1 Introduction

Experience of the industrial use of object-oriented technology indicates that a disciplined software process is the essential factor determining success [4]. Key components of a software process are systematic analysis and design techniques. The first efforts at employing such techniques for object-oriented software development attempted to use traditional methods such as SA/SD ([6] [11]). However it rapidly became apparent that object-oriented methods were necessary because methods based on the functional decomposition of a system clash with the object-oriented approach.

Recently there has been a profusion of object-oriented analysis and design methods coming from a variety of backgrounds. Some, such as HOOD [7] and Buhr's [2] are targetted at the ADA community. Booch's [1] method is similar but has been extended to make it more truly object-oriented. Entity-Relationship Modelling [3] is the basis of methods like Rumbaugh's et al. [8], whereas the responsibility-driven method of Wirfs-Brock et al. [10], stems from an operational view of object interaction.

The question that naturally arises for the practitioner is which is the appropriate method for the type of development they are engaged in. This paper evaluates the aforementioned methods by scoring them against a set of criteria. It is not the goal of the paper to answer the question "Which one is the best?", but rather to show the differences between methods and to allow conclusions be drawn as to their applicability.

The criteria are presented as a set of questions together with some preceding commentary. The aim of the commentary is to clarify the associated question(s) and to indicate some technical terms that may be useful in framing the answer. We do not provide definitions, except in the case of terms that may be ambiguous or unfamiliar to members of the object-oriented community. We then consider how the five selected methods address these issues. The results are summarised in tabular form, the presence or absence of a black dot (•) indicating whether a method supports or possesses a given feature. In rare cases, a question mark is used to indicate that no evaluation can be made due to lack of evidence. A brief explanation of the reasoning behind the most interesting points of comparison accompanies each table.

The document is structured according to the four main categories of criteria: Concepts, Models, Process and Pragmatics. The concepts section addresses the question of what makes a development method object-oriented. The core of all software development methods, namely the models advocated and the process for developing those models, are examined in subsequent sections. Finally, the pragmatics section considers non-technical features of methods such as availability of resources. We also offer some directions for future research and conclusions.

Note In this document, evaluation criteria are indicated by use of italics.

2 Concepts

In order to be considered object-oriented a method should support the expression of those concepts which assume the most prominent role in object-oriented software systems. These concepts are mostly derived from object-oriented programming languages. As with object-oriented languages [9], there is no universally agreed upon set of features (apart from objects) that a method should support. This section introduces criteria for describing the semantics of the object model and thus provides a framework for evaluating the extent to which a method is object-oriented.

2.1 Objects and Classes

The fundamental concept that must be supported by an object-oriented method is the *object*. An object encapsulates its internal state (or attributes) and provides an interface (a set of operations) for manipulating the state. The way information hiding interacts with the type system to provide objects differentiates Ada based methods from those coming from the object-oriented world.

In *class-based* methods the information hiding module, called a class, has an associated type. A *class* is a template which describes the attributes and interface of a set of objects. Object instances are produced by defining class variables.

Package-based methods separate the type system from modules by providing untyped modules for encapsulating data. If a module exports a private data type then each variable of the type produces an object instance; otherwise the module corresponds to a single object.

Both package-based and class-based methods can employ *generic* modules. A generic module or class is a parameterised template that can be instantiated to give a simple module or class. According to the method, the parameters can be types, classes or operations.

A *metaclass* is a template whose instance is a *class object*. A class object has attributes that contain information common to an entire set of objects of one class and an operation to create new instances of that class.

> *Is the method class-based or package-based?*
> *Does it support generic modules and/or metaclasses?*

Evaluation:

> Both Rumbaugh and Wirfs-Brock were developed for object-oriented software and fully support classes. In contrast, HOOD and Buhr are package based. The main focus of Booch's method is class based, although it does provide a notion of packages.
>
> Booch provides generic classes and modules, as well as meta-classes. In Rumbaugh, class maintained information can be indicated on class icons.

Method	Package	Class	Generic	Metaclass
Booch	●	●	●	●
Buhr	●		●	
HOOD	●		●	
Rumbaugh		●		●
Wirfs-Brock		●		

Figure 1: Objects and Classes

2.2 Inheritance

Inheritance is a relationship between classes in which the features of one class, called a *subclass*, are defined in terms of one or more ***superclasses***. This facility permits the incremental development of designs and implementations. In ***single inheritance*** each subclass is allowed just one immediate superclass, whereas ***multiple inheritance*** permits more than one immediate superclass.

A design method can support ***subtype inheritance*** in which the subclass behaves like its superclass for all the operations of the superclass. It may also support *unrestricted* inheritance which permits the subclass to change the signature or behaviour of operations.

Some methods allow the definition of ***abstract classes*** which cannot have instances and exist solely to partially define the properties of its subclasses.

> *What type of inheritance does the method support?*

Evaluation:

> All the class-based methods support single and multiple inheritance. In addition, Rumbaugh supports a number of specific properties such as whether subclasses have overlapping features. Wirfs-Brock devotes considerable attention to the design of inheritance hierarchies. In both methods, abstract classes are discussed extensively.
>
> Wirfs-Brock and Rumbaugh recommend subtype inheritance. Booch explicitly distinguishes subtype inheritance from derived type inheritance.

2.3 Visibility

An object uses another object to perform a service by invoking an operation in the used object's interface. This is called the ***client/server*** relationship between objects and is fundamental because work in object-oriented systems is accomplished by collections of interacting objects. In order for a client to be able to use a server, the

Method	Single	Multiple	Subtype	Unrestricted	Abstract Class
Booch	●	●	●	●	●
Buhr					
HOOD					
Rumbaugh	●	●	●		●
Wirfs-Brock	●	●	●		●

Figure 2: Inheritance

server has to be visible to the client. Methods differ in the degree to which object visibilities can be expressed.

The *aggregation* relationship holds when one object is a component of another object. Components have the same lifetime as the whole and are visible to the whole.

Objects can use objects other than their components. A method may make no restrictions and assume *static global* visibility. Scoping allows visibility to be *restricted statically*. Visibility may be *dynamic*, for example a server object can become visible to a client by parameter passing.

What visibility relationships does the method support?

Evaluation:

The aggregation relationship is precisely defined in HOOD and it is the basis of design. Rumbaugh has an extensive notation to differentiate the various types of aggregation. Booch supports aggregation implicitly. Wirfs-Brock only mentions aggregation in passing.

