A Laboratory For Teaching Object-Oriented Thinking

Kent Beck, Apple Computer, Inc.
Ward Cunningham, Wyatt Software Services, Inc.

It is difficult to introduce both novice and experienced procedural programmers to the anthropomorphic perspective necessary for object-oriented design. We introduce CRC cards, which characterize objects by class name, responsibilities, and collaborators, as a way of giving learners a direct experience of objects. We have found this approach successful in teaching novice programmers the concepts of objects, and in introducing experienced programmers to complicated existing designs.

1. Problem

The most difficult problem in teaching object-oriented programming is getting the learner to give up the global knowledge of control that is possible with procedural programs, and rely on the local knowledge of objects to accomplish their tasks. Novice designs are littered with regressions to global thinking: gratuitous global variables, unnecessary pointers, and inappropriate reliance on the implementation of other objects.

Because learning about objects requires such a shift in overall approach, teaching objects reduces to teaching the design of objects. We focus on design whether we are teaching basic concepts to novices or the subtleties of a complicated design to experienced object programmers.

Rather than try to make object design as much like procedural design as possible, we have found that the most effective way of teaching the idiomatic way of thinking with objects is to immerse the learner in the "object-ness" of the material. To do this we must remove as much familiar material as possible, expecting that details such as syntax and programming environment operation will be picked up quickly enough once the fundamentals have been thoroughly understood.

It is in this context that we will describe our perspective on object design, its concrete manifestation, CRC (for Class, Responsibility, and Collaboration) cards, and our experience using these cards to teach both the fundamentals and subtleties of thinking with objects.

2. Perspective

Procedural designs can be characterized at an abstract level as having processes, data flows, and data stores[1], regardless of implementation language or operating environment. We wished to come up with a similar set of fundamental

principles for object designs. We settled on three dimensions which identify the role of an object in a design: class name, responsibilities, and collaborators.

The class name of an object creates a vocabulary for discussing a design. Indeed, many people have remarked that object design has more in common with language design than with procedural program design. We urge learners (and spend considerable time ourselves while designing) to find just the right set of words to describe our objects, a set that is internally consistent and evocative in the context of the larger design environment.

Responsibilities identify problems to be solved. The solutions will exist in many versions and refinements. A responsibility serves as a handle for discussing potential solutions. The responsibilities of an object are expressed by a handful of short verb phrases, each containing an active verb. The more that can be expressed by these phrases, the more powerful and concise the design. Again, searching for just the right words is a valuable use of time while designing.

One of the distinguishing features of object design is that no object is an island. All objects stand in relationship to others, on whom they rely for services and control. The last dimension we use in characterizing object designs is the collaborators of an object. We name as collaborators objects which will send or be sent messages in the course of satisfying responsibilities. Collaboration is not necessarily a symmetric relation. For example in Smalltalk-80[2], View and Controller operate as near equals (see example below) while OrderedCollection offers a service with little regard or even awareness of its client.

Throughout this paper we deliberately blur the distinction between classes and instances. This informality is not as confusing as it might seem because the concreteness of our method substitutes for naming of instances. This also makes our method for teaching independent of whether a class or prototype-based language is used.

3. CRC Cards

The second author invented CRC cards in response to a need to document collaborative design decisions. The cards started as a Hypercard[3] stack which provided automatic indexing to collaborators, but were moved to their current form to address problems of portability and system independence.

Like our earlier work in documenting the collaboration of objects[4], CRC cards explicitly represent multiple objects simultaneously. However, rather than simply tracing the details of a collaboration in the form of message sending, CRC cards place the designer's focus on the motivation for collaboration by representing (potentially) many messages as a phrase of English text.

As we currently use them, all the information for an object is written on a 4" x 6" index card. These have the advantages that they are cheap, portable, readily available, and familiar. Figure 1 shows an idealized card. The class name appears underlined in the upper-left hand corner, a bullet-

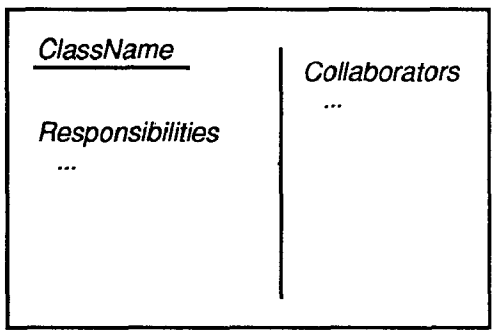


Figure 1. A Class-Responsibility-Collaborator (CRC) index card.

