Abstract— An approach to error-based testing is described that uses simple programmer error models and focus-directed methods for detecting the effects of errors. Errors are associated with forgetting, ignorance, bandwidth and perversity. The focus-directed approach was motivated by the observation that focus is more important than methodology in detecting such errors. The strengths and weaknesses of error-based versus more methodological methods are compared using three underlying assumptions called the faith, coincidence and hindsight effects. The weaknesses of error-based testing are compensated for by establishment of an expertise-based foundation that uses research from the study of natural decision making. Examples of the application of error-based methods are given from projects in which the author had access to the programmers, making it possible to track failure back to both defect and error. The relationship of error-based testing to contemporary methods, such as context-driven and exploratory testing, is described.

Keywords - errors; defects; testing; analysis; comments; focus; functions; checklists; expertise; learning; context-driven; exploratory

I. INTRODUCTION

The error-based test selection paradigm that is used here has several components. One is the identification of general classes of errors that occur during program construction. Another is the assumption that the expert tester knows how to interpret general error classes in particular contexts, and choose relevant tests and test methods.

The following section looks a little deeper into the nature of an error, and is followed by error-oriented descriptions of various kinds of defects. This is followed by sections on: development versus testing and analysis, comparative effectiveness of methodological versus error-based testing, expertise, test-set completeness, error-based testing support tools, related work and conclusions.

II. ERRORS AND FOCUS

An early, specifically error-based, program testing and analysis method was described in [1]. In this project, errors were associated with "forgetting". Forgetting was assumed be responsible for the following kinds of programming errors: forgetting what kind of data was currently stored in a variable or register, and forgetting the meaning of particular data values. The technique involved inserting program comments that documented variable and data interpretation "assumptions", which were checked by the analysis tool against other comments, called "facts".

The comments analysis approach in [1] was applied to the avionics code for the AV-8B aircraft. The code was re-commented with assumptions and facts, which were then analyzed for consistency. Several critical errors were uncovered. Error-based analysis was successful in this project, although it appears to have been tied to the kind of programming language that was being analyzed.

In the comments analysis work, the initial goal had been to analyze the existing natural language comments. After a preliminary analysis, this was abandoned for reasons of accuracy and practicality. Instead, the complete code base was re-commented with comments that were amenable to automated analysis. Commenting a program after the fact had the advantage of being able to reflect on different aspects of the code, which is not possible in the middle of the program construction process.

General observations on the nature and the detection of errors also occurs in [2]. In this research it was discovered that what was needed for defect detection was not so much a new testing and analysis method as "focus". A program was interpreted as consisting of a collection of "functions". At the lower levels, functions were associated with things like array index calculations, or loop exit decisions. At the higher levels they were associated with design level concepts. By focusing on each of these aspects of the code separately it was observed that relatively simple test methods, such as black box testing, were adequate for defect detection.

There are many models of human error. In the model used here we assume that when a program is being developed, the programmer or designer focuses on one or more aspects of the program. Errors occur because of "defocus" or "exclusion". In defocus, some aspect of the program receives weak focus in the sense of not having received undivided attention. This may be due to the limited bandwidth of the brain, only capable of actively considering a limited number of concepts at the same time. In exclusion,
some aspect is not considered at all, either because it did not occur to the programmer, due to forgetting or ignorance, or it was simply disregarded.

The primary means of error detection that we will consider is the use of checklists. The elements of these lists remind the tester to separately focus on the associated views or aspects of the code, probably in several separate passes to avoid defocus errors during testing. The lists are not meant to be detailed descriptions of possible kinds of defects, which have been used since the beginnings of programming [3]. They are meant to serve as a link between the tester's knowledge and the application at hand. The mental process used in applying checklists is one of analogy and metaphor rather than detailed pattern matching. Checklists may be accompanied by stories and examples that can be used in this process.

III. FOCI AND ERROR STORIES

This section contains examples of errors and of focal checklist entries that could have been useful in discovering them. The goal is for the checklist items to be general enough so that they can be used across different applications and application domains. This can be contrasted with "rote" checklists, which go as far back as [3], whose goal is to be detailed enough so that the tester would know exactly what to look for. A consequence of this is that we have to have faith in the expertise of the tester to fill in the details for a particular application. In the extreme case, testing would rely completely on the experience of the tester, with no aids or guides. Part of the argument given here is that, due to the limitations of the mind, this cannot be relied on. The checklists are seen as a codification of expert knowledge that is necessary for expert performance.

