methods) in attempting to overcome this problem; the Reference Manual in [Cole94] offers a (semi-formal) semantics for the Fusion notations, claiming that notations with formal semantics "are only practical in ..systems where defects must be avoided at all costs" ([Cole94], p59). This is an interesting statement since their presented case study deals with a chemical storage system, which could be viewed by some as a system where "*defects must be avoided at all costs*". Despite the informality of this effort, considerable benefits do seem to accrue from it, notably the greater degree of consistency checking between models which Fusion supports when compared to (say) OMT.

3.2 Comparison with Booch

The major strength of the Booch method [Booch91] and [Booch94] is its extensive and expressive notation. Booch's weakness is its "very ill-defined and loose process" as observed in [Cole94]. Nevertheless, the authors of Fusion state in chapter 8-8.2 [Cole94] that the Fusion design stage is partly based on the Booch method. However, they only emphasise the differences between the methods and don't mention any correspondence.

3.2.1 Object interaction graph versus object diagram

Looking at the Fusion process and its notation one can see that Fusion's *object interaction graph* is in fact a modified version of Booch's *object diagram*. A minor difference is that Fusion changed Booch's *amorphous blob* to a rectangular shape, which is a friendlier icon for computer based drawing tools. A more important modification is that Fusion includes numbers on the message paths. The numbers indicate the order in which the messages are sent. This is basically a common sense improvement. Booch did not say that such numbers should not be included in his object diagrams, but rather omitted to emphasise their usefulness.

Generally, Fusion's notation provides less detail in this comparison. For example, the *synchronisation symbols* that can be attached to Booch's message arcs have been omitted. Also, the *visibility symbols* in the Booch notation are not included, instead they have been extracted into a separate diagram within the Fusion process, i.e. the *visibility graph*.

In order to make the *object interaction graphs* easier to understand, Fusion encourages the use of descriptive comments. Booch uses its *object template* and *message template* instead which gives the comments more structure.

3.2.2 Other Fusion graphs versus other Booch diagrams

The rich set of diagrams that is offered by the Booch notation has largely been ignored by Fusion. Apart from the *object diagram* there is merely the *state transition diagram* which found its way into the Fusion process. This is of course not particular to the Booch method. State transition diagrams have been widely used, both outside and inside the object-oriented world, for example in Rumbaugh's OMT method [Rumb91].

Booch's *timing diagram*, *module diagram* and *process diagram* have neither been included in the Fusion method nor been replaced by an equivalent notation.

The *class diagram* has also been left out, but instead the *object model* of OMT [Rumb91] has been used which provides a comparable amount of detail.

3.2.3 Descriptive text versus templates

Most of the descriptive text used within the Booch method is placed in separate templates and is not included in the diagrams. Fusion's *operational model* could have been influenced by Booch's *operation template* [Booch91]. However, Fusion's notation for textual information provides much less details than Booch does. This is true for Fusion's *operational model* compared with Booch's *operation template*. Also Fusion's *class description* is not nearly as detailed as Booch's *class template*. Booch's *class utility template* does not even have an equivalent in the Fusion method.

3.2.4 Summary

Summarising, one could say that the influence of the Booch method on Fusion is apparent, but not overwhelming. Many, in my opinion, very useful features and details of the Booch approach have been ignored in the Fusion method for the sake of simplicity, as I guess.

3.3 Comparison with CRC

The idea of CRC cards (Class Responsibility Collaborator) as presented in [BeCu89] is not a method but rather a technique which can be used within other object-oriented design methods. The authors of the CRC card technique refer to [Wirf89] which became the basis for [Wirf90] - an object-oriented design method which uses CRC cards as one of its main mechanisms.

The authors of Fusion acknowledge the value of CRC cards and they claim that the CRC technique was one of their major influences on Fusion. However, they don't employ the CRC technique within their process. If at all, then there are a few minor overlaps which could be interpreted as influences. For example, the Fusion analysis phase starts with a brainstorming session in which a list of candidate classes and relationships is produced. This could be related to the *class dimension* within the CRC technique which also tries to find the potential classes and, even more important, appropriate names for them. Fusion does not use CRC cards to protocol the findings of the brainstorming session, but uses a data dictionary instead. After that Fusion continues to build its *object model* in a notation that is strongly related to Rumbaugh's OMT [Rumb91]. The next time Fusion gets anywhere near an analogy to CRC cards is when it creates the *operational model*. However, the notation of Fusion's *operational model* holds more resemblance to Meyer's *design by contract* approach [Meyer88] or to Booch's *operation template* [Booch91] than to CRC cards.

