

1

Software engineering perspectives



In the previous chapter we looked at idioms and patterns that resulted from object-oriented software development. In this chapter we will focus on the software engineering of object-oriented systems and issues of design in particular, including the identification of objects and the specification of contractual obligations.

Software engineering perspectives

- methods of development
- the identification of objects
- contracts – refinement
- validation – a formal approach

Additional keywords and phrases: *requirements, analysis, implementation, design as transition, CRC cards, responsibilities, heuristics, contractual obligations, validation*

31-1

Slide 1-1: Software engineering perspectives

First we will explore what methods are available to guide us in the development of object-oriented systems. Then we will look more closely at the heuristics of actual design. After establishing what is involved in specifying contractual obligations, we will discuss what is needed for a more formal approach to object-oriented development.

1.1 Development methods

Object-oriented software development is a relatively new technology. Consequently, ideas with respect to methodologies supporting an object-oriented approach are still in flux. Nevertheless, a plethora of methods and tools does exist supporting object-oriented analysis and design. See slide ??.

Methods		1-2
• OOA/D – <i>incremental</i>	[CY91b]	
• Objectory – <i>use-case analysis</i>	[Jacobs92]	
• OOSA – <i>model-driven</i>	[Kurtz90]	
• OOSD – <i>structured</i>	[Wasserman89]	
• CRC – <i>cards</i>	[BC89]	
• RDD – <i>responsibility-driven</i>	[Wirfs89]	
• OMT – <i>object modeling</i>	[Rum91]	
• OOD – <i>development</i>	[Booch91]	
• Fusion – <i>lifecycle</i>	[Fusion]	
Unified Modeling Language – <i>standard notation</i>		
• class diagrams, object interaction, packages, state and activity		
	<div>UML</div>	

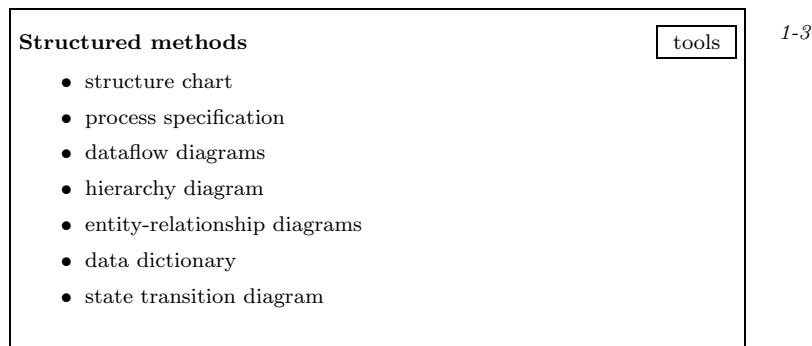
Slide 1-2: Software development methods

Some of these methods (and corresponding tools) directly stem from a more conventional (read *structured*) approach to software development. Others are more radical and propose new tools to support the decomposition principles underlying object-oriented technology. Naturally, those who wish to make a gradual shift from conventional technology to adopting an object-oriented approach may benefit from methods that adapt familiar techniques to the new concepts. In this section we will look at a variety of existing methods and the tools they offer. We do not discuss the tools and diagram techniques used in any detail. However, we will discuss the Fusion method in somewhat more detail. Fusion is a strongly systematic approach to object-oriented software development that integrates various concepts and modeling techniques from the other methods, notably OMT, Booch OOD, Objectory and CRC. We will discuss the process view underlying Fusion and sketch the models it supports in relation to the other methods. For the reader this section may supply an overview and references needed for a more detailed study of a particular method or tool. A recent development is the Unified Modeling Language (UML), which has been approved as a standard in 1998. UML brings together the models and notations featured by the various methods.

Jim Rumbaugh, Grady Booch and Ivar Jacobson, all leading experts in object-

oriented development, joined forces to achieve this. The importance of such a standardization can hardly be overemphasized. However, it must be noted that UML does not provide a method, in the sense of delineating the steps that must be taken in the development of a system. UML itself may be regarded as a toolbox, providing notations and modeling techniques that may be deployed when needed. A brief overview of UML is given in ???. An excellent introduction to UML, including advice how to apply it in actual projects may be found in [Fowler97].

