A Strategy for Optimizing Item-Pool Management

Adelaide Ariel, Wim J. van der Linden, and Bernard P. Veldkamp
University of Twente

Item-pool management requires a balancing act between the input of new items into the pool and the output of tests assembled from it. A strategy for optimizing item-pool management is presented that is based on the idea of a periodic update of an optimal blueprint for the item pool to tune item production to test assembly. A simulation study with scenarios involving different levels of quality of the initial item pool, item writing, and management for a previous item pool from the Law School Admission Test (LSAT) showed that good item-pool management had about the same main effects on the item-writing costs and the number of feasible tests as good item writing, but the two factors showed strong interaction effects.

An item pool is a collection of test items for a given domain usually stored in computer memory along with a list of codes for their attributes. All item attributes can be classified into three different categories, namely categorical attributes (e.g., item content, cognitive level, and item format), quantitative attributes (e.g., word counts, exposure rates, and item-parameter values), and logical attributes (e.g., relations of exclusion or inclusion in subsets of items) (van der Linden, 2005, chap. 2).

Although item pools are typically calibrated using an item-response theory (IRT) model, for example the three-parameter logistic (3PL) model (Hambleton, Swaminathan, & Rogers, 1991), their quality depends only partially on the distribution of the item parameters. Key to the performance of a test-assembly algorithm is the match between the composition of the pool and the complete list of specifications for the tests assembled from it.

For example, in spite of their quality, highly discriminating items are not chosen if their content attributes are not the ones required by the test specifications. In an extreme case, if the pool misses one item with a specific combination of content attributes, it can even become impossible to select a valid test from the pool at all. Also, items in the pool with relatively scarce combinations of content attributes are easily overused. The latter problem creates potential item-security risk in computerized adaptive testing and leads to the necessity of an adaptation of the item-selection algorithm to prevent items from being overexposed (Ariel, Veldkamp, & van der Linden, 2004; Boyd, Dodd, & Fitzpatrick, 2002; McBride, Wetzel, & Hetter, 1997; Revuelta & Ponsoda, 1998; Stocking & Lewis, 1998; Sympson & Hetter, 1985; van der Linden & Veldkamp, 2004).

Suppose that an item pool is optimal for a testing program. If items from the pool have been used for test production, ideally they are replaced by new items with the same statistical and content attributes. Otherwise, the pool may get depleted of sufficiently good items after some time. The strategy for item-pool management studied in this paper is based on this simple principle. In practice, however, a testing program seldom begins with an optimal item pool. The reason often is restrictions on
the resources of the testing organization, which forces it to begin test assembly while
the pool has not yet been completed. Under such conditions, it is difficult to imple-
ment the principle. We will show how item-pool management can nevertheless be
optimized no matter the initial quality of the item pool. Our basic idea is not to begin
with an actual item pool but with a design of it in the form of an optimal blueprint.
The blueprint is periodically updated allowing for the attributes of the new items
that were added to the pool during the last planning period as well as the attributes
of the items that were used in tests assembled from it. The updated blueprints are
used to govern the production of the items needed to complete the pool. Each step in
this process is optimized using integer programming (IP) modeling. The notion of a
blueprint for an item pool will be made more precise below.

In a testing program with paper-and-pencil tests of a linear format, the tests are
frequently assembled from a master pool (Way, Steffen, & Anderson, 1998). For a
program with adaptive tests, the tests are usually not administered from a master
pool but from an operational pool assembled from it (Veldkamp & van der Linden,
2000). Recently, it has been proposed to assemble this operational pool also as a set
of linear tests (van der Linden, Ariel, & Veldkamp, 2006), but even then the adaptive
test is administered from an operational pool and not directly from a master pool. The
same practice occurs in linear on-the-fly testing and multistage testing, where a pool
of exchangeable linear tests or subtests has to be maintained to guarantee uniform
quality of testing for all examinees.

In the current paper, the focus is on the management of a pool for a program with
linear tests assembled from the master pool. This includes the cases of a program
with linear paper-and-pencil tests as well as adaptive, linear on-the-fly, or multistage
testing from a pool of linear tests or subtests. Fluctuations in the quality of the master
pool immediately translate into fluctuations in the quality of the sets of (sub)test.
Optimizing item-pool management to guarantee uniform high
quality of testing over time for these types of testing seems therefore important.