Most methods assume global scoping except HOOD, which supports name space partitioning, and Booch, which provides mechanisms for expressing and restricting the visibility of names.

Method	Aggregation	Global Scope	Restricted Scope	
			Static	Dynamic
Booch	●	●	●	●
Buhr	●	●		
HOOD	●	●	●	
Rumbaugh	●	●		
Wirfs-Brock		●		

Figure 3: Visibility

2.4 Lifetimes

A method may be restricted to dealing with *static systems* of objects in which all objects have the same lifetime as the system. If this is not the case then the method must contain some facility for dynamically *creating* objects, for example by instantiating a class. It is also desirable to be able to specify object *destruction*. These two operations allow fully general systems to be modelled.

Does the method support object creation and destruction?

Not all objects are transient. There are a number of reasons why mechanisms to maintain objects that live indefinitely are necessary. Some objects may simply outlive one (or all) the executions of a program. In long lived object systems, some objects may have to be written to storage for reasons of resource management. To provide for these circumstances, a method may contain a facility for indicating object *persistence*.

Does the method support object persistence?

Evaluation:

Wirfs-Brock implicitly assumes that objects can be created and destroyed.

Method	Creation	Destruction	Persistence
Booch	•	•	•
Buhr	•	•	
HOOD	•		
Rumbaugh	•	•	•
Wirfs-Brock			

Figure 4: Lifetimes

2.5 Concurrency

Because the real world is concurrent, concurrent objects are often used in the analysis stage to model it. Objects mesh nicely with concurrency since their logical autonomy makes them a natural unit for concurrent execution. However, concurrent sharing is more complex than sequential sharing, requiring mutual exclusion and temporal atomicity. The interfaces, internal structure and communication protocols of concurrent objects are more complex.

Normally objects are *passive*, because they are inactive until an operation is invoked by a client. In contrast *active* objects have their own thread of control and may be

executing when the client attempts to send a message (i.e. invoke an operation). An active object is *internally concurrent* if it has more than one thread of control. Methods should support ways (e.g. monitors) for guaranteeing mutually exclusive access to shared data in concurrent systems.

What models of concurrency does the method support?

Evaluation:

Buhr's method supports the design of concurrent real-time systems. The design of object interfaces, synchronisation and mutual exclusion is dealt with extensively. HOOD also supports concurrency although no mention is made of mutual exclusion. The Booch method supports passive and active objects, which are explicitly labelled as such.

Internal concurrency in Buhr and HOOD is provided by recursive decomposition of objects. The composition of state machines in statecharts supports internal concurrency in Rumbaugh's method.

Method	Passive	Active	Internally Concurrent	Mutual Exclusion
Booch	•	•		
Buhr	•	•	•	•
HOOD	•	•	•	
Rumbaugh	•	•	•	•
Wirfs-Brock	•			

Figure 5: Concurrency

2.6 Communication

Objects constitute a loosely coupled model of computation, in which communication provides both information flow and synchronisation. The usual model of communication is that only two objects are involved in any one communication with the sender having to know the receiver's identity but not vice-versa. The information flow however, may be uni- or bi-directional.

Synchronous communication requires the sender to suspend execution until the receiver accepts the message, whereas ***asynchronous*** communication allows the sender to continue. Further qualifications of synchronous communication are ***balking*** (abort if receiver not ready) or ***timeout*** (abort if receiver not ready after some specified period). Communication is ***reliable*** if the sent message is guaranteed to remain available until the receiver is ready to accept it.

At the analysis stage methods often use an ***event model*** in which the communication is instantaneous and atomic. For design, development methods often use more complex communication primitives like those provided by implementations, e.g.

the *procedure call* for sequential systems and the *rendezvous* and *remote procedure call* for concurrent or distributed object systems.

Mutual messaging between objects is important because of its use in model-view-controller type designs. This category of object communication includes *recursion*, where an object sends a message to itself and *callbacks*, where a server sends a message to a client during the evaluation of a message from a client.

What models of communication does the method support?

Evaluation:

HOOD provides a rich set of communication primitives which are implemented by procedure calls and rendezvous. In Rumbaugh's and Buhr's methods, the event is used during early design and it is refined to a procedure call or rendezvous for implementation. The Booch method supports most of the interesting types of communication, including synchronous, asynchronous, balking and timeout and procedure call for implementation.

HOOD explicitly describes circumstances under which mutual messaging is permitted. Rumbaugh supports recursive messaging.

Method	Synchronous	Asynchronous	Event	Procedure	Rendezvous	Mutual Messaging
Booch	•	•	•	•		
Buhr	•	•	•	•	•	
HOOD	•	•	•	•	•	•
Rumbaugh		•	•	•	•	•
Wirfs-Brock	•					

Figure 6: Communication

3 Models

A development method proceeds by developing abstract descriptions, or *models*, of the system under analysis or design. Each model is expressed in some *notation*. In assessing a method it is necessary to consider the models it constructs and the notations that it uses. The prime requirement is that the set of models should form a *complete* and *consistent* description.

3.1 Kinds of Models

Three kinds of models can be produced. A *physical* model is concrete and concerned with the actual structure of the software system and typically deals with such things

as code modules and processors. *Logical* models capture the key abstractions of the system. Logical models can be separated into *static* models which emphasise the structure of a system and *dynamic* models which deal with temporal and functional behaviour. Another distinguishing feature is whether a model pertains to the *system* or an individual *component*. A component can be atomic, i.e an individual class or package, or a *subsystem*. In cases where more than one model captures the same information there should be rules for checking *consistency* between the models.

> *What models does the method prescribe and what notation is used for each?*
> *Are there any aspects of a system that are omitted or any that are covered by more than one model?*

Evaluation:

> All the methods considered use directed graphs to represent the usage and inheritance structure of a system. Wirfs-Brock uses separate diagrams, whilst Booch and Rumbaugh combine the information in a single diagram.
>
> HOOD uses state machines and/or pseudo-code to define system behaviour. Rumbaugh presents a data-flow diagram to define the functional behaviour of a system in response to an event. Booch uses object diagrams to show the interactions between the objects involved in responding to system events. Booch also uses timing diagrams, to illustrate example scenarios associated with system events. Rumbaugh advocates a textual representation of timing diagrams, event traces, for this purpose. Buhr relies solely on the use of timimg diagrams to illustrate system-level behaviour.
>
> The Wirfs-Brock method does not build models of dynamic system behaviour. Instead the Class-Responsibility-Collaboration (CRC) cards constructed for the system are used for scenario walkthroughs. The collaboration links on the CRC cards duplicate the usage structure shown by collaboration graphs.
>
> Wirfs-Brock has special notations for capturing subsystem structure, whereas the other methods allow system level notations to be applied recursively. The individual CRC cards and contract specifications contain natural language descriptions of component behaviour in Wirfs-Brock's method. The other methods use extended state machines and/or pseudo-code to capture the dynamic behaviour of atomic components.
>
> Rumbaugh and Booch have explicit notations for showing the allocation of classes to modules and active objects to processors. In Buhr and Hood the package is a physical module.

