RATIONAL EVALUATION OF THE METHODOLOGIES OF OBJECT-ORIENTED SOFTWARE ENGINEERING

by

DAVID R. GILLER, JR.

A THESIS

Presented to the Department of Computer and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Master of Science

December 1997

RATIONAL EVALUATION OF THE METHODOLOGIES OF OBJECT-ORIENTED SOFTWARE ENGINEERING

by

DAVID R. GILLER, JR.

A THESIS

Presented to the Department of Computer and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Master of Science

December 1997

"Rational Evaluation of the Methodologies of Object-Oriented Software Engineering," a thesis prepared by David R. Giller, Jr. in partial fulfillment of the requirements for the Master of Science degree in the Department of Computer and Information Science. This thesis has been approved and accepted by:

[illegible]

Dr. Kent Stevens, Chair of the Examining Committee

December 2, 1997

Date

Committee in charge: Dr. Kent Stevens

Accepted by:

[illegible]

Vice Provost and Dean of the Graduate School

An Abstract of the Thesis of

David R. Giller, Jr. for the degree of Master of Science

in the Department of Computer and Information Science to be taken December 1997

Title: RATIONAL EVALUATION OF THE METHODOLOGIES OF OBJECT-ORIENTED SOFTWARE ENGINEERING

Approved: [illegible]
Dr. Kent Stevens

Software engineers considering the selection of an object-oriented methodology for a new software project have available several mature options but no sophisticated discourse on the process of selecting between them. This work studies the application of two popular methodologies to a sample project and draws conclusions about the metrics by which the methods can be judged. The Fusion method is practical and narrowly-defined, seeking to reduce risk by rationalizing a strict process from start to finish. The Booch method is more sophisticated and generalized, trading complexity for coverage. Comparison of the application of each method illustrates that three primary metrics are the decomposition of software engineering tasks along time and personnel, the efficiency of the language of method notation, documentation, and other artifacts, and the probable expense of educating the project's human resources in proper execution of the methods.

CURRICULUM VITA

NAME OF AUTHOR: David R. Giller, Jr.

PLACE OF BIRTH: Visalia, California

DATE OF BIRTH: March 31, 1972

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon
Portland State University
Occidental College

DEGREES AWARDED:

Master of Science in Computer and Information Science, 1997, University of Oregon
Bachelor of Science in Computer and Information Science, 1996, University of Oregon
Bachelor of Arts in Cognitive Science, 1994, Occidental College

AREAS OF SPECIAL INTEREST:

Object-Oriented Software Engineering
Human Resource Management in Software Engineering

PROFESSIONAL EXPERIENCE:

Graduate Teaching Fellow, Department of Computer and Information Science, University of Oregon, Eugene, 1996

Co-Founder and Vice President, Graydog Internet, Inc., Portland, Oregon, 1997

ACKNOWLEDGEMENTS

I express my sincere gratitude to the friends and associates without whom this work would not have been completed. Foremost among these is Dr. Kent Stevens, whose guidance was fundamental in the development of my graduate career. I must also acknowledge the unconditional support actively offered by my entire family for all of my academic and entrepreneurial endeavors.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

PROJECT GOALS AND OVERVIEW

The purpose of the present work is to evaluate the issues facing a software engineering team interested in deploying an object-oriented methodology for a software development project. Two leading methodologies are evaluated and applied to a small sample software project, noting the subjective experience of both methodologies.

The project case used to study these methodologies is a real system developed at the University of Oregon, called the Object Deschutes Mobile Robot System (ODMR). Like most real-world systems, it does not exist in a vacuum; it has a history that precedes its existence as software, and this history figures in some of the design choices made during development of the system. This history then is worth brief coverage here.

The Deschutes Mobile Robot is a 12 inch diameter by three foot tall cylindrical wireless mobile robot with microcontroller-driven rotation and translation servos, and with Polaroid sonar transducers and bump panels for limited sensory input. It was purchased for undergraduate research and handed over to a class of software engineering students tasked with development of a control program for autonomous operation.

The basic robot hardware is non-programmable, but is equipped with a pair of wireless modems for communication with a host computer running the control program. As a result, any general-purpose computer can be used as the computing environment for the control program, and the host can be changed at will. This means that computational power is essentially unlimited and infinitely flexible.

The first system produced by the undergraduates over the course of several terms of study was eventually dubbed the X Deschutes Mobile Robot System (XDMR); it was so named because it was a graphical client of the X Window System, designed to be run under the SunOS operating system. Also developed was a software-only robot simulator which allowed development and testing without the inconvenience of using a physical robot for each test run.

The XDMR system eventually failed for several reasons. The project had no lead architect, and while the participants had agreed upon a concept of operation, no design documentation was ever produced, so different developers' perspectives diverged. Eventually, despite its simple goals, the implementation, which due to the lack of design documentation was the sole reference material for the project, became riddled in complexity, redundant functionality, and undocumented and misunderstood module interfaces. The system was too entangled in its failings for an undergraduate to digest within a 10-week term, and undergraduate development stalled.

As an independent academic project in software maintenance, I documented and evaluated the XDMR system in its final form. I concluded (uncontroversially) that the project had lost its focus and that while individual algorithms and concepts were valuable, the project would be best served by restarting development from scratch. I subsequently undertook that effort in the ODMR project, the case study for the present work.

The design goals for ODMR were to utilize powerful but simple abstraction mechanisms to provide a practical software framework for research and development using the available Deschutes Mobile Robot hardware, while limiting the size of the system so that it is approachable by inexperienced programmers.

For the purposes of this thesis, I summarize in the following sections the steps taken when analyzing and designing the ODMR system under both the Fusion and Booch methodologies. For clarity and brevity, some details are omitted. In particular, only a representative subset of the generated artifacts (charts, diagrams, and models) is presented, as a full set of project documentation would be prohibitively large. Also, implementation issues are largely ignored, as they are intended to follow in a straightforward manner from analysis and design; the purpose of the methodologies, after all, is to capture difficult decision-making junctions into documented analysis and design steps. Experience with ODMR in particular found this to be the case.

That said, it is clear that with only one architect, it was fundamentally impossible to either simultaneously or sequentially perform two unrelated passes through the software processes studied here. In reality, the project was first seen through a complete cycle to first release with the Booch method. Following this, a cycle of the Fusion process was performed, reevaluating the analysis, design, and implementation decisions made previously but from the perspective of the Fusion method. There are obvious questions which can be raised as to the effects of such an arrangement on the conclusions which might be eventually drawn. However, it does allow one to make a more direct comparison of the mechanisms employed in the respective methods, because the same questions are asked each time. I decided that this direct comparison would be the most valuable contribution this project could make, and is the intended product of this thesis.

The following studies assume a basic familiarity with the methods in use, offering only a discussion of the relative value of the processes. For those not familiar with the methods, the appendix summarizes Booch and Fusion methods. The next chapter describes an application of the Fusion method to ODMR, and is followed by a similar chapter showing application of the Booch method. The intent is to identify the aspects of the methods which we will later compare.

CHAPTER II

THE FUSION METHOD

The primary strength of the Fusion system is its precisely and unambiguously defined procedure, briefly illustrated here on the ODMR problem. Its intent is to firmly guide the developer from concept to product, minimizing confusion which can side-track a development team. Like most systems, Fusion begins with analysis of the project's requirements.

Requirements

ODMR is a software system for control of robot hardware. The robot contains one non-programmable microcontroller for motion and one for sonar sensors placed around its circular enclosure. These controllers communicate via a pair of radio modems to a host computer, which runs the control software.

ODMR is intended to be a flexible control framework into which behaviors can be programmed. The method for building behaviors is to add them into a module which is to be called the Behavior module. This module should have assistance from several other modules in order to carry out the tasks desired of the robot system. Behavior programmers will see the system much like a library against which their individual Behavior code is linked. ODMR thus has some of the characteristics of an application framework, but will be viewed as an extensible application, because most application-level tasks, such as scheduling and control flow, will be handled inside the portion of the system which will not be modified by Behavior programmers.

Analysis

Fusion analysis begins with creation of an object model for the problem domain. This model contains not only those elements expected to be implemented in the delivered executable, but should include all objects, whether they will eventually be represented in software or not. The inclusion of external agents in this stage of analysis frames the problem; a single diagram can describe in the same language both the system and the agents it will interact with.

Communication between robot and host is carried out via messages encoded and transmitted over the radios. These messages need to be interpreted and converted into symbolic system events by an Interpreter module of ODMR. A Knowledge database will store information about the physical environment and the robot device collected from these events. A Controller module will manage the scheduling and communication multiplexing issues. A Behavior module is the locus of planning and reactive logic.

We can represent logical events in the system as a hierarchy of classes, each concrete class representing a concrete defined event type that can be generated by the robot or the system. Reactions will trigger when specified events are generated. Reaction subtypes can be defined for various conditions as needed by the Behavior or Knowledge modules. The object model diagram below in Figure 1 elaborates on these objects and their relationships. This diagram is the first graphical product of the Fusion process. Not a one-time product, it should be a living artifact of the design process which is updated to reflect changing ideas about the landscape of the problem space. Were it decided later, for example, that the Controller and Interpreter modules represent redundant or conflicting centers of control, then the object model diagram would be modified to reflect the change.

The next important artifact created by the Fusion process is the system interface model, which illustrates how the system interacts with its users and other systems. Development of this model formalizes the boundaries of the system, and defines the operations which cross these boundaries, effectively framing the problem.

The system interface model is developed using scenarios of system operation. A notation is provided by Fusion for scenario diagrams. A simple example is given below. Here, we propose a hypothetical behavior, "go home", which maneuvers the robot from its current location in the test room to its 'home' location. For simplicity at this stage of analysis, we assume that there is nothing in the way. This might be a quick way to move the robot back to its battery recharger. The scenario is very simple, and is illustrated in Figure 2.

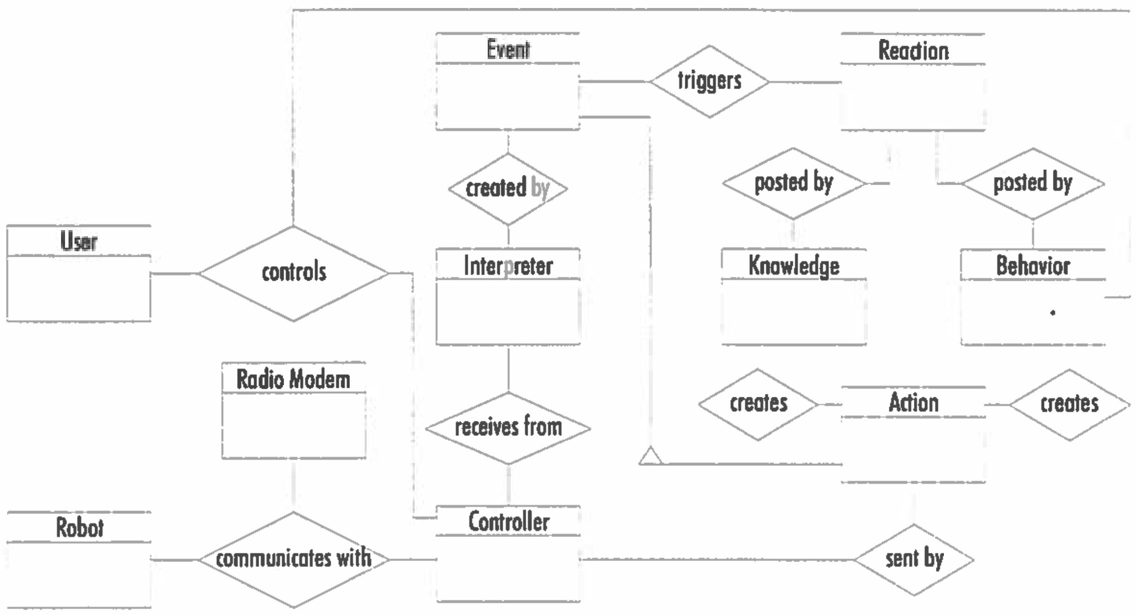


Figure 1: Fusion Object Model Diagram.

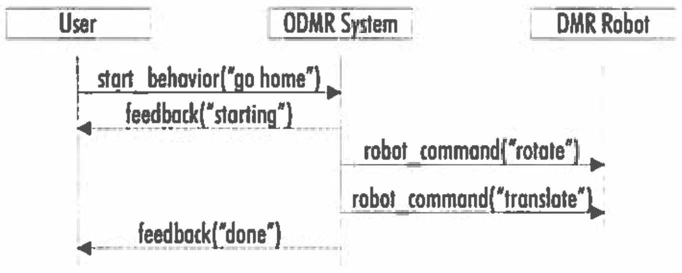


Figure 2: Fusion "Go Home" Behavior Scenario.

There are a very limited number of agents which will interact with the ODMR system, as will be the case with many other kinds of system. Consequently, the system interaction scenarios for the ODMR system are exceedingly simple and all very similar, and we will not explore them further here. Although it is not always necessary, the scenarios can be summarized in a system interaction diagram as shown below in Figure 3. This broad overview of the system and its surroundings helps to concisely convey the place the system holds in its environment; such broad, redundant diagrams are essential for efficiently bringing new members into an existing project.

