

# *Teaching Testing: A Skills-Based Approach*

---

Cem Kaner, J.D., Ph.D.  
Department of Computer Sciences  
Florida Institute of Technology

May, 2001  
Software Quality Week

# Abstract

- Training software testers involves teaching culture, vocabulary, concepts and skills. I think that many of the commercial seminars (and certification review courses) teach vocabulary and many concepts quite well. Some of them address cultural issues. Fewer address skills, but skills development is essential for new testers.
- I've been training testers for 18 years, sometimes personally coaching them, sometimes teaching a pretty successful commercial seminar, and now teaching undergraduate and graduate students. Over the past year, I've rethought my approach to teaching.
- When I studied mathematics, we learned a lot of conceptual material, but we also did a lot of drill--exercise upon exercise--from the course text and from exercise books like the Schaum's Outlines. These exercises forced the student to learn how to work with the concepts, and how to apply them under a wide range of circumstances.
- Working primarily with James Bach, Helene Astier, and Alan Jorgensen, I've been trying to develop a list of specific skills that testers use in the normal course of their work, and then develop exercises that will practice them in those skills. I have several such exercises now and am developing more as I go.
- This session will go through practice exercises in bug reporting, domain testing, combination testing (all pairs), specification analysis for ambiguity, specification analysis for holes, and possibly some other areas. The more time we have, the more techniques we look at. In this session, I'll share course notes, quizzes, exam questions, and explain how I use them.

# Scope

- This workshop is focused on classroom-style teaching, though much of it will apply to programs involving self-paced self-study.
- I've been involved in classroom-style teaching:
  - As a senior tester, conducting classes for my staff (in-house “Tester College”.)
  - As an instructor on the road, teaching 3-day seminars in hotels.
  - As a University of California Extension instructor, teaching courses to working professionals on nights and weekends, covering my 3-day seminar in some courses and test automation architecture in others.
  - As a traveling test teacher, delivering courses onsite at several companies.
  - As a university professor, teaching senior undergrads and various graduate students.

# *Work In Progress*

---

- These slides are far from complete. It feels to me as though I am at the beginning of skills analysis for software testing tasks and techniques, not the end. These slides reflect that early stage:
  - They are inconsistent in depth
  - The exercises are of varying quality
  - The range of exercises per topic varies widely
- I expect to work on this for several years. Your comments, criticisms, and suggestions would be very much appreciated. Please write me at [kaner@kaner.com](mailto:kaner@kaner.com).

# Learning Styles

- People learn in different ways. There are several models of learning styles. Different presentation methods work best for different styles of learners.
- I'm not up to date on this literature. Back when I was a grad student in psychology, we talked about
  - Visual vs. auditory learners. *Visual learners have more success in lectures if they have a copy of every slide*
  - Active vs. passive learners. *Some people learn better in lecture or by reading / memorizing. Others learn better through exercises.*
  - Individual vs. group learners. *Some people learn better on their own, others on project teams.*
  - Concrete vs. conceptual learners. *Some people learn better from detailed stories, others from the formal presentation of the underlying principles.*

# *Learning Styles*

---

- I'm still thinking about how to develop a course that works well with the different styles. (I'm especially interested in this for building on-line courses that are worth teaching/taking).
- For now, my approach involves:
  - Lecture of key material
  - Hard copies of every slide
  - Many stories with rich detail to support key points
  - Several assignments, which can be done by individuals or jointly
  - Sample questions that can be studied by individuals or jointly.
- And will soon involve
  - Self-study drills
  - A library of examples that students can read on their own.

# Learning Styles

---

- There's nothing magical about the division that I learned, and several others are more popular in the literature.
- For a useful introduction and discussion of several style classifications, see R.M. Felder, *Matters of Style*, <http://www2.ncsu.edu/unity/lockers/users/f/felder/public/Papers/LS-Prism.htm>
- Felder makes the point that the basic lessons for instructional design end up being pretty similar across the different classification schemes. The next slides, taken verbatim from his paper, show his advice to chemistry teachers:

# *Examples from Felder's Matter of Style*

---

“Here are some strategies to ensure that your courses present information that appeals to a range of learning styles. These suggestions are based on the Felder-Silverman model.

- *Teach theoretical material by first presenting phenomena and problems that relate to the theory* (sensing, inductive, global). For example, don't jump directly into free-body diagrams and force balances on the first day of a statics course. First describe problems associated with the design of buildings and bridges and artificial limbs, and perhaps give the students some of those problems and see how far they can go before they get all the tools for solving them.
- *Balance conceptual information* (intuitive) *with concrete information* (sensing). Intuitors favor conceptual information--theories, mathematical models, and material that emphasizes fundamental understanding. Sensors prefer concrete information such as descriptions of physical phenomena, results from real and simulated experiments, demonstrations, and problem-solving algorithms. For example, when covering concepts of vapor-liquid equilibria, explain Raoult's and Henry's Law calculations and nonideal solution behavior, but also explain how these concepts relate to barometric pressure and the manufacture of carbonated beverages.



# *Examples from Felder's Matter of Style*

---

- *Make extensive use of sketches, plots, schematics, vector diagrams, computer graphics, and physical demonstrations (visual) in addition to oral and written explanations and derivations (verbal) in lectures and readings.* For example, show flow charts of the reaction and transport processes that occur in particle accelerators, test tubes, and biological cells before presenting the relevant theories, and sketch or demonstrate the experiments used to validate the theories.
- *To illustrate an abstract concept or problem-solving algorithm, use at least one numerical example (sensing) to supplement the usual algebraic example (intuitive).* For example, when presenting Euler's method for numerical integration, instead of simply giving the formulas for successive steps, use the algorithm to integrate a simple function like  $y = x^2$  and work out the first few steps on the chalkboard with a hand calculator.
- *Use physical analogies and demonstrations to illustrate the magnitudes of calculated quantities (sensing, global).* For example, tell your students to think of 100 microns is about the thickness of a sheet of paper and to think of a mole as a very large dozen molecules. Have them pick up a 100 ml. bottle of water and a 100 ml. bottle of mercury before talking about density.