The reader is referred to the summaries in the appendix for the details of each notation.

Method	Logical				Physical
	System		Component		
	Static	Dynamic	Static	Dynamic	
Booch	Class Diagram, Object Diagram	Timing Diagram	Class Diagram	State m/c	Module Diagram, Process Diagram
Buhr	Structure Chart	Timing Diagram	Structure Chart	State m/c, pseudo-code	Structure Chart
HOOD	Obj Defn Skeleton	State m/c, pseudo-code	Obj Defn Skeleton	State m/c, pseudo-code	Ada
Rumbaugh	ER Diagram	DF Diagram, Event Trace	ER Diagram	State m/c	System Arch
Wirfs-Brock	Hierarchy Graph, Collab Graph		Subsystem Card, Collab Graph	CRC card, Contract Spec	

Figure 7: Models

3.2 Notation

This section is concerned with the properties of notations used to capture models. We consider their expressive power, whether their syntax and semantics are well-defined and how well they scale-up. The evaluation for this section appears at the end of section 3.2.3.

3.2.1 Expressivity

The main issue for a notation is *fitness for purpose*. Notations can be pitched at different levels of abstraction: they can use *abstract* or *concrete* data types and can be *declarative* or *operational*. If a notation cannot directly represent the essential concepts of the model, then the user has to encode this representation explicitly in the terms of the notation. This leads to more complex and less easily understood descriptions. These kinds of problem also afflict notations that are too *verbose*.

Are the method's notations appropriately expressive?

3.2.2 Syntax and Semantics

Not only should the notation be sufficiently expressive but it should also be well-defined. The *syntax* of a notation is a set of rules which describe the primitive components of a notation and the legal combinations of those symbols. Notations can be *textual* or *diagrammatic*. There are well-known techniques, such as BNF, for formally defining textual syntax. Techniques for defining the syntax of diagrams are less well-established, however there should be a clear definition of the icons and their legal combinations. A defined syntax is a requirement for effective use and also for automated tool support.

Is there a syntax definition or does the syntax have to be deduced from examples?

The *semantics* of a notation is a set of rules which gives the meanings of the syntactic primitives and their combinations. In general, semantic definitions are more complex than syntactic definitions. A well-defined semantics eliminates ambiguity and is a pre-requisite for advanced tool support such as code generation or simulation. More importantly, a semantics is necessary for allowing analysis and design models to be examined and evaluated during development. There should be rules for *reasoning* about models and for *transforming* one model into another.

Is there a semantic definition or does the semantics have to be deduced from examples?
Does the semantics have a formal foundation?
Is there a logic for reasoning about or transforming models?

3.2.3 Scalability

Scalability is concerned with whether a notation can be used effectively on large systems. Notations need a mechanism for *partitioning* descriptions into smaller and more manageable modules and *composing* the whole from those modules. It should also provide some means of *controlling the visibility of names* across modules, in much the same way as programming languages provide mechanisms for controlling the scope of names.

Does the notation provide a partitioning mechanism?
Are there rules for composing the meaning of a system from the meaning of its modules?
Is there an explicit mechanism for defining the scope of names?

Evaluation:

The notations used in all the methods are informal, expressive and tend to be verbose; this is particularly true of Booch's method.

Booch, Rumbaugh and Wirfs-Brock provide short guides to their notations. HOOD has a reference manual in which textual syntax is defined using in BNF. Buhr does not provide a reference guide to his diagrammatic notation.

All the methods rely on informal examples to explain semantics. However HOOD has design transformation rules and Buhr includes some rules for handling synchronisation. Optimisation rules are included in Rumbaugh.

HOOD has a comprehensive set of rules covering scoping and object visibility. Booch has scaling mechanisms for some types of diagrams and also the scope of names can be indicated. Rumbaugh uses Harel's notation[5] for reducing the complexity of state machines.

Method	Expressivity	Syntax	Semantics	Reasoning & Transformation	Partitioning	Scoping
Booch	•	•	•		•	•
Buhr	•			•		
HOOD	•	•	•	•	•	•
Rumbaugh	•	•	•	•		
Wirfs-Brock	•	•	•		•	

Figure 8: Notation

4 Process

We use the term *process* to characterise the steps that make up a method. A process has two main roles: to drive the development to an appropriate implementation and to assist progress tracking through the definition of milestones and deliverables.

First we look at the context of the software development in which a method is useful and what part of the lifecycle it covers. We then discuss the properties of a process including pragmatic issues such as flexibility and heuristics.

4.1 Development Context

Software development occurs in many different contexts. Most development methods are aimed at *greenfield* developments where there is no previous history of software development and the only environment is that provided by for example an operating system.

Adding functionality and *reengineering* requires a provision for the capture of functionality and the extraction of suitable abstractions of an existing system before the design can be modified to include the new functionality.

> *Does the process provide support for adding functionality to existing systems and reengineering?*

A further kind of development context is that of design with reuse. A process which supports reuse requires a look ahead approach such that the common, useful and hence reusable components can be identified. Once candidates for reuse are identified one can search a library to see if reusable components already exist.

> *Does the process address the issue of design WITH reuse?*

Reusable components and designs have to be developed, they are not just a by product of using objects. A process needs to explicitly provide activities which are intended to identify reuseability and support the development of reusable components and designs. Typically the development of a reusable component will be a

design exercise in its own right, as a reusable component must not only satisfy the immediate needs of the current development but must take a broader view of the requirements for reuse.

Does the process address the issue of design FOR reuse?

Evaluation:

All the methods emphasise greenfield development. The coverage of reuse and reengineering is weak.