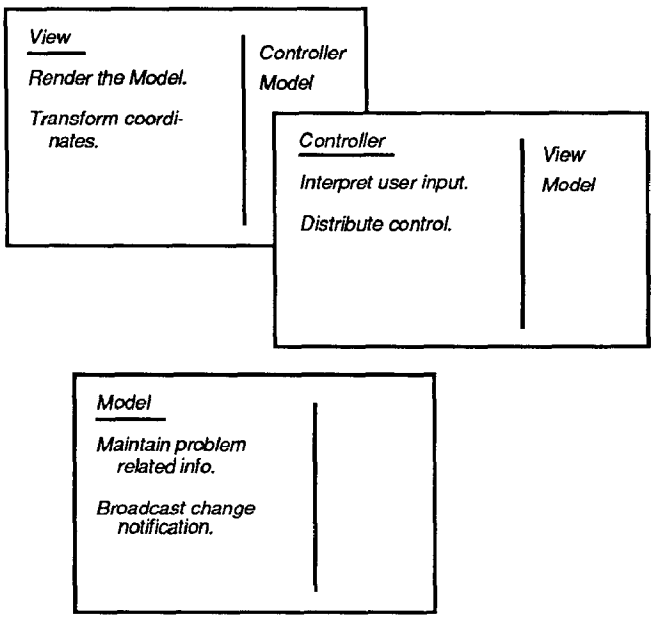


Figure 2. CRC-cards describing the responsibilities and collaborations of Smalltalk's Model, View and Controller.

list of responsibilities appears under it in the left two-thirds of the card, and the list of collaborators appears in the right third.

Figure 2 shows an example taken from the Smalltalk-80 image, the much-misunderstood model-view-controller user interface framework. We have deliberately shown only a portion of the responsibilities each of these objects assumes for clarity of exposition. Note that the cards are placed such that View and Controller are overlapping (implying close collaboration) and placed above Model (implying supervision.) We find these and other informal groupings aid in comprehending a design. Parts, for example, are often arranged below the whole. Likewise, refinements of an abstraction can be collected and handled as a single pile of cards with the most abstract card on top where it can represent the rest.

The ability to quickly organize and spatially address index cards proves most valuable when a design is incomplete or poorly understood. We have watched designers repeatedly refer to a card they intended to write by pointing to where they will put it when completed.

Design with the cards tends to progress from knowns to unknowns, as opposed to top-down or bottom up. We have observed two teams arriving at essentially the same design through nearly opposite sequences, one starting with device drivers, the other with high-level models. The problem demanded a certain set of capabilities which both teams discovered in the course of fulfilling the requirements of the design.

We suggest driving a design toward completion with the aid of execution scenarios. We start with only one or two obvious cards and start playing "what-if". If the situation calls for a responsibility not already covered by one of the objects we either add the responsibility to one of the objects, or create a new object to address that responsibility. If one of the object becomes too cluttered during this process we copy the information on its card to a new card, searching for more concise and powerful ways of saying what the object does. If it is not possible to shrink the information further, but the object is still too complex, we create a new object to assume some of the responsibilities.

We encourage learners to pick up the card whose role they are assuming while "executing" a scenario. It is not unusual to see a designer with a card in each hand, waving them about, making a strong identification with the objects while describing their collaboration.

We stress the importance of creating objects not to meet mythical future needs, but only under the demands of the moment. This ensures that a design contains only as much information as the designer has directly experienced, and avoids premature complexity. Working in teams helps here because a concerned designer can influence team members by suggesting scenarios aimed specifically at suspected weaknesses or omissions.

4. Experience

One of the contexts in which we have used CRC cards is a three-hour class entitled "Thinking with Objects," which is intended for computing professionals who have programmed, but whose jobs do not necessarily involve programming every day. The class proceeds by introducing a data flow example (a school, with processes for teaching and administration) which is then recast in terms of objects with responsibilities and collaborators (such as Teacher, Janitor, and Principal). The class then pairs off and spends an hour designing the objects in an automatic banking machine, an exercise chosen because of everyone's familiarity with the application and its ready breakdown into objects to control the devices, communicate with the central bank database, and control the user interface. (See the appendix for a sample solution.) The exercise is followed by a definition of the terms "class", "instance", "method", and "message", and the class concludes with a brief discussion of the history and features of a variety of object-oriented programming languages.