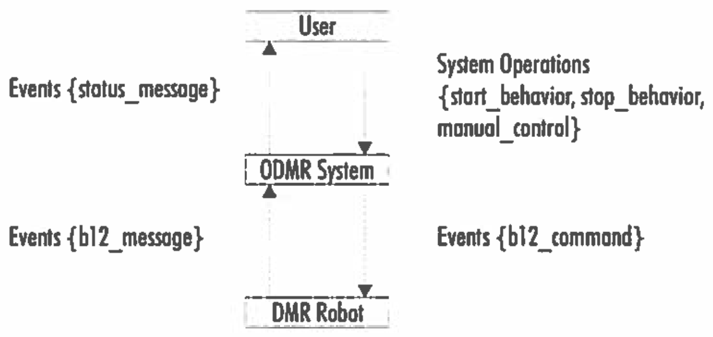


Figure 3: Fusion System Interaction Model for ODMR.

After exploring scenarios, analysis moves on to the interface model. At this stage, architects develop lifecycle graphs for the system. A lifecycle graph is a set of regular expressions defining the 'language' of system operation. All system operations should be expressed in a lifecycle graph, as they will later be used to develop the next set of artifacts.

Lifecycle graphs offer an early place to express the behavior of the system. One complaint which has often been made by newcomers to object-oriented methodologies is that the analysis stages seem very 'static', describing a taxonomy of classes and objects, but only implicitly describing their behavior by labeling relationships among objects. Using Fusion's lifecycle graphs, behaviors can be formalized without interfering with the object model, because lifecycle graphs intentionally omit assignment of responsibility to individual objects. In this sense, they bring some of the intuitive benefits of structured programming to bear on object-oriented projects.

Lifecycle graphs are intended to read as a grammar for the system's operation, so many linguistic conventions apply. Terminal symbols are written in lowercase, while nonterminals are in uppercase. Figure 4 shows a portion of the lifecycle graph for ODMR, illustrating the event generation and reaction triggering mechanism.

```
lifecycle ODMR: Initialization . Operation*

Initialization = start_behavior("initialize")

Operation =     ProcessRobotMessage
                | ProcessUserMessage
                | ProcccessTimer

ProcessRobotMessage =
                process_robot_message .
                        (#invalid_robot_message | GenerateEvent)

GenerateEvent =
                create_event . trigger_reaction*
```

Figure 4: Fusion Lifecycle Graph.

After creating the interface and lifecycle models, we apply the notions of boundaries and operations across them to refine the object model diagram into a what is called a system object model diagram. The notational difference between the object model diagram and the system object model diagram is slight: merely the addition of a dotted line denoting the boundary between the system and its environment, and more comprehensive coverage of the cardinalities of the relationships described within. Anything outside the boundary line is an agent, and all lines crossing the boundary are either event or operation relationships between an agent and a component of the system. During generation of the system object model is also a good time to double-check the semantic soundness of the model, and fill in any important missing details as object and relationship attributes and cardinality relationships. A system object model for ODMR is illustrated below in Figure 5.

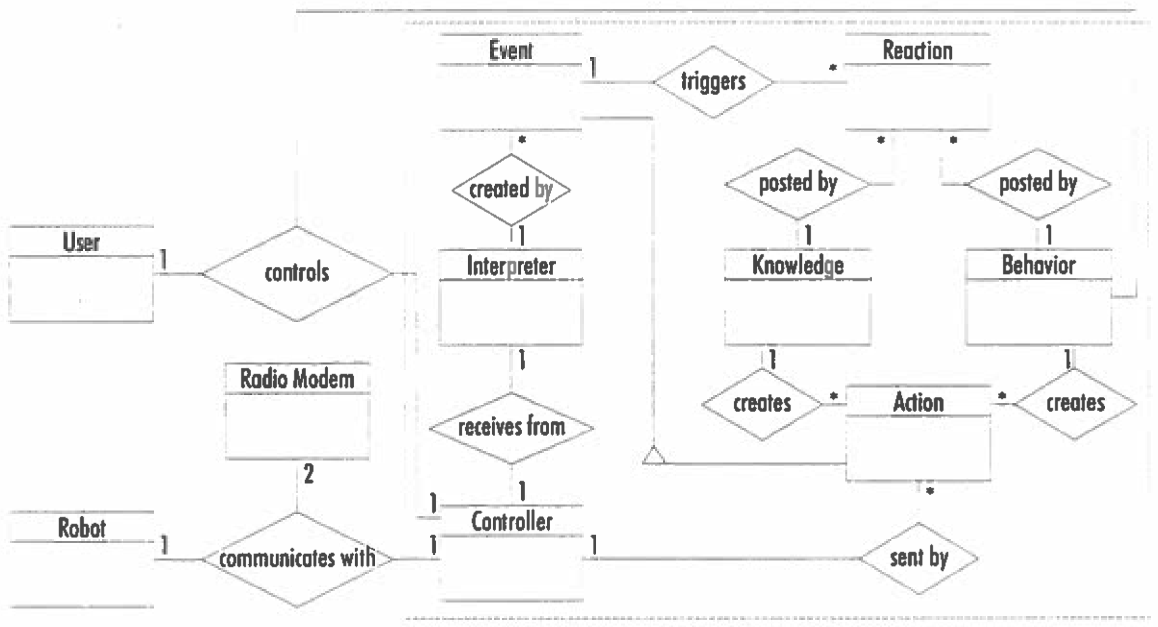


Figure 5: Fusion System Object Model.

An operation model is a set of schemata for each system operation. Each schema is the formal contract for that operation, and defines the semantics for that part of the system's interface with its environment. Schemata can be used to determine if the designer's views of the system agree with the original system requirements, because while sufficiently formal to serve as reference documentation, they are also easy enough to read and digest that they can be compared directly to earlier work.

Figure 6 shows two schemata from the ODMR system. These are for top-level operations visible to the user, *manual_control*, which is invoked by an external user interface to allow manual remote control of the robot by the user (to put it in a specific starting location, for instance), and *start_behavior*, which the user specifies to start a predefined behavior.

```
Operation:    manual_control
Description:  Exert manual control of robot motion by the user
Reads:        supplied direction, supplied servo
Changes:      nothing
Sends:        Robot: {b12_command}
Assumes:      No behaviors are active
Result:       If servo is rotate then
                  If direction is left then
                      Send {b12_command( "rotate left" )}
                  Else if direction is right then
                      Send {b12_command( "rotate right" )}
                  Else
                      Send {b12_command( "rotate stop" )}
              Else
                  If direction is forward then
                      Send {b12_command( "move forward" )}
                  Else if direction is backward then
                      Send {b12_command( "move backward" )}
                  Else
                      Send {b12_command( "move stop" )}

Operation:    start_behavior
Description:  Start a specified behavior (push it on the top of the
                  active behavior stack)
Reads:        supplied behavior
Changes:      nothing
Sends:        nothing
Assumes:      nothing
Result:       behavior is started, suspending active control by
                  the current behavior
```

Figure 6: Fusion Schemata for *manual_control* and *start_behavior*.

The Fusion developers define a procedure for schema creation. The first step is generally to define the *Result* clause. This clause specifies the semantics of the operation; it is generally the aspect of the operation foremost in the designers' minds during this process, so it is natural to specify it first. The *Result* clause is simply an English or pseudo-code description of the actions taken by the operation. It is specified at the level of abstraction most appropriate for communicating the purpose of the operation to a reader, but all possible events, messages, and operations that may be generated are mentioned (with the exception of those exceptional or error-handling conditions which are more properly seen as implementation issues).

Once these *Results* are specified, the events that the operation generates should be summarized in the *Sends* clause. This clause serves as a quick reference, which can be consulted when investigating the possible outcomes of a scenario by tracing through the operation schemata.

Having specified the *Sends* clause, the designer should create an *Assumes* clause listing all assumptions that must be valid for the operation to make sense. These items are in more traditional terms the preconditions for the operation; in order to fulfill their responsibilities of the contract, the clients of the operation must ensure that these assumptions are valid before calling upon the operation. For the example operation of *manual_control*, we have specified that the user should only use manual manipulation of the robot if it is not currently under control of an active behavior. Manipulating the robot while a program is driving it will almost certainly confuse the software, so it is expressly disallowed. As a matter of proper user-interface design, of course, management of this precondition will be the responsibility of the user interface, rather than the end user directly. Explicitly listing the assumptions of the operation in this design artifact commits to paper an essential aspect of the operation.

Finally, the *Reads* and *Changes* clauses should contain the values that are, respectively, read and modified by the operation. These summaries are used to identify data dependencies and coupling issues between classes and operations.

After the completion of schemata for all the system operations, the analysis phase is complete. Designers should pause here to review the consistency and completeness of their analysis products. Check completeness by ensuring that all aspects of the system's operation are described by appropriate scenarios, a schema describes each system operation, and that the system object model completely describes the static composition of the system. Furthermore, a complete data dictionary should be kept throughout the analysis phase and rechecked at this point for agreement with the graphic models.

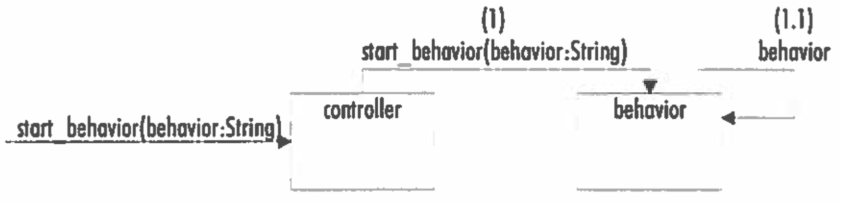
In summary, during the analysis phase of Fusion, designers are involved in a structured study of the problem domain, identifying natural abstractions which can be used to describe the problem, and defining the role of the project system in the domain. Thus analysis, though not as passive as the name implies, is a process of discovery, guided by the experience of the architect. Design, on the other hand, is a process of active invention.

Design

In the Fusion design phase, developers describe a system which will exhibit the properties and behaviors described during the analysis phase. This will involve creating and documenting new abstractions which will cooperate to become the software expression of the system object model. This distinction between analysis and design is useful to make; while a set of requirements will probably only result in a small number of largely similar analysis models, each these models could yield a wide variety of designed systems depending on the designers. The result of design is a specific solution to the problem explored during analysis.

This said, however, Fusion developers realize that software engineering is an organic process that should not always respect strict divisions and dichotomies. Sometimes aspects of design creep into analysis, and portions of analysis are discovered only after some design. This is natural, and shouldn't be feared: the important contribution of Fusion's division is to remind the designer to separate the questions of *what* the system does from *how* it works. If part of an answer to the first question suggests an answer to the second, there is no advantage to artificially suppress it.

Object interaction graphs (Figure 7) are the first artifact of the Fusion design phase. These are graphical depictions of the objects interior to the system and their specific roles during collaboration. For each system operation, an object interaction graph is generated, showing all objects involved in the performance of the desired task.


(1)
start_behavior(behavior:String)
(1.1)
behavior
start_behavior(behavior:String)
controller
behavior

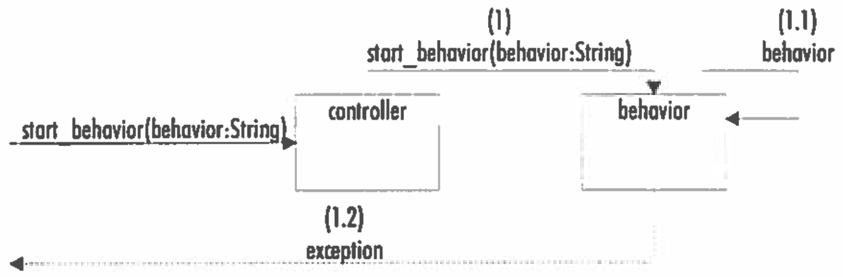
Description:

The controller passes the request to the behavior object (1), which invokes the behavior function (1.1) associated with the named behavior. The implicit call stack interrupts any currently-active behavior.

Figure 7: Fusion Object Interaction Graph for *start_behavior*.

The Object interaction graph for *start_behavior* shows how simple its function is; the controller passes the command to the behavior object, which looks up the behavior name in a table and invokes the proper behavior function. It also implies the system mechanism of using the source language's call stack as a mechanism for stacking active behaviors. These graphs have the disadvantage, from this perspective, that they cannot effectively depict exceptions for source languages that provide them; the diagram notation was not designed with exceptions in mind. In the case of ODMR, this omission is significant, because there are two common ways the start_behavior function can terminate: normal function return, if the behavior completes its task, or via source-language exception, if there is some non-recoverable problem which prevents normal termination. ODMR uses language-level exceptions to express abstract behavior-level conditions such as collision, fulfillment of abortion conditions, and so on. Fusion cannot satisfactorily express these uses directly; the Object Interaction Diagrams illustrates only normal execution, not error handling, even though high-level error handling may be an important part of a system's design.

In order to express important error-recovery procedures using exception-handling mechanisms in an Object interaction graph, the Fusion method developer might express exceptions either as method invocations or events, parallel to and in the reverse direction to the original call. By extending the graph notation as shown in Figure 8, these additional relationships can be expressed at the cost of some complexity and reduced readability of the graph. Such extensions are not part of the Fusion specification, but adhere to its spirit.