HOOD has notations for describing the environment and system context for reengineering and guidance on incorporating library objects but not on developing them. The other methods offer heuristics for developing reusable components. The Booch method addresses design with reuse by providing a notation for indicating the use of libraries.

Method	Greenfield	Reengineering	With Reuse	For Reuse
Booch	•		•	•
Buhr	•			•
HOOD	•	•	•	
Rumbaugh	•			•
Wirfs-Brock	•			•

Figure 9: Development Context

4.2 Coverage of Lifecycle

In this section we identify some of the activities which constitute a software development process. Many methods cover different parts of the lifecycle, not just analysis or design. Therefore it is more useful to describe a method in terms of the development activities it supports. These activities can be combined in various ways to make up a particular process model, for example the spiral model.

The term design is applied very loosely by authors of methods, so that many methods which claim to be design methods also include aspects of analysis and implementation. For our purposes we will use the following definitions:

Analysis The purpose of analysis is to construct the logical model of the system and its environment. At this stage there is an emphasis on describing properties rather than the mechanisms which implement them.

Design In the design phase the system to be built is differentiated from its environment. The logical models produced during analysis are successively refined and made more concrete, and a physical model is produced. The emphasis is on the realisation of the properties as a software structure.

Implementation Implementation encodes the physical and logical models in a particular programming language. At this point in the process all of the structure and behaviour of a system will have been defined, and the emphasis is on providing an encoding of the design using the primitives of a particular language.

Which of these activities does the process support?

Evaluation:

Wirfs-Brock's and Rumbaugh's methods are appropriate for analysis, providing mechanisms and notations respectively to explore the consequences of decisions.

The Booch, HOOD and Buhr methods are strong in design. The former provides annotations in the design diagrams which can be used to capture some implementation decisions. The latter two discuss extensively how to transform abstract models to concrete implementations.

In Rumbaugh, implementation is discussed in detail. In addition, attention is paid to performance, storage considerations and the allocation of components of the system to processors.

Method	Analysis	Design	Implementation
Booch		●	●
Buhr		●	●
HOOD		●	●
Rumbaugh	●	●	●
Wirfs-Brock	●	●	

Figure 10: Coverage of Lifecycle

4.3 Process Properties

A process should be repeatable and flexible so that it can be reused and adapted to meet local requirements. Each process step must be defined in terms of its inputs and outputs, or in some other way. Since one step may produce an input for another step, there are usually constraints on the order in which the steps can be tackled. However a process definition should not force unnecessary sequentialisation. Wherever possible it should allow steps to be overlapped in time, in order to exploit potential parallelism in the development. Similarly deliverables should not be tied to particular notations because this makes it difficult to substitute alternative approaches for particular activities.

Are the process steps well-defined?
Is the process flexible?

The adaptability of a process can be improved by the inclusion of guidelines, or *heuristics*. These provide a means of identifying common situations and tackling them in a previously used way. They should help to identify when it is appropriate to perform a particular activity and how to begin the activity.

Are there heuristics?

The reasoning behind the decisions embodied in an implementation is invaluable during software maintenance. It is important therefore to be able to *trace* the connection between requirements and implementation. Traceability is aided if the deliverable from one step is explicitly refined or developed during some subsequent step. Naming conventions can also help by indicating relationships between models.

Is it possible to locate the origin of design decisions made during the development?

A process should provide mechanisms to *verify* that an implementation meets its requirements. Verification involves demonstrating that at each phase the models are consistent with each other and with those from the previous phase. This requires steps which show consistency by inspections, testing or proof.

Does the process provide for verification?

A process should also include steps for *validating* whether a development meets the customer's needs. This can be done through the construction of executable models. e.g. through simulation or the use of prototypes, or by using notations which allow the properties of models to be deduced.

Does the process provide for validation?

Evaluation:

Buhr presents the underlying philosophy of his design process and presents two variants. The spiral process proposed by Booch is loosely defined. It is described as a set of steps which although they are presented in a particular order, can actually be applied in a number of different ways.

Wirfs-Brock's and Rumbaugh's methods provide many useful heuristics to guide the development process.

Most methods work by successive refinement of the classes and objects so it is therefore possible to trace the evolution of a design. HOOD emphasises traceability and verification of designs at the expense of flexibility. It mandates the verification and documentation of intermediate

designs before proceeding to further stages. The many complicated relationships between the different diagrams used in Booch's method could impede verification of design steps. In contrast, the relationships between the diagrams in Rumbaugh are explicitly discussed.

Rumbaugh and Wirfs-Brock emphasise the use of scenario walkthroughs to validate the models. Booch recommends prototyping to validate design.

Method	Well-defined	Flexibility	Heuristics	Traceability	Verification	Validation
Booch		●	●	●		●
Buhr	●	●	●	●		
HOOD	●			●	●	
Rumbaugh	●	●	●	●	●	●
Wirfs-Brock	●	●	●	●		●

Figure 11: Process Properties

5 Pragmatics

There are many pragmatic concerns that influence a methods uptake in the software engineering community. These concerns can be divided into two categories: those having to do with the human-method interaction and those pertaining to the utility of a method in a particular application domain. Within these two categories further distinctions can be made between those properties that are intrinsic to the method itself and others that are external and possibly even transient in nature. The evaluation for this section appears at the end.

5.1 Resources

A concern when considering which method to adopt are the variety of resources available to support its introduction and use. A course is often an appropriate vehicle for the first introduction. A textbook may be sufficient for more experienced developers and can serve as a reference document. Other discriminants for a method include whether it is supported by more than one consultancy firm or CASE vendor. Similarly the existence of user groups, workshops and conference tutorials tend to suggest that a method is in widespread usage.

What resources are available to support the method?

No matter how straightforward a method is, almost all projects beyond a certain size will require some form of tool support to assist the development of analysis and design models. CASE tools can be distinguished by whether they provide syntactic or type checking. Semantic processing is also desirable; simulation, code generation and proof tools fall into this category. In general the existence of CASE tools encourages the development of defacto standards for the method.

Are there CASE tools available to support the method?

5.2 Accessibility

Users are also concerned with how difficult it is to learn to use the method and once learned, how usable it is. The background required of the user must be taken into account. A distinguishing characteristic of methods is the level of mathematical sophistication required to use its notations.

What background is necessary for someone learning the method?

5.3 Applicability

A method that is targeted at a particular implementation language is likely to have limited applicability, since it may not fit well with languages that have a different underlying semantic model.