The examples used in this section are taken from two sources. One is a Survey system installed on a handheld device. The other is a computerized dating system (DS). In both the Survey and the DS examples, the developers were available for discussion and analysis, providing invaluable information about how the errors occurred that led to the defects in the examples.

Each example includes a story and one or more relevant checklist items. The proposed checklist items correspond to the excluded or degraded foci. The idea is that the checklist items, together with associated stories, ensure the tester's discovering similar focal related errors in other examples.

The examples are ordered according to whether they are requirements, design, or programming oriented.

A. Requirements Foci

In this case, the foci correspond to high-level external points of view. Sample viewpoints include "stakeholders", such as users and customers. Errors correspond to not looking at a program from a point of view, due to simple ignorance or due to knowing about it but not giving it undivided attention. By focusing on a point of view, the tester is able to determine if a defect-creating error has occurred that is related to that point of view.

Example 1. Survey Program - Customers Set Themselves Up: The customer supplied a list of tests that was required to be used during acceptance testing. Without critical reflection, the testers made sure the program ran flawlessly for these tests, i.e. they focused on the aspects of the software that were high value. The demo staff also focused on these tests. When the demo occurred, the customer was impressed with the ease of using the system and did not require further testing. This had several effects. One was that sometime later there were complaints of awkwardness of use. This was because the actual users were not performing the test cases for which usability had been polished. In addition, defects were found for other input cases.

This example illustrates two points. One is the necessity of a customer test requirements checklist item, perhaps nested under a generic customer item. The other is that the expert tester, in successfully using the list item in this example, would have had to have had a range of experience that included cases in which the customer led the developer astray. The association of this story with the checklist item would help it be applied to new projects, using reasoning by analogy. The story could also be referenced from an (un)expected user actions checklist item.

Example 2. Survey Program - You're Not the Real User: After releasing the system, the users complained about the formatting of the user interface, about the ordering of menu items, and the text used for the items.

The checklist item here is user. We might also consider having a real users, subordinate checklist item. The developer interviews revealed that the developers did the testing, pretending to be real users. This entry, along with the associated story, will provoke the tester into thinking of similar tests in future testing contexts. The example emphasizes the importance of being able to interview development staff to find out why an error occurred.

B. Design Foci

We will look at two examples here, both relating to sequences of actions taken by a user of the Survey system. In both cases, the problem is one in which the developers focused on the "normal" cases, and failed to give undivided focus to the "oddball" cases.

Example 3. Survey Program - I Changed my Mind: The Survey system allows the user, at any intermediate step, to review previous entries by going to a "review" mode from the top level "edit" mode. While in review mode, the user can set a pointer to one of the listed entries, and then return to the edit mode to change it. In this case, the user went to review mode, set the pointer, went back to the edit mode and then changed his mind and went back to review mode. At this point all the entries in the entry list after the pointed at item had disappeared. The designers had assumed they would not be needed and had caused them to be deleted.

We might tie this story to a user checklist item, a real
user checklist item, or to an (un)expected user action item. The construction of revealing tests by a tester would have depended on the tester's experience in having seen this kind of thing before. The accumulation of user checklist item stories like this helps to pass along experience from one tester to the next.

Example 4. Survey Program - I Want this to End Here and Now: This example is also from the review/edit mode feature of the Survey system. The user went into review mode, and then decided to terminate the input and get the final computation. Consequently he moved the pointer to the last data item and went back to edit mode, where an empty entry box was displayed. At this point, the closure button had been depowered because it was assumed that the user wanted to enter yet another data item, so that he was not able to terminate the session.

This is another example of an (un)expected user actions checklist item story, or perhaps also a desired user action checklist item. Exploration of these foci might require consultation with real testers.

Example 5 Survey Program - What Other Formula?: The user's location is computed from bearings from three points. Different formulae should be used depending on how big the total angle is between the directions. The programmer/designer did not know that there were different formulae and that another formula was needed for a narrow band of angles.

In this case there is a missing aspect/focus due to simple ignorance. The focus approach suggested in [2] used the single focus checklist item function, together with what is called here edgecase focus. Recall that the idea was to identify all the "functions" in the code, implicit or explicit, and test them separately, i.e. give them undivided focus. The application of this kind of focused testing would have found this error, because edgecase testing would have forced execution of the code over the angle sets for which it is inaccurate. This example could serve as a story which directs attention to missing subfunctions.

C. Programming Foci

When a program is being constructed, it is necessary to simultaneously consider several things while the idea for the program is being translated into code artifacts. For example, in the case of a loop, we have the principal alternative normal flows, special cases where no flow occurs, and loop termination conditions.