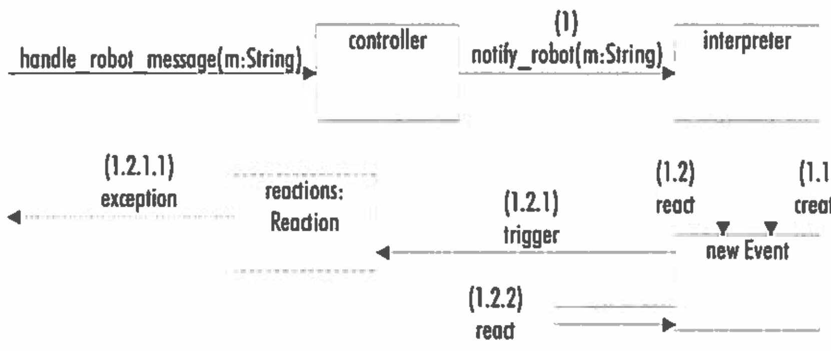

(1)
start_behavior(behavior:String)
(1.1)
behavior
start_behavior(behavior:String)
controller
behavior
(1.2)
exception

Description:

The controller passes the request to the behavior object (1), which invokes the behavior function (1.1) associated with the named behavior. The implicit call stack interrupts any currently-active behavior. In the event of abnormal termination, the behavior function may throw an exception (1.2) to the caller of start_behavior.

Figure 8: Fusion Object Interaction Diagram for *start_behavior* with exceptions.

Because object interaction graphs are central to documenting the responsibilities and relationships of objects, and describing the dynamic behavior of the system, we will look at another example here, *handle_robot_message*. This operation is invoked in response to a *b12_message* event from the B12 robot hardware. The object interaction diagram for *handle_robot_message* is shown in Figure 9.



Description:

The controller collects the robot message and notifies the interpreter (1). The interpreter collects the input, and when it finds a complete message it parses its contents.

Depending on the type of message, the interpreter creates a new event object with the appropriate information about the event (1.1), then instructs the event object to trigger reactions (1.2).

For each reaction registered to the event type, the reaction is triggered (1.2.1). This reaction object may perform virtually any task, including throwing an exception (1.2.1.1) which will travel back to the active behavior.

After triggering all reactions to the event type, the event's base type is instructed to trigger its own reactions (1.2.2), allowing polymorphic event reaction using recursion over the event type's base class.

Figure 9: Fusion Object Interaction Graph for *handle_robot_message*.

This is a more complex operation, resulting in a more complex object interaction graph. There are several aspects of this diagram which are not self-explanatory. Item 1.2.1 is a method invoked on all members of a collection, in this case all reaction objects collected in a registration list. The dotted outline of the target object marks it as a collection. If the collection were to be filtered, the criteria would be indicated on the diagram inside square brackets near the arrow label. If reactions could be individually activated and deactivated without removal from the registration list, for example, the *trigger* method might be annotated with "[active = true]" to indicate that the method is applied only to reaction objects whose *active* attribute is true. The exception event shown as item 1.2.1.1 is another example of a common exception being illustrated as an event as described before. Because this extension to the standard notation may be unfamiliar, it is noted explicitly in the description that accompanies the diagram.

Since the object interaction graphs describe both responsibility and relationship information at a relatively fine level of detail, work on object interaction graphs represents a large portion of the Fusion process documentation and effort. The diagram is somewhat more complex than other portions of the notation, because it resolves more complex details in a compact format, and is a large part of the overall system documentation. Prospective or new Fusion developers should spend time familiarizing themselves with the object interaction graph's uses.

After the object interaction graphs are complete, developers can generate from them visibility graphs for each class. Visibility graphs summarize the collaboration relationships from the perspective of each object, and are useful when writing class interface specifications (such as C++ class declarations). Construction of these graphs is straightforward, and follows almost mechanically from the object interaction graphs. Visibility graphs for the *Controller*, *Interpreter*, and *Event* classes are shown in Figure 10.

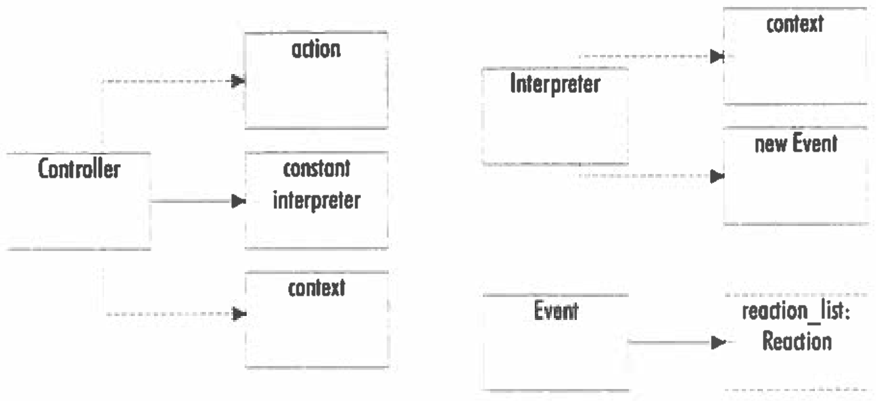


Figure 10: Fusion Visibility Graphs for *Controller*, *Interpreter*, and *Event* classes.

Once again, consistency checking must be done to ensure that all the graphs and diagrams agree with one another. The visibility graphs must identify all object relationships indicated by the object interaction graphs, and all references noted as exclusive in the visibility graphs should be instantiated by exactly one class in the object interaction diagrams. Inconsistent sets of artifacts at this point are serious potential design deficiencies, because not all aspects of the system's operation have been adequately examined.

```
class Controller
        method request( action: Action )
        method handle_robot_message( message: String )
        method handle_user_message( message: String )
        method start_behavior( behavior: String )
endclass

class Interpreter
        method notify_robot( m: String )
        method notify_user( m: String )
        method notify_open( c: Context )
        method notify_close( c: Context )
endclass
```

Figure 11: Fusion Class Descriptions for Controller, Interpreter, and Event.

Class descriptions are a language-independent documentation tool for the Fusion method. They fulfill a purpose similar to C++ class declaration, and have a similar look as well. They are designed, however, to be read by human designers, and to convey all attribute information relevant to design, rather than that relevant to compilation.

The ODMR event class hierarchy is rather large, because there are a significant number of event and action types necessary to capture the events generated by the robot hardware and the actions it accepts. A portion of the class derivation tree is displayed in Figure 12.

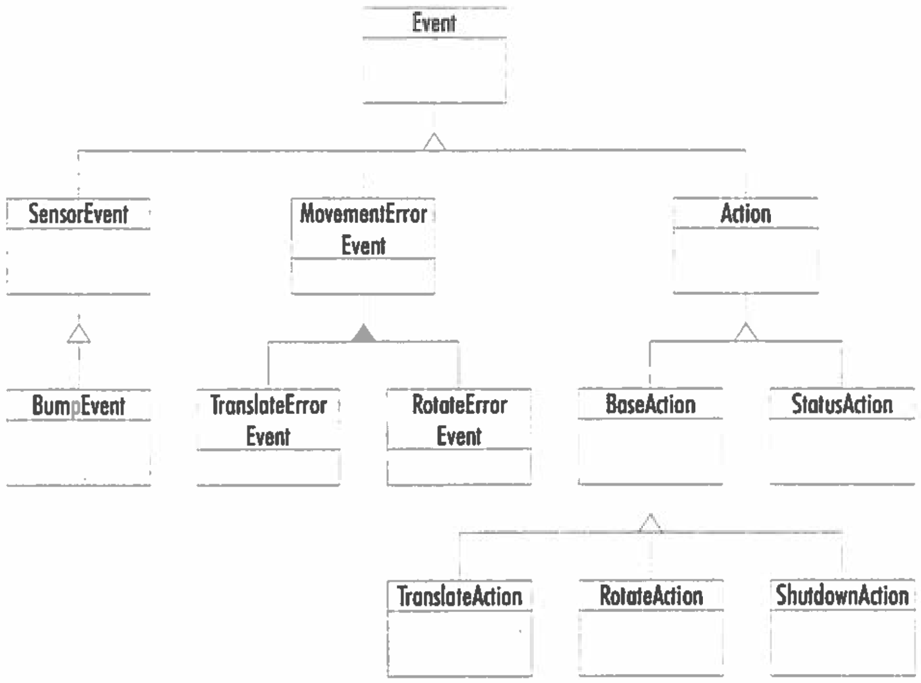


Figure 12: Fusion Inheritance Graph for Event Hierarchy.

There remain no further design steps. Before moving on to implementation, the designers should once again pause to evaluate the design decisions by reviewing all of the design procedures and artifacts. Having examined the whole system, they can now draw on a broader perspective from which to judge the abstractions so far created. After this review, the remaining phase of Fusion method is implementation of the system. These pauses specified in the Fusion method help gain closure for the developers at more regular intervals than whole revision cycles. Since Fusion is less cyclic than many other methods, such instances of closure are valuable opportunities for goal-setting and recognition of outstanding and overdue issues.

Implementation

The fusion method is designed to offer specific instructions and a reasonable guarantee of progress by proscribing detailed procedures and notations for all software engineering activities from problem analysis through implementation. At implementation time, adherence to this regimen pays off, as codification of well-documented design decisions is a nearly mechanical process.

The class descriptions developed late in the design process translate easily into C++ class declarations. Attributes become member variables and methods become member functions. Attribute parameters (constancy, binding exclusivity, etc.) translate easily into C++ declaration modifiers for member variables. Methods are also straightforward; virtual specifiers are added to methods that might be overridden in descendant classes, and type signatures are given directly in the class descriptions.

Many of the necessary method implementations can be achieved by translating object interaction graphs directly into code. Some operations that appear simple in the graphs will be more complex, as some abstractions which are easy to depict graphically are nontrivial to code, or require extensive error-handling or boundary condition checking. For instance, the *notify_robot* method of the *Interpreter* class appears relatively straightforward in Figure 9, but in practice takes some care to implement properly due to the limited capabilities of the available robot hardware; it must perform text processing on potentially incomplete messages, and be prepared to interrupt and restart parsing of partial messages. In addition, the number of different event types is significant, leading to a fan-out of private methods performing the task of interpreting message arguments and creating properly initialized event objects. Finally, a context is maintained between interpreter and controller to ensure that the robot controller hardware, which cannot effectively accept more than one command at a time, is not confused.

Fusion has relatively little to say about the period between initial code generation and software release. Although guidelines for testing and code review are offered, the method focuses on the architect's and designer's task of converting system requirements into software, and views the surrounding tasks as management issues which are best handled when considered separately from core analysis, design, and implementation.

The Software Development Process

Both the Fusion and Booch methods draw a distinction between the micro-development process, as described above, and the "macro-process," or management structure of the project. The overall development macro-process of Fusion is largely similar to a traditional "waterfall" model of software engineering. As such, it seems designed more to encourage an atmosphere of discipline than to offer much real benefit over other management methods, quite in contrast to the intent of the micro-development portion described above. Figure 13 gives a graphical outline of the procedure.

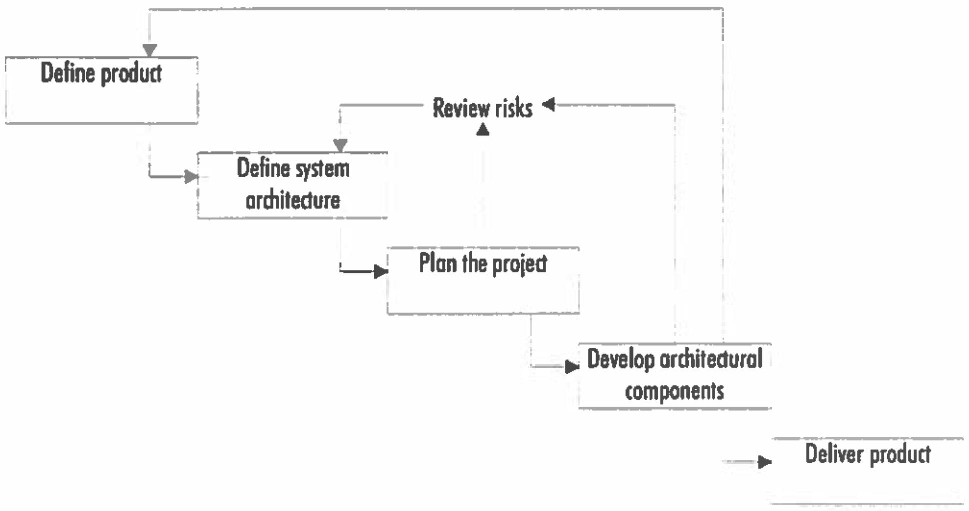


Figure 13: Fusion Software Development Process.