Is the method targeted at a specific language?

Methods may be restricted to certain application domains. Rumbaugh et al [8] give the following list as a starting point for what areas a method might reasonably be expected to address:

Batch - A data transformation executed once on an entire input set.

Continuous transformation - A data transformation performed continuously as inputs change.

Interactive interface - A system dominated by external interactions.

Dynamic simulation - A system that simulates evolving real world objects.

Real-time system - A system dominated by strict timing constraints.

Transaction manager - A system concerned with storing and updating data, often including concurrent access from different physical locations.

Distributed system - A system subject to communication latency.

For what application areas is the method suitable?

Evaluation:

Currently, there is no HOOD textbook available although courses are readily available from many European CASE vendors. We do not know the current availability of courses for the other methods.

Wirfs-Brock is the most accessible method as it is based on common sense and there is not much overhead to learning it. Buhr's quirky vocabulary may provide a learning barrier to some.

The Booch method is language independent, although use of a particular target language will determine which of the features can be used. Rumbaugh discusses implementation in object and non object-oriented languages in detail. Buhr's method discusses implementation in Ada and is particularly suited to use in the real-time domain.

Since Wirfs-Brock's method does not deal explicitly with object creation and deletion, it provides no notation for expressing storage management during design.

Both Booch and Rumbaugh present case studies from a variety of domain areas. Rumbaugh also provides a discussion of the application of the method to the areas enumerated above. Buhr is aimed at real-time but is generally applicable.

Method	Course	Book	Tool	Accessibility	Language	Domains
Booch	?	●	●	●	independent	●
Buhr	?	●	●	●	Ada	real-time
HOOD	●		●	●	Ada	●
Rumbaugh	?	●	●	●	independent	●
Wirfs-Brock	?	●	?	●	auto storage mgt	●

Figure 12: Resources and Applicability

6 Conclusion

We have presented a set of criteria for systematically comparing methods and have applied them to five object-oriented analysis and design methods. From the evaluations we conclude:

The Booch method fully supports object-oriented concepts. The notation used is scalable for large developments, although it is somewhat verbose. The method is weak on process and concentrates on design and implementation.

The Buhr method is package based and omits inheritance. Its notations and vocabulary are comprehensive, but quirky. Its process is well-defined and flexible. Buhr is focussed on the design and implementation of real-time systems.

The HOOD method is package based and omits inheritance. The notations are well-defined. The process provides a rigorous and transformational approach to design. The level of definition makes it suitable for large team developments.

The Rumbaugh method fully supports object-oriented concepts. The notations it uses are concise and are borrowed from SA/SD and Harel. The process is well-defined and covers analysis, design and implementation.

The Wirfs-Brock method fully supports object-oriented concepts. The process is exploratory and informal and is thus suited for the individual developer, rather than the large team. The method is appropriate for analysis or high-end design.

The evaluations have highlighted similarities and differences and this may suggest ways of combining methods to deal with particular classes of applications. For instance, attempting to synthesise the Wirfs-Brock and Buhr methods might be a suitable starting point for producing a method for the analysis, design and implementation of object-oriented real-time systems.

Our evaluations have convinced us that the criteria we have used are workable, though not without flaws. In particular, the process properties criteria have proved to difficult to apply in an objective manner. We would like to encourage others to use and improve the criteria by applying them to other methods and we are keen to receive feedback on any such efforts.

7 References

[1] G. Booch. *Object-Oriented Design with Applications.* Benjamin Cummings, Redwood City, CA (USA), 1991.

[2] R.J.A. Buhr. *Practical Visual Techniques in System Design: with Applications to Ada.* Prentice Hall, Englewood Cliffs, NJ (USA), 1991.

[3] P. Chen. The entity-relationship model - toward a unified view of data. *ACM Transactions on Database Systems*, pages 9–36, March 1976.

[4] D. Coleman and F. Hayes. Lessons from Hewlett-Packard's experience of using object-oriented technology. In *TOOLS 4*, pages 327–333, Paris, 1991.

[5] Harel D. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[6] T. DeMarco. *Structured Analysis and System Specification.* Yourdon Press, New York, NY (USA), 1979.

[7] HOOD Technical Group. HOOD reference manual, October 1990.

[8] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, NJ (USA), 1991.

[9] P. Wegner. Concepts and paradigms of object-oriented programming. *OOPS Messenger*, 1(1):7–87, August 1990.

[10] R. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software.* Prentice Hall, Englewood Cliffs, NJ (USA), 1990.

[11] E.N. Yourdon and L.L. Constantine. *Structured Design.* Prentice Hall, Englewood Cliffs, NJ (USA), 1979.

A Booch

A.1 Concepts

Objects and Classes In Booch's method an object is something which has identity, state and behaviour. A class is a template for a set of objects which share a common structure and behaviour. A generic class is a template for other classes. Booch's method also provides for meta-classes. Note that there is a single icon for all these classes and generic or meta-classes are distinguished by the relationships they participate in.

Inheritance Booch supports the inheritance relationship. He separates the notion of subtype inheritance, which simply places constraints on the supertype, from derived inheritance which introduces a new type.

Visibility Booch provides a general use relationship which is specialised into use in interface and use in implementation. Use in interface means that used objects are accessible as parameters whereas use in implementation means that the use of the objects is entirely hidden in the using class. It is also possible to indicate whether objects which interact are visible by one being a component of the other, or being passed as a parameter or being in lexical scope.

Lifetimes Objects which are persistent can be indicated in the template either for an instance or, if all instances are to be persistent, in the class. Creation and destruction of objects can be captured as part of the dynamic model in a timing diagram.

Concurrency Objects which can have their own thread of control can be labelled as *active* in the diagrams. The method does not address the issue of internal object concurrency.

Communication There are five types of message passing:

- **simple** For sequential systems where the message passing is based on procedure call.
- **synchronous** The sender will wait, possibly indefinitely, for the receiver to accept the message.
- **balking** The sender will abandon the operation if the receiver is not immediately ready to accept the message.
- **timeout** The sender will abandon the operation after a specified amount of time if the receiver isn't ready to accept the message.
- **asynchronous** The sender will send the message and continue regardless of the state of the receiver.

A.2 Notation

Booch uses six basic diagrams made up of a small set of icons. Many of these icons can have additional annotations for capturing more detailed design information. This makes the notation very detailed.