Summarising one could say that there is very little evidence of the influence of CRC cards on the Fusion method and its notation.

3.4 Comparison with Responsibility Driven Design

As previously mentioned, Wirfs-Brock's responsibility driven design approach [Wirfs90] is primarily based on the CRC card idea of Beck and Cunningham [BeCu89]. In addition to CRC cards which are also used for recording details on subsystems, Wirfs-Brock provides other supporting tools such as *hierarchy graphs* to capture the inheritance relationships between classes, *Venn diagrams* to show common responsibilities between classes and *collaborations graphs* which describe how the individual classes work together to achieve the system's goals.

The responsibility driven design approach is quite helpful during the analysis phase of a project where the vocabulary of the problem domain is defined. Wirfs-Brock provides some useful heuristics for mastering the first steps in the system development process. Fusion's initial brainstorming session which tries to find a list of candidate classes and relationships shows a degree of overlap in this respect. However, apart from this, Wirfs-Brock's method has a weakly defined process combined with a set of tools which probably make any simple problem look complicated as demonstrated by the authors themselves in the appendix of their book.

Fusion did well to stay away from the design philosophy and notation presented in [Wirfs90] and to take on board only the useful features of the analysis phase of the responsibility driven design approach.

3.5 Comparison with Objectory

According to Fusion's authors, the main influence of Ivar Jacobson's Objectory method [Jaco92] on Fusion has been the *use case scenario*. Although scenarios are embedded in some form or another by various other methods too, their application within Objectory is particularly interesting, since they provide the foundation for the entire method. Fusion uses the concept of scenarios during the determination of the system interface. However, the notation used by Fusion seems to be more based on the *interaction diagram* of Objectory than on the actual *use case scenario*.

Apart from the scenario concept there is not much overlap between Fusion and Objectory. The authors of Fusion criticise the models and the notation used within Objectory as poorly defined. Also the lack of checks for completeness and consistency throughout the Objectory's development process is observed as a major weakness. While this is certainly true to some extent, there are also a few issues that are covered better in Jacobson's book [Jaco92] than in the Fusion publication [Cole94]. In particular, Jacobson devotes a chapter each to *Real-time specialisation, Database specialisation* and *Testing*. Fusion does not provide much insight in how it would work in different problem domains.

The title of Jacobson's book already indicates that its objectives are much wider than Fusion's, i.e. *"Object-Oriented Software Engineering"* versus *"Object-Oriented Development"*. In addition to the standard introductions to object-oriented concepts, Objectory takes great care in discussing the particular needs of, for example, real-time systems and databases in the light of object-orientedness. Also the production and usage of software components within object-oriented systems is discussed. In addition, Jacobson delivers a relatively detailed coverage of issues concerning the testing of object-oriented software. This is a topic which received particularly little attention within the Fusion process.

Summarising, it can be noted that Fusion adopted probably the single most important contribution of Objectory to the field of object-oriented software development, i.e. the *use case scenario*. The weaker parts of Objectory have been ignored. Unfortunately, some other, stronger parts of Objectory, such as testing, have been ignored too.

3.6 Comparison with Formal Methods

Fusion claims that one of its major influences has come from formal methods as prominently advocated on the front of the hard cover print of [Cole94]. Looking inside the book though, reveals that the only influence to which the authors can refer is the usage of pre- and post-conditions in the *operational model* where they are used to specify "declaratively what a system does". However, pre- and post-conditions can also be found in many other object-oriented methods and even in object-oriented programming languages such as Eiffel (see [Meyer88]). Formal methods such as VDM and Z have certainly more to offer than this single aspect, but in the opinion of the authors "all formal methods suffer from the problem that they are too costly to introduce" (chapter 8-7 : *Other Methods* in [Cole94]).

Those costs are partly related to the training aspect. Formal methods tend to rely heavily on mathematical notations with which many designers and programmers do not feel comfortable. Hence, Fusion made an effort to avoid mathematical notation in its models as much as possible.