Defining the product entails achieving consensus on the goals of the project, and developing a set of requirements. This task will be highly dependent upon the domain and the development team and organization, as in any software project, but the result must be an understanding of what the project should accomplish. In some situations, it might be the result of a market analysis, while in others it is a clear need to solve a problem. In all cases, it clearly states the reason for the software's existence. A system architecture is a broad overview of the strategy for creating the software. Major sequential system components are determined, each of which will be developed separately, and the data flow between these components is identified. At this point, the paradigm of the system (batch, real-time, graphical, etc.) is identified and specified.

Planning the project properly results in an understanding of its feasibility, measured by its scale and possible risks. It is largely dependent upon the raw experience of the architects and developers to compare the project to existing systems and recognize applicable similarities. Developing the architectural components is the application of the core Fusion process as outlined above, and results in the production of an executable system. Finally, delivering the product moves it into the mature state of continuous testing and maintenance.

CHAPTER III

THE BOOCH METHOD

While the designers of Fusion aimed to provide a systematic method with strict guidelines and well-defined procedures, the method of object-oriented analysis and design created by Grady Booch forms a more abstract approach to software engineering. The Booch method divides the process into two parallel levels, as with Fusion, but in Booch the macro-process is described as that followed by the project team as an organization, while the micro-process is followed by each team member independently. As the project progresses, the goals of project as a whole follow the macro process timeline, while each member independently follows the micro process in an iterative manner. This arrangement, reasons Booch, more closely agrees with the reality of software development. Creativity and productivity, which are the most valuable human resources during analysis and design, come when a developer is personally and individually involved with the work; a process which forces all members into the same single-minded track, while attractive to management due to its strong feedback, does not scale well.

The Booch method, then, is a non-linear progression from requirements to product. This documentation of its application to ODMR is largely concerned with the macro-process as the software engineering tool employed by the development team. Discussion of the macro-process, however, is incomplete without first addressing the micro-process which forms the daily routine of a project team member. Below is an abbreviated summary of the most important aspects of this process as it affects the larger process described subsequently.

The Micro Process

During the progression of a Booch project from concept to product, an individual developer follows a cycle of discovery, invention, and implementation that Booch calls the micro-process of software development. The process starts with an identification of class and object abstractions capturing parts of the problem domain. For ODMR project this meant, for example, describing events and actions as classes in an inheritance hierarchy which could be traversed at run-time to trigger a reaction mechanism. Recognition of such abstractions and their consequences is the goal of this first stage of the micro process. Scenarios, class responsibility and collaborator (CRC) cards, and simplified class and object diagrams are the most useful tools.

Having identified an abstraction or a group of cooperating abstractions, the next task is to determine the semantics of the abstractions and their contribution to the system using more detailed and formal techniques, such as primary and secondary scenarios, detailed class diagrams, state machines, and prototype executable programs for proof of concept. Integral to this task is the investigation of the relationships between abstractions. Before a class may be accepted as a valid abstraction, its relationship with respect to each other class in the system must be understood. Classes with overlapping or conflicting responsibilities result in a confused architecture: proper analysis of class relationships avoids this.

The main system architecture of ODMR was originally created using this Booch method of class and abstraction analysis; the top-level classes were implemented in skeleton form, and synthetic test events were generated to test the interaction of behavior functions and the recursive nature of the program activation stack with the *controller* and *interpreter* objects playing referee. Experience with a vastly-simplified prototype yielding proof of concept allowed the use of this abstraction as a fundamental building block for the entire system.

The final step for any abstraction, of course, is implementation. As each system abstraction is formalized, its prototype can be evolved into an implementation. The advantage to incremental implementation is clear: substantive results can be used to meter the health of the project, and project-threatening problems can be identified much earlier than in a method where implementation is delayed for as long as possible, as is potentially the case with the Fusion method. While the Booch method encourages early implementation of well-understood abstractions, it still warns that even classes whose implementations are complete are subject to review, modification, or possibly elimination as later analysis and design redefines their roles. It is for this reason that the micro process is iterative on a small, rapid scale; it allows structured development with the ability to revise without altering the schedule of the entire project.

The broader view of the Booch method, the macro process, is discussed below.

Analysis

Analysis in the Booch system has a similar character and goal as its counterpart in Fusion, but with a substantially different notation and vocabulary. The starting point for Booch analysis is the requirements as presented above in Fusion. The ODMR requirements are the same, so they will not be reiterated. A useful diagram in the Booch notation similar to Fusion's system interaction diagram is the perhaps slightly misnamed process diagram, which depicts as graphical blocks the system-level entities involved in a problem domain. If desired, specific icons representing the different hardware platforms that make up the system can replace the blocks, but shaded icons tend to represent programmable components, such as embedded CPUs or as in the case of ODMR, application servers, while light icons represent non-programmable elements controlled by the active systems. Figure 14 is a process diagram for ODMR, showing the hardware that makes up the DMR system. Being a control program application, ODMR can take advantage of such a diagram; many applications will not interact with outside hardware in any substantial form, so the process diagram will be less useful.

A substantive difference between this diagram and the system interaction diagram is that the Booch method does not consider interaction with humans in these diagrams, but only other electronic devices. User-interface, it is reasoned, is not well-mannered enough to formally define in the same terms as the operations and events of other software systems and hardware devices.

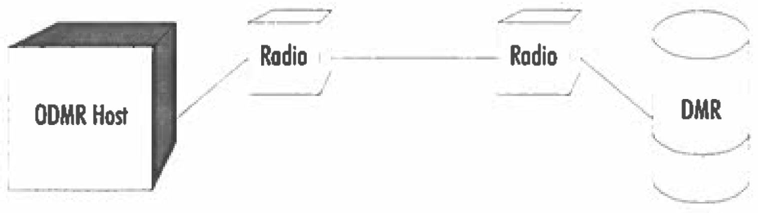


Figure 14: Booch Process Diagram.

Since the micro process is more explicitly decoupled from the macro process in Booch, any of the diagrams in the notation may naturally appear at any point in development. This is useful for communication between team members, because it allows system to be documented as the creative ideas of its developers become available. However, in any software project, it is always possible to become lost in the details; the less stringent scheduling of the Booch method as compared to Fusion means that Booch developers must exercise more self-discipline to avoid becoming lost in a sea of random diagrams. This is avoided by always keeping the current state of the macro process in mind during individual execution of the micro process.

As such, it is common in Booch for inheritance diagrams to appear very early in the process, as the early ODMR event abstraction diagram shows in Figure 15. Inheritance is a very powerful abstraction mechanism; aggressive and effective application requires inheritance to be part of the system concept from its inception. Figure 15 is a class diagram in the Booch notation showing how inheritance is a basic concept in the ODMR event mechanism. The notation is considerably more complex than Fusion's object model diagram, both in symbology and visual style, but there are good reasons for both differences.

A Booch class diagram expresses several related kinds of information about classes at once. As just mentioned, it shows inheritance relationships, using directed arrows from derived class to base. Other adornments shown here include "A" triangles marking abstract base classes, and filled circles attached to open squares, representing containment by reference of objects in the class on the square end of the link. These relationships are also labeled with their description ("registration") and roles ("trigger" and "finished reaction").

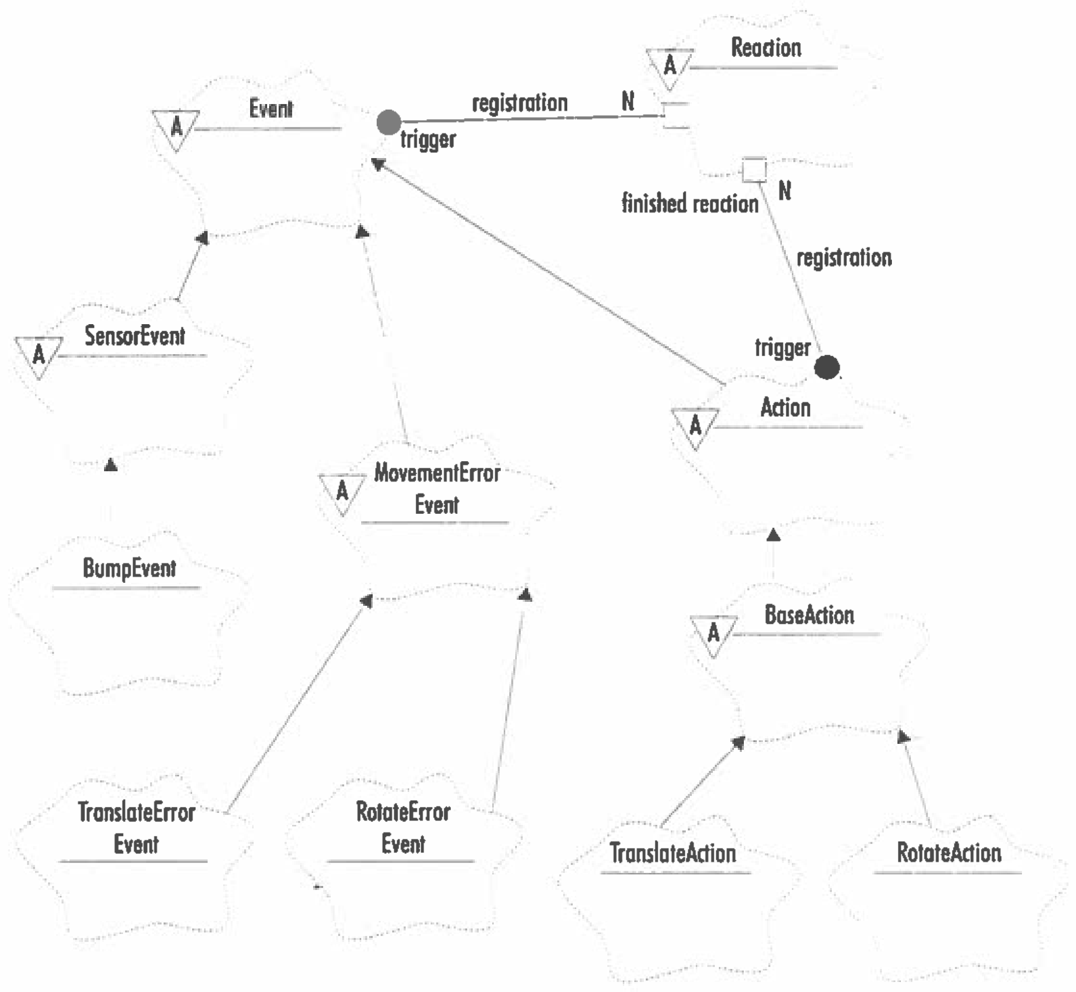


Figure 15: Booch Class Diagram for *Event* Hierarchy.

Two techniques that are actively promoted by Booch for discovering main system abstractions are scenarios and class responsibility and collaborator (CRC) cards. Booch method scenarios and notation are quite similar to those of Fusion, and serve the same purposes, but while Fusion scenarios are intended to express interaction between system agents (such as users) and the system, Booch scenarios are used to describe interaction between any number of abstraction entities internal to the system.

Interaction diagrams graphically illustrate scenarios in the Booch method; a scenario for a *Move* behavior, including intervening events, is illustrated in

Figure 16. Again, the Booch version of the interaction diagram is somewhat more complex, expressing at one time more information about the scenario's details than the Fusion equivalent. In particular, the Booch notation places boxes along the timelines for each object showing the duration of the method's activation, and encourages English descriptions along the left edge to clarify the meaning of clusters of method invocations. These additional elements however, are quite useful for furthering a reader's understanding of the scenario, and overall are a valuable extension to the notation.

CRC cards have been around far longer than the Booch method; they are a very low-tech means of collecting and reviewing the details about class abstractions and how they may interact in the system. A CRC card in its simplest form is just a 3- by 5- inch note card divided into three sections. Across the top is the class name and attributes (such as base class, abstract or virtual nature, and other fundamental characteristics); down the left are the responsibilities of the class; and across on the right are the collaborating classes which work together with the class to fulfill its responsibilities.