In teaching over a hundred students this course we have encountered no one who was unable to complete the exercise unaided, although one pair in each class usually needs a few hints to get started. Although we have done no follow-up studies, the class is considered a valuable resource in the company and is still well attended with a long waiting list almost a year after its inception.

We have also asked skilled object programmers to try using CRC cards. Our personal experience suggests a role for cards in software engineering though we cannot yet claim a complete methodology (others[5],[6] have more fully developed methodologies that can take advantage of CRC methods). We know of one case where finished cards were delivered to a client as (partial) design documentation. Although the team that produced the cards was quite happy with the design, the recipient was unable to make sense of the cards out of context.

Another experiment illustrates the importance of the context established by the handling and discussing of cards. We had videotaped experienced designers working out a problem similar to the bank machine. Our camera placement made cards and the designers' hands visible but not the writing on the cards. Viewers of the tape had no trouble following the development and often asked that the tape be stopped so that they could express their opinions. The most telling moments came when a viewer's explanation required that he point to a blurry card in the frozen image on the screen.

Finally, we have used CRC cards to advantage in explaining complex designs. A few minutes of introduction is sufficient to prepare an audience for a card based presentation. Cards can be made out in advance or written on the spot. The latter allows the complexity in a design to be revealed slowly, a process related to Dave Thomas' "lie management". The cards are being used as props to aid the telling of a story of computation. The cards allow its telling without recourse to programming language syntax or idiom.

5. Conclusion

Taking our perspective as a base we give novices and experienced programmers a learning experience which teaches them something valuable about objects. CRC cards give the learner who has never encountered objects a physical understanding of object-ness, and prepares them to understand the vocabulary and details of particular languages. CRC cards also give useful and convincing experience with objects to those who has learned the mechanisms of objects but do not yet see their value.

Ragu Raghavan[7] has said that in the switch to objects strong programmers become stronger, but weaker programmers are left behind. Using the cards in group settings we found that even weaker programmers, without a deep understanding of objects, could contribute to object designs. We speculate that because the designs are so much more concrete, and the logical relationship between objects explicit, it is easier to understand, evaluate, and modify a design.

We were surprised at the value of physically moving the cards around. When learners pick up an object they seem to more readily identify with it, and are prepared to deal with the remainder of the design from its perspective. It is the value of this physical interaction that has led us to resist a computerization of the cards.

It is just this problem–integrating the cards with larger design methodologies and with particular language environments, that we feel holds the most promise for the future. The need to retain the value of physical interaction points to the need for a new kind of user interface and programming environment as far beyond what we have today as our current systems are beyond the tool-oriented environments of the past.

References

[1] DeMarco, T.: Structured Analysis and System Specification, Yourdon, 1978.

[2] Smalltalk-80 image, Xerox Corp, 1983.

[3] Hypercard manual, Apple Computer, Inc.

[4] Cunningham, W. and Beck, K.: "A Diagram for Object-Oriented Programs," in Proceedings of OOPSLA-86, October 1986.

[5] Wirfs-Brock, R. and Wilkerson, B. "Object-Oriented Design: a Responsibility-Driven Approach," submitted to OOPSLA '89.

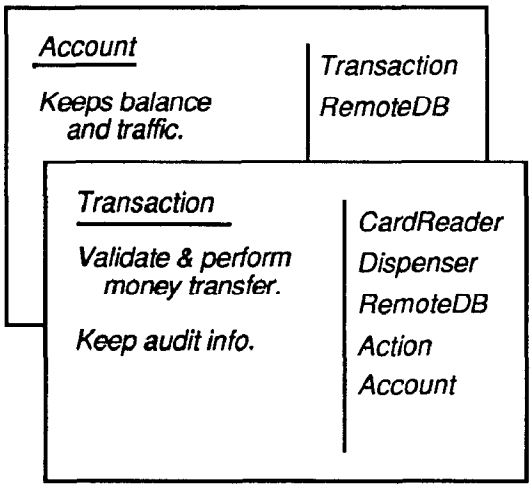
[6] Reenskaug, T.: "A Methodology for the Design and Description of Complex, Object-Oriented Systems," technical report, Center for Industrial Research, Oslo, Norway, November 1988.