# *Examples from Felder's Matter of Style*

---

- *Occasionally give some experimental observations before presenting the general principle, and have the students (preferably working in groups) see how far they can get toward inferring the latter (inductive).* For example, rather than giving the students Ohm's or Kirchoff's Law up front and asking them to solve for an unknown, give them experimental voltage/current/resistance data for several circuits and let them try to figure out the laws for themselves.
- *Provide class time for students to think about the material being presented (reflective) and for active student participation (active).* Occasionally pause during a lecture to allow time for thinking and formulating questions. Assign "one-minute papers" near the end of a lecture period, having students write on index cards the lecture's most important point and the single most pressing unanswered question. Assign brief group problem-solving exercises in class that require students to work in groups of three or four.
- *Encourage or mandate cooperation on homework (every style category).* Hundreds of research studies show that students who participate in cooperative learning experiences tend to earn better grades, display more enthusiasm for their chosen field, and improve their chances for graduation in that field relative to their counterparts in more traditional competitive class settings.

# *Examples from Felder's Matter of Style*

---

- *Demonstrate the logical flow of individual course topics (sequential), but also point out connections between the current material and other relevant material in the same course, in other courses in the same discipline, in other disciplines, and in everyday experience (global). For example, before discussing cell metabolism chemistry in detail, describe energy release by glucose oxidation and relate it to energy release by nuclear fission, electron orbit decay, waterfalls, and combustion in fireplaces, power plant boilers, and automobiles. Discuss where the energy comes from and where it goes in each of these processes and how cell metabolism differs. Then consider the photosynthetic origins of the energy stored in C-H bonds and the conditions under which the earth's supply of usable energy might run out.”*

# *Blooms Taxonomy Slides*

---

- “Bloom's Taxonomy, created in 1956 by Dr. Benjamin Bloom of the University of Chicago and his group of educational psychologists, is a categorization of verbs describing cognitive skills verbs into six classes (knowledge, comprehension, application, analysis, synthesis, and evaluation). The classes are ranked from least complex (knowledge) to most complex (evaluation) in terms of the level of thinking required for students to achieve these objectives. In general, **critical thinking** skills encompass only the three most complex categories (analysis, synthesis, and evaluation).”
- (From Michael Bowen's web page, <http://207.233.105.16/curriculum/bloomtax.htm>, to be distributed in class.)

# *Testing Training*

---

- It is easy to teach to the first few levels (tests would have students recall what was taught, define terms, make simple applications).
- It is much tougher to teach the underlying skills of testing.
- Putting students through tests, exercises and exams has been eye opening in terms of how much they can learn with personal involvement and how little they get from traditional lectures.

# *Where are you training?*

---

- Different constraints for
  - Undergraduates
  - Working professionals in a hotel room
  - Working professionals in a short course onsite
  - Working professionals at a many-week extension course
  - Working professionals in a long-term course onsite
  - On-line training
  - One-on-One coaching

# *Example of a Task Breakdown*

---

- Bug Reporting
  - Reproduce the bug
  - Simplify the bug
  - Follow-up testing
  - Describe the screen
  - Describe the sequence
  - Determine customer impact
  - Determine technical impact

***What sub-tasks and skills are involved in each of these?***

# *Bug Reporting Practice Example*

---

- You will be given a copy of a computer screen.
  - Please write a description (all words, no pictures) of that screen
  - When asked, please pass your description to your partner
  - You will receive a description of a different screen from your partner.
  - Please draw the screen that your partner is describing.



# *Basic Testing Styles*

---

What tasks and skills are involved in each of these?

- Domain testing
- Function testing
- Risk-based testing
- Exploratory testing
- Stress testing
- Specification-based testing
- User testing
- Scenario testing
- Regression testing
- Random testing
- Configuration testing

*So, Let's Start the Analysis  
with Domain Testing*



# Domain Testing

- AKA partitioning, equivalence analysis, boundary analysis
- Fundamental question or goal:
  - This confronts the problem that there are too many test cases for anyone to run. This is a stratified sampling strategy that provides a rationale for selecting a few test cases from a huge population.
- General approach:
  - Divide the set of possible values of a field into subsets, pick values to represent each subset. Typical values will be at boundaries. More generally, the goal is to find a “best representative” for each subset, and to run tests with these representatives.
  - Advanced approach: combine tests of several “best representatives”. Several approaches to choosing optimal small set of combinations.
- Paradigmatic case(s)
  - Equivalence analysis of a simple numeric field.
  - Printer compatibility testing (*multidimensional variable, doesn't map to a simple numeric field, but stratified sampling is essential.*)

# *Domain Testing*

---

- Some of the Key Tasks
  - Partitioning into equivalence classes
  - Discovering best representatives of the sub-classes
  - Combining tests of several fields
  - Create boundary charts
  - Find fields / variables / environmental conditions
  - Identify constraints (non-independence) in the relationships among variables.

# *Domain Testing*

---

- Some Relevant Skills

- Identify ambiguities in specifications or descriptions of fields
- Find biggest / smallest values of a field
- Discover common and distinguishing characteristics of multi-dimensional fields, that would justify classifying some values as “equivalent” to each other and different from other groups of values.
- Standard variable combination methods, such as all-pairs or the approaches in Jorgensen and Beizer’s books

# *Domain Testing*

---

- Ideas for Exercises
  - Find the biggest / smallest accepted value in a field
  - Find the biggest / smallest value that fits in a field
  - Partition fields
  - Read specifications to determine the actual boundaries
  - Create boundary charts for several variables
  - Create standard domain testing charts for different types of variables
  - For finding variables, see notes on function testing

*Here are a few of the exercises I actually use:*

# *An Introductory Exercise*

---

The program reads three integer values from a card. The three values are interpreted as representing the lengths of the sides of a triangle. The program prints a message that states whether the triangle is scalene, isosceles, or equilateral.

- From Glen Myers, *The Art of Software Testing*

1. *Write a set of test cases that would adequately test this program.*
2. *Imagine testing this program “completely.” What proportion of complete testing does your set of test case represent?*
3. *Hand it in when you are done.*

# Myers' Answer

- Test case for a *valid* scalene triangle
- Test case for a valid equilateral triangle
- Three test cases for valid isosceles triangles ( $a=b$ ,  $b=c$ ,  $a=c$ )
- One, two or three sides has zero value (5 cases)
- One side has a negative
- Sum of two numbers equals the third (e.g. 1,2,3) is invalid b/c not a triangle (tried with 3 permutations  $a+b=c$ ,  $a+c=b$ ,  $b+c=a$ )
- Sum of two numbers is less than the third (e.g. 1,2,4) (3 permutations)
- Non integer
- Wrong number of values (too many, too few)



# Notes on the Exercise

---

- Several classes of issues were missed by most students. For example:
  - Few students checked whether they were producing valid triangles. (1,2,3) and (1,2,4) cannot be the lengths of any triangle.
    - *Knowledge of the subject matter of the program under test will enable you to create test cases that are not directly suggested by the specification. If you lack that knowledge, you will miss key tests. (This knowledge is sometimes called “domain knowledge”, not to be confused with “domain testing.”)*
  - Few students checked non-numeric values, bad delimiters, or non-integers.
  - The only boundaries tested were at MaxInt or 0.