Class Diagram (CD) A class diagram shows the existence of classes and the relationships between them. Class diagrams can be organised into chunks called class categories, which can make complicated class diagrams simple to understand. The class diagram is part of the logical design of a system.

Object Diagram (OD) An object diagram shows the existence of objects in a system and the relationships (i.e. message passing) between them. The mechanism by which one object can pass messages to another can be denoted. These include component, parameter or lexical scope. It is part of the logical design of a system.

State Transition Diagram (STD) The state transition diagram shows the state space of a class, the events that cause transitions from one state to another and the actions triggered as a result of state changes. It is part of the dynamic model.

Timing Diagram (TD) Timing diagrams are used to show the dynamics of message passing in an object diagram. There are three notations suggested for this: enumerating the arcs of an object diagram, pseudo code and timing diagrams similar to those used in hardware to indicate the flow of control between objects and methods.

Module Diagram (MD) A module diagram shows the allocation of classes and objects to modules. Module diagrams can be broken up into chunks called subsystems, which can be used to make complicated module diagrams simpler to understand. A module diagram is part of the physical design of a system.

Process Diagram (PD) A process diagram is used to capture the allocation of processes to physical processors. It is also part of the physical design of a system.

Many of the icons used in the diagrams have associated templates which are used to store non-diagrammatic information about the entity, such as a textual description.

A.3 Process

Booch advocates the use of a spiral development model. The steps performed for each cycle in the spiral are:

Identifying Classes and Objects This step concentrates on identifying the key abstractions in the problem domain, and the mechanisms which will implement the required functionality. The products can be as vague as a list of

classes and methods or a formal as complete class and object diagrams. This step would make use of techniques such as domain analysis to perform the identification.

Identify Semantics of Classes and Objects This step focuses on the classes and objects identified in the previous step. It involves identifying the protocol for the classes and objects. The products will be refined versions of the templates and diagrams from the previous step. Booch suggests trying to describe the lifetime of an object from creation to destruction, including the characteristic behaviour.

Identify Relationships between Classes and Objects This step is an extension of the activities of the previous one. Relationships between classes are identified and the dynamic semantics of the key mechanisms established. Techniques such as CRC cards are advocated for this step. The product of this step will be a complete logical design of the system.

Implement Classes and Objects This step involves making implementation decisions about the logical design, including allocation of classes and objects to modules, processes to processors etc. Note that implementation is used in a rather loose way here. Classes at one level of abstraction are *implemented* by classes at a lower level, not necessarily by program code. The products are a refined form of the logical design and some of the physical design of the system.

B Buhr

B.1 Concepts

Objects and Classes The method deals with design for Ada rather than object-oriented languages. The term that is used is *box*, which is an abstraction of an instance of Ada package, i.e. an encapsulated data type. Thus Buhr is a package-based method and the diagrams used in Buhr denote package instances rather than classes. Packages can be parameterised corresponding to generic Ada packages.

Inheritance Inheritance is not dealt with.

Visibility Different kinds of client/server relationships are supported. There is no notion of scope, but objects can be shown as components of other objects.

Lifetimes Objects can be dynamically created and deleted. Persistence is not covered.

Concurrency Active and passive objects are covered, including semi-active objects such as monitors. The active objects have the semantics of Ada tasks. Emphasis is placed on the design of interfaces and efficient mutual exclusion and synchronisation structures.

Communication There are two levels of interaction between objects, *events* which are abstract and used during preliminary design and *visits* which are concrete and used during detailed design.

Events indicate *what* interactions happen without commitment to *how* they take place. They abstract away from a model of synchronisation, however the communication is reliable, no messages get lost. The event model is as follows:

- An event is a named, abstract unit of communication between objects that is independent of the concrete interface details of the objects.
- may have data associated with it.
- an event defines an interaction path between two objects, with *source* and *sink* ends.
- the temporal aspect of an event defines points on timelines when the event is *sourced* and *sunk*.
- Event sourcing and sinking take place instantaneously but possibly take some time to transfer between timelines so that sourcing and sinking times may not be identical.

During detailed design the notion of *visit* is used. A visit is an abstraction of method (i.e. procedure) call, Ada rendezvous etc. One object *visits* another object in order to obtain a service or wait for some occurrence. Visits are one-to-one (i.e. when a object is visiting it cannot be visiting elsewhere at the same time). Visits trigger computation in another object. The visit concept covers:

- Method call in which the computation is performed immediately and the visitor leaves when it is done.
- An Ada entry call, which is like a procedure-call except that it requires the active cooperation-operation of an acceptor to complete.
- Both *synchronous* and *asynchronous* communication between objects are covered. Though asynchronous communication requires implicit buffers.

The same *event* can be implemented by many different visiting patterns.

B.2 Notation

Diagrammatic notations are used to give abstract models of Ada programs:

Structure Chart (SC) is a static model of a system that shows the interconnection between objects. The design process successively refines the structure charts to give more information about interfaces and subsystem structure.

State Transition Diagrams (STD) are used to give a dynamic model of objects at the abstract structure chart level. Buhr suggests the use of Harel's Statechart machines but does not use them in the book. The transitions are events and actions are informally specified in natural language or a MCL (a high level pseudo-code).

Concrete Structure Chart (CSC) is an output of the design process which shows the names, data flows, and mutual exclusion requirements associated with the interfaces of each object are shown. The internal details of each atomic object are shown by MCL pseudo-code. Concrete structure charts are a kind of graphical Ada.

Abstract Controller Machine (ACM) is a state machine together with mappings between events and concrete visits that can be used to model dynamic object behaviour at the concrete structure chart level.

Timeline Diagrams (TD) which show sequences of event flows between components are used to model the dynamic behaviour of a system. Their concrete analogue, visit scenarios, are used at the concrete structure chart level.

B.3 Process

The method covers the design and implementation of greenfield developments. Analysis is not covered at all. Some consideration is given to the design of reusable Ada packages.

Buhr advocates an overall design strategy which proceeds jointly in the temporal and structural domains, paying attention first to external aspects of each, and then to internal ones, while deferring details of functionality (in other words leaving it in stub form). Design is performed in two stages:

Preliminary Design
Partition system under design into subsystems (possibly recursively), informally allocate functionality items to each, and explore the nature of the interactions that need to take place among the subsystems to do the functionality, while deferring details of both functionality and interfaces.

Deferring interface details is important because the underlying nature of the interactions needs to be explored first, to ensure that inappropriate overheads are not built in at the interface level.