An additional cost factor in formal methods is the *proof of correctness of a program* as demonstrated for VDM in [AnIn91]. Proving that a program is correct can be an expensive and resource consuming task. Therefore, the authors of Fusion advise that "formal methods are really only viable in circumstances where the cost of defects is very high". Moreover, the results of proofs can potentially be unreliable. It is possible that the person who carries out the proof introduces new errors. This may stem from a lack of mathematical skills or simply be due to the complex process involved. However, the formal proof approach could be another, although unclaimed, influence of formal methods on Fusion. The emphasis in Fusion on checking the results of individual phases against each other could be interpreted as a simplified proof process. Fusion provides a set of guidelines for each phase to validate results and ensure consistency. At the end of these checks, Fusion has not proved that the results are correct, but has improved the confidence in the correctness of the results.

There is perhaps one more potential influence of formal methods on one of Fusion's models, namely the *life-cycle model*. Although the authors refer to the Jackson System Design [Jack83] as the source of the model, the underlying idea is strongly reminiscent of Robin Milner's CCS or 'process calculus' as he prefers to call it in [Mill89]. In Fusion's analysis stage the *life-cycle model* is used to characterise the allowable sequences of system operations and events. The model describes how a system communicates with its environment by using a syntax that is a simple extension to regular expressions. This notation can express repetition, alternation, optionality and concatenation. The *life-cycle model* is then used as a basis for creating *state machines* or *state transition diagrams*. This gives rise to the question why *state machines* are not used in the first place. Why the detour via an incomplete semi-formal notation when the end result is a *state machine*? The authors of Fusion express the concern that "state machines describe mechanisms and therefore invite the analyst to indulge in design". Fusion wants to keep a clear separation between objects in the analysis phase and objects in the design phase. The authors also emphasise strongly that Fusion's analysis objects do not have an interface or any dynamic behaviour. Hence, the translation of *life-cycle models* into *state machines* is deferred to the implementation phase of the Fusion process. Additionally the authors report that *state machines* are difficult to handle as soon as the task's complexity increases. They refer to the *state explosion problem* which has not been solved yet. However, this equally applies to the corresponding regular expressions.

This rigorous distinction between the different phases of the development cycle is characteristic of the Fusion method, as described in the process overview early in this report, where the Fusion approach is compared with the traditional waterfall model.

Summarising, it appears that there might have been a number of subtle influences of formal methods on the Fusion development method. However, Fusion does not really implement any of the formal methods concepts fully. One could get the impression that the buzz word *formal methods* was catchy enough to serve for raising Fusion's profile in the advertising campaigns, but the philosophy behind *formal methods* was too difficult to tame for Fusion's purposes.

3.7 Some general observations

An interesting characteristic of Fusion is that the notion of an object changes while working through the three phases (i.e. analysis, design and implementation). Fusion enforces a number of restrictions on objects during the analysis phase in order to ensure a clean separation between analysis and design. The analysis phase is concerned with *"what a system does rather than how it does it"*. Therefore an analysis object has a unique identification but it is not specified how this could be accomplished until later. Also, an analysis object can possess various attributes, but the values of these attributes are not allowed to be objects themselves, instead they have to be simple types such as integer, boolean, enumeration and text. Most important, analysis objects cannot have a method interface. It is only during the design phase that the object model gets extended to specify the available methods. Here Fusion goes much further in the separation between analysis and design than its ancestor methods do (compare with [Booch91] and [Rumb91]). The authors of Fusion also emphasise that there is not necessarily a one to one relationship between the objects of the different phases. Although a design object is an extension of the analysis object, it refers to some high-level design component. Therefore it may be decomposed further during the design phase and be implemented as a group of objects.

It is also notable to see how the Fusion method deals with inheritance. This is largely ignored during the analysis phase and during most of the design phase. Inheritance is introduced as almost the last step in the design phase, followed only by an update to the other models to reflect any changes that are due to the discovered inheritance relationships. The authors distinguish clearly between specialisation and generalisation on the one hand, which *"are properties of the domain model and not of the system design or implementation"*. During the design phase, on the other hand, inheritance is regarded as a *"property of the system and not necessarily of the domain"*. Again, other methods tend to introduce the concept of inheritance much earlier in their process.