# *Notes on the Exercise*

---

- James Bach uses a variation on this simple example to teach several lessons. He's probably willing to share his demo program. Contact him at [james@satisfice.com](mailto:james@satisfice.com).
- In Bach's example:
  - You actually enter three numbers (a test case) on a client computer
  - All the clients send back student tests to the teacher's machine, so James can look at trends
  - He can ask for specific tests, like try to find the largest number the program can accept.
  - Remarkably, students don't test numbers anywhere near as big as they could. For example, if the input field window is 17 characters wide, some students will type an input 17 characters long, but few will go wider.

# *Another Example*

- Here is the program's specification:

1. This program is designed to add two numbers, which you will enter.
2. Each number should be one or two digits.
3. The program will print the sum. Press Enter after each number.
4. To start the program, type ADDER.

*Before you start testing, do you have any questions about the spec?*

# *Let's Brainstorm Some Tests*

---

- Brainstorming Rules:
  - The goal is to get lots of ideas. You are brainstorming together to discover categories of possible tests.
  - There are more great ideas out there than you think.
  - Don't criticize others' contributions.
  - Jokes are OK, and are often valuable.
  - Work *later*, alone or in a much smaller group, to eliminate redundancy, cut bad ideas, and refine and optimize the specific tests.
  - Facilitator and recorder keep their opinions to themselves.

# *Let's Brainstorm Some Tests*

What?

---

---

---

---

---

---

---

---

---

---

---

---

Why?

---

---

---

---

---

---

---

---

---

---

---

---

# *Traditional Presentation (Myers)*

---

- One input or output field
  - The “valid” values for the field fit within one (1) easily specified range.
  - Valid values: -99 to 99
  - Invalid values
    - value <- 99
    - value > 99

# *Myers' Boundary Table*

Variable	Valid Case Equivalence Classes	Invalid Case Equivalence Classes	Boundaries and Special Cases	Notes
First number	-99 to 99	> 99 < -99	99, 100 -99, -100	
Second number	same as first	Same as first	same	
Sum	-198 to 198			Are there other sources of data for this variable? Ways to feed it bad data?

# *Notes on the Traditional Presentation*

---

- In the traditional approach, we analyzed the code from the point of view of the programmer and the specification.
- The specification provides us with only one of several classes of risk. We have to test against a broader set.
- Experts in different subject matters will see different opportunities for failure, and different classes of cases that might reveal these failures.
- Thinking outside of the box posed by the explicit design is at the essence of black box testing.



# *Expanding the Notion of Equivalence*

---

Consider these cases. Are these paired tests equivalent?

(55+56, 56+57)

(57+58, 58+59)

(59+60, 60+61)

(61+62, 62+63)

(63+64, 64+65)

(65+66, 66+67)

(67+68, 68+69)

(69+70, 70+71)

# *Equivalence Classes and Partitions are Subjective.*

---

## *Example: The Hockey Game*

My working definition of equivalence:

*Two test cases are equivalent if you expect the same result from each.*

This is fundamentally subjective. It depends on what you expect. And what you expect depends on what errors you can anticipate:

*Two test cases can only be equivalent by reference to a specifiable risk.*

Two different testers will have different theories about how programs can fail, and therefore they will come up with different classes.

# *Another Example*

---

	<u>Character</u>	<u>ASCII Code</u>
	/	47
lower bound	0	48
	1	49
	2	50
	3	51
	4	52
	5	53
	6	54
	7	55
	8	56
upper bound	9	57
	:	58
	A	65
	a	97

# *Revised Boundary Analysis Table*

Variable	Equivalence Class	Alternate Equivalence Class	Boundaries and Special Cases	Notes
<b>First number</b>	-99 to 99  digits	> 99 < -99 non-digits  expressions	99, 100 -99, -100 /, 0, 9, : leading spaces or 0s null entry	
<b>Second number</b>	same as first	same as first	same	
<b>Sum</b>	-198 to 198 -127 to 127	??? -198 to -128 128 to 198	??? 127, 128, -127, -128	Are there other sources of data for this variable? Ways to feed it bad data?

Note that we've dropped the issue of "valid" and "invalid." This lets us generalize to partitioning strategies that don't have the concept of "valid" -- for example, printer equivalence classes. (For discussion of device compatibility testing, see Kaner et al., Chapter 8.)

# *Additional Basic Exercises*

- Send students to common dialog boxes, such as these in MS Word:
  - Print dialog
  - Page setup dialog
  - Font format dialog
- For each dialog
  - Identify each field, and for each field
    - Name the type of the field (integer, rational, string, etc.)
    - List the range of entries that are “valid” for the field
    - Partition the field and identify boundary conditions
    - List the entries that are almost too extreme and too extreme for the field
    - List a small number of test cases for the field and explain why the values you have chosen are best representatives of the sets of values that they were selected from.
    - Identify any constraints imposed on this field by other fields

# *Additional Basic Exercises*

---

- Basic exercises like these can be presented with solutions. I'm working on developing a series of these. Imagine giving a student 20 examples of questions like these, with 20 solutions/comments in the back of the book.
- There might not be one true solution, but I think we can give three, one being a typical weak solution, one being an average acceptable solution, and one being excellent.
- In some cases, there are alternative good solutions.
- In any case, I think the answers should include commentary.

# *Specifications and Boundaries*

---

- In general, word puzzles are important exercises. Math students can often solve equations but have real trouble identifying the equations when they are embedded in simple sentences.

# *Specifications and Boundaries*

---

For each of the following, list

- The variable(s) of interest
- The valid and invalid classes
- The boundary value test cases.

1. FoodVan delivers groceries to customers who order food over the Net. To decide whether to buy more more vans, FV tracks the number of customers who call for a van. A clerk enters the number of calls into a database each day. Based on previous experience, the database is set to challenge (ask, “Are you sure?”) any number greater than 400 calls.