Structured methods Initially, structured methods (which were developed at the beginning of the 1970s) were primarily concerned with modeling processes in a modular way. Based on software engineering principles such as *module coupling* and *cohesion*, tools were developed to represent the structure of a design (within what we have previously called the procedural or modular paradigm); see, for example, [Yourdon79]. Apart from diagrams to describe the modular architecture of a system (such as *structure charts* and *process specifications*), structured methods also employ *data flow diagrams* to depict processes and the flow of data between them, and *hierarchy diagrams* to model the structure of the data involved. See slide ??.



Slide 1-3: Tools for a structured approach

Later, structured methods were extended to encompass analysis, and the focus shifted to modeling the data by means of *entity-relationship diagrams* and *data dictionaries*. Also, *state transition diagrams* were employed to represent the behavioral aspects of a system.

As observed in [Fichman], in the late 1970s and early 1980s, planning and modeling of data began to take on a more central role in system development, culminating in data oriented methodologies, such as information engineering (which may be regarded as precursors to object-oriented methods). Information engineering, however, is primarily concerned with analysis and strategic planning. In addition to the modeling techniques mentioned, tools were developed to model the information needs of an enterprise and to perform risk analysis. Also, extensions to the *data dictionary* were proposed in order to have an integrated repository, serving all phases of the development. Currently, repository-based techniques are

again of interest since, in combination with modern hypermedia technology, they may serve as the organizational basis for reuse.

1.1.1 Perspectives of modeling

Understanding a problem domain may be quite demanding. Understanding is even more difficult when the description of the domain is cast in some representation pertaining to the solution domain. An object-oriented approach is said to require less translation from the problem domain to the (software) solution domain, thus making understanding easier. Many proponents of an object-oriented approach, however, seem to be overly optimistic in their conception of the modeling task. From an epistemological point of view, modeling may be regarded as being essentially colored by the mechanisms that are available to express the model. Hence, rather than opposing the functional and object-oriented approach by claiming that an object-oriented approach aims at *modeling reality*, I would prefer to characterize the distinction in terms of (modeling from) a different vernacular, a different perspective due to different modeling mechanisms. In other words, a model is meant to capture some salient aspects of a system or problem domain. Dependent on what features are considered as most important, different means will be chosen to construct a model.

Even within the confines of an object-oriented approach, there appear to be radically different perspectives of the modeling required in the various phases of the software life-cycle.

Modeling reality – *vernacular*

- requirements – *use cases*
- analysis – *domain concepts*
- design – *system architecture*
- implementation – *language support*

Design model – *system oriented*

- provides a justification of the architecture

1-4

Slide 1-4: Perspectives of modeling

An important contribution of [Jacobs92] is the notion of *use cases* that describe the situations in which a user actually interacts with the system. Such a (use case) model is an important constituent of the *requirements* document, since it precisely describes what the system is intended for. For the purpose of analysis, it may be helpful to develop a more encompassing (conceptual) model of the problem domain. The advantage of such an approach is that the actual system may later easily be extended due to the generality of the underlying analysis model. In contrast to the model used in analysis, both the design model and the implementation model are more *solution oriented* than *domain oriented*. The

implementation model is clearly dependent on the available language support. Within a traditional life-cycle, the *design* model may be seen as a transition from analysis to implementation. The notion of *objects* may act as a unifying factor, relating the concepts described in the analysis document to the components around which the design model is built. However, as we have noted, object-oriented development does not necessarily follow the course of a traditional software life-cycle. Alternatively, we may characterize the function of the design document as a *justification* of the choices made in deciding on the final architecture of the system. This remark holds insofar as an object-oriented approach is adopted for both design and implementation. However, see [Hend90] for a variety of combinations of structured, functional and object-oriented techniques.

Dimensions of modeling When restricting ourselves to *design models*, we may again distinguish between different modeling perspectives or, which is perhaps more adequate in this context, *dimensions of modeling*. In [Rum91], it is proposed to use three complementary models for describing the architecture and functionality of a system. See slide ??.

Dimensions of modeling – OMT

1-5

- object model – *decomposition into objects*
- dynamic model – *intra-object state changes*
- functional model – *object interaction* (data-flow)

Model of control

- procedure-driven, event-driven, concurrent

Slide 1-5: The OMT method

The OMT method distinguishes between an *object model*, for describing the (static) structure of object classes and their relations, a *dynamic model*, that describes for each object class the state changes resulting from performing operations, and a *functional model*, that describes the interaction between objects in terms of a *data-flow* graph.