The assistance given to the developer in identifying when a particular phase in the process is finished and hence when the developer can proceed to the next one, is a very helpful feature in the Fusion method. This is done mainly through deliverables and checks, i.e. if the developer has produced the set of required models and if they have been validated, then it is time to proceed. However, a weakness seems to be that Fusion does not provide any guidance for identifying scenarios when an additional iteration, i.e. going back to the previous phase, might be more appropriate than proceeding. As an example, it would be useful during the design phase to be able to find flaws in the analysis models that could not have been detected by the simple consistency checks at the end of the analysis phase, but which could be revealed in the wider contents of the design models. Such findings should then initiate a partial or entire rework of the analysis phase.

4. Requirements for object-oriented methods

This chapter attempts to show how far the authors of Fusion progressed towards meeting requirements for object-oriented methods in general as they have been published by Walker [Walk92] and Monarchi [MoPu92]. It also shows where Fusion has its shortcomings which may or may not influence potential users on deciding about the usability of the Fusion method as an aid to object-oriented software engineering.

In [Cole94] the Fusion method is compared with its ancestor methods as well as with a set of requirements for object-oriented methods in general. Coleman et al refer to a number of published comparisons and evaluations of object-oriented methods, such as [Arn91], [Crib92], [MoPu92], [Walk92]. However, they conclude that there is no agreed way of evaluating methods and hence they produce their own list of criteria (see Appendix A) to assess their method.

The criteria used in the assessment process are certainly tailored to put Fusion in a particularly good light. The five questions used for the evaluation are of a very general and high-level nature. If the Fusion method were be assessed against the criteria presented by Ian Walker [Walk92] (see Appendix B) or Monarchi [MoPu92] (see Appendix C) then many shortcomings of the method would be revealed. However, this does not mean that the older methods would do any better against Walker's requirements.

One of the obvious limitations of Fusion, especially when compared with Booch, is that there is no support for concurrency (see Appendix B: 2.3) in the notations of the various Fusion models. The authors regard this as "an inconvenience rather than a major disadvantage" ([Cole94], p. 209), since they believe that concurrent programs can be split into sequential components so that each of them could be developed using the Fusion method. Although this is certainly correct, it would be very helpful if the developer could be supported by a method in the process of identifying candidates for concurrent components. In addition the management of groups of co-operating sequential programs including their asynchronous message passing is probably in many cases a non-trivial task for which appropriate support would be reasonable.

The authors also admit that the Fusion method would have problems of adaptation for "developing applications for distributed object-oriented systems" ([Cole94], p. 209). Distributed systems seem to be generally neglected by object-oriented methods which is fairly surprising since the industry develops an increasing interest in Distributed Object Computing (DOC) with IBM's Distributed System Object Model (DSOM), Sun's Distributed Object Environment (DOE) [Suth94] and Microsoft's Common Object Model (COM) which serves as foundation technology for Microsoft's Object Linking and Embedding OLE [Betz94]. It may need a third generation object-oriented development method to catch up this technology. Specialised concepts such as metaclasses in DSOM would need to be covered as well as more general issues such as problem recovery, e.g. system behaviour when connections to remote objects break down.

Browsing further through Walker's criteria the support for test generation and metrics for test coverage (see Appendix B: 3.4) does not seem to merit much attention by Fusion either. The issue of testing is reduced to the mentioning of black-box testing and white-box testing, in addition to a brief discussion of some testing requirements that are specific to object-oriented systems, namely additional levels of abstraction, object state and inheritance. The discussion contains some useful hints which are better than nothing, but in my opinion this can not be called an in-depth treatment of this very important part of software construction. Unfortunately the Fusion method is no exception with regards to neglecting testing in its process. Few other methods spend an appropriate amount of effort in ensuring that the system that has been built in fact works as required and complies to some form of quality standard. However, the topic of testing object-oriented systems has been covered outside method publications, for example by Robert Binder [Bind94] who also provides numerous references to earlier discussions of testability issues.