In addition to "normal" data, we have what has been referred to as edgecase data. This includes boundary values, empty data structures, or initial and final cases. It is commonly understood in black box testing that this is where errors often occur. It is possible to define simple kinds of edge case tests, but the concept of an edge case can be interpreted differently in different contexts, relying on the experience of the tester. The approach suggested here is to use a general checklist item, edgecases, which is interpreted by the knowledgeable tester according to the context, guided by stories.

The following example combines the edgecases checklist item with a loop termination item. Neither of these points of view received undivided attention by the programmer when the associated code was being constructed.

Example 6. DS Program - Respecting Variable Array Bounds: If the DS (Dating System) administrator attempts to delete a member, and that member is non-existent, while at the same time no existing member has previously been deleted, then the system fails.

Analysis, including programmer interviews, indicated that the error was implementation-oriented. The defect traces back to an array reference inside a loop whose loop counter is used as the basis for the array index. The (virtual) array bounds for the loop are adjusted dynamically with local variables. An out-of-bounds reference will occur if the maximum value of the loop counter is achieved. The checklist item in this case is array bounds, requiring testing over maximal and minimal array indices.

Interviews indicated that the error occurred during programming, when focus was on the loop termination conditions and the array reference was defocused. Discovery of errors like this requires both focus on the array reference as well as the use of an invariant that checks the index bounds during testing. The invariant is necessary because an out of bounds reference does not always cause incorrect program behavior. The invariant, like the focus on array references, is needed to take the defect back to its source.

This error was previously used in [4] as an example where BET (Bounded Exhaustive Testing) at the user level would have been effective. Recall that BET is an approach to practical testing of input combinations. In that paper, this error was described as discoverable if a sequence of user input actions was used in which the user deletes a non-existing member before having deleted an existing one. This short sequence would be covered in a BET approach. In the description here, the defect is traced back to its source, the illegal array reference, which provides a more direct approach to its detection without the necessity of considering combinational coverage.

Example 7. DS Program - Cannibalizing Input Streams: The checklist item in this case is input streams. In the story here, the input comes from a token-separated file. Under certain circumstances the file can be generated with empty items, i.e. the separators are adjacent with no entry between them. When this occurs the next separator is used as the data, so that attempts to read beyond the end of the file or subfile will occur.

The error occurred in this case because the programmer, when writing the code, focused on the mainline case in which data is read in to be stored in a vector. Little consideration was given to the kinds of edge cases that could occur in the input data. A tester focusing on input streams, combined with edgecase, has a good chance of finding the defect. The experience factor, perhaps prompted
or informed by this story, would raise the need to test "virtual" edge cases like this one, which do not directly correspond to programming language constructs such as files or vectors.

This is also an example that was previously used in discussions of user-level BET combinational testing. If we tested over short user action sequences, with black box user input data cases, the defect would manifest itself. It corresponds to the combination in which a new user was added by the administrator and the system was shut down before the new user logged on and added his or her personal data to the new user record. When a subsequent start-up occurred, the system malfunctioned. Closer analysis of this problem, traces the testing problem back to an input data case where it is more directly discoverable, without the need for considering user action combinations.

IV. DEVELOPMENT VS ANALYSIS AND TESTING

One obvious question is "why not provide these foci lists to the developer so that the errors can be prevented from occurring in the first place?" An assumption of the focal point error model is that developers cannot keep all aspects of the code in their heads at once and it is necessary to add post-initial-phase analysis or testing phases in order that undivided attention can be made to each aspect of the code.

An argument can also be made that it is necessary to have a tester as well as a developer. An expert in testing and analysis will have more experience with defects across a variety of systems, leading to superior capabilities for investigating focal failures.

Another question might be, "since you have checklists, why not just analyze the code for problems directly, instead of running tests". For some of the above examples, such as I Want This to End Here and Now, this is precisely the case. For others, such as I Changed My Mind, or Cannibalizing Input Streams, analysis of the code or design may not help because they require testing to see if the code has certain operational properties.