Recursive decomposition can proceed by informally doing preliminary design through all levels OR by beginning with a detailed design at some deeply nested level and then moving up to preliminary design at higher levels and then back down again.

Detailed Design
The interfaces are first made concrete to meet the needs of preliminary design. Then the internal details are fixed to achieve the desired temporal and functional behaviour.

In more detail, the steps of the design process are:

Preliminary Design

1. The first realisation of a design is called an *abstract structure chart*. It shows event flows between faceless black box objects. A faceless object has an interface which is yet to be defined; all that is known is its input and output event alphabets.

2. Explore temporal behaviour by showing event interactions on *timeline* diagrams. Show different scenarios for how the system behaves. These may be considered as an abstract form of test-case generation.

3. Where useful, draw Mealy style *abstract state machines* showing how machines behave. It is suggested that the state machines be "discovered" by analysing the event scenarios.

4. Concurrency Commitment and Placement. The abstract structure chart is refined to show which objects are active. A box containing a parallelogram is used to indicate an active object.

5. The recursive decomposition proceeds through all levels until one arrives at a point where primitive internal machinery (i.e. engines) is required.

Detailed Design

1. The concurrent structure chart is further refined to show how the event flows have been mapped into visits. Each event flow is mapped onto a "channel" which shows the mode of communication. That is whether the same object always initiates the visit and whether the communication is synchronous or asynchronous.

2. Resolve interface mechanisms by showing the data flows for each visit. At this point the concrete structure chart shows the external details of each object.

3. *Visit Scenarios* are used to confirm that the expected temporal behaviour will occur. Threaded visit patterns can be drawn on concrete structure charts.

4. *Concrete Structure Charts (internal).* At this point decomposition stops. Pseudo-code or ***abstract controller machines*** are used to define the behaviour of atomic objects.

This design process adopts a structure first approach to design. Buhr also shows how it is possible adopt a temporal behaviour-first approach.

C HOOD

C.1 Concepts

Objects and Classes An object in HOOD is an entity that has internal state and provided and required operations. A class is a template for objects with type and data parameters. Classes may be generic.

Inheritance HOOD does not support inheritance in any form. Partially defined objects like *abstract* objects are refined during design.

Visibility There are two relationships in HOOD – *uses* which is the client/server relationship and *includes* which is an aggregation relationship used during design.

Lifetimes Object creation is supported by class instantiation. Object deletion is not considered. Neither is object persistence handled.

Concurrency Objects may be active or passive. Active objects can be internally concurrent with many threads of control. However, HOOD does not have any mutual exclusion mechanisms.

The *virtual node* object, representing a node in a distributed system, is used for distributed systems' design.

Communication HOOD defines five different asynchronous and synchronous communication primitives:

Highly synchronous the call is to an active object. The client is suspended until the call returns.

Loosely synchronous the call is to an active object. The client waits until it receives an acknowledgment from the server.

Asynchronous the client continues without suspension.

Timeouts the client requests the server to respond within a certain time limit.

Synchronous which is procedure call to passive objects.

Rules for mutual message passing and cyclic calls are defined.

C.2 Notation

Object Definition Skeleton (ODS) which is a template for defining each object – its provided and required operations and its internal state.

Object Control Structure (OBCS) This is a description of the synchronisation between the provided operations and/or asynchronous events for active objects. The standard notation uses Ada rendezvous semantics although state machines are also mentioned for this.

Operation Control Structure (OPCS) This is a description of the implementation of each operation for the primitive objects in the system. The exceptions, both raised and handled, the pseudo code and the other operations called in an operation's implementation are defined.

The features of an Object Definition Skeleton can also be shown graphically. Each field in the Object Definition Skeleton is defined formally in BNF syntax. HOOD aims to allow smooth successive transformations from high-level design to implementation. The semantics are described by example and also in a set of consistency rules for language constructs.

C.3 Process

HOOD spans high level design through to implementation. The process recommends using structured techniques such as SA/SD or SSADM for analysis although no support is given for transforming the output of analysis to HOOD.

There are two phases in HOOD design: architectural design and detailed design with implementation. The goal in architectural design is a complete definition of the system with object definition skeletons for all the objects in the system. Detailed design is concerned with transforming the HOOD description into the implementation language i.e. Ada.

The design strategy is top-down and proceeds by object decomposition. A *parent* object is decomposed into a set of component ***child*** objects which compose to provide the functionality of the parent.

The process of decomposition starts with a root object which is an abstract model of the system to be designed. Each intermediate object is decomposed recursively until the bottom level primitive objects are defined.

Each step has four parts:

Define the context of each object define the interface of the parent object,

Produce initial decomposition define potential child objects and explore how they combine to provide the parent functionality.

Define the child objects' interfaces complete the provided and required operations definition for each child respecting the decomposition rules,

Define the child objects' object definition skeleton complete the child objects' definition. Define the internal state, the object control structure (for active objects) etc.

The steps in detailed design are as follows:

- complete all type, data and exception declarations for each operation

- refine each operation control structure into (Ada) pseudo code
- produce a design prototype making the active objects into subprograms
- incorporate the library objects needed for the system
- generate Ada code for each object

D Rumbaugh

D.1 Concepts

Objects and Classes The method supports objects, classes and metaclasses.

Inheritance The method supports both single and multiple inheritance as well as a number of properties such as whether subclasses have overlapping features.

Visibility The method provides a rich set of aggregation primitives including recursion.

Lifetimes The method provides minimal support for object creation, destruction and persistence.

Concurrency The method supports the expression of inter and intra object concurrency.

Communication The method employs an asynchronous model of communication.

D.2 Notation

The method uses three notations to capture Object, Dynamic and Functional models of the system:

Enhanced Entity Relationship (EER) which captures the main entities of the system under development and their static relationships.

Harel Statechart (HSC) which captures the sequences of events, states and operations that occur between systems of objects.

Data Flow Diagram (DFD) which shows the flow of values from external inputs, through operations and internal data stores, to external outputs.

Various structuring mechanisms for the diagrams are possible: object and event classes can be arranged into a hierarchy, state transition and data flow diagrams can be nested.

D.3 Process

The methodology is presented as consisting of three phases: analysis, system design and object design. The input to analysis is a problem statement and the output is a formal model that captures the objects and their relationships, the dynamic flow of control and the transformation of data. With the formal model as a guide, the system is organised into subsystems during system design. During object design, the analysis models are refined and optimised.

