Part 3: Implementation

17

Implementation Modeling

17.1 An arrow indicates that the association is implemented in the given direction. Because of the large amounts of data for an ATM we used two-way pointers for associations that are traversed in both directions—we avoided one-way pointers combined with backward searching. Note that most of the associations in the domain model may be traversed either way, but associations for the application model are traversed only one way.

- **ATM <–> RemoteTransaction**. This association would be implicit for the ATM. However, it would be needed for the consortium and bank computers. Given a transaction we must be able to find the ATM. Given an ATM we might want to find all the transactions for a summary report.
- **RemoteTransaction <–> Update**. Obviously we must be able to find the updates for a transaction. However, we also must be able to go the other way. We might find the updates for an account and then need to find the transactions, such as to get the date and time.
- **RemoteTransaction <–> CardAuthorization**. Given a transaction, we might want to find how it was authorized. Given an authorization we might want to see all of its activity (such as for a fraud investigation).
- **RemoteTransaction <– RemoteReceipt**. Given a receipt, we should be able to find the transactions. (This might happen if a customer brings in a receipt and has questions.) We presume that the receipt is unimportant to the bank, and is just an artifact for the customer's benefit, so we do need to traverse from transaction to receipt.
- **RemoteReceipt –> CashCard**. The receipt should indicate which cash card was used. We presume there is no need to traverse from a cash card to a receipt. (Note, however, that we can traverse from cash card to authorization to transaction.
- **CashCard <–> CardAuthorization**. A cash card must know its authorization. An authorization must also know about its cash cards.

- **Bank <–> CardAuthorization**. An authorization must be able to reference its bank and a bank must be able to find all authorizations (to facilitate issuing of card codes).
- **Consortium <–> ATM**. The consortium must track all of its ATMs and the ATM must know its consortium.
- **Consortium <–> Bank**. Same logic as the prior bullet.
- **Bank <–> Account**. A bank must know all its accounts. An ATM must be able to find the bank for a card authorization's account.
- **Customer <–> Account**. An account must be able to retrieve customer data. A customer must be able to look up all of his or her accounts.
- **Customer <–> CardAuthorization**. A customer must be able to find their authorizations and an authorization must be able to find customer data.
- **CardAuthorization <–> Account**. A card authorization may cover only some of a customer's accounts. So a card authorization must be able to find those accounts. An account may need to find the card authorizations to which it is subject.
- **Account <–> Update**. An account must be able to tally its updates. An update must know which account is affected.
- **CashCardBoundary <– AccountBoundary**. This association is purely an artifact for the convenience of importing and exporting data. Consequently, there is no need to implement the association in both directions. We will choose to have pointers that retrieve cash card data for an account. (The decision on which way to implement is arbitrary and we could do it the other way.
- **TransactionController –> RemoteTransaction**. A transaction controller must know about its transactions. There is no need to go the other way.
- **ATMsession –> CashCard**. An ATM session must know which cash card is involved. There is no need to go the other way, since the ATM session is transient and there is at most one active per ATM machine.
- **ATMsession –> Account**. Same logic as previous bullet.
- **SessionController –> ATMsession**. A session controller has at most one ATM session active at a time, but has many ATM sessions over time. The session controller must be able to find the current, active ATM session. There is no need to go the other way.
- **SessionController –> ControllerProblem**. The session controller must be able to find its problems. There is no need to go the other way.
- **ControllerProblem –> ProblemType**. We must be able to find the type for a controller problem, but do not need to go the other way.

17.2 An arrow indicates that the association is implemented in the given direction.

- **Text <–> Box**. The user can edit text and the box must resize, so there should be a pointer from text to box. Text is only allowed in boxes, so we presume that a user

may grab a box and move it, causing the enclosed text to also move. So there should be a pointer from box to text.

- **Connection <–> Box**. A box can be dragged and move its connections, so there must be pointers from box to connections. Similarly, a link can be dragged and move its connections to boxes, so there must also be a pointer from connection to box. There is no obvious ordering.
- **Connection <–> Link**. Same explanation as above bullet.
- **Collection –> ordered Box**. Given a collection we must be able to find the boxes. There does not seem to be a need to traverse the other way. There likely is an ordering of boxes, regarding their foreground / background hierarchy for visibility.
- **Collection –> ordered Link**. Same explanation as above bullet.

17.3 The answer depends on whether the model is used only to generate output (going from a document to the individual lines) or whether the model is also used to handle input (the user picks a line at random and operates on it). For the first case, the associations are likely to be traversed in only one direction, from page to column and from column to line. In the latter case, it would be best to implement each association in both directions.

The lines within a column should be ordered. We need to know more about the problem to determine if the columns on a page should be ordered. If text is to flow from page to page when changes are made, columns should be ordered. Otherwise there is little advantage to having ordered columns.

17.4 This association would likely be traversed in both directions. Operations on a collection of cards, such as *insert*, require access to the ordered set of cards. On the other hand, the user may be able to select a card from the screen, in which case the collection containing the card must be found.

17.5 The simplest approach is to implement every association in both directions. In the rest of this answer we indicate when a one-way implementation of an association might be feasible. Note that the choice of one-way or two-way traversal depends on the details of an application and that some of our decisions may be arguable. An arrow indicates that the association is implemented in the given direction. Association ends are unordered, except where specified.

- **Season –> ordered Meet**. Probably want to order meets by date. Probably do not need to go from meet to season.
- **Meet –> Station**. There is no reason to order stations nor any basis to do so. An implementation might introduce station numbers but they have no intrinsic meaning. Probably do not need to go from station to meet.
- **Station <–> Scorekeeper**. Need to go both ways. Need to know which scorekeepers are assigned to a station. Each scorekeeper needs to know their assigned stations.
- **Station <–> Judge**. Same comments as previous bullet.

- **Station <–> ordered Event**. Probably want to order events by time because each station can support only one event at a time. We will probably want to query an event to see what station it is on.
- **Meet <–> Event**. Because a meet (unlike a station) may have simultaneous events, ordering is not so useful. We may need the back pointer to answer questions such as "Which meets did a competitor participate in during a season?"
- **Figure <–> ordered Event**. We need to go both ways; an event needs to know its figure, and we may want to know the meets at which a figure is held. Events could be ordered by meet date.
- **Event <–> ordered Trial**. We will need to go both ways. The trials are held in a specific sequence that is important for scheduling.
- **Trial –> (Judge, rawScore)**. No need to order the scores. No reason to go backwards from judge to (Trial, rawScore), unless possibly we need to audit each judge's performance.
- **Competitor <–> ordered Trial**. Obviously a competitor can only do one thing at a time, and the trials are ordered by time.
- **League <–> Team**. Need to go both ways, no obvious ordering.
- **Team <–> Competitor**. Same as previous bullet.