2. FoodVan schedules drivers one day in advance. To be eligible for an assignment, a driver must have special permission or she must have driven within 30 days of the shift she will be assigned to.



# *Ambiguity in Specifications*

- Most people have trouble with Question 2 on the previous slide. For example, what does “within” mean? If today is May 29, what is the longest-ago date that would qualify them to drive? What if someone has permission to drive for their first time tomorrow morning--can they be scheduled for a second shift tomorrow evening?
- For more on ambiguity analysis, see the discussion of spec-based testing, later.
- Self-study exercises can work with these, but the problem is discovering what ambiguities are troublesome to the students (and thinking about how to prompt them). I think that I would deal with these at two levels:
  - First, prompt with questions to get the student to puzzle further through the problem
  - Second, provide answers that describe the ambiguities commonly spotted by students.

# *Equivalence Classes: A Broad Concept*

---

The notion of equivalence class is much broader than numeric ranges. Here are some examples:

- Membership in a common group
  - such as employees vs. non-employees. (Note that not all classes have shared boundaries.)
- Equivalent hardware
  - such as compatible modems
- Equivalent event times
  - such as before-timeout and after
- Equivalent output events
  - perhaps any report will do to answer a simple the question: Will the program print reports?
- Equivalent operating environments
  - such as French & English versions of Windows 3.1

# *Variables Suited to Equivalence Class Analysis*

---

There are many types of variables, such as:

- input variables
- output variables
- internal variables
- hardware and system software configurations, and
- equipment states.

Any of these can be subject to equivalence class analysis.

Here are some examples:

# *Variables Suited to Equivalence Class Analysis*

- ranges of numbers
- character codes
- how many times something is done
  - (e.g. shareware limit on number of uses of a product)
  - (e.g. how many times you can do it before you run out of memory)
- how many names in a mailing list, records in a database, variables in a spreadsheet, bookmarks, abbreviations
- size of the sum of variables, or of some other computed value (think binary and think digits)

- size of a number that you enter (number of digits) or size of a character string
- size of a concatenated string
- size of a path specification
- size of a file name
- size (in characters) of a document
- size of a file (note special values such as exactly 64K, exactly 512 bytes, etc.)
- size of the document on the page (compared to page margins) (across different page margins, page sizes)

# *Variables Suited to Equivalence Class Analysis*

- size of a document on a page, in terms of memory requirements for the page. This may be in terms of resolution x page size, but is more complex if we have compression.
- equivalent output events (such as printing documents)
- amount of available memory (> 128 meg, > 640K, etc.)
- visual resolution, size of screen, number of colors
- operating system version
- variations within a group of “compatible” printers, sound cards, modems, etc.

- equivalent event times (when something happens)
- timing: how long between event A and event B (and in which order--races)
- length of time after a timeout (from JUST before to way after) -- what events are important?
- speed of data entry (time between keystrokes, menus, etc.)
- speed of input--handling of concurrent events
- number of devices connected / active

# *Variables Suited to Equivalence Class Analysis*

---

- system resources consumed / available (also, handles, stack space, etc.)
- date and time
- transitions between algorithms (optimizations) (different ways to compute a function)
- most recent event, first event
- input or output intensity (voltage)
- speed / extent of voltage transition (e.g. from very soft to very loud sound)

# *Constraints / Interactions Among Variables*

---

Rather than thinking about a single variable with a single range of values, a variable might have different ranges, such as the day of the month, in a date:

1-28

1-29

1-30

1-31

We analyze the range of dates by partitioning the month field for the date into different sets:

{February}, {April, June, September, November}

{Jan, March, May, July, August, October, December}

For testing, you want to pick one of each. There might or might not be a “boundary” on months. The boundaries on the days, are sometimes 1-28, sometimes 1-29, etc

This is nicely analyzed by Jorgensen: Software Testing--A Craftsman's Approach. It works well as a puzzle for students, alone or in groups.

# *Another example of interaction*

---

- Interaction thinking is important when we think of an output variable whose value is based on some input variables. Here's an example that gives my students headaches on tests:

I, J, and K are integers. The program calculates  $K = I * J$ . For this question, consider only cases in which you enter integer values into I and J. Do an equivalence class analysis from the point of view of **the effects of I and J (jointly) on the variable K**. Identify the boundary tests that you would run (the values you would enter into I and J) if

- I, J, K are unsigned integers
- I, J, K are signed integers



# *Interaction example*

- K can run from MinInt (smallest integer) to MaxInt.
- For any value of K, there is a set of values of I and J that will yield K
  - The set is the null set for impossible values of K
  - The set might include only two pairs, such as  $K = \text{MaxInt}$ , when MaxInt is prime (7 could be a MaxInt)
    - $(I,J) ? \{(1, \text{MaxInt}), (\text{MaxInt}, 1)\}$
  - The set might include a huge number of pairs, such as when K is 0:
    - $(I,J) ? \{(0,0), (1, 0), (2, 0), \dots, (\text{MaxInt},0), (0,1), \dots, (0, \text{MaxInt},1)\}$
- A set of pairs of values of I and J can be thought of as an equivalence set (they all yield the same value of K) and so we ask which values of K are interesting (partition number 1) and then which values of I and J would produce those K-values, and do a partitioning on the sets of (I,J) pairs to find best representatives of those.
- As practitioners, we do this type of analysis often, but many of us probably don't think very formally about it.

# *Fuzzy Boundaries*

---

- In theory, the key to partitioning is dividing the space into mutually exclusive ranges (subsets). Each subset is an equivalence class. This is nice in theory, but let's look at printers.
- Problem:
  - There are about 2000 Windows-compatible printers, plus multiple drivers for each. We can't test them all.
- But maybe we can form equivalence classes.
- *An example from two programs (desktop publishing and an address book) developed in 1991-92.*

# *Fuzzy Boundaries*

---

- Primary groups of printers at that time:
  - HP - Original
  - HP - LJ II
  - PostScript Level I
  - PostScript Level II
  - Epson 9-pin, etc.
- LaserJet II compatible printers, huge class (maybe 300 printers, depending on how we define it)
  1. Should the class include LJII, LJII+, and LIIP, LJIID-compatible subclasses?
  2. What is the best representative of the class?