An important contribution of [Rum91] is that it identifies a number of commonly used *control models*, including *procedure-driven* control, *event-driven* control and *concurrent* control. The choice for a particular control model may deeply affect the design of the system.

The OMT approach may be called a *hybrid method* since it employs non object-oriented techniques for describing intra-object dynamics, namely state-charts, and a functional approach involving data-flow diagrams, for describing inter-object communication.

Coherent models The OMT *object model*, however, only captures the static structure of the system. To model the dynamic and functional aspects, the object

model is augmented with a *dynamic model*, which is given by state diagrams, and a *functional model*, which is handled by data flow diagrams. From a formal point of view this solution is rather unsatisfactory since, as argued in [Hayes91], it is hard to establish the consistency of the combined model, consisting of an object, dynamic and functional model.

Model criteria – formal approach

- *unambiguous* – single meaning
- *abstract* – no unnecessary detail
- *consistent* – absence of conflict

1-6

Slide 1-6: Coherent models – criteria

Consistency checking, or at least the possibility to do so, is important to increase our belief in the reliability (and reusability) of a model. To be able to determine whether a model is consistent, the model should be phrased in an unambiguous way, that is, in a notation with a clear and precise meaning. See slide ???. Also, to make the task of consistency checking manageable, a model should be as abstract as possible, by leaving out all irrelevant details. To establish the consistency of the combined model, covering structural, functional and dynamic aspects, the interaction between the various models must be clearly defined.

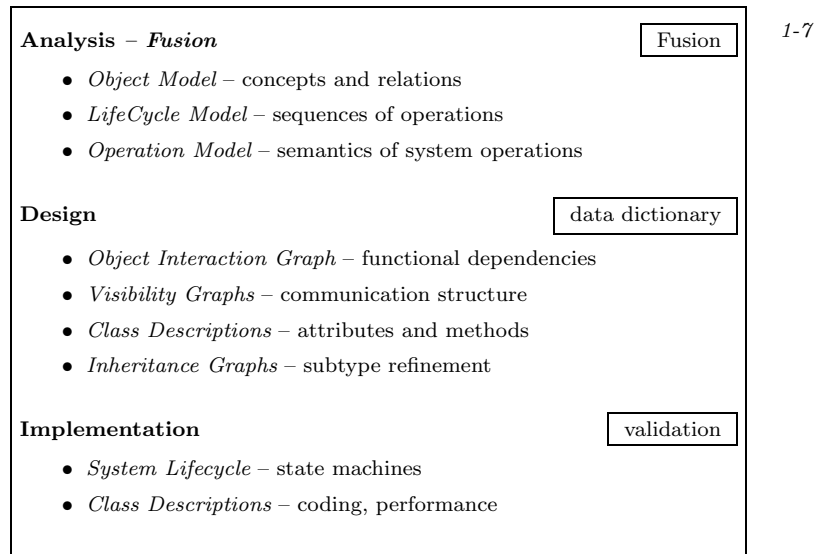
1.1.2 Requirements engineering – *Fusion*

The Fusion method is presented in [Fusion] as a second generation object-oriented method. The phrase *second generation* is meant to indicate that the method transcends and incorporates the ideas and techniques employed in the early object-oriented methods.

Above all, the Fusion method focuses on a strongly systematic approach to object-oriented software development, with an emphasis on the *process* of development and the validation of the consistency between the models delivered in the various phases of a project.

The software life-cycle model underlying Fusion is the traditional waterfall model, consisting of the subsequent phases of analysis, design and implementation. Each phase results in a number of models describing particular aspects of the system. See slide ???. A *data dictionary* is to be kept as a means to unify the terminology employed in the various phases.

The models produced as the result of analysis, design and implementation serve to document the decisions made during the development. Each of the phases covers different aspects of the system. Analysis serves to document the system requirements from a user perspective. The Fusion method describes how to construct an *Object Model* that captures the basic concepts of the application domain. These concepts are represented as entities or objects and are connected



Slide 1-7: The Fusion method

by relations, similar to *entity-relationship* diagrams employed in semantic modeling. Analysis also results in an *Operation Model*, describing the semantics of the operations that may be performed by a user by means of pre- and post-conditions, in a formal manner. In addition, Fusion defines a *Lifecycle Model* that describes, by means of regular expressions, which sequences of operations are allowed.