V. ERROR-BASED VS METHODOLOGICAL TESTING

In this section we will briefly compare the error-based approach with other methods for several of the examples given above. The methods are branch coverage, mutation testing, black box, and random testing. They are referred to as "methodological approaches". The goal is not to attempt to determine the "best" method but to gain additional insight into error-based testing. In the case of Customers Set Themselves Up, traditional test methods would work because they would force the use of test cases outside the customer test base. In the case of You're Not The Real User, a focus on unexpected user-step choices is the only approach that would seem to be effective. In the case of Respecting Variable Bounds, the use of an array bounds checklist item, with its associated invariance assertion, is necessary. This is because other test methods will not force an out of bounds element has not been previously set to a non-null value. In the case of Cannibalizing Input Streams, tester expertise is necessary to test over virtual edge cases, something that is not forced by any of the methodological approaches. Random testing may not be effective in these examples because the failure domain footprints are not large enough to guarantee probable choice of error revealing tests.

In general, one gets the impression that the methodological approach may simply not be in tune with the ways in which defects actually occur, and that the effectiveness of such approaches is dependent on certain coincidences, such a defect's failure footprint being large enough for statistical testing, or the choice of data that covers a defect-containing statement to be just the right data that will cause a program failure. This observation will be referred to as the coincidence effect.

However, there are also potential criticisms of the checklist error-based approach, which we will call the hindsight effect and the faith effect. The hindsight effect refers to the practice of inventing focal checklist items on the basis of experience, i.e. of having seen an error that prompts the creation of that list item. The faith effect results from the generality of checklist items. This is necessary so that they can be used across applications and application domains. But this means that their effectiveness depends on the testers being able to interpret them for a particular application, leading to tests for discovering errors associated with that checklist focal item. One must have faith that the tester can effectively explore the application of a checklist item in a particular context. The hindsight effect is mitigated if the checklist items are general and make sense, based on repeated experience. The faith effect is mitigated through the use of stories associated with focus items, through the assumption of expertise, and through the use of focal coverage tools.

VI. EXPERTISE

One of the goals in testing research has been to make testing as methodical and systematic as possible. Coverage-based testing, mutation testing, and random testing all try to make the selection of test data rational and rote-like. This methodological emphasis can be compared to that of methodological decision-making methods. In his seminal book [5], Klein contrasts rational/methodological decision making with what he calls "recognition-prime decision making" (RPD). In the latter approach, the expert observes the situation and "knows" what to do next. This may involve exploratory actions to gain more information, but choices are based on reasoning by analogy and metaphor, as opposed to rote-like rules that are based on precise pattern-matching. Klein argues that RPD is what experts do in practice, and that methodological approaches are best suited to novices.

Error-based testing is expert-oriented, like Klein's RPD approach, in that the tester, guided by focus checklist items and "experience stories", recognizes by analogy how to
Finally, at the end of the day, software development, like based methods.

Klein's and other work on expertise, [e.g. 6], together with the foci-based error model, provide a foundation for the error-based approach. The error model directs the focus of the tester to specific aspects of the software for undivided attention, and the expert consideration justifies the assumption that experienced testers will know how to apply general checklist items to new applications and application domains.

VII. TEST SET COMPLETENESS

It is common to see statements that test set incompleteness is an unavoidable problem because of the large or infinite number of possible inputs. In fact, large numbers of possible inputs is not necessarily the problem. The underlying problem is that the selection of completely effective test sets is known to be undecidable. This indicates the need for some working definition of test completeness other than correctness, or even approximations to correctness. In the systematic/methodological approach, "completeness" is defined by the process. e.g. a branch-coverage test set is complete when all (or in practice a high percentage of) branches has been covered. A mutation test set is adequate when all the (non-equivalent) mutants have been killed. A random test set is complete when a certain number of (failure-free) tests have been observed. There does not seem to be any comparable well-defined measure of completeness for error-based testing. Examples may indicate that methodological approaches are weaker, but at least you know when you are done.

Several arguments can be used to mitigate completeness anxiety associated with error-based approaches. The first is the observation that for some focal checklist items, such as acceptability of user interface formats, the focus may reduce the test selection problem to the identification of an acceptable, specific finite set of tests. A second is that if focal coverage tools are used, like those described below, error-based test requirements may subsume standard code and model coverage measures, so that if we are willing to feel secure with their concept of completeness, we should be willing to accept that of more intuitively directed error-based methods.

Finally, at the end of the day, software development, like all human engineering projects, relies on faith in the professionalism and expertise of the engineers and technicians. Certain aspects of the process, such as proofs of algorithm and model properties are scientific and mathematical, but others, such as context-specific design features, or physical implementation, may not be. Following Klein [5], completeness in this context must go beyond the novice-like completion of a set of rote rules. It has to also include acceptance of a process in which an expert's recognition of critical aspects of the project, using metaphors and reasoning by analogy, leads to the context-tailored application of methods and techniques that have been seen to be effective. For example, in Cannibalizing Input Streams, the defect would have been detected because the knowledgeable tester, applying the general concepts of an edge case, would know to test with empty token-separated data items.