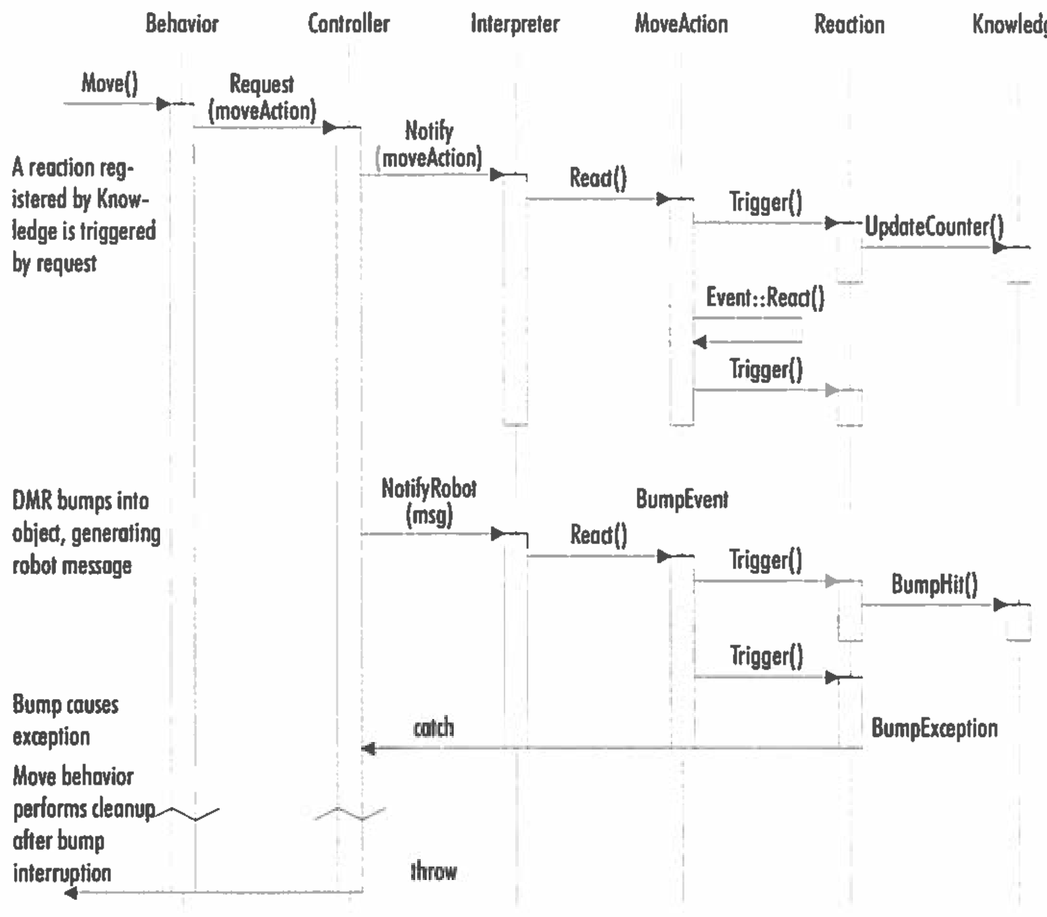


Figure 16: Booch Interaction Diagram for *Move* Behavior.

The cards are somewhat an anachronism in a method such as Booch which in many deployments relies on computer-aided drawing and CASE tools for effective use of much of its notation, but their use is strategically important. CRC cards are a brainstorming and visualization technique; the ability to lay out cards on a table, rearranging or replacing cards by simply taking from and replacing to a deck, is an unobtrusive mechanism which would be difficult to reproduce in a computer. The designer can grab several cards at once and hold them up next to each other to see if the class collaborations agree with one another, or quickly arrange inheritance hierarchies visually. If desired, it can even be a social process involving several team members and a potentially large number of cards.

Design

Transition from analysis to design in the Booch method is gradual, with no border point in between. It is marked by a shift in focus from identification of abstractions to formalization of the relationships and collaborations between them necessary to turn the model into software. In particular, the fundamental product of design is a careful specification of the architectural framework in which the classes and objects will operate. Unlike in Fusion where each step of the method employs different artifacts, the notation for this stage is no different than analysis; it is the character of the goals and the purpose for the diagram elements that have changed. Accordingly, the architectural framework developed during Booch design of the ODMR system is illustrated by another class diagram as in Figure 17.

The class diagram for the architectural framework shows the top-level objects and the collaborative relationships between them. It also shows the nature of the endpoints of the relationships: circles represent the 'source' or controlling side of a relationship, filled meaning ownership of an object, while squares show the kind of aggregation, either filled for by-value or open for by-reference. The dotted line attaching the *Action* class to the *request* relationship of *Behavior* to *Controller* in Figure 17 specifies that the *Action* is the requested item. Any relationship adornments that are left out of a class diagram simply mean that the particular details are unspecified, such as the unspecified number or role of the *Reaction* objects which are "updated" by the *Knowledge* class.

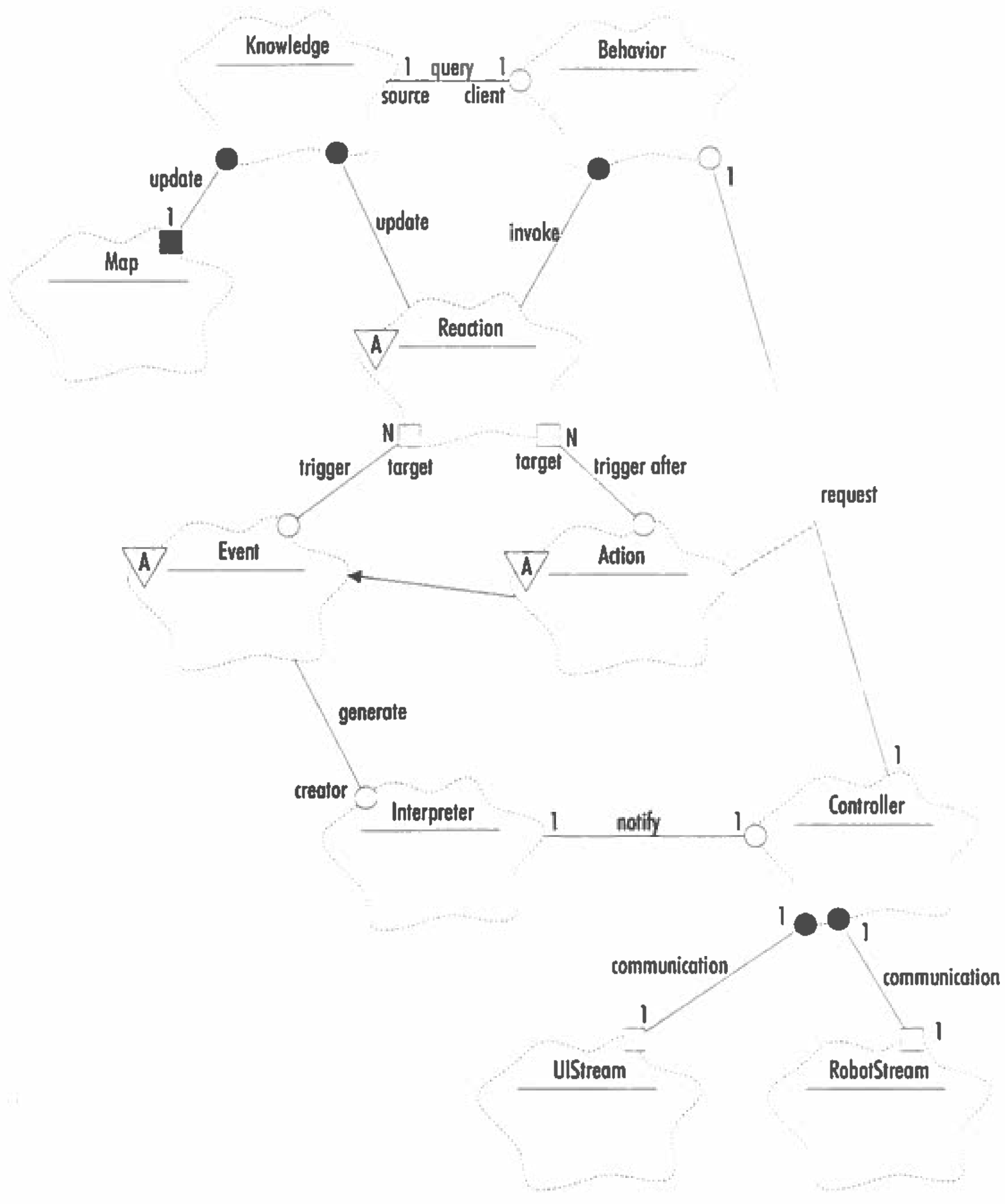


Figure 17: Booch Class Diagram for ODMR Architectural Framework.

The particular visual style of the Booch class diagram notation has been a source of controversy by many users and reviewers of the method. Booch himself, while collaborating with other scientists to create a successor (the "Unified Method") to the current method, has apparently given in to pressure to revise the notation, eliminating the class and object 'cloud' shapes in favor of rectilinear shapes which are ostensibly easier to draw and manage. However, there were and are valid reasons for choosing a distinctive, non-rectangular, and uncommon symbol for the classes. First, as originally reasoned by Booch, the irregular shape is possibly mnemonic for the clustering of attributes and methods that form the very conceptual basis for object-oriented methodology. In addition, the rounded amorphous icon makes the important class and object symbols quickly recognizable to the eye and quite distinct from connecting lines and grouping rectangles which proliferate in more complex and involved applications of the notation. This allows a complex diagram, which would otherwise quickly become a painfully garbled tangle of parallel lines and 90 degree angles, to remain amenable to scanning and tracing by a reader's eye.

One remaining artifact of the Booch method not yet described is the object diagram. Like a cross between the class diagram and a scenario, the object diagram depicts a dynamic slice of the system during a specific operation, with focus on the ontology of objects and their relationships rather than on the order of the action as was illustrated in the interaction diagrams above. Figure 18 is an object diagram for the first part of the scenario given in Figure 16.

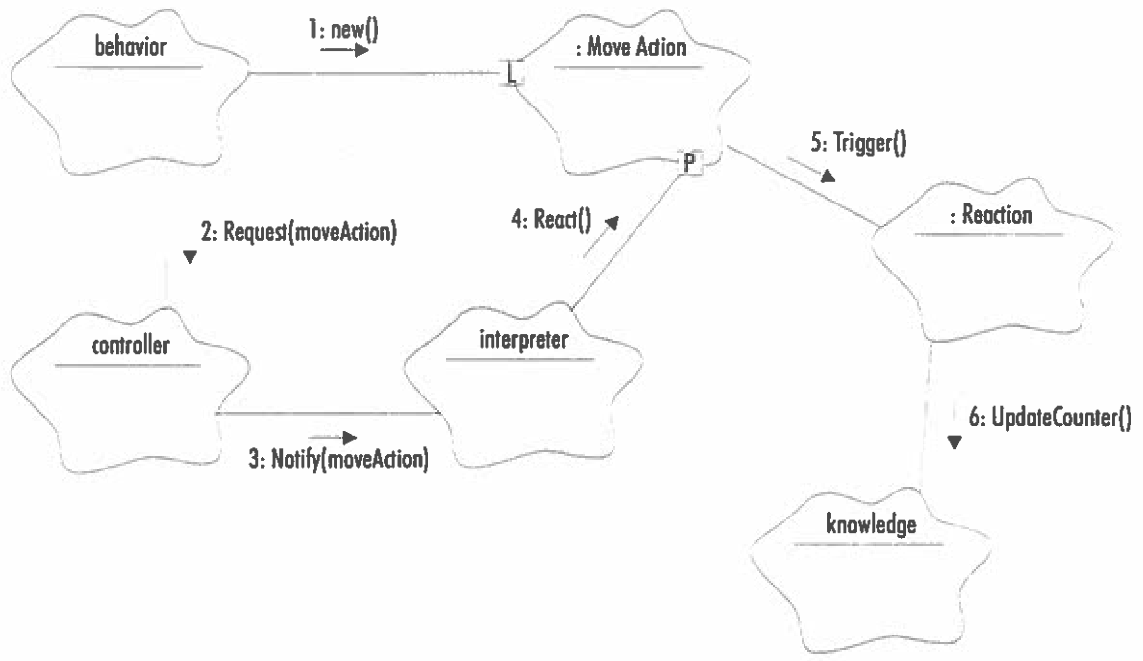


Figure 18: Booch Object Diagram for *Move* Event Scenario.

Evolution

In the Booch method, an evolutionary refinement stage follows design. After the basic abstractions have been designed and agreed upon by the project members, the project consciously shifts into a state of cyclic refinement. Although the basic system logic works with some early-implemented functionality already intact as a result of the design stage, major features are added and performance is analyzed and optimized to reach the goals set forth in the requirements documents. This state is called Evolution in the Booch method, and if differs from design primarily in two ways.

First, system abstractions and the classes created to express them are extended and revised, but rarely are the underlying principles significantly modified. In ODMR, for instance, *Action* classes were originally not a subclass of *Event*, but rather a separate class hierarchy. It was found desirable, however, to allow reactions to be registered to *Action*s as well as *Event*s, so that behaviors could be written to modify other behaviors.

Consequently, the *Event* class was inserted as a base of *Action*, and *Action* was modified to allow reactions either before or after the robot action is performed.

This was modification was performed as a refinement of one working ODMR revision to yield another working program, illustrating the second major difference between design and evolution. The Booch method encourages experimentation via executable prototypes and even early implementation during the design step, but the focus is on architectural development. During evolution, the project places more value on releasable executables as it nears delivery date. The most effective way to manage evolution, then, is to base development on a pattern of incremental extensions to a source base, each aimed at getting the system closer to the goal.

Maintenance

A final difference between Fusion and Booch is the termination point; the published Booch method extends its reach past delivery into the maintenance stage of the software's lifecycle. Since most software spends the vast majority of its useful life in this state, and most researchers seem to agree that software maintenance is the most time-consuming portion of a software engineer's job, a more comprehensive outlook on the software process is a valuable attribute of the method.

In addition to simply extending the tasks begun in the evolution stage, project members in the maintenance state keep a prioritized list of problem items that should be remedied. In many projects, an entirely new team may carry out the maintenance stage, while the development team moves on to begin another project, although in others the development team, which has a maternal familiarity with the software, can be used to rapidly and efficiently implement any desired changes.