Design may be considered as the transition between analysis and implementation. During design, decisions are made with respect to the realization of the system operations identified during analysis. Design according to the Fusion method results in an *Object Interaction Graph*, that for each system operation describes which objects are involved and which methods are invoked. Fusion also allows one to label the arrows representing method calls in the interaction diagram with sequencing information. In addition, design involves the construction of *Visibility Graphs*, indicating the attribute and method interface for each object, *Class Descriptions*, defining the attributes and methods of objects, and *Inheritance Graphs*, specifying the subtype refinement relation between classes.

Implementation is considered in the Fusion method as a phase in which to work out the details of the decisions taken during analysis and design. It results in a *System Lifecycle* description for each object identified in the *Object Model*, in the form of a finite state machine, and precise *Class Descriptions*, in the form of (preferably) efficient code.

Validation An important aspect of the Fusion method is the validation of the completeness and consistency of the collection of models. Completeness, obviously, is a relative matter and can only be established with respect to explicitly

stated user requirements. However, the models developed in a particular phase impose additional requirements upon the efforts engaged in the later phases and in the end maintenance. Consistency involves verifying whether the various models are not contradictory. For both development and validation, the data dictionary plays an important role, as a common point of reference.

1.1.3 Methods for analysis and design – a comparative study

In [Fichman] a comparative review of a selected number of object-oriented analysis and design methods is given. Criteria for selection were the availability of documentation and acceptance in the object-oriented community, measured in terms of refereed articles.

Paraphrasing [Fichman] again: *As with traditional analysis, the primary goal of object-oriented analysis is the development of an accurate and complete description of the problem domain.*

The three analysis models described in [Fichman] share a number of diagram techniques with both structured methods and methods for object-oriented design. However, the method proposed in [Shlaer88] in particular reflects the domain-oriented focus of analysis.

A similar focus on domain requirements and analysis may be found in the Objectory method. See slide ???. Objectory is one of the methods that has inspired Fusion, in particular because it presents a systematic approach to the process of software development. The Objectory method centers around *use case* analysis. Use case analysis involves a precise description of the interaction between the user of a system and the components representing domain-specific functionality. The Objectory method gives precise guidelines on how to proceed from the identification of *use cases*, which include user interface aspects, to their realization in the subsequent phases of design and implementation. Objects are called blocks in Objectory. Use case analysis corresponds in a loose way with the identification of system operations in Fusion.

There is a close correspondence between the OMT object model and the analysis object model of Fusion. Both OMT and Fusion employ extended entity-relationship diagrams. Also, the dynamic model of OMT reoccurs in the Fusion method, albeit in a later phase. The functional model of OMT, which has the form of a dataflow diagram, is generally considered to be inappropriate for object-oriented analysis. Instead, Fusion employs a model in which the semantics of system operations are captured by means of formal pre- and post-conditions. In [Fusion], OMT is characterized as a very loose method, giving few rules for discovering inconsistencies between the various models and lacking a clear view with respect to the process of development. OMT is strongly focused on analysis, giving nothing but heuristics to implement the models that result from analysis. However, what is called the *light-weight Fusion method* almost coincides with OMT.

A lack of detailed guidelines for the process of software development is also characteristic of the Booch OOD method. Booch offers a wealth of descriptive

Objectory – *systematic process*

1-8

- requirements – *use cases, domain object model, user interface*
- analysis – *subsystems*
- design, implementation – *block model, interaction diagrams*

OMT – *few rules for inconsistencies*

- analysis – *object model, dynamic model, functional model*
- design, implementation – *heuristics to implement analysis models*

Booch – *descriptive*

- diagrams – *class, object, timing, state, module, process*

CRC – *exploratory*

- analysis, design – *class, responsibilities, collaborators*

Formal methods

- operations – *pre- and post-conditions*

Slide 1-8: Comparison of methods (1)

diagrams, giving detailed information on the various aspects of a system, but offers merely heuristics for the actual process of development.