# *Fuzzy Boundaries*

---

- Example: graphic complexity error handling
  - HP II original was the weak case.
- Example: special forms
  - HP II original was strong in paper-handling. We worked with printers that were weaker in paper-handling.
- Examples of additional queries for almost-equivalent printers
  - Same margins, offsets on new printer as on HP II original?
  - Same printable area?
  - Same handling of hairlines? (Postscript printers differ.)
- ***What exercises can we do to support development of this type of analysis?***

# *Test Matrices and Domain Testing*

---

- Use a test matrix to show/track a series of test cases that are essentially the same.
  - For example, for most input fields, you'll do a series of the same tests, checking how the field handles boundaries, unexpected characters, function keys, etc.
  - As another example, for most files, you'll run essentially the same tests on file handling.
- The matrix is a concise way of showing the repeating tests.
  - Put the objects that you're testing on the rows.
  - Show the tests on the columns.
  - Check off the tests that you actually completed in the cells.

# *Typical Uses of Test Matrices*

---

- You can often re-use a matrix like this across products and projects.
- You can create matrices like this for a wide range of problems. Whenever you can specify multiple tests to be done on one class of object, and you expect to test several such objects, you can put the multiple tests on the matrix.
- Mark a cell green if you ran the test and the program passed it.
- Mark the cell red if the program failed and write the bug number of the bug report for this bug.
- Write (in the cell) the automation number or identifier or file name if the test case has been automated.

# Reusable Test Matrix

Numeric (Integer) Input Field												
	Nothing	LB of value	UB of value	LB of value - 1	UB of value + 1	0	Negative	LB number of digits or chars	UB number of digits or chars	Empty field (clear the default value)	Outside of UB number of digits or chars	Non-digits

This includes only the first few columns of a matrix that I've used commercially, but it gets across the idea.

# *Homework: File Name Matrix*

---

- Thursday, we'll illustrate the process of creating test matrices by brainstorming a specific one.
  - File name field
  - Windows 95 / 98 / NT / 2000
  - Treat the path as a separate issue
- Please spend 15 minutes at home writing a list of file name tests. Bring your notes with you on Thursday.

-----



# *Homework: File Name Matrix*

---

You could do this for almost any type of variable. For example, imagine listing all of the hardware (including connection, power, etc.) error conditions that could cause failure of a file save operation.

At this point, though, it is interesting to look at variables that have values that can be treated as equivalent.

Simpler examples are:

- string fields, numeric (rational numbers), percentages
- date fields, time fields

# *Matrix Construction Brainstorm*

---

- Take up the assignment with a brainstorming session.
- Expect to spend 4 – 15 hours if the students are prepared and 2 – 3 hours if they are not.
- Brainstorming Rules:
  - Don't criticize others' contributions
  - Jokes are OK, and are often valuable
  - Goal is to get lots of ideas, filter later.
  - Recorder and facilitator keep their opinions to themselves.

# *Matrix Construction Brainstorm*

---

- Facilitating and Recording Rules:
  - Exercise patience: Goal is to get lots of ideas.
  - Encourage non-speakers to speak.
  - Use multiple colors when recording
  - Echo the speaker's words.
  - Record the speaker's words
  - The rule of three 10's. Silence is OK.
  - Switch levels of analysis.
  - Some references:
    - S. Kaner, Lind, Toldi, Fisk & Berger, *Facilitator's Guide to Participatory Decision-Making*
    - Freedman & Weinberg, *Handbook of Walkthroughs, Inspections & Technical Reviews*
    - Doyle & Straus, *How to Make Meetings Work*.

# Useful Readings

---

- *The Impossibility of Complete Testing* at [www.kaner.com/imposs.htm](http://www.kaner.com/imposs.htm)
- Clarke, Hassell, & Richardson: A Close Look at Domain Testing, *IEEE Transactions on Software Engineering*, Vol. SE-8, No. 4, July 1982. A clear presentation of the formal assumptions.
- Ostrand & Balcer: The category-partition method for specifying and generating functional tests. *Communications of the ACM*, Vol. 31 (6), pages 676-686, June 1988. An excellent description of the method, laying it out for the reader step by step.

# *Analysis of Function Testing*

---

(Let's work on this one together)

# *Function Testing*

---

- Tag line: “Black box unit testing.”
- Fundamental question or goal
  - Test each function thoroughly, one at a time.
- Paradigmatic case(s)
  - Spreadsheet, test each item in isolation.
  - Database, test each report in isolation
- Strengths
  - Thorough analysis of each item tested
- Blind spots
  - Misses interactions, misses exploration of the benefits offered by the program.

# *Function Testing: Tasks*

---

# *Function Testing: Tasks*

---



# *Function Testing: Tasks*

---

# *Function Testing: Skills*

---

# *Function Testing: Skills*

---

# *Function Testing: Skills*

---

# *Function Testing: Exercises*

---

# *Function Testing: Exercises*

---

# *Function Testing: Exercises*

---

# *Function Testing: Exercises*

---



# *Some Function Testing Tasks*

---

- Identify the program's features / commands
  - From specifications or the draft user manual
  - From walking through the user interface
  - From trying commands at the command line
  - From searching the program or resource files for command names
- Identify variables used by the functions and test their boundaries.
- Identify environmental variables that may constrain the function under test.
- Use each function in a mainstream way (positive testing) and push it in as many ways as possible, as hard as possible.

# *Risk-Based Testing and Risk-Based Test Management*



# *Risk-Based Testing*

---

- Karl Popper, in his famous essay *Conjectures and Refutations*, lays out the proposition that a scientific theory gains credibility by being subjected to (and passing) harsh tests that are intended to refute the theory.
- We can gain confidence in a program by testing it harshly (if it passes the tests).
- Subjecting a program to easy tests doesn't tell us much about what will happen to the program in the field.
- ***In risk-based testing, we create harsh tests for vulnerable areas of the program.***

# *Risk-Based Testing*

---

- Two key dimensions:
  - **Find errors** (risk-based approach to the technical tasks of testing)
  - **Manage the process of finding errors** (risk-based test management)
- Let's start with risk based testing and proceed later to risk based test management.

# *Risk-Based Testing: Definitions*

---

## Hazard:

A dangerous condition (something that could trigger an accident)

## Risk:

Possibility of suffering loss or harm (probability of an accident caused by a given hazard).

## Accident:

A hazard is encountered, resulting in loss or harm.