CHAPTER IV

EVALUATING THE METHODS

With the case study of the methods in deployment complete, we turn to drawing conclusions and extracting some principles for comparing object-oriented methodologies. To do so in a meaningful way means understanding what makes software engineering difficult, and what a method should offer to mitigate these difficulties.

The primary enemy of the software developer is complexity. In its purest form, a developer's task is to take as input a problem description and produce from it a program executable on a computer that solves the problem. Although computer programs are often comfortingly compared to cooking recipes, the complexity of a solution to a useful problem is much more realistically compared to a recipe for the chemistry of the human digestive system than that for chicken noodle soup.

Complexity in the software development process comes from many sources. The project's problem domain itself is usually the first and most direct source of complexity for team. Reality, it is believed, contains infinite complexity, and any software system tasked with interacting with its environment is tasked with categorizing and generalizing significant partitions of reality in order to act rationally. Even programs which interface only with users bear the unpredictability of human behavior. Furthermore, computer systems themselves, being discrete systems, can exhibit behavior that is notoriously difficult for people to characterize. While the human mind typically thinks in terms of continuous, linear, or differentiable processes, digital computers discretely sample their environment, so small changes in input can result in arbitrarily large changes in output. Finally, as implementation languages grow in power, they inevitably also grow in size and scope, and hence fundamental complexity.

Virtually all aspects of the software engineering task potentially contribute vast complexity. In order to perform any useful work at all, the developer must adopt a well-designed and well-defined process which divides and categorizes the issues into smaller, more manageable pieces in such a way that they may later be reassembled into a complete solution, and which can be performed using the human resources available to the project. We examine here each of these needs as criteria for evaluation of the Fusion and Booch methods.

Decomposition of Tasks

The primary feature of an object-oriented method is its process of task decomposition. The overall task of software development is divided at least into analysis, design, and implementation, and usually others depending on the method. These tasks are further subdivided in various ways, until the individual segments are separately approachable.

As shown, the Fusion method was designed to focus primarily on this facet of the process. By providing an unambiguous roadmap with tools for guaranteeing forward momentum, the designers sought to minimize the risk and uncertainty seemingly inherent in software analysis and design. They have certainly been successful in this respect; the method is straightforward and relatively easy to learn, with a specific sequence of steps from requirements to delivery.

We can be critical, however, that so rigid a structure may be unrealistic. Software is a very flexible medium, and specific problem domains and software architectures are sometimes better served by different development cultures. For instance, safety-critical software, by its nature, demands nearly religious attention to correctness, but may still be well served by object-oriented implementation by virtue of its reputed ability to simplify the resulting software system, reducing the opportunities for errors. Fusion, however, does not address the issue of provable correctness, nor is it clear where the proper techniques should be inserted into the method.

Fusion also downplays the use of aggressively object-oriented concepts such as inheritance and polymorphism, which may result in simpler object-based or component-based architectures, where classes are used primarily for their encapsulation and modularity features, rather than for object typing, dynamic binding, and stronger abstraction. Although over-zealous application of such aggressive linguistic tools has been a criticism of much object-oriented work, disregard for them is equally dangerous.

The scope of Fusion, kept limited so that the method might be easier to adopt, may likewise be a liability. Since the high-level management process discussed briefly by the method's designers is relatively unsophisticated, it seems likely that Fusion's success in real-world projects will depend on its insertion into an already effectively managed organization; young organizations still developing their management strategies may find Fusion inadequate guidance in that respect.

With respect to this criterion, the Booch method is nearly opposite to Fusion in principle. Rather than narrow the focus to disambiguate the process, Booch attempted to provide a flexible method with two parallel processes for the individual and the project as an organization. Furthermore, the Booch macro-process offers a much broader scope, from conceptualization and requirements formulation through delivery and later product maintenance. The procedures offered by Booch form a comprehensive set of resources for an organization using the Booch method.

Inside this method of broader scope, the Booch method also delivers more attention on discovering inheritance, polymorphism, aggregation methods, attribution relationships, class genericism, and other advanced or highly descriptive design and implementation mechanisms offered in modern object-oriented languages. Booch chose not to present a language-independent view of software development, instead discussing constructs specific to C++ and Ada. For the majority of mainstream developers, this is a direct benefit, reducing the semantic gap between analysis and design artifacts, and the implementation tasks that follow. However, with recent rising interest in more special-purpose languages such as Java which offer a slightly different set of mechanisms and abstractions, this slant towards the status quo in languages may become a limitation in the future. For example, analysis using aggressive Booch methods may suggest an architecture which makes extensive use of multiple-inheritance and template genericism, while the chosen implementation language may be Java, which supports neither. Although architects under either method can avoid such problems by understanding the limitations of the intended implementation environment and language from the start, neither method addresses this issue.

The ODMR system was first developed under the Booch method, by a single architect with a small number of additional developers assisting with specific subproblems. Itself a small project, it is not sufficiently large to illustrate any of the issues of scale which will be of interest to most potential users of either Fusion or Booch methods. Reanalyzed under the Fusion method and the design process revisited, few changes to the basic design were made, although it did become apparent that many of the artifacts generated by the Fusion process seemed to oversimplify the relationships compared to their Booch representations.

Notation and Integration of Artifacts

After the method guides the developer through the analysis of the problem domain and again after the design of individual abstractions to model the solution, it must then provide resolution and integration of these products into a system which can be understood by all team members, implemented and finally delivered. The efficiency of a method during this reintegration process is highly dependent on the notation which expresses its analysis and design artifacts.

Notation is an important part of any engineering methodology. The notation a methodology creates is its language, and since the purpose of software engineering is the organization and communication of ideas about software abstractions, the notation's linguistic properties will largely determine its effectiveness. Both Fusion and the Booch method include their own notations. Since these methods use somewhat different sets of artifacts, the notations are difficult to compare on a direct, side-by-side basis. However, we can still establish some objective rules of measure.

One of the most important diagrams in an object-oriented methodology, and the one most directly associated with the system's central abstractions, is the object model. Fusion utilizes an object model directly as one of its artifacts, and expands the model with object interaction graphs, while the Booch method builds one out of class diagrams and object diagrams. The purpose of this artifact is to express the basic relationships between the system's abstractions.

An object model diagram, like any expressive artifact, is subject to the influences of style; even for the same project, it can fill any of several roles depending on the style in which it is drawn. In both methods, the object model diagram is most affected by the level and distribution of detail used when creating the object model. Minute detail can be specified when specific implementation choices are critical to the effectiveness or performance of a system component, or omitted when it would obfuscate the relationship lattice and make a complex collaboration more difficult to understand.

From these desired properties of object model diagrams, we can derive some rules of evaluation. First, since our task is to manage complexity, the notation should be easy to read but expressive. These attributes may seem mutually inconsistent, but a good compromise can be reached after a bit of digression into linguistic and information science. If we view a diagram as a one-way message exchange between a speaker and a listener, we can see that the speaker's job is to encode a complex message (the relationships between abstractions) into a medium (the diagram), while the listener must decode the symbols from the medium to reconstruct the message. The speaker's task is made easier if the language medium is robust and unambiguous, with simple and direct symbols or constructs for each idea to be expressed. The listener can best decode the message if the symbols are unambiguous and quickly recognizable. Hence, the vocabulary of the language must be large enough to cover all possible ideas to be conveyed, constructs of symbols must be simple to decode.

Information science tells us that we can achieve these goals by dividing the elements of the language into classes, each of which is used to resolve different levels of ambiguity. Just as a Huffman code uses varying code lengths to express data of varying frequency, a notation can efficiently express concepts of varying complexity by defining symbols with varying diversity.

To make this abstract discussion concrete, consider an old numeric construct: Roman numerals. While this form of numerals are less practical than the Arabic numerals typically in use today, they illustrate nicely how a notation can use constructs of symbols to optimize communication efficiency.

Most numbers we encounter in everyday life—excluding the increasingly complex realm of personal finance—are very small. To express common situations, such as the number of objects on a table or the number of people in a room, we usually only need precise numbers less than five. The simplest numbering system for this range is to use hash marks. As a result, the first three roman numerals are 'I', 'II', and 'III'. (In fact, through much of the active lifetime of these numerals, the number four was represented in the vernacular as 'IIII'.) Five, as the number of items countable in one hand, in expressed as 'V'. The first five numbers, then, are instantly recognizable and easy for the number-shy to learn.

After these few very common numbers, however, a more concise and efficient mechanism is needed. The first nontrivial construction, addition, is used for symbols 'VI' through 'VIII', followed by subtraction for 'IX' (and the more proper representation for four, 'IV'). The next, more complicated step, is to replace all the symbols to express numbers larger than ten, and concatenate with the smaller numbers to build precise numbers of higher order.

This numerical digression illustrates how simple symbols can be used to make simple concepts very easy to express, while more complex constructs can be added to express ideas of arbitrary size. Careful selection of the symbols and the constructs makes for an efficient language. The languages in which we are interested for this discussion are the notations of the Fusion and Booch methodologies.

The notation of Fusion is simple and direct. Consistent with the linguistic principles suggested above, it is designed to disambiguate specifically the information that is intended to be expressed in the particular diagram. For example, the object model diagram is intended to show two primary categories of information: the objects involved in the system, and the named relationships between them. The notation was designed so that each of these is shown by a distinct symbol, either a rectangle for an object or a diamond for a relationship, and the connective lines are marked with auxiliary annotations such as cardinality and, where appropriate, subtyping. Another look at Figure 5 illustrates the efficiency of the notation.

Fusion, in some cases, uses diagrams to summarize the details of analysis or design. Visibility graphs, an example of which is Figure 10, contain information which is largely already derivable from the object interaction graphs. By gathering them together in one place, however, Fusion intends to simplify the chore of a reader trying to understand the system's operation.

The Booch notation is considerably more complex, as the diagrams are intended to have much more power to express intricate relationships concisely. The Booch class diagram of Figure 17 gives more information about the system architecture than does Fusion's system object model diagram. Part of the additional complexity derives from the 'cloud' symbols and their adornments, as discussed earlier. The shapes of these symbols are more complex and organic than those of Fusion, which are dominated by lines and rectangles. The Booch class diagrams also embrace auxiliary adornments, such as the filled and open circles and rectangles with optional single-letter extensions, the triangular class adornments, and role labeling on relationship arcs. Most of these adornments were shunned when the Fusion notation was designed, but while this decision made many diagrams easier to read, it also placed hard limits on the utility of these diagrams for many projects.

The ODMR project's *Event* mechanism is one example of a system which needs some of the Booch notation's extra expressive power. Figure 15 describes the relationships of the classes derived from *Event* and *Reaction*. The attributes of collection aggregation by reference and the roles of the *Event*, *Reaction*, and *Action* classes would not be easily expressible in a Fusion object model diagram without additional English notes, while they are reasonably simple to express using the Booch class diagram.

Education

The Fusion method post-dates the Booch method, however, and there was another principal reason for its significant reduction in scope. A commercial project is composed of software engineers which are an expensive and scarce resource. The larger a project gets, furthermore, the more sophisticated its developers' communication abilities must be in order to make an effective contribution toward the goal.

In object-oriented methodologies, it is the method and the notation which provide the primary means of communication among architects, developers, and other programmers. If the architects use a method, then the developers and programming teams will need to be able to understand the method and read its artifacts in order to carry out their duties. The more complex and time-consuming it is to learn the method and its notation, the more expensive it is to educate the project members into effective team members.

In light of this fact, the designers of Fusion made not only the notation easier to learn and use, but simplified the process of software development as well. By laying out a rational process with few forking paths, they hoped to create a development environment in which even the least sophisticated member would not feel lost, resulting in a sense of unity and progress, a common ground under which all the members could effectively communicate and work.

It is undeniable that the Fusion method is easier to teach and learn than the Booch method; on the other hand, it is unclear that the entire method must be employed by all members. The architects of the system, who are ultimately responsible for the creation of a workable design, must certainly be fluent in the method in able to steer its course in a strategically sound way. Designers should also be experienced in the method so that they can work efficiently with the architects to create and develop the abstractions on which the end product will be based.

In all but the smallest projects, however, there will be a pool of team members, developers and programmers, who need a less sophisticated familiarity with the method. They will be largely unconcerned with strategic decisions, focusing mostly on the tactical development of small numbers of classes, or even just implementing classes that have been designed and documented by the architects or designers. For these members, it may be necessary only to effectively apply the micro process within the context of the macro process as executed by the architectural team. Indeed, for large projects, it is possible that top-level comprehension of the entire project is unlikely to be possible for any but a few managing team members; in these kinds of projects, Fusion method teams will be unable to take advantage of the common ground the method attempts to offer.

APPENDIX A

FUSION PROCESS SUMMARY

This appendix and the following are distillations of the specifications offered in the literature for the respective methodologies. It serves as a common-format reference for the methods and a model for the kind of meta-analysis which should be performed when an architect is deciding between methodologies.

This appendix is adapted from:

D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, P. Jeremaes, 1994. *Object-Oriented Development: The Fusion Method*, Prentice Hall.

Fusion – Process

Important attributes:

1. The method should be viewed as an ideal for making plans and decisions, and for facilitating production, not a rulebook for developer conduct.