5. Concluding comments

The influence of various existing object-oriented methods on Fusion is clearly observable. In particular Rumbaugh's Object Modelling Technique [Rumb91] has left its traces in Fusion's analysis phase more prominently than any other method. However, it also seems that Fusion has used many of its ancestor methods mainly as warning examples of how not to do things. This becomes particularly clear in the section of the book [Cole94] where the other methods are analysed and summarised. The authors of Fusion identified many of the problems in the existing methods and they tried to avoid making the same mistakes. In general this has been achieved. Although in some instances Fusion has done so by not providing the same level of detail and in depth coverage as its ancestor methods, for example in comparison with the expressiveness and richness of the Booch diagrams. In other areas, such as testing, Fusion simply provides as little guidance as most other methods. Taking this observation into account makes it difficult to decide whether the Fusion method really succeeded in *reusing the best parts of the existing object-oriented methods* which was one of its main ambitions. In my opinion some valuable concepts of other methods did not make it into the Fusion method as discussed earlier in "Fusion and its ancestors".

However, it must be noted that Fusion is one of the younger object-oriented development methods, and as such it continues to evolve and improve continuously. Evidence for this can be found in the *Fusion Forum* on the Internet where practitioners and authors of Fusion are participating in constructive dialogues about the strengths and weaknesses of the method. News on further developments are also reported on the forum, such as Derek Coleman's work on extending Fusion to include requirements capture.

Overall the Fusion method is well structured and clearly presented. It should be noted that the authors do not present any earth shattering novel concepts or ideas. Instead they combine and extend existing knowledge from different sources in a unique form which could help software developers to find a more structured, organised and guided path to the construction of object-oriented system. In that sense, my opinion is that the Fusion method lives up to its goal of being a *second generation object-oriented software development method.*

6. References

AnIn91 Derek Andrews, Darrel Ince
Practical Formal Methods with VDM
McGraw-International (UK) Ltd, 1991, ISBN 0-07-707214-6

Arn91 P. Arnold, S. Bodoff, D. Coleman, H. Gilchrist, F. Hayes
Evaluation of five object-oriented development methods
Journal of object-oriented programming: Focus on analysis and design
SIGS Publication 1991, pp. 101 - 121

BeCu89 K. Beck, W. Cunningham
A laboratory for teaching object-oriented thinking
ACM OOPSLA '89 Conference Proceedings, 1989

Betz94 Mark Betz
Interoperable Objects
Dr. Dobb's Journal, #220 October 1994, pp. 18 - 39

Bind94 Robert V. Binder
Design for testability in object-oriented systems
Communications of the ACM, September 1994, Vol. 37, No. 9

Booch91 Grady Booch
Object-Oriented Design with Applications
The Benjamin Cummings Publishing Company, 1991, ISBN 0-8053-0091-0

Booch94 Grady Booch
Object-Oriented Analysis and Design with Applications
The Benjamin Cummings Publishing Company, 1994, ISBN 0-8053-5340-2

Cole92 Derek Coleman, Paul Jeremes, Chris Dollin
Fusion: A systematic method for object-oriented development
HP internal paper, November 26th 1992

Cole94 Derek Coleman et al
Object-Oriented Development - The Fusion Method
Prentice Hall, 1994, ISBN 0-13-101040-9

Crib92 J. Cribbs, C. Roe, S. Moon
An evaluation of object-oriented analysis and design methodologies
SIGS Publications, Inc., New York, 1992

FiKe92 Robert G. Fichman, Chris F. Kemerer
Object-oriented and conventional analysis and design methodologies
Computer, October 1992, pp. 22 - 39

Goss94 Sanjiv Gossain
Book Review : Object-Oriented Development - The Fusion Method
Journal of object-oriented programming, SIGS Publication
March/April 1994, Vol. 7, No. 1, pp. 84 - 86

Hay94 Wayne Haythorn
What is object-oriented design ?
Journal of object-oriented programming, SIGS Publication
March/April 1994, Vol. 7, No. 1, pp. 67 - 78

Jack83 M. Jackson
System Development
Prentice Hall International, Englewood Cliffs, NJ, 1983

Jaco92 I. Jacobson
Object-oriented software engineering
Addison-Wesley, reading, MA, 1992, ISBN 0-201-54435-0

Jere93 Paul Jeremes, Derek Coleman
Fusion: A second generation object-oriented analysis and design method
IEE Colloquium on "Object-Oriented Development", 14-Jan-1993, (paper)