In the next two sections, we first explain the notions of item-pool design and an
item-pool blueprint and then discuss a strategy for item-pool management based on
the latter. In the last two sections, we present a simulation study with this strategy
and discuss the results.

### Designing the Item Pool

Our method of item-pool design is basically a nonstatistical extension of optimal
design methods as they have been developed in statistics (Silvey, 1980) and used,
for example, to optimize item calibration and ability estimation problems in IRT
(Berger, 1997; van der Linden, 1994). A key notion in optimal design theory is that of
a design space. For instance, if the problem is to design an experiment for estimating
the parameters in a linear regression equation, the design space for this experiment
is the set of possible values for the predictor variables. Typically, the full space is
not considered but a subset of points that covers the space in sufficient detail. The
design of the experiment is the distribution of the observational units over these
design points. A design is optimal if the distribution maximizes the accuracy of the
estimators of the parameters.
van der Linden, Veldkamp, and Reese (2000) suggested to approach the problem of item-pool design from an optimal-design perspective. The design space in this approach is the Cartesian product of all (categorical and quantitative) item attributes addressed in the specifications of the tests that the item pool has to serve. The objective function that is optimized is not the accuracy of certain statistical parameters but a function that represents the costs of writing the items.

More specifically, suppose the item pool has to serve a program with parallel versions of a set of different tests \( t = 1, \ldots, T \). The tests are from the same pool but differ in some of their specifications (e.g., an easier and a more difficult test or a system of \( T \)-short tests for use in multistage testing). We use \( n_t \) to denote the number of parallel versions needed for test \( t \). This description of a testing program is general. If the program has only one type of test, we set \( T = 1 \). Likewise, if only one version of test \( t \) is needed, we set \( n_t = 1 \).

Let \( D \) denote the design space for this item pool. The space is defined as the Cartesian product of the categorical and quantitative item attributes used in the specifications for the set of \( T \) tests. The individual points in \( D \) are denoted as \( d = 1, \ldots, D \). Each point represents a possible combination of the attributes that an item can have,

\[
d = (\text{attribute 1, attribute 2, \ldots}).
\]

An example of a design point is (magnetism, four-choice item, presence of a graph, calculus required, 120 words, \( b_i = .50, a_i = 1.00 \)). We will use the term item blueprint to denote a combination of attributes at a given design point. Item blueprints can be used to govern the item-writing process. They give item writers more specific instructions than the usual taxonomy, list of learning objectives, or description of a content domain. We expect item writers to be able to write items with the categorical attributes in the blueprint and with such quantitative attributes as word counts. But they will generally have difficulty producing items with prespecified IRT parameter values. We will return to this point later.

Let \( x_{dt} \) denote the number of items in the pool at design point \( d \) needed to assemble test \( t \). How these numbers can be calculated is addressed in the next section. The total number of items needed in the pool at design point \( d \) for the planning period is

\[
N_d = \sum_{t=1}^{T} n_t x_{dt}.
\]

The total number of items needed (i.e., the size of the item pool) is

\[
I = \sum_{d=1}^{D} N_d.
\]

We will refer to the array of numbers

\[
(N_1, \ldots, N_D)
\]

as the design of the item pool. Alternatively, since (3) can be viewed as the collection of item blueprints for the pool, we will also refer to it as an item-pool blueprint.
The choice of a finite set of points implies a discretization of the quantitative item attributes. This is not a problem as long as we choose a set of values that is typical of the range of the attribute. In fact, the reduction is the same as in IRT-based test assembly when we control a test information function over the range of \( \theta \) for a population of students at a set of discrete values \( \theta_k, k = 1, \ldots, K \), where \( K \) typically is chosen to be in the range 3–5. Also, we do not need to address such logical item attributes as relations of exclusion between items (“enemy items”). Constraints on these attributes have to be imposed only when the tests are actually assembled. For these and other issues in item-pool design, see van der Linden (2005, chap. 10).

**Design Model**

The basic numbers \( x_{dt} \) can be calculated using IP. In the IP model, \( x_{dt} \) are the integer decision variables that tell us how many items we need with the attributes at point \( d = 1, \ldots, D \) for test \( t = 1, \ldots, T \). The variables are used to formulate the sets of specification for the \( T \) tests as constraints in the model. As objective function, we use a cost function based on the cost \( \varphi_d \) of writing an item with the set of attributes associated with design point \( d = 1, \ldots, D \). A suggestion for a cost function is discussed later in this paper.