# *Risks: Where to look for errors*

## **Quality Categories:**

- Capability
- Reliability
- Usability
- Performance
- Installability
- Compatibility
- Supportability
- Testability
- Efficiency
- Localizability
- Portability
- Maintainability

*Each quality category is a risk category, as in: “the risk of unreliability.”*

- Derived from James Bach’s Satisfice Model

# *Risks: Where to look for errors*

---

- **New things:** newer features may fail.
- **New technology:** new concepts lead to new mistakes.
- **Learning Curve:** mistakes due to ignorance.
- **Changed things:** changes may break old code.
- **Late change:** rushed decisions, rushed or demoralized staff lead to mistakes.
- **Rushed work:** some tasks or projects are chronically underfunded and all aspects of work quality suffer.
- **Tired programmers:** long overtime over several weeks or months yields inefficiencies and errors

- Adapted from James Bach's lecture notes

# *Risks: Where to look for errors*

---

- **Other staff issues:** alcoholic, mother died, two programmers who won't talk to each other (neither will their code)...
- **Just slipping it in:** pet feature not on plan may interact badly with other code.
- **N.I.H.:** external components can cause problems.
- **N.I.B.:** (not in budget) Unbudgeted tasks may be done shoddily.
- **Ambiguity:** ambiguous descriptions (in specs or other docs) can lead to incorrect or conflicting implementations.

- Adapted from James Bach's lecture notes



# *Risks: Where to look for errors*

---

- **Conflicting requirements:** ambiguity often hides conflict, result is loss of value for some person.
- **Unknown requirements:** requirements surface throughout development. Failure to meet a legitimate requirement is a failure of quality for that stakeholder.
- **Evolving requirements:** people realize what they want as the product develops. Adhering to a start-of-the-project requirements list may meet contract but fail product. (check out <http://www.agilealliance.org/>)
- **Complexity:** complex code may be buggy.
- **Bugginess:** features with many known bugs may also have many unknown bugs.
  - Adapted from James Bach's lecture notes

# *Risks: Where to look for errors*

---

- **Dependencies:** failures may trigger other failures.
- **Untestability:** risk of slow, inefficient testing.
- **Little unit testing:** programmers find and fix most of their own bugs. Shortcutting here is a risk.
- **Little system testing so far:** untested software may fail.
- **Previous reliance on narrow testing strategies:** (e.g. regression, function tests), can yield a backlog of errors surviving across versions.
- **Weak testing tools:** if tools don't exist to help identify / isolate a class of error (e.g. wild pointers), the error is more likely to survive to testing and beyond.

- Adapted from James Bach's lecture notes

# *Risks: Where to look for errors*

---

- **Unfixability:** risk of not being able to fix a bug.
- **Language-typical errors:** such as wild pointers in C. See
  - Bruce Webster, *Pitfalls of Object-Oriented Development*
  - Michael Daconta et al. *Java Pitfalls*
- **Criticality:** severity of failure of very important features.
- **Popularity:** likelihood or consequence if much used features fail.
- **Market:** severity of failure of key differentiating features.
- **Bad publicity:** a bug may appear in PC Week.
- **Liability:** being sued.

- Adapted from James Bach's lecture notes

# *Bug Patterns as a Source of Risk*

---

- *Testing Computer Software* lays out a set of 480 common defects. You can use these or develop your own list.
  - *Find a defect in the list*
  - *Ask whether the software under test could have this defect*
  - *If it is theoretically possible that the program could have the defect, ask how you could find the bug if it was there.*
  - *Ask how plausible it is that this bug could be in the program and how serious the failure would be if it was there.*
  - *If appropriate, design a test or series of tests for bugs of this type.*

# *Build Your Own Model of Bug Patterns*

---

Too many people start and end with the TCS bug list. It is outdated. It was outdated the day it was published. And it doesn't cover the issues in *your* system. Building a bug list is an ongoing process that constantly pays for itself. Here's an example from Hung Nguyen:

- This problem came up in a client/server system. The system sends the client a list of names, to allow verification that a name the client enters is not new.
- Client 1 and 2 both want to enter a name and client 1 and 2 both use the same new name. Both instances of the name are new relative to their local compare list and therefore, they are accepted, and we now have two instances of the same name.
- As we see these, we develop a library of issues. The discovery method is exploratory, requires sophistication with the underlying technology.
- Capture winning themes for testing in charts or in scripts-on-their-way to being automated.

# *Building Bug Patterns*

---

- There are plenty of sources to check for common failures in the common platforms
  - [www.bugnet.com](http://www.bugnet.com)
  - [www.cnet.com](http://www.cnet.com)
  - links from [www.winfiles.com](http://www.winfiles.com)
  - various mailing lists

# *Risk-Based Testing*

---

- Tasks
  - Identify risk factors (hazards: ways in which the program could go wrong)
  - For each risk factor, create tests that have power against it.
  - Assess coverage of the testing effort program, given a set of risk-based tests. Find holes in the testing effort.
  - Build lists of bug histories, configuration problems, tech support requests and obvious customer confusions.
  - Evaluate a series of tests to determine what risk they are testing for and whether more powerful variants can be created.

# *Risk-Based Testing*

---

- Skills

- 

- 

- 

-



# *Risk-Based Testing*

---

- Exercises
  - Given a list of ways that the program *could* fail, for each case:
    - Describe two ways to test for that possible failure
    - Explain how to make your tests more powerful against that type of possible failure
    - Explain why your test is powerful against that hazard.
  - Given a list of test cases
    - Identify a hazard that the test case might have power against
    - Explain why this test is powerful against that hazard.

# *Risk-Based Testing Exercises*

---

- Take a function in the software under test. Make a hypothetical change to the product. Do a risk analysis:
  - For N (10 to 30) minutes, the students privately list risk factors and associated tests (ask for 2 tests per factor)
  - Break
  - Class brainstorm onto flipcharts--list risk factors.
  - Pairs project--fill out the charts:
    - Students work in pairs to complete their chart
    - Each chart has one risk factor as a title
    - List tests derived from risk (factor) on body of page
    - When a pair is done with its page, they move around the room, adding notes to other pairs' pages.
  - Paired testing, one chart per pair, test SUT per that risk factor.