The CRC method must be regarded primarily as a means to explore the interaction between the various objects of a domain. It is powerful in generating ideas, but offers poor support for documenting the decisions with respect to the objects and how they interact.

Formal methods have been another important source of inspiration for the Fusion method. The description of system operations during analysis employs a characterization of the functionality of operations that is directly related to the specification of operations in model-based specification methods such as VDM and Z. See section ??.

The Fusion method may be regarded as being composed of elements of the methods mentioned above. It shares its object model with OMT, its approach to the characterization of system operations with formal methods, its focus on object interaction with CRC and its explicit description of classes and their relations with Booch. See slide ??.

In comparison with these methods, however, it provides a much more systematic approach to the process of development and, moreover, is explicitly concerned with issues of validation and consistency between models. In addition, [Fusion] claim to provide explicit semantics for their various models, whereas the other methods fail to do so. However, it must be remarked that the Fusion method remains somewhat obscure about the nature of system operations. System operations are characterized as *asynchronous*. Yet, if they are to be taken as methods,

Comparison - as a systematic approach						1-9
	Objectory	OMT	Booch	CRC	Fusion	
development	+	+-	-	x	+	
maintenance	+	+-	+	-	+	
structure	+-	+-	+	+	+	
management	+	+-	+-	-	+	
tool support	+-	+-	+-	-	+	

Slide 1-9: Comparison of methods (2)

such operations may return a result, which is quite hard to reconcile with their asynchronous nature. The claim that the models have a precise semantics, which is essential for tool support, must be substantiated by providing an explicit semantics in a formal manner!

With regard to the process of development, both Objectory and Fusion provide precise guidelines. The CRC method may be valuable as an additional exploratory device. For maintenance, the extent to which a method enforces the documentation of design decisions is of utmost importance. Both the Objectory and Booch method satisfy this criterion, as does the Fusion method. OMT is lacking in this respect, and CRC is clearly inadequate.

Whether a method leads to a *good* object-oriented design of the system architecture, depends to a large extent upon the ability and experience of the development team. Apart from Fusion, both the Booch method and CRC may be characterized as purely object-oriented, whereas Objectory and OMT are considered to be impure.

A strongly systematic approach to the process of development is important in particular from the point of view of project management. Project management support entails a precise definition of the deliverables associated with each phase, as well as an indication of the timing of their deliverance and validation. Both the OMT method and Booch are lacking in this respect, since they primarily provide techniques to develop descriptive models. Clearly, CRC lacks any support for project management.

Tool support is dependent on the existence of a well-defined semantics for the models employed. For both Objectory and OMT commercial tools are available, despite their loosely specified semantics. The Fusion diagramming techniques are also supported.

For CRC, tool support is considered to be useless. The success of the method depends upon flexibility, the ease with which new ideas can be tried, a flexibility which even hypertext cannot offer, according to its authors.

1.2 Identifying objects

Object-oriented design aims at describing a system in terms of objects (as the primary components) and the interaction between them. Motivated by the wish to arrive at stable abstractions, object-oriented design is often characterized as *modeling reality*, that is the application domain. However, many applications require, at least partly, a *system-oriented* view towards design, since they involve system artifacts for which there exist no clearly identifiable counterparts in the application domain. As an example, think of a window-based system. Many of the items (widgets) introduced in such a system belong to an artificial reality, which at best is only vaguely analogous with reality as we normally understand it.

Irrespective of whether the design is intended as a preliminary study before the implementation or as a *post hoc* justification of the actual system, the most important and difficult part of design is the *identification of objects* and the characterization of their role in the system and interaction with other objects.

As observed in [McGregor92], object-oriented design is best seen as *class oriented*, that is directed towards the *static* description of (classes of) objects, rather than a description of the dynamic interaction between actual objects. In section ??, we will discuss *class-less* languages which are well suited for exploratory programming. However, from the perspective of design, we are more interested in a (static) abstract specification of the components that constitute the system.