Our example of a design model is for a program with IRT-based test assembly. Let \( I_d(\theta_k), k = 1, \ldots, K \) be the information at ability level \( \theta_k \) in the response to an item at design point \( d \). This quantity can be calculated from the values of the IRT parameters in the definition of point \( d \). (If the 3PL model is used, we can adopt a common value for \( c_i \), for example, an empirical average and ignore this parameters in our definition of \( D \)). Suppose the tests \( t = 1, \ldots, T \) should have information functions that satisfy absolute target values \( T_{tk} \) at \( \theta_k, k = 1, \ldots, K \), for test \( t \). To show an example of a model with constraints on categorical attributes, we use sets \( V_c, c = 1, \ldots, C \) to denote the partition of the design space by the attribute, for example, a content classification (i.e., if \( d \in V_c \), then \( d \) represents a potential item for content category \( c \)). Finally, let \( q_d \) be the value of an arbitrary quantitative attribute \( q \) for an item at design point \( d \).

A general version of the design model for a program with \( T \) tests is:

\[
\text{minimize } \sum_{t=1}^{T} \sum_{d=1}^{D} \varphi_d x_{dt} \text{ (minimal costs)} \tag{4}
\]

subject to

\[
\sum_{d=1}^{D} I_d(\theta_k)x_{dt} \geq T_{tk}, \quad k = 1, \ldots, K, \quad t = 1, \ldots, T, \text{ (test information)} \tag{5}
\]

\[
\sum_{d=1}^{D} x_{dt} = l_t, \quad t = 1, \ldots, T, \text{ (test length)} \tag{6}
\]

\[
\sum_{d \in V_c} x_{dt} \leq n_{ct}^{\max}, \quad t = 1, \ldots, T, \quad c = 1, \ldots, C, \text{ (categorical attributes)} \tag{7}
\]
The constraints in (5) require the test information functions to satisfy the target values \( T_{tk} \). The constraints in (6) specify the lengths of the tests. In (7) and (8), the numbers of items with categorical attribute \( c \) are constrained by lower bounds \( n_{ct}^{\text{min}} \) and upper bounds \( n_{ct}^{\text{max}} \). Likewise, (8) and (9) impose lower bound \( b_{qt}^{\text{min}} \) and upper bounds \( b_{qt}^{\text{max}} \) on the quantitative attributes of the tests. We need to include as many constraints of the types in (7)–(9) as we have bounds on the categorical and quantitative attributes in the set of test specifications for the tests. The final set of constraints defines the decision variables as integer.

If the tests should be assembled to relative target values \( R_{dt} \), we can maximize the height of the test information function subject to the shape defined \( R_{dt} \) (for the notions of relative and absolute targets for information functions, see van der Linden, 2005, section 5.1.1). This is realized if (4) and (5) are replaced by

\[
\text{maximize } wy - (1 - w) \sum_{t=1}^{T} \sum_{d=1}^{D} \phi_d x_{dt} \quad \text{(test information)}
\]

subject to

\[
\sum_{d=1}^{D} I_d(\theta_k) x_{dt} \geq R_{dk} y \quad k = 1, \ldots, K, \quad t = 1, \ldots, T, \quad \text{(test information)}
\]

where \( y \) is a nonnegative (real-valued) variable and \( 0 < w < 1 \) is a weight for the relative importance of the objectives of minimal item-writing costs and maximal test information. Variable \( y \) serves as a common factor in the lower bounds to the test information in (13) that is maximized in the objective function.

A solution to a design model is a set of values for the variables \( x_{dt} \) that meets all constraints and involves minimal costs. The values can be used to calculate the optimal blueprint for the item pool in (3). Solutions can easily be calculated using a general IP solver, for instance, the solver in CPLEX 9.0 (ILOG, Inc., 2003).

Cost Function

If direct estimates of writing the items for the combinations of attributes at the design points are available, they should be used in the objective function. Alternatively,
we could use subjective estimates of these costs or an empirical estimate of a proxy variable.

A useful proxy to direct costs was recommended in van der Linden et al. (2000). Their class of functions was estimated from a previous item pool for the same testing program. Let $\eta_d$ be the number of items in the previous pool at design point $d$. The proposed class of functions was

$$\varphi_d = f\left(\frac{1}{\eta_d + \delta}\right),$$

with $f(\cdot)$ a monotonically increasing function and $\delta$ an arbitrary small constant needed to prevent infinite values at points with $\eta_d = 0$. Since $f(\cdot)$ is monotone and the function is used in an optimization problem, any choice of function in this class leads to the same result.