Analysis

In analysis, models which focus on different aspects of what the system is required to do, are developed as follows:

Object Model From the initial description of the problem, identify objects and classes. Prepare a *data dictionary* which consists of descriptions of each class. Identify associations between classes and class attributes. Organise classes using inheritance. Iterate this process eliminating redundant classes and associations.

Dynamic Model Write out *scenarios* of typical interaction sequences. Identify events (signals, inputs, decisions, etc. to or from users or external devices) in each scenario. Show each scenario as an *event trace* **(ET)** - an ordered list of events between different objects. Show events between a group of classes in an *event flow diagram* which summarises events between classes disregarding sequence. Arrange events pertaining to each object in a *state transition diagram*.

Functional Model Identify input and output values (parameters of events between system and outside world) from problem statement. Construct *data flow diagrams* showing how each output value is computed from input values. Write a *description* of each function. Identify constraints between objects. Specify optimisation criteria.

Refinement The models are verified and refined iteratively using more detailed *scenarios*.

Key class operations are determined from the models as follows:

- Object Model - reading and writing attribute values and association links.
- Events - an event sent to an object corresponds to an operation.
- State Diagrams - activities and actions may be operations.
- Functions - a function in the data flow diagram corresponds to an operation on an object.
- Real-world behaviour of classes.

System Design

System design involves deciding on the organisation of the system into subsystems and the allocation of subsystems to hardware and software components.

The steps in constructing the system design are:

- Organise the system into subsystems.
- Identify concurrency.

- Allocate subsystems to to processors and tasks.
- Choose an approach for management of data stores (e.g. files stores or databases) and global resources (e.g. physical units such as processors, space such as disk space, logical names, and shared resources such as databases).
- Choose the implementation of control: event driven, concurrent systems or procedure driven.
- Handle boundary conditions and set tradeoff priorities.

Common architectural frameworks are discussed, along with the relevance of the three models. The output of this process is normally a *system architecture diagram* (SA) which captures the static relationships between the major subsystems.

Object Design

Object design involves further refinement of the initial models to address the requirements of an execution environment.

Classes are carried from analysis into design. As the design phase proceeds, classes may be added or broken up for efficiency. The class structure may have to be adjusted to increase inheritance by abstracting common behaviour out of groups of classes or using delegation where inheritance is semantically invalid. For each object, the representation must be chosen; choices include using primitive types or other objects. Object associations must be designed taking into account traversal direction and multiplicity. Then algorithms to implement operations to optimise measures such as ease of implementation, understandability and performance are designed.

E Wirfs-Brock

E.1 Concepts

Objects and Classes Both objects and classes are supported with standard definitions.

Inheritance The method supports single and multiple inheritance, abstract and concrete classes.

Visibility The using relationship is supported.

Lifetimes There is no explicit discussion in the method of the lifetime of object instances. Consequently there is no support for managing dynamic object creation and deletion. It is implicitly assumed that instances can be created and destroyed as required.

Object persistence is not supported.

Concurrency No consideration is given to whether objects are active or passive, and thus no support is provided for concurrency.

Communication Communication between objects is only discussed at an abstract level and uses the client-server model. No detail is given on whether the communication is synchronous etc.

E.2 Introduced Terminology

Responsibilities The knowledge an object maintains; the actions an object performs. The first of these is refined into attributes, and the second into method signatures.

Collaborations Collaborations represent requests from a client to a server in fulfillment of a client responsibility.

Contract A set of requests that a client can make of a server. The server is bound to respond to these requests.

E.3 Notation

Class Responsibility Collaboration (CRC) cards are used throughout the design process to record information relating to classes. On a CRC card, its super and subclasses are recorded as well as the responsibilities of a class, together with collaborator classes. Towards the end of the design process, the CRC card is developed into a class specification.

Subsystem Card (SC) On a subsystem card, a short description of the subsystem is recorded. Contracts required by clients external to the subsystem are noted together with the class within the subsystem which supports the contract. Towards the end of the design process, the SC card is developed into a subsystem specification.

Hierarchy Graphs (HG) are a standard representation for inheritance hierarchies. Single and multiple inheritance can be denoted as well as concrete and abstract classes.

Venn Diagram (VD) are used as a tool to explore and refine inheritance hierarchies. Each class is viewed as being a set of responsibilities. Common responsibilities are drawn in the overlapping part of the venn diagram and independent parts in the non-overlapping parts.

Collaboration Graphs (CG) are a notation used to display and analyse the paths of communication between classes. In collaboration graphs, classes, inheritance relationships, contracts and collaborations are represented.

Contract Specifications (CS) are templates in which the details of the contracts are filled out. A contract specification contains the name of the server and clients which collaborate to fulfill the contract, as well as an informal description of what the contract does.

E.4 Process

The process is presented as a set of sequential steps, but it is clearly acknowledged that design is an iterative and incremental activity. The method strongly advocates 'walkthroughs of scenarios'. These are performed for two reasons. Firstly they entail exploration of the domain area, thus helping understanding. Secondly they can be used to check that required behaviour of the system has not been omitted. Walkthroughs can be done at any time during the process.

The assumed input to the process is a natural language requirements specification and the output is:

- A graph of each class hierarchy
- A graph of the paths of collaboration for each subsystem
- A specification of each class [1]
- A specification of each subsystem
- A specification of the contracts supported by each class and subsystem.

[1] In this context, specification means certain information recorded about the class, for example its super and sub-classes, and for each responsibility, a method signature together with a description of the behaviour of the method.

The main steps in the process are identified as:

Exploratory phase

- Identifying the classes
  The main approach advocated here is noun phrase analysis.
- Identifying the responsibilities
  Techniques mentioned to achieve this are: recalling the main purpose of the classes and verb phrase analysis.
- Identifying the collaborations
  Collaborations are found by going through all of the responsibilities and identifying which objects are needed to fulfill them.

Refining the design

- Building and refining class hierarchies
  The class hierarchies are reviewed and improved. Abstract and concrete classes are identified.
- Identifying the subsystems
  Subsystems are a sets of classes which collaborate closely together to fulfill a set of responsibilities. Subsystems should form good abstractions of components of the system.
- Constructing the protocols for each class
  In this step, the responsibilities are refined into a sets of protocol signatures. Then a specification is written for each class, subsystem and contract.