<p>Object-oriented design – <i>decomposition into objects</i></p> <ul style="list-style-type: none"> • application/system/class oriented <p>Identifying objects – <i>responsibilities</i></p> <ul style="list-style-type: none"> • data/procedure oriented <p>Layers of abstraction</p> <ul style="list-style-type: none"> • components, subsystems, frameworks 	1-10
---	------

Slide 1-10: Object-oriented design

In comparison with a functional approach, object-oriented design is clearly *data oriented*. However, although a data-oriented approach may provide a first guideline in developing the system, the primary concern in object-oriented design should be the *responsibilities* of an object rather than how it acts as a *data manager*, so to speak.

For larger systems, the complexity of the design may necessitate the introduction of additional layers of abstraction. Apart from objects, which must be regarded as the basic components of a system, we may need to isolate subsystems, consisting of a number of related object classes. When we have developed a

subsystem that can be used in a variety of contexts, such a subsystem may be used as a *framework*. A framework is generally not only a collection of classes but must also be seen as an approach or method in its own way, since it usually imposes additional constraints on the development. For example, most development environments for window-based applications provide a framework consisting of a number of predefined classes and functions, and guidelines or recipes that prescribe how to use or adapt these classes and functions. Also, most frameworks impose a specific control model, such as the *event-driven* control model imposed by window programming environments.

1.2.1 Modeling heuristics

Following [Booch86], we may characterize objects as ‘crisp’ *entities* that *suffer* and *require* actions. From the perspective of system development, *objects* must primarily be regarded as *computational entities*, embodying the means by which we may express a computation. Modeling a particular problem domain, then, means defining abstractions in terms of *objects*, capturing the functional characteristics of that domain. The question is, how do we arrive at such a model?

Objects – *crisp entities*

- *object* = an entity that *suffers* and *requires* actions

The method:

- [1] Identify the objects and their attributes
- [2] Identify operations suffered and required
- [3, 4] Establish visibility/interface

1-11

Slide 1-11: The Booch method

In [Booch86], a straightforward method of object oriented development is proposed, which consists of the successive identification of objects and their attributes, followed by a precise characterization of the interobject visibility relations. In [Booch91], a shift of emphasis has occurred towards determining the semantics of an individual object and the interaction between collections of objects.

As a heuristic to arrive at the proper abstractions of the problem domain (in terms of object classes), [Booch86] proposes scanning the requirements document for *nouns*, *verbs* and *adjectives*, and using these as initial suggestions for respectively *objects*, and *operations* and *attributes* belonging to objects (see slide ??). This technique has been adopted and augmented by a number of other authors, among which [WWW90] and [Rum91]. For example, [WWW90] illustrate the technique in fine detail in several examples, including the design of an automated teller machine and a document processing system.

In addition to the interpretation of nouns as possible objects, verbs as possible operations on objects, and adjectives as possible attributes of objects, [Rum91]

Heuristics <ul style="list-style-type: none">• <i>model of reality</i> – balance <i>nouns</i> (objects) and <i>verbs</i> (operations) Associations <ul style="list-style-type: none">• <i>directed action</i> – drives, instructs• <i>communication</i> – talks-to, tells, instructs• <i>ownership</i> – has, part-of• <i>resemblance</i> – like, is-a• <i>others</i> – works-for, married-to	1-12
---	------

Slide 1-12: Heuristics for modeling

suggest this technique to determine other relations and associations between object classes as well. For instance, a model of control and object interaction may be suggested by phrases indicating directed action or communication. Similarly, structural issues, such as whether an object owns another object or whether inheritance should be used, may be decided on the basis of resemblance or subordination relations.

Example – ATM (1) The example of an automated teller machine discussed in [WWW90] nicely illustrates a number of the notions that we have thus far looked at only in a very abstract way. A *teller* machine is a device, presumably familiar to everyone, that allows you to get money from your account at any time of the day. Obviously, there are a number of constraints that such a machine must satisfy. For instance, other people should not be allowed to withdraw money from your account. Another reasonable constraint is that a user cannot overdraw more than a designated amount of money. Moreover, each transaction must be correctly reflected by the state of the user's account.

An initial decomposition into objects based on these requirements is shown in slide ?? In [WWW90], a fully detailed account is given of how one may arrive at such a decomposition by carefully reading (and re-reading) the requirements document. What we are interested in here, however, is how we may establish that we have not overlooked anything when proposing a design, and how we may verify that our design correctly reflects the requirements.