# *Risk-Based Testing Exercises*

---

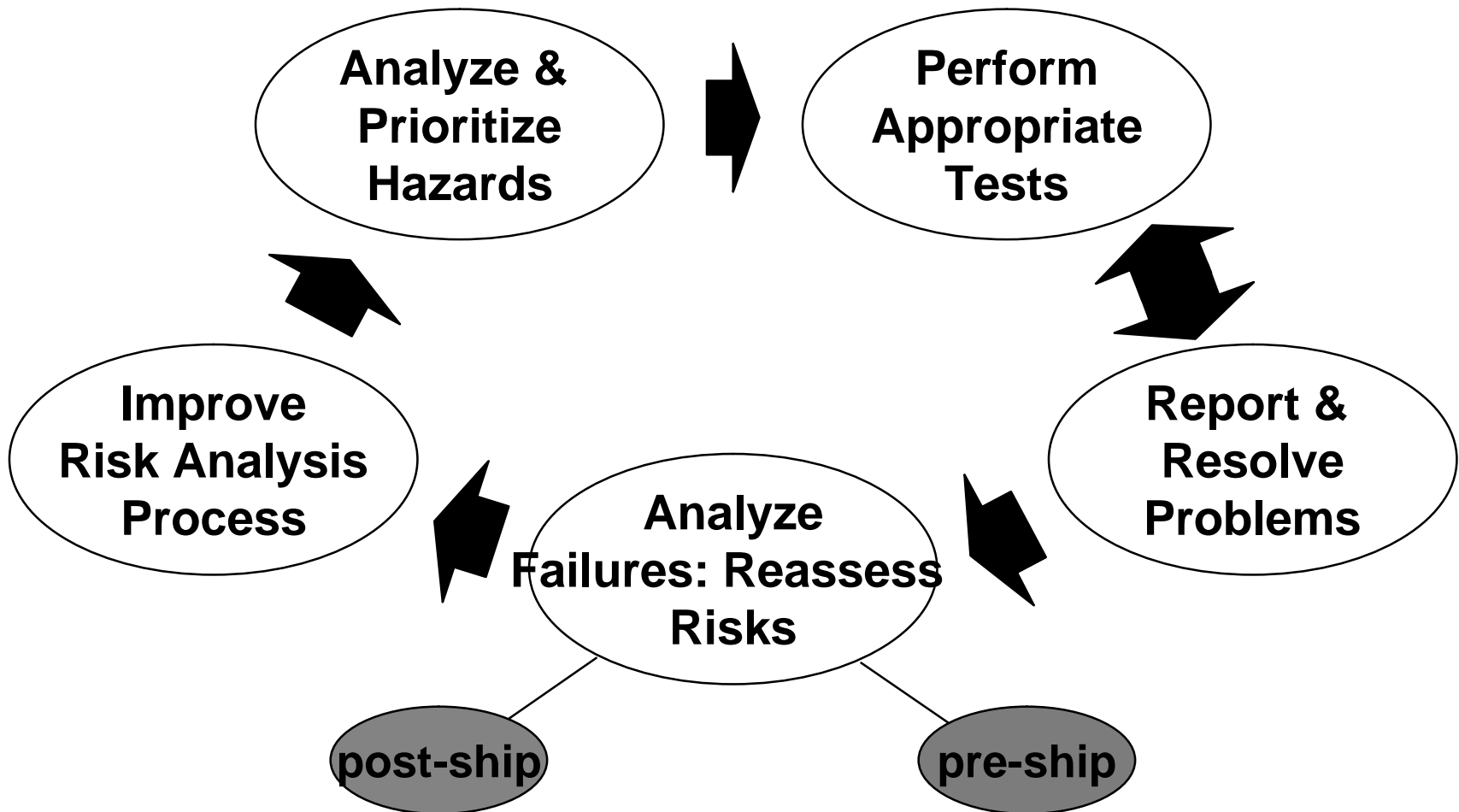
- Collect or create some test cases for the software under test. Make a variety of tests:
  - Mainstream tests that use the program in “safe” ways
  - Boundary tests
  - Scenario tests
  - Wandering walks through the program
  - Etc.
  - If possible, use tests the students have suggested previously.
- For each test, ask:
  - How will this test find a defect?
  - What kind of defect did the test author probably have in mind?
  - What power does this test have against that kind of defect? Is there a more powerful test? A more powerful suite of tests?

# *Risk-Based Test Management*

---

- Project risk management involves
  - Identification of the different risks to the project (issues that might cause the project to fail or to fall behind schedule or to cost too much or to dissatisfy customers or other stakeholders)
  - Analysis of the potential costs associated with each risk
  - Development of plans and actions to reduce the likelihood of the risk or the magnitude of the harm
  - Continuous assessment or monitoring of the risks (or the actions taken to manage them)
- Useful material available free at <http://seir.sei.cmu.edu>
- <http://www.coyotevalley.com> (Brian Lawrence)
- Good paper by Stale Amland, *Risk Based Testing and Metrics*, 16th International Conference on Testing Computer Software, 1999.

# *Risk-Driven Testing Cycle*





## Categories of Risk Sources

- Mission and goals
- Decision drivers
- Organization management
- Customer / end user
- Budget / cost
- Schedule
- Project characteristics
- Development process
- Development environment
- Personnel
- Operational environment
- New technology



## Project Consequences

- Cost overruns
- Schedule slips
- Inadequate functionality
- Canceled projects
- Sudden personnel changes
- Customer dissatisfaction
- Loss of company image
- Demoralized staff
- Poor product performance
- Legal proceedings

# *Risk-Based Test Management*

---

- Tasks
  - List all areas of the program that could require testing
  - On a scale of 1-5, assign a probability-of-failure estimate to each
  - On a scale of 1-5, assign a severity-of-failure estimate to each
  - For each area, identify the specific ways that the program might fail and assign probability-of-failure and severity-of-failure estimates for those
  - Prioritize based on estimated risk
  - Develop a stop-loss strategy for testing untested or lightly-tested areas, to check whether there is easy-to-find evidence that the areas estimated as low risk are not actually low risk.

# *Risk-Based Test Management*

---

- Skills
  - 
  - 
  -
- Exercises
  - 
  -



# *Specification-Driven Testing*



# *Specification-Driven Testing*

---

- Tag line:
  - “Verify every claim.”
- Fundamental question or goal
  - Check the product’s conformance with every statement in every spec, requirements document, etc.
- Paradigmatic case(s)
  - Traceability matrix, tracks test cases associated with each specification item.
  - User documentation testing

# *Specification-Driven Testing*

---

- **Strengths**
  - Critical defense against warranty claims, fraud charges, loss of credibility with customers.
  - Effective for managing scope / expectations of regulatory-driven testing
  - Reduces support costs / customer complaints by ensuring that no false or misleading representations are made to customers.
- **Blind spots**
  - Any issues not in the specs or treated badly in the specs /documentation.