VIII. FOCAL COVERAGE TOOLS

Coverage tools can be used to support error-based testing and analysis, particularly at the implementation and design levels. Traditional coverage tools may be useful, but perhaps in a non-standard way. The goal is not so much to achieve full coverage with a set of tests, but to help the programmer explore the foci on a checklist. For example, when focusing on loop termination alternatives, an error-focused coverage tool might be used that would report the ways in which a program loop has terminated over a set of tests, together with the edgecase data that was involved in the terminations. For some examples, this may correspond directly to reporting test coverage of loop expression conditions.

The goal in constructing error-based coverage tools is to provide the information the tester needs to confirm that a focus item has been appropriately explored. This information may be necessarily more primitive than the item if the item suggests the examination of abstract constructs that are not directly reflected in the code. In the Cannibalizing Input Streams example, the application of edgecase testing requires that the programmer consider token-separated files with no data between the tokens, a concept that is not directly reflected in the code. A coverage tool that reported the data over which the input statement was executed during testing could be effectively used by expert testers, through their ability to identify higher level token-separated input subsequences.

Other kinds of test coverage tools include those for reporting user-system interactions, in which finite-state models are constructed from user action traces over a set of tests. From these the tester can evaluate whether focal checklist items, such as (un)expected user actions, have been appropriately examined.

IX. RELATED WORK

In addition to the work described above, there are several other important associations. The concept of an expert
tester, and the rejection of methodological approaches is found in Bach's published talks [7]. His description of an expert tester matches the concept of an expert introduced by Klein [5]. In addition, Bach supplies checklists that he refers to as "test models". They are more eclectic that the lists suggested above, including a variety of kinds of items in addition to what have been called focal points here.

A group of testing people, including Bach, announced in 1999 what they call the "Context-Driven School of Testing". There is a general connection with the error-based approach described here, since the details of the application of a focal point general concept will be determined by the system/context in which it is applied. In addition to their occurrence in Bach's work, expert-oriented test checklists can also be found in Kaner's online lectures. A 14 item list used in [8], for example, contains a set of different kinds of goals for testing. Entries include "block premature product release" and "check interoperability with other products". These might occur in an error-based approach as high level process-oriented foci. The idea that programs have aspects, or themes that direct testing efforts, was used by Marick in his book [9], where he referred to them as "clichés". In this case, however, they are limited to several low level aspects such as "searching" and "sorting".

The work described here represents a departure from recent methodological approaches whose goal was to avoid the faith effect. In the Test Selection Patterns approach described in [10], the goal was to identify testing patterns consisting of a test context and an associated test rule. The approach was defect-oriented, and the patterns were low level, avoiding the faith effect. The patterns were grouped into application domain and subdomain categories. The idea was that the programmer/tester would determine if the application was in a particular category, and then examine the test patterns for relevance.

There are two problems with the test patterns approach: the hindsight effect and the classification problem. The first problem occurs because each pattern is too closely tied to the defect example that inspired it. This means that we would have test rules for the defects we had seen before, but not for the new ones we might see in the future. The second problem occurs because it was found to be difficult to come up with unambiguous, decisive categories that would always direct the tester to the relevant set of patterns.

As indicated above, in the checklist-driven approach, where the entries are suitably general, the expert tester is expected to come up with relevant tests for the application at hand, guided by the meaning of the checklist item. This mitigates both the hindsight and the classification problems.

Alternative methodological work that attempted to avoid the hindsight problem in the test patterns approach adopted a strategy of testing combinations of input data which might be associated with kinds of bugs not yet seen, i.e. combinations that did not necessarily match current hindsight-derived test patterns [10]. The methodology that was used, called BET (Bounded Exhaustive Testing) [11] requires the selection of finite test input subdomains corresponding to "small" versions of the application. The subdomains include typical edgecase data and sample data from non-edgecase "uniform" subdomains. A program is tested over all combinations of data from the finite subdomains. In general, this requires both automated test data generation and an automated oracle. The referenced work describes techniques for constructing "partial" automated oracles in cases where it is impractical to construct full automated oracles but it is still possible to automatically check necessary output or behavioral properties. In some cases, it is necessary to apply BET starting in an intermediate state of a program's computations. This can be done with a set-up pretest in which the program is run on special tests that take it to that state, where a test tool could then apply BET. The BET approach certainly mitigates the faith effect, when compared with the test patterns approach, but it has its own faith component. One must have faith in the programmer's choice of finite subdomains, and also in the ability to identify intermediate states in which to initiate a BET test process. In addition, BET has an oracle problem. It generally relies on large numbers of tests, requiring automated test generation, which leads to the need for an automated oracle that may be difficult to construct.