The direct assumption on which this proxy is based is that items written more frequently in the past are easier to produce and therefore involve less cost. But the choice of (14) also has a hidden advantage: It allows us to capitalize on the empirical dependencies between the item attributes in the program, in particular on the dependencies between the IRT parameters and all nonstatistical attributes. As already observed, we expect item writers to be able to produce items with all attributes in the blueprints, except the values of the IRT parameters. Although item writers know how to tell a difficult from an easy item, it is typically hard for them to predict their exact difficulty. This observation holds more strongly for the discrimination parameter. Minimization of the function in (14) means preference of the items with combinations of IRT parameter values and the other attributes that occurred more frequently in the past and apparently are easier to write. This strategy may become more effective if the program has fixed item writers and we use these item writers as an attribute in our definition of $D$. We then capitalize on the specific practice of the individual item writers (van der Linden, 2005, section 10.7.2).

If the size of the previous pool is relatively small, the cost estimates in (14) are unstable and it is recommended to smooth the estimates over the quantitative attributes in the design space. A useful smoothing method is $k$-nearest neighbor regression, which replaces the estimate at point $d$ in (14) by the average found in a small neighborhood of $d$. This method was used in the empirical example below.

**Managing the Item Pool**

When item pools are developed dynamically, with items added to the pool and taken from it to assemble tests continuously, the blueprint in (3) should be used as a tool for monitoring and permanent optimization of this developmental process of an item pool rather than in a one-shot approach. In this approach, at appropriate points of time, we compare the actual composition of the item pool with the optimal blueprint and use the comparison to calculate a projection of the additional numbers of items needed at the design points.

The updates consist of the following steps:

1. Establishing the sets of parallel versions of the tests to be assembled in the next planning period;
2. Updating the model to allow for possible changes in the test specifications for the next planning period;
3. Adapting the model to account for the current composition of the item pool (see below);
4. Updating the cost function, using the numbers of items written in the preceding period to reestimate the function in (14);
5. Running the model using an integer solver to determine the numbers of new items that have to be written for the next planning period.

The update in Step 3 of this procedure consists of a simple operation. Let \( p \) denote the planning period for which the update is needed, \( \eta_d^{(p-1)} \) the numbers of items in the pool available at point \( d \) at the end of period \( p - 1 \), and \( x_{d_t}^{(p)} \) the decision variables for the additional numbers of items in the update of the model in (4)–(11) for period \( p \).

Optimal values for \( x_{d_t}^{(p)} \) can be calculated using the updated version of the model with the constraint

\[
\sum_{t=1}^{T} x_{d_t}^{(p)} \geq \eta_d^{(p-1)}, \quad d = 1, \ldots, D \quad \text{(current item pool)}
\]

added to it. This constraint is necessary to account for the attributes of the current items in the pool. (It thus also accounts for the actual values of the item difficulty and discrimination parameters produced by the item writers!)

The projection of the additional numbers needed for \( p \) follows from the following updates of (1) and (3):

\[
N_d^{(p)} = \sum_{t=1}^{T} n_t^{(p)} x_{d_t}^{(p)}
\]  

(15)

and

\[
\left( N_1^{(p)}, \ldots, N_d^{(p)} \right)
\]

**Simulation Study**

The goal of this study was to assess the impact of the management tool above on the quality of a testing program. The quality was measured both by the average costs per written item and the maximum number of tests that could be assembled from the pool. Since these two variables are also dependent on the quality of the initial item pool and the quality of item writing, these two factors were also varied in this study. For the evaluation of the measurement quality of the tests assembled in this study, see the end of this section.

The study was run in the form of a computer simulation of several scenarios for a fictitious testing program for one of the sections of the Law School Admission Test (LSAT). The program was assumed to consist of six testing periods per year, with six versions of the test needed for each period. The size of the item pool was fixed at
350 items. The test-assembly model consisted of 40 constraints on such attributes as test length, content distribution, and word count. The test versions were not allowed to have any overlap in items. The set of test versions for each testing period was assembled simultaneously; that is, we first tried to assemble a set of six versions. If this problem was not feasible, we tried to assemble a set of five, and so forth.