- ANALYSIS
  - Analysis is about describing *what* a system does and is, rather than *how* it does it.
  - Focus is on the domain of the problem and is concerned with externally visible behavior; isolate and specify a model of the system and its environment.
  - Products
    - Object Model
    - System Interface
    - Interface Model
      - Life-Cycle Model
      - Operation Model

- Activities
  - Develop the Object Model.
    - Brainstorm a list of candidate classes and relationships.
    - Enter classes and relationships into a data dictionary.
    - Incrementally produce an object model looking for:
      - Generalizations: "kind of" or *is-a* relationships (inheritance).
      - Aggregations: "part of" or *has-a* relationships.
      - Attributes of classes.
      - Cardinalities of relationships between objects.
      - General constraints that should be recorded in the data dictionary.
  - Determine the System Interface.
    - Identify agents, system operations, and events.
    - Produce the system object model, a refinement of the object model developed in the first step of analysis:
      - Using information from the system interface, identify classes and relationships on the object model that embody the operational state of the system.
      - Identify and document the system boundary to produce the system object model. Use scenarios to represent external agents involved with the system and what their expectations of the system are.
  - Develop an Interface Model.
    - Create a life-cycle model:
      - Generalize scenarios and form named life-cycle expressions.
      - Combine life-cycle expressions to form life-cycle model.
    - Create an operation model. For each system operation:
      - Develop *Assumes* and *Results* clauses.
        - Describe each aspect of the result as a separate subclause of *Results*.

            - Use the life-cycle model to find events that are output in *Results*.
            - Check that *Results* does not allow unwanted values.
            - Add relevant system invariants to *Assumes* and *Results*.
            - Ensure that *Assumes* and *Results* are satisfiable.
            - Update data dictionary entries for system operations and events.
        - Extract *Sends*, *Reads*, and *Changes* clauses from the *Results* and *Assumes*.
    - Check the analysis models for:
        - Completeness against the requirements.
        - Simple name consistency.
        - Semantic consistency.

- DESIGN
  - Software structures are introduced to satisfy the abstract definitions produced from the analysis. Transform the artifacts of analysis into specifications of design.
  - <u>Products</u>
    - Object Interaction Graphs
    - Visibility Graphs
    - Class Descriptions
    - Inheritance Graphs
  - <u>Activities</u>
    - Create Object Interaction Graphs. For each operation:
      - Identify the objects which cooperate to perform the computation.
      - Establish the role of each object:
        - Identify controller.
        - Identify collaborators.
      - Decide on messages between objects (establish contracts).
      - Diagram the interaction of the identified objects: object interaction graph.

  - Check consistency with analysis models: each of the classes in the system object model is represented in at least one object interaction graph.
  - Verify functional effect. Double-check that each object interaction graph describes the same effect as the specification of its system operation given the operation model.
- Extract Visibility Graphs.
  - Inspect all object interaction graphs. Each message on an object interaction graph implies a visibility reference.
  - Decide on the kind of visibility reference required taking into account:
    - Lifetime of reference.
    - Visibility of target object.
    - Lifetime of target object.
    - Mutability of target object.
  - Draw a visibility graph for each design object class.
  - Check consistency with analysis models: for each relation on the system object model there is a path of visibility for the corresponding classes on the visibility graphs.
  - Check mutual consistency: exclusive target objects should not be referenced by more than one class and shared targets are in fact referenced by more than one class.
- Write Class Descriptions.
  - Record methods and parameters from the object interaction graph.
  - Collect data attributes from the system object model and data dictionary.
  - Determine object containment and reference attributes from the visibility graph for the class.
  - Check that all methods and attributes from previous sources are represented.
- Create Inheritance Graphs.
  - Find generalizations and specializations in the object model.

        - Collect common methods in object interaction graphs and class descriptions.
        - Find common visibility in the visibility graphs.
    - Update Class Descriptions
        - Add inheritance information from the inheritance graphs.
        - Check that object interaction graphs are not invalidated by the creation of abstract classes.
- IMPLEMENTATION
    - Translate artifacts into software. Following specific procedures, a revision-zero executable can be generated directly from the design artifacts.
    - <u>Products</u>
        - Executable code
    - <u>Activities</u>
        - Coding
            - Translate system life-cycle.
                - Implement regular expressions as state-machines.
            - Implement class descriptions.
                - Specify interface of the classes in the target language.
                    - Attribute declarations
                    - Method declarations
                    - Inheritance
                - Implement method bodies.
                    - Error handling (violation of preconditions)
                        - Error detection
                        - Error recovery
                    - Iteration
                - Cover data dictionary.

    - Implement functions, predicates, and types that are both found in the data dictionary and used by methods.
    - Ensure that assertions are respected by adding any necessary code to all affected methods.
- Performance analysis (optional).
  - Design for performance and utilize profiling data to focus performance maintenance.
- Review
  - Inspections
    - Simple review of code, both static and dynamic (via analysis of a running system) to ensure that conditions have been preserved through assumptions.
  - Testing
    - Checking state observation and manipulation.
    - Applying algebraic properties such as associativity and identity preservation to member function invocation.
    - Checking that destructors in C++ are consistent with corresponding constructors.
    - Checking proper use of initialization.
    - Checking that casting in C++ is being used in safe ways.
    - Trying to trigger exception-handling capabilities via extreme boundary value inputs.

APPENDIX B

BOOCH PROCESS SUMMARY

The following is adapted from:

Grady Booch, 1996. *Object Solutions: Managing the Object-Oriented Project.* Menlo Park, California: Addison-Wesley.

Grady Booch, 1994. *Object-Oriented Analysis and Design with Applications, Second Edition.* Menlo Park, California: Addison-Wesley.

Booch – Macro Process

Important attributes:

1. The macro process of object-oriented development is one of *continuous integration.*
2. At regular intervals, the continuous process yields executable releases that grow in functionality at every release. Utilize rapid prototyping to reduce cost of these executables and multiply their accessibility.
3. It is through these "milestones and measures" that management can measure progress and quality, and hence anticipate, identify, and then actively attack risks on an ongoing basis. (Risk management through situational awareness)
   - Iterative, incremental releases force the development team to closure at regular intervals.
   - Management can schedule a suitable response and fold these activities into future iterations, rather than disrupt ongoing production.

        - A rhythmic development process allows a project's supporting elements (including testers, writers, toolsmiths, and domain experts) to give timely feedback.

- CONCEPTUALIZATION – ESTABLISH CORE REQUIREMENTS
    - Bracket the project's risks by building a proof of concept
    - <u>Products</u>
        - An executable prototype
        - A risk assessment
        - A general vision of the project's requirements
            - During conceptualization, focus upon establishing the project's vision by quickly developing a fairly broad yet shallow executable prototype. At the end of this phase, throw the prototype away, but retain the vision. This process will leave the project with a better understanding of the risks ahead.
    - <u>Activities</u>
        - Establish a set of goals for the prototype including a delivery date.
            - For projects of modest complexity whose full life cycle is about one year, the conceptualization phase typically lasts about one month.
        - Assemble an appropriate team to develop the prototype, and let them proceed, constrained only by the prototype's goals and schedule.
        - Evaluate the resulting prototype, and make an explicit decision for product development or further exploration. A decision to develop a product should be made with a reasonable assessment of the potential risks, which the proof of concept should uncover.
    - <u>Agents</u>
        - Architect(s), perhaps one or two other developers who collectively continue their engagement through analysis and design.
- ANALYSIS – DEVELOP A MODEL OF THE SYSTEM'S DESIRED BEHAVIOR

- Develop a common vocabulary and understanding of the system's desired behavior by exploring scenarios with end users and domain experts.
- <u>Products</u>
  - A description of the system's "context"
    - Establishes the overall boundaries of the project
  - A collection of scenarios that define the behavior of the system
    - A "use case" specifies a given portion of a system's behavior.
    - A scenario is an instance of a use case, and thus represents a single path through that use case.
  - A domain model
    - Visualizes all of the central classes responsible for the essential behavior of the system.
    - Not merely a product of invention; largely a product of discovery.
  - A revised risk assessment
    - Identifies the known areas of technical and non-technical risk that may impact the design process.
- <u>Activities</u>
  - Scenario Planning
    - Enumerate all of the primary system functions and, where possible, organize them into use cases denoting clusters of functionally related behaviors.
    - Specify the system's context.
    - Make a list of primary scenarios that describe each system function.
    - For each interesting set of system functions, storyboard primary scenarios.
    - Interview domain experts and end users to establish the vocabulary of the problem space. First through techniques such as CRC cards, then more formally through scenarios that define the system's behavior in this vocabulary.

        - As needed, generate secondary scenarios that illustrate special behavior or behavior under exceptional conditions.
        - Organize the web of scenarios that defines a system along two dimensions: first in use cases according to major system functions, then between primary scenarios and secondary ones (representing variations on the theme of primary ones, often reflecting exceptional conditions).
        - Update the evolving domain model to include the new classes and objects identified in each scenario with their roles and responsibilities.
        - Where the life cycle of certain objects is significant or essential to the behavior of the system, develop a scenario or a state machine diagram for this class of objects.
        - Where external events tend to drive the system's behavior, enumerate all such events and analyze their impact on upon the system by tracing these events through the domain model. Discover which classes are responsible for detecting each event, which classes are responsible for handling each event, and which classes are responsible for reacting to each event.
        - Scavenge for patterns among scenarios and the domain model, and express these patterns in terms of generalized scenarios or abstract classes that embody common structure or behavior.
      - Domain analysis (analysis by analogy)
        - Study existing systems that are similar to the one under development.
        - For those areas of great uncertainty, develop a prototype to be used by the development team to validate its assumptions about the desired behavior of the system, or to serve as a basis of communication with end users.
    - <u>Agents</u>
      - Small (2-6) team including architect, analysts/developers, domain experts/end users, testing team member.
- DESIGN – CREATE AN ARCHITECTURE FOR THE IMPLEMENTATION

- Establish a strategy for the solution and tactical policies for implementation by constructing the system's architecture.
- <u>Products</u>
  - An executable and baselined architecture:
    - Exists as a real application that runs in some limited way.
    - Carries out some or all of the behavior of a few interesting scenarios chosen from the analysis phase.
    - Is production-quality code.
    - Either constitutes a vertical slice that cuts through the complete system from top to bottom, or goes horizontal by capturing most of the interesting elements of the domain model.
    - Touches upon most if not all of the key architectural interfaces.
    - Makes a careful set of explicit simplifying assumptions which do not assume away reality.
  - The specification of all important architectural patterns:
    - All well-structured, object-oriented systems embody a number of patterns at various levels of abstraction:
      - Idioms, at the level of the implementation language.
      - Mechanisms, representing important object collaborations.
      - Frameworks, which codifying well-understood solutions to large functional sub-domains.
    - They capture the architect's intent and vision, and propagate that vision in a tangible way.
    - They provide common solutions to related problems, so are necessary for efficient creation of simple systems.
  - A release plan
    - Later product of design which serves as a management planning tool that drives the next phase of development.
  - Testing plan

        - Test criteria, and where possible, actual test scripts.
    - A revised and updated risk assessment
- <u>Activities</u>
    - Architectural planning:
        - Consider the clustering of functions from the products of analysis, and allocate these to layers and partitions of the architecture. Functions that build upon one another should fall into different layers; functions that collaborate to yield behaviors at a similar level of abstraction should fall into partitions which represent peer services. This builds a hierarchy and taxonomy of abstractions.
        - Validate the architecture by creating an executable release that partially satisfies the semantics of a few interesting system scenarios as derived from analysis.
        - Instrument the architecture and assess its weaknesses and strengths. Identify the risk of each key architectural interface so that resources can be meaningfully allocated as evolution commences.
    - Tactical design:
        - Relative to the given application domain, enumerate the common policies that must be addressed by disparate elements of the architecture.
        - Consider whether any existing frameworks can be adapted to satisfy the needs of these mechanisms.
        - For each common policy, develop the scenarios that describe the semantics of that policy. Further capture its semantics in the form of an executable prototype that can be instrumented and refined.
        - Document each policy and carry out a peer walkthrough, so as to broadcast its architectural vision.
    - Release planning:
        - Given the scenarios identified during analysis, organize them in order of fundamental to peripheral behaviors. Prioritizing scenarios can best be

accomplished with a team that includes a domain expert, analyst, architect, and quality-assurance personnel.

- Group these scenarios and assign them to a series of architectural releases whose final delivery represents the production system.
- Adjust the goals and schedules of the stream of releases so delivery dates allow adequate development time, and releases are synchronized with other development activities, such as documentation and field-testing.
- Create a task plan, wherein a work breakdown structure is identified, and development resources are identified that are necessary to achieve each architectural release.

- ▪ Agents
  - Small team of the project's top people: architect and one/two developers.