As mentioned above, in Examples 6 and 7, considering a defect in terms of its originating focal aspect error, as opposed to the application's input data, replaced a combinational interpretation of the defect requiring BET with one in which a carefully chosen, relatively small set of tests was adequate.

The newest related work occurs in Whittaker's interesting book on Exploratory Testing [12]. Whittaker lists a number of test selection metaphors that describe a general approach to test selection. In his discussion of some of the metaphors, he lists possible related program defects or originating errors. In some cases, there is a direct relationship between one of his metaphors, and the foci listed in this paper. For example, the Antisocial Tour is so named because it is associated with people who do not act according to accepted social norms. In the case of testing, the intention is to simulate users that perform unexpected steps. Tests derived from this metaphor are the same as tests for testing the (un)expected next steps foci. The advantage of metaphors is that they are very general and can be applied to very different application domains. However, when the number of them starts to proliferate it is hard to keep track of what they mean. They may also be less effective when they suggest a category of tests rather than a set of steps to be followed by a tester, [cf. 6], or when there is no obvious connection between the metaphor and testing.

X. CONCLUSIONS

The paper introduces a simple error model that is used to synthesize ideas from traditional testing, common testing
practice, development frameworks, and cognitive science.

In the approach described in this paper, errors occur because programmers fail to fully focus on some aspect of a program. Underlying causes include forgetting, bandwidth, perversity, and ignorance. Bandwidth relates to a programmer's inability to hold more than a small number of things in his mind at the same time. Perversity refers to examples in which a programmer was neither ignorant, nor forgetful, nor bandwidth limited, but chose not to focus on some aspect of the software anyway.

Focus errors may be inevitable due to human limitations. Consequently, testing (and analysis) is viewed as an unavoidable integral phase in development, a later phase in which it is possible to give each aspect undivided attention. Test selection is guided by checklists of focal items. Test output may take a variety of forms, including traces. Three trade-off factors were introduced that can be used to compare approaches to testing and analysis that are unavoidable because of undecidability: the hindsight, coincidence and faith factors. Traditional research in testing has concentrated on methodological approaches that can be applied by novices with little experience, i.e. you do not have to have "faith" in their expertise. But defect discovery using these methods may be a "coincidence", as in happening to choose just the right data to cause some branch to be executed in coverage-based methods. They may also be "hindsight" limited, involving specific descriptions of defects that occurred in the past, which are then used to attempt to find future defects.

The error-based, focus approach discussed here involves checklists of aspects, or views, of the software. The lists are developed from past experience, but the hindsight factor is mitigated through the use of general aspects, as opposed to specific defect patterns. The coincidence factor is avoided because the views narrow the testers focus on potential kinds of errors associated with a viewpoint, rather than finding them by coincidence while pursuing a possibly orthogonal goal, such as coverage. The potential weakness in the approach is the faith factor: we have to trust that testers will have the expertise needed to apply a general viewpoint to a new application, guided by associated stories about past errors. We have to have faith they will be able to construct, by analogy, tests that will expose related errors in that application. This potential weakness is mitigated through an understanding of expertise: what it consists of, how it is learned, and how it is recorded so that it can be referenced or passed on to others. The seminal work of Klein on decision making was used to supply an expertise-based foundation for the error-based approach.

Several examples were given of the application of the error-based approach, which has been investigated for an industrial survey system and for an experimental dating system. Relevant focal checklist items were identified. The results of the investigation were positive. The error-based approach provided a more meaningful, less coincidental approach to testing. It also provides a framework for method integration. Black box testing, for example, can be viewed as a subset of error-based testing. Coverage testing is more meaningful when it gathers directed feedback information, corresponding to different error-based tests. The use of models in model based-testing can be viewed as a means to avoid bandwidth errors. Models allow the designer to record important design entities and interrelationships. Because they are recorded, different aspects can then be separately analyzed. In the case of a user interface model, for example, instead of generic coverage measures such as the Postman Tour, we can use focused coverage criteria corresponding to checklist items such as "unexpected user actions", or "functional side tours".

REFERENCES