The cost function was the one in (14) estimated from a previous pool of 2,436 items. The function was smoothed using the regression method referred to above. All items had been calibrated using the 3PL model but we omitted the guessing parameter in the definition of the design space and used the average value for this parameter in the previous pool to calculate the values of the item-information functions in the design model. The test-information function was controlled using the actual bounds for the function for the LSAT at $\theta = -1.2, .0, \text{and} 1.2$.

The size of each test was 50 items. Once an item was used in a test, it was permanently removed from the pool and replaced by a new item. Each new period thus began with a complete pool of 350 items. In principle, the maximum number of forms possible from the pool was thus seven per testing period. However, to make the study realistic we simulated attrition of the item pool due to items becoming outdated, showing parameter drift, detection of flaws in their formulation, and so on. Way et al. (1998) report that the number of such items should be expected to be approximately 5% of the inventory. This percentage was used in our study.

The design of the study was the fully balanced three-factor design in Table 1. Each factor had two levels, which we classified as optimal (+) and less than optimal (−). More specifically, the levels were implemented as follows:

1. **Initial pool.** The optimal initial pool was assembled using a blueprint calculated according to the model in (4)–(11), whereas the less-than-optimal pool was randomly drawn from the previous pool.

2. **Management.** Optimal item-pool management was based on the dynamic version of the design model with a new update for each testing period, as proposed in this paper. For the conditions with less-than-optimal management, we sampled the blueprints for the new items from the old item pool. That is, for each new item needed, we sampled an item from it, classified its attributes, and used the list of attributes as the blueprint for the new item. This level of management

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Initial Pool</th>
<th>Management</th>
<th>Item Writing</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>2</td>
<td>+</td>
<td>+</td>
<td>−</td>
</tr>
<tr>
<td>3</td>
<td>+</td>
<td>−</td>
<td>+</td>
</tr>
<tr>
<td>4</td>
<td>+</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td>5</td>
<td>−</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>6</td>
<td>−</td>
<td>+</td>
<td>−</td>
</tr>
<tr>
<td>7</td>
<td>−</td>
<td>−</td>
<td>+</td>
</tr>
<tr>
<td>8</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
</tbody>
</table>
simulates a continuation of the current practice without any explicit attempt to improve or even intervene.

(3) Item writing. It was assumed that all content attributes in the item blueprints were realized but the authors had two levels of command as to the quantitative attributes. For both levels we perturbed the attribute values in the blueprint randomly, but used a level of random error for the less-than-optimal conditions approximately twice as large as for the optimal conditions. For example, the random error in the difficulty parameter was uniformly in [.15–.20] for the optimal level of item writing, but in [.20–.30] for the less-than-optimal level.

Each combination of levels of these three factors was implemented using the following steps:

(1) The initial pool was assembled;
(2) The set of tests for the first period was assembled simultaneously;
(3) The items selected for a test were removed from the pool. Also, from the remaining items 5% was removed to simulate attrition;
(4) The blueprint for the items to be added to the pool for the next period was prepared;
(5) New items were written to this blueprint by randomly disturbing the values of the quantitative parameters in the blueprints.

The simulation of each scenario was repeated five times, and the results were averaged to get stable estimates of the impact of the scenario on the item-writing costs and the number of feasible tests for each period. It should be noted that each test assembled in this study was constrained by the actual bounds on the test-information function for the LSAT. Thus, the number of feasible tests in a scenario also evaluates the measurement quality made possible by the item pool.

**Results**

The results are presented in Table 2. Of course, the case with all the three factors at optimal level (scenario 1) yielded the best results both for the number of feasible tests and the average cost per item. For the optimal pool, the main impact of optimal management but less-than-optimal item writing (scenario 2) was an increase in the item-writing costs, whereas, for the converse case of less-than-optimal management but optimal item writing (scenario 3), the main effect was a systematic decrease of the number of feasible tests. When the less-than-optimal levels of management and item writing were combined (scenarios 4 and 8), both effects occurred. A strong interaction effect between item-pool management and item writing was observed both for the optimal initial item pool (scenarios 1–4) and the less-than-optimal initial pool (scenarios 5–8). As a matter of fact, regardless of the initial quality of the item pool, it was no longer possible to compensate bad management by good item writing, and both the average costs and number of feasible tests deteriorated quickly.
### Table 2

*Number of Feasible Tests and Average Costs per Item in Each Planning Period for the Eight Scenarios*