[7] Raghavan, R.: "Panel: Experiences with Reusability," in the Proceedings of OOPSLA '88, October, 1988.

Appendix

Here we provide a sample solution to the banking machine problem discussed in section 4.

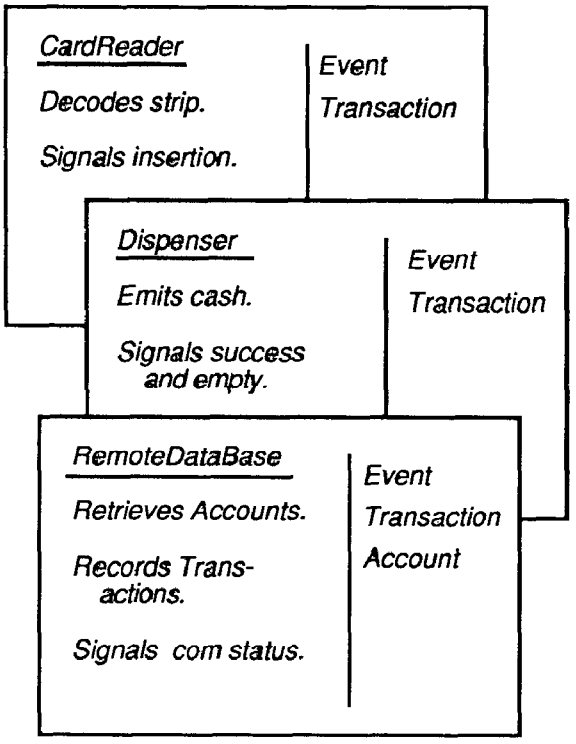
Account and Transaction provide the banking model. Note that Transaction assumes an active role while money is being dispensed and a passive role thereafter.



Transactions meet their responsibilities with the aid of several objects that serve as device drivers. The Dispenser object, for example, ultimately operates the dispensing device.

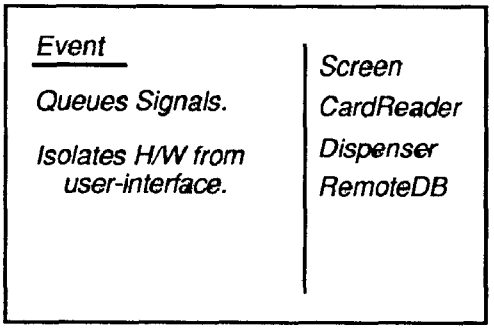
The CardReader object reads and decodes the information on the bank card's magnetic strip. A common mistake would be to itemize all of the

information stored on the bank card. Card encoding formats must certainly be well thought out and documented. However, for the purpose of designing the objects, we need only identify where that knowledge will be placed in the program.

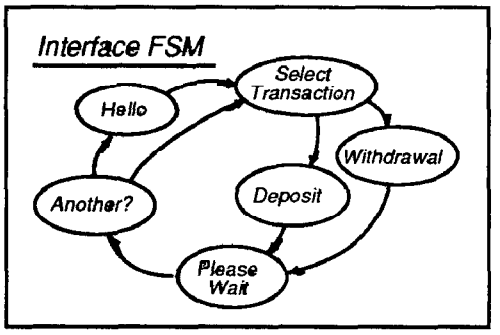


The RemoteDataBase drives the communication lines and interprets data transmitted across them. It creates Account objects and consumes Transaction objects.

The device drivers signal exceptional or asynchronous events by adding Event objects to a shared queue.



Events drive the human interface by triggering Actions that sequence through Screens. The actual format and sequence of screens will be determined by the user-interface design and will probably vary from bank to bank. We offer objects from which state-machine like interfaces can be built.



Screen objects correspond to the states and Action objects correspond to the transitions. Screens may vary in how they dispatch Actions. Actions themselves will vary in how they process events. Actions ultimately construct Transactions to which they delegate the further operating of the bank machine.