This particular example nicely illustrates the need for an analysis of the *use cases*. To develop a proper interface, we must precisely know what a user is expected to do (for instance, insert a bank card, key in a PIN code) and how the system must respond (what messages must be displayed, how to react to a wrong PIN code, etc.). Another decision that must be made is when the account will be changed as the result of a transaction. Also, we must decide what to do when a user overdraws.

A very important issue that we will look at in more detail in the next sections

Candidate classes

- *account* – represents the customer’s account in the banks database
- *atm* – performs financial services for a customer
- *cardreader* – reads and validates a customer’s bankcard
- *cashdispenser* – gives cash to the customer
- *screen* – presents text and visual information
- *keypad* – the keys a customer can press
- *pin* – the authorization code
- *transaction* – performs financial services and updates the database

ATM

1-13

Slide 1-13: The ATM example (1)

is how the collection of objects suggested above will interact. What means do we have to describe the cooperation between the objects, and how do we show that the proposed system meets all the requirements listed above? Moreover, can we verify that the system satisfies all the constraints mentioned in the requirements document?

Validation However, before examining these questions and trying out different scenarios, we may as well try to eliminate the spurious classes that came up in our initial attempt. In [Rum91], a number of reasons are summarized that may be grounds on which to reject a candidate class. See slide ??.

Eliminating spurious classes

- *vague* – system, security-provision, record-keeping
- *attribute* – account-data, receipt, cash
- *redundant* – user
- *irrelevant* – cost
- *implementation* – transaction-log, access, communication

Good classes

- our candidate classes

1-14

Slide 1-14: Eliminating spurious classes

For example, the notion underlying the candidate class may be too *vague* to be represented by a class, such as the notion of *system* or *record-keeping*. Another reason for rejecting a suggested class may be that the notion represents not so much a class, but rather a possible attribute of a class. Further, a proposed class may either be *redundant*, for example the class *user*, or simply *irrelevant*, as is the

class *cost*. And finally, a class may be too *implementation* oriented, such as the class *transaction-log* or classes that represent the actual communication or access to the account.

Looking back, our choice of candidate classes seems to have been quite fortunate, but generally this will not be the case, and we may use the checklist above to prune the list of candidate classes. An interesting architectural issue is, how may we provide for future extensions of the system? How easily can we reuse the design and the code for a system supporting different kinds of accounts, or different input or output devices? And how can we establish that the objects, as identified, interact as desired?

1.2.2 Assigning responsibilities

Design is to a large extent a matter of creative thinking. Heuristics such as performing a linguistic scan on the requirements document for finding objects (nouns), methods (verbs) and attributes (adjectives) may be helpful, but will hopelessly fail when not applied with good taste and judgement. Not surprisingly, one of the classical techniques of creative writing, namely the *shoe-box method*, has reappeared in the guise of an object-oriented development method. The shoe-box method consists of writing fragments and ideas on note cards and storing them in a (shoe) box, so that they may later be retrieved and manipulated to find a suitable ordering for the presentation of the material. To find a proper decomposition into objects, the method creates for each potential (object) class a so-called CRC card, which lists the *Class* name, the *Responsibilities* and the possible *Collaborators* of the proposed class. In a number of iterations, a collection of cards will result that more or less reflects the structure of the intended system.

According to the authors (see Beck and Cunningham, 1989), the method effectively supports the early stages of design, especially when working in small groups. An intrinsic part of the method consists of what the authors call *dynamic simulation*. To test whether a given collection of cards adequately characterizes the functionality of the intended system, the cards may be used to simulate the behavior of the system. When working in a group, the cards may be distributed among the members of the group, who participate in the simulation game according to their cards. See slide ??.

A number of authors have adopted this method, or developed a very similar method, for identifying objects and characterizing their functionality in an abstract way. It is doubtful, however, whether the method has any significance beyond the early stages of analysis and design. Without any more formal means to verify whether the responsibilities listed adequately characterize the intended functionality of the system, the method amounts to not much more than brainstorming. Clearly, the method needs to be complemented by more formal means to establish whether the (implicit) protocols of interaction between the objects satisfy the behavioral requirements of the system.

Nevertheless, the elegant simplicity of the method is appealing, and the card format lends itself to easy incorporation in an on-line documentation system. Moreover, since the method imposes no strict order, and has relatively little