<table>
<thead>
<tr>
<th>Initial Pool: Optimal</th>
<th>Scenario 1</th>
<th>Scenario 2</th>
<th>Scenario 3</th>
<th>Scenario 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Period</td>
<td>Tests</td>
<td>Costs</td>
<td>Tests</td>
<td>Costs</td>
</tr>
<tr>
<td>1</td>
<td>6.0</td>
<td>.17</td>
<td>6.0</td>
<td>.17</td>
</tr>
<tr>
<td>2</td>
<td>6.0</td>
<td>.17</td>
<td>6.0</td>
<td>.50</td>
</tr>
<tr>
<td>3</td>
<td>6.0</td>
<td>.17</td>
<td>6.0</td>
<td>.38</td>
</tr>
<tr>
<td>4</td>
<td>6.0</td>
<td>.17</td>
<td>6.0</td>
<td>.50</td>
</tr>
<tr>
<td>5</td>
<td>6.0</td>
<td>.17</td>
<td>6.0</td>
<td>.37</td>
</tr>
<tr>
<td>6</td>
<td>6.0</td>
<td>.17</td>
<td>6.0</td>
<td>.42</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Initial Pool: Less than Optimal</th>
<th>Scenario 5</th>
<th>Scenario 6</th>
<th>Scenario 7</th>
<th>Scenario 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Period</td>
<td>Tests</td>
<td>Costs</td>
<td>Tests</td>
<td>Costs</td>
</tr>
<tr>
<td>1</td>
<td>5.0</td>
<td>.48</td>
<td>5.0</td>
<td>.48</td>
</tr>
<tr>
<td>2</td>
<td>3.3</td>
<td>.17</td>
<td>3.0</td>
<td>.39</td>
</tr>
<tr>
<td>3</td>
<td>4.4</td>
<td>.19</td>
<td>4.0</td>
<td>.65</td>
</tr>
<tr>
<td>4</td>
<td>5.2</td>
<td>.18</td>
<td>4.2</td>
<td>.66</td>
</tr>
<tr>
<td>5</td>
<td>5.4</td>
<td>.18</td>
<td>4.4</td>
<td>.41</td>
</tr>
<tr>
<td>6</td>
<td>6.0</td>
<td>.17</td>
<td>3.0</td>
<td>.53</td>
</tr>
</tbody>
</table>

**Concluding Comments**

The previous results are from one study only, and we should be careful not to over-generalize. Nevertheless, we believe that they underscore the importance of high-quality item-pool management. For a testing program, we expect the number of feasible tests per testing period to be the most important success parameter. As Table 2 shows this number was always considerably higher for the cases with optimal management (scenarios 1, 2, 5, and 6) than with less-than-optimal management (scenario 3, 4, 7, and 8). The strong interaction between item-pool management and item writing in Table 2 also points at the critical importance of good management. It turned out to be impossible to compensate for bad management by good item writing.

Further, the results show that once the number of feasible tests starts to decrease for a scenario, it tends to keep decreasing. This makes sense because of the cumulative nature infeasibility over time. If a set of tests is assembled, the good items in the pool disappear, while the bad items remain. As a result, the likelihood of a lower number of feasible tests in the next period increases.

The last results thus confirm the observation in Way et al. (1998) that item pools tend to decrease in quality over time, and, consequently, an increasing number of new items have to be produced to maintain the testing program. The management tool presented in this paper is one instrument that could be used to break this tendency.
References


van der Linden, W. J. (1994). Optimum design in item response theory: Applications to test assembly and item calibration. In G. H. Fischer & D. Laming (Eds.), *Contributions to mathematical psychology, psychometrics, and methodology* (pp. 308–318). New York: Springer.


Authors

ADELAIDE ARIEL is a Research Assistant, Department of Research Methodology, Measurement, and Data Analysis, University of Twente, PO Box 217, 7500 AE Enschede, The Netherlands; a.ariel@utwente.nl. Her primary research interests include optimal test assembly and item-pool design.

WIM J. VAN DER LINDEN is a Professor of Measurement and Data Analysis, Department of Research Methodology, Measurement, and Data Analysis, University of Twente, PO Box 217, 7500 AE Enschede, The Netherlands; w.j.vanderlinden@utwente.nl. His primary research interests include test theory, applied statistics, and research methods.

BERNARD P. VELDKAMP is an Assistant Professor, Department of Research Methodology, Measurement, and Data Analysis, University of Twente, PO Box 217, 7500 AE Enschede, The Netherlands; b.p.veldkamp@utwente.nl. His primary research interests include optimal test assembly, computerized adaptive testing, and operations research.