- EVOLUTION – EVOLVE AND DEPLOY THE IMPLEMENTATION
  - ▪ Refine the architecture and package releases for deployment. Tthis phase typically requires further analysis, design, and implementation
  - ▪ For projects of modest complexity whose full life cycle is about one year, the evolution phase typically lasts about nine months
  - ▪ Products
    - A stream of executable releases exhibiting:
      - A growth in functionality, measured by new scenarios.
      - Greater depth, as measured by a more complete implementation of the system's domain model and mechanisms.
      - Greater stability, as measured by a reduction in the changes to the system's domain model and mechanisms.
    - Behavioral prototypes
    - Quality assurance results
    - System and user documentation
  - ▪ Activities
    - Application of the micro process

        - Release assessment and change management
            - Qualify each release against the project's minimal essential characteristics as well as against certain other predictors of health, such as stability, defect discovery rates, and defect density.
            - Identify the project's next highest risks, and adjust the scope and schedule of the next series of releases as necessary.
    - <u>Agents</u>
        - Staffing reaches its peak:
            - Implementation of the classes and collaborations of classes is specified by the architect and abstractionists.
            - Completion of system functions through the assembly of the classes and patterns invented by the abstractionists and implemented by other application engineers.
            - Completion of behavioral prototypes used to explore design alternatives or new avenues of technology.
        - During evolution, a full 80% of the team should be focused on pumping out each new release. The remaining 20% or so should be assigned to secondary tasks that attack new risks and that prepare the groundwork for the next series of releases.
- MAINTENANCE – MANAGE POST-DELIVERY EVOLUTION
    - Continue the system's evolution in the face of newly-defined requirements.
    - Largely a continuation of the previous phase of development (evolution) except that architectural innovation is less of an issue.
    - <u>Products</u>
        - All those of evolution
        - A punch list of new tasks
    - <u>Activities</u>
        - All those of evolution, plus:
        - Prioritization and assignment of tasks on the punch list

        - Prioritize requests for major enhancements or bug reports that denote systemic problems, and assess the cost of redevelopment.
        - Establish a meaningful collection of these changes and treat them as function points for the next evolution.
        - If resources allow, add less intense, more localized enhancements (the so-called low-hanging fruit) to the next release.
        - Manage the next maintenance release.
    - <u>Agents</u>
        - Typically carried out by a truncated core of the original development team, or by an entirely new team.
        - Virtually identical to that of the development team during evolution, with two exceptions: there is no architect, and there are few if any abstractionists.

<u>Booch – Micro-process</u>

Important attributes:

1. The process is cyclic, like the macro-process, with each path through the process focusing on different partitions of the system or bringing light to a different level of abstraction.
2. It is opportunistic, meaning that each cycle begins with only that which is best known, with the opportunity to refine that work on every subsequent pass.
3. It is focused on roles and responsibilities rather than on functions and control.
4. It is pragmatic, meaning that it achieves closure by regularly building real, executable things.

- IDENTIFYING CLASSES AND OBJECTS
    - Select the right abstractions that model the problem at hand.
    - The development team focuses on discovering the abstractions under the vocabulary of the problem domain.
    - <u>Products</u>

        - A dictionary of abstractions:
            - In most cases either an intellectual product that lives in the minds of its developers, on a collection of CRC cards, or in the form of any related notation; or it is a byproduct of other artifacts, such as a view upon the system's domain model or its executable architecture.
            - As development proceeds, this dictionary will grow and change, reflecting the team's deeper understanding of the problem and its solution.
    - <u>Activities</u>
        - Discovery and invention of abstractions:
            - Examine the vocabulary of those familiar with the domain.
            - Pool the wisdom and experience of parties interested in the product.
            - Use scenarios to drive the process of identifying classes and objects; CRC card techniques are particularly effective at getting interested parties together to work through scenarios.
    - <u>Agents</u>
        - Not every developer has the skills to perform this step, but then, not every developer needs to be involved here. Rather, this activity is primarily the work of a project's architect and abstractionists, who are responsible for a system's architecture.
- IDENTIFYING THE SEMANTICS OF CLASSES AND OBJECTS
    - Determine proper distribution of responsibilities among classes and objects identified up to this point in the development process.
    - This phase involves a modest amount of discovery (to understand the deeper meaning of each abstraction), an equal measure of invention (to determine the right set of roles and responsibilities in the domain and then to attach these decisions to a concrete form).
    - <u>Products</u>
        - A specification of the roles and responsibilities of key abstractions.

    - As design and evolution proceed, these roles and responsibilities are transformed into specific protocols and operations that carry out these contracts.
  - Software that codifies these specifications (that is, their interfaces).
  - Diagrams or similar artifacts that establish the meaning of each abstraction.
    - During analysis, scenario diagrams are quite useful products at this stage to formally capture the team's storyboarding of key scenarios. During design and evolution, it is common to introduce class diagrams, scenario diagrams, state machine diagrams, and other kinds of diagrams as well.
    - Start with the essential elements of any notation and apply only those advanced concepts necessary to express details that are essential to visualizing or understanding the system that cannot otherwise be expressed easily in code.
    - The primary benefit of these more rigorous products at this stage is that they force each developer to consider the pragmatics of each abstraction's deeper meaning. Indeed, the inability to specify clear semantics at this stage is a sign that the abstractions themselves are flawed.
- Activities
  - Scenario planning
    - Select one scenario or a set of scenarios related to a single function point; from the previous step in the micro process, identify those abstractions relevant to the given scenario.
    - Walk through the activity of this scenario, assigning responsibilities to each abstraction sufficient to accomplish the desired behavior. As needed, assign attributes that represent structural elements required to carry out certain responsibilities. CRC cards are a particularly effective technique to use here.
    - As storyboarding proceeds, reallocate responsibilities so there is a reasonably balanced distribution of behavior. Where possible, reuse or

adapt existing responsibilities. Splitting large responsibilities into smaller ones is a very common action; less often, but still not rarely, trivial responsibilities are assembled into larger behaviors.

- Isolated class design
    - Select one abstraction and enumerate its roles and responsibilities.
    - Devise a minimal and sufficient set of operations that satisfy these responsibilities. (Where possible, try to reuse operations for conceptually similar roles and responsibilities.)
    - Consider each operation in turn, and ensure that it is primitive, meaning that it requires no further decomposition or delegation to other abstractions. If it is not primitive, isolate and expose its more primitive operations. Composite operations may be retained in the class itself (if it is sufficiently common, or for reasons of efficiency) or be migrated to a class utility (especially if it is likely to change often). Where possible, consider a minimal set of primitive operations.
    - Particularly later in the development cycle, consider the life cycle of the abstraction, particularly as it relates to the creation, copying, and destruction. Unless there is compelling reason to do so, it is better to have a common strategic policy for these behaviors, rather than allowing individual abstractions to follow their own idiom.
    - Consider the need for completeness: add other primitive operations that are not necessarily required for the immediate clients, but whose presence rounds out the abstraction, and therefore would probably be used by future clients. Realizing that it is impossible to have perfect completeness, lean more toward simplicity than complexity.
- Pattern scavenging
    - Given a reasonably complete set of scenarios at the current level of abstraction, look for patterns of interaction among abstractions. Such collaborations may represent implicit idioms or mechanisms, which

should be examined to ensure that there are no gratuitous differences among each invocation. Patterns of collaboration that are nontrivial should be explicitly documented as a strategic decision so that they can be reused rather than reinvented. This activity preserves the integrity of the architectural vision.

- Given a set of responsibilities generated at this level of abstraction, look for patterns of behavior. Common roles and responsibilities should be unified in the form of common base, abstract, or mixin classes.
- Particularly later in the life cycle, as concrete operations are being specified, look for patterns within operations signatures. Remove any gratuitous differences, and introduce mixin classes or utility classes when such signatures are found to be repetitious.
- Employ pattern scavenging as an opportunistic activity to seek out and exploit global as well as local commonality. Ignore this practice and you run the high risk of architectural bloat which, if left untreated, will cause your architecture to collapse under its own sheer weight.

- Agents
  - This work is generally performed by developers in conjunction with domain experts, and should always be an open process (subject to peer reviews).
  - Pattern scavenging may be undertaken by 'tiger teams' which are tasked with searching out opportunities for simplification.

- IDENTIFYING RELATIONSHIPS AMONG CLASSES AND OBJECTS
  - Products
    - A specification of the relationships among key abstractions
      - These products serve to capture the patterns of collaboration among a system's classes and objects, and represent an important dimension in any system's architecture.
    - Software that codifies these specifications

    - Successful projects try to capture these relationships concretely in the form of code or similar executable artifacts. Typically this means refining the interfaces of classes specified in the previous phases, by introducing semantic connections from one class to another.
  - Diagrams or similar artifacts that establish the meaning of each relationship as well as larger collaborations
    - Because these relationships by their very nature span many individual abstractions, various kinds of diagrams become an even more important product at this phase. The best use of diagrams is to illustrate relationship semantics that are important to the problem, yet cannot be easily enforced by the linguistics of any programming language.
- Activities
  - Association specification
    - Start identifying the relationships among abstractions by considering their associations first. Once these are reasonably stable, begin to refine them in more concrete ways.
    - Collect a set of classes that are at the same level of abstraction or that are germane to a particular family of scenarios; populate this set (via CRC cards, in scenario diagrams, or in class diagrams) with each abstraction's important operations and attributes as needed to illustrate the significant properties of the problem being solved.
    - In a pair-wise fashion, consider the presence of a semantic dependency between any two classes, and establish an association if such dependency exists. The need for navigation from one object to another and the need to elicit some behavior from an object are both cause for introducing associations. Indirect dependencies are cause for introducing new abstractions that serve as agents or intermediaries. Some associations (but probably not many) may immediately be identified as specialization/generalization or aggregation relationships.

  - For each association, specify the role of each participant, as well as any relevant cardinality or other king of constraint.
  - Validate these decisions by walking through scenarios and ensuring the associations that are in place and are necessary and sufficient to provide the navigation and behavior among abstractions required by each scenario.
- Collaboration identification
  - Mechanisms (which represent patterns of collaboration that yield behavior that is greater than the sum of the individual participants in the collaboration)
  - Generalization/specialization hierarchies
  - The clustering of classes into categories
  - The clustering of abstractions in modules
  - The grouping of abstractions into processes
  - The grouping of abstractions into units that may be distributed independently
- Association refinement
  - Given a collection of classes already related by some set of associations, look for patterns of behavior that represent opportunities for specialization and generalization. Place the classes in the context of an existing inheritance lattice, or fabricate a lattice if an appropriate one does not already exist.
  - If there are patterns of structure, consider creating new classes that capture this common structure, and introduce them either through inheritance as mixin classes or through aggregation.
  - Look for behaviorally similar classes that are either disjointed peers in an inheritance lattice or not yet part of an inheritance lattice, and consider the possibility of introducing common parameterized classes.
  - Consider the navigability of existing associations, and constrain them as possible. Replace with simple using relationships, if bi-directional

navigation if not a desired property. Expand these associations if navigation requires significant underlying behavior to carry out.

- As development proceeds, introduce tactical details such as statements of role, keys, cardinality, friendship, constraints, and so on. It is not desirable to state every detail: just include information that represents an important analysis or design position, or that is necessary for implementation.

- <u>Agents</u>
  - The identification of relationships among classes and objects follows directly from the previous step. As such, the activities of this stage are carried out by the same kinds of agents as for the identification of the semantics of classes and objects.

- IMPLEMENTING CLASSES AND OBJECTS
  - Whereas the first three phases of the micro process focus upon the outside view of abstractions, this step focuses upon their inside view.
  - <u>Products</u>
    - Software that codifies decisions about the representation of classes and mechanisms
  - <u>Activities</u>
    - Selection of the structures and algorithms that complete the roles and responsibilities of all the various abstractions identified earlier in the micro process.
      - For each class or each collaboration of classes, consider again its protocol. Identify the patterns of use among its clients in order to determine which operations are central, and hence should be optimized.
      - Before choosing a representation from scratch, consider adapting existing classes, typically by subclassing or by instantiation. Select the appropriate abstract, mixin, or template classes or create new ones if the problem is sufficiently general.

- Next, consider the objects to which you might delegate responsibility. For an optimal fit, this may require a minor readjustment of their responsibilities or protocol.
- If your abstraction's semantics cannot be provided through inheritance, instantiation, or delegation, consider a suitable representation from primitives in the language. Keep in mind the importance of operations from the perspective of the abstraction's clients, and select a representation that optimizes for the expected patterns of use, remembering that it is not possible for optimize for every use. As you gain empirical information from successive releases, identify which abstractions are not time- and/or space-efficient, and alter their implementation locally, with little concern that you will violate the assumptions clients make of your abstraction.
- Select a suitable algorithm for each operation. Introduce helper operations to divide complex algorithms into less complicated, reusable parts. Consider the trade-offs of storing versus calculating certain states of an abstraction.

- Agents
  - This phase requires substantially different skills than the other three of the micro process. For the most part, these activities can be carried out by application engineers who do not have to know how to create new classes, but at least must know how to reuse them properly and how to adapt them to some degree.

BIBLIOGRAPHY

[1] G. Booch, 1996. *Object Solutions: Managing the Object-Oriented Project.* Menlo Park, California: Addison-Wesley.

[2] G. Booch, 1994. *Object-Oriented Analysis and Design with Applications, Second Edition.* Menlo Park, California: Addison-Wesley.

[3] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, P. Jeremaes, 1994. *Object-Oriented Development: The Fusion Method*, Prentice Hall.