Meyer88 Bertrand Meyer
Object-oriented Software Construction
Prentice Hall International, Englewood Cliffs, NJ, 1988, ISBN 0-13-62903-10

Mill89 Robin Milner
Communication and Concurrency
Prentice Hall International (UK) Ltd, 1989, ISBN 0-13-115007-3

Mon94 David Monarchi, Grady Booch, Brian Henderson-Sellers, Ivar Jacobson, Steve Mellor, James Rumbaugh, Rebecca Wirfs-Brock
Methodology Standards : Help or Hindrance ?
ACM OOPSLA '94, Conference Proceedings

MoPu92 David E. Monarchi, Gretchen I. Puhr
A research typology for object-oriented analysis and design
Communications of the ACM, September 1992, Vol. 35, No. 9, pp. 33 - 47

Rumb91 J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen
Object-oriented modelling and design
Prentice Hall International, Englewood Cliffs, NJ, 1991

Stein93 W. Stein
Objekt-orientierte Analysemethoden - ein Vergleich
Informatik Spektrum, December 1993, Vol. 16, No. 6, pp. 317 - 332

Suth94 Jeffrey Sutherland
Distributed object architecture for IS applications
Distributed Object Computing, Supplement to SIGS Publications, 1994, pp. 7 - 12

Walk92 Ian Walker
Requirements of an object-oriented design method
Software Engineering Journal, March 1992, pp. 102 - 113

Wirf89 R. Wirfs-Brock, B. Wilkerson
Object-Oriented Design: a Responsibility-Driven Approach
SIGPLAN Notices vol. 24 (10), October 1989

Wirf90 R. Wirfs-Brock, B. Wilkerson, L. Wiener
Designing object-oriented software
Prentice Hall International, Englewood Cliffs, NJ, 1990, ISBN 0-13-629825-7

Your94 E Yourdon
Object-oriented systems design; an integrated approach
Yourdon Press, 1994

Appendix A: The "Fusion Criteria"

Taken from [Cole94] p. 189

- Does it make it easier to develop a system ?
- Does the method make the system easier to maintain ?
- Does it help produce software with good object-oriented structure ?
- Does it assist project management ?
- Can it be given effective tool support ?

Appendix B: The "Walker Criteria"

Taken from [Walk92], p. 107

1. Abstraction

   1.1. Rules, guidelines, heuristics for the separation of the problem domain into aggregation/composition, class/inheritance and functional abstraction.
   1.2. Methods for identifying state, services and interfaces for encapsulating objects.
   1.3. Strategy for abstracting commonality for inclusion in abstract superclasses or mix-ins.

2. Notation and representational verisimilitude

   2.1. Notation for representing aggregation, inheritance and functional hierarchies.
   2.2. Notation for representing message sending/calling/parameter passing between objects in a seniority hierarchy, within an inheritance hierarchy.
   2.3. Representation of autonomous objects and asynchronous message passing - concurrency.
   2.4. Support for constraint representation and enforcement.

3. Validation

   3.1. Transformation rules between alternative representations of the design, e.g. graphical, textual, algebraic, code.
   3.2. Support for validation/consistency checking between message sends and interface protocols.
   3.3. Mechanisms for ameliorating implementation language shortcomings.
   3.4. Support for test generation and metrics for test coverage.

4. Reuse

   4.1. Heuristics for identifying pre-existing classes from which to develop new application classes.
   4.2. Support for different categories of object class (e.g. application, system, functional, data) and their differing reusability characteristics.
   4.3. High degree of language independence for logical or architectural design phase.

Appendix C: The "Monarchi & Puhr Criteria"

Taken from [MoPu92] p. 45

Strengths and Weaknesses of current OOAD research	
Strengths	**Weaknesses**
• Identifying semantic classes • Identifying attributes and behaviour • Placing methods (Law of Demeter) • Identifying and representing generalisation and aggregation structures • Representing static views (structure)	• Identifying interface, application and system classes • Determining when an attribute, relationship or behaviour should be a class • Placing classes • Identifying and representing other kinds of relationships • Maintaining consistent and correct semantics for relationships • Representing dynamic views (message passing, control, ...) • Integrating static and dynamic models • Maintaining consistent levels of abstraction/granularity