# *Specification*

---

- Tasks
  - review specifications for
    - Ambiguity
    - Adequacy (it covers the issues)
    - Correctness (it describes the program)
    - Content (not a source of design errors)
    - Testability support
  - Create traceability matrices
  - Document management (spec versions, file comparison utilities for comparing two spec versions, etc.)
  - Participate in review meetings

# Specification

---

- Skills
  - Understand the level of generality called for when testing a spec item. For example, imagine a field X:
    - We could test a single use of X
    - Or we could partition possible values of X and test boundary values
    - Or we could test X in various scenarios
    - Which is the right one?
  - Ambiguity analysis
    - Richard Bender teaches this well. If you can't take his course, you can find notes based on his work in Rodney Wilson's **Software RX: Secrets of Engineering Quality Software**

# Specification

---

- Skills
  - Ambiguity analysis
    - Another book provides an excellent introduction to the ways in which statements can be ambiguous and provides lots of sample exercises: Cecile Cyrul Spector, *Saying One Thing, Meaning Another : Activities for Clarifying Ambiguous Language*

# *Example: Requirements Questions*

---

- Important to trace from requirements to implications
- Exercise
  - Give a list of questions
    - Examples are the test documentation requirements questions (next slides) and the automation maintainability questions at ([www.kaner.com/pdfs/shelfwar.pdf](http://www.kaner.com/pdfs/shelfwar.pdf))
  - For each question
    - Ask the students to name at least two decisions that they would make on the basis of the answer to the question
  - Make this a small group exercise, splitting up the questions, groups fill flipcharts, then bring back to full class.

# *Test Docs Requirements Questions*

---

- Is test documentation a product or tool?
- Is software quality driven by legal issues or by market forces?
- How quickly is the design changing?
- How quickly does the specification change to reflect design change?
- Is testing approach oriented toward proving conformance to specs or nonconformance with customer expectations?
- Does your testing style rely more on already-defined tests or on exploration?
- Should test docs focus on what to test (objectives) or on how to test for it (procedures)?
- Should the docs ever control the testing project?



# *Test Docs Requirements Questions*

---

- If the docs control parts of the testing project, should that control come early or late in the project?
- Who are the primary readers of these test documents and how important are they?
- How much traceability do you need? What docs are you tracing back to and who controls them?
- To what extent should test docs support tracking and reporting of project status and testing progress?
- How well should docs support delegation of work to new testers?
- What are your assumptions about the skills and knowledge of new testers?
- Is test doc set a process model, a product model, or a defect finder?

# *Test Docs Requirements Questions*

---

- A test suite should provide prevention, detection, and prediction. Which is the most important for this project?
- How maintainable are the test docs (and their test cases)? And, how well do they ensure that test changes will follow code changes?
- Will the test docs help us identify (and revise/restructure in face of) a permanent shift in the risk profile of the program?
- Are (should) docs (be) automatically created as a byproduct of the test automation code?

# *Reviewing a Specification for Completeness*

---

- Reading a spec linearly is not a particularly effective way to read the document. It's too easy to overlook key missing issues.
- We don't have time to walk through this method in this class, but the general approach that I use is based on James Bach's "Satisfice Heuristic Test Strategy Model" at <http://www.satisfice.com/tools/satisfice-tsm-4p.pdf>.
  - You can assume (not always correctly, but usually) that every sentence in the spec is meant to convey information.
  - The information will probably be about
    - the project and how it is structured, funded or timed, or
    - about the product (what it is and how it works) or
    - about the quality criteria that you should evaluate the product against.

# *Reviewing a Specification for Completeness*

---

- Spec Review using the Satisfice Model, continued
  - The Satisfice Model lists several examples of project factors, product elements and quality criteria.
  - For a given sentence in the spec, ask whether it is telling you project, product, or quality-related information. Then ask whether you are getting the full story. As you do the review, you'll discover that project factors are missing (such as deadline dates, location of key files, etc.) or that you don't understand / recognize certain product elements, or that you don't know how to tell whether the program will satisfy a given quality criterion.
  - Write down these issues. These are primary material for asking the programmer or product manager about the spec.

# *Combination Testing*

---

These notes are reminders,  
see demonstration in class.

# Combination Testing

- Imagine a program with 3 variables, V1 has 3 possible values, V2 has 2 possible values and V3 has 3 possible values.
- If V1 and V2 and V3 are independent, the number of possible combinations is 12 ( $3 \times 2 \times 2$ )
- Building a simple combination table:
  - Label the columns with the variable names, listing variables in descending order (of number of possible values)
  - Each column (before the last) will have repetition. Suppose that A, B, and C are in column K of N columns. To determine how many times (rows in which) to repeat A before creating a row for B, multiply the number of variable values in columns K+1, K+2, . . . , N.

# Combination Testing

- Building an all pairs combination table:
  - Label the columns with the variable names, listing variables in descending order (of number of possible values)
  - If the variable in column 1 has  $V1$  possible values and the variable in column 2 has  $V2$  possible values, then there will be at least  $V1 \times V2$  rows (draw the table this way but leave a blank row or two between repetition groups in column 1).
  - Fill in the table, one column at a time. The first column repeats each of its elements  $V2$  times, skips a line, and then starts the repetition of the next element. For example, if variable 1's possible values are A, B, C and  $V2$  is 2, then column 1 would contain A, A, blank row, B, B, blank row, C, C, blank row.

# Combination Testing

- Building an all pairs combination table:
  - In the second column, list all the values of the variable, skip the line, list the values, etc. For example, if variable 2's possible values are X,Y, then the table looks like this so far

A	X
A	Y
B	X
B	Y
C	X
C	Y



# Combination Testing

- Building an all pairs combination table:
  - Each section of the third column (think of AA as defining a section, BB as defining another, etc.) will have to contain every value of variable 3. Order the values such that the variables also make all pairs with variable 2.
  - Suppose variable 2 can be 1,0
  - The third section can be filled in either way, and you might highlight it so that you can reverse it later. The decision (say 1,0) is arbitrary.

A	X	1
A	Y	0
B	X	0
B	Y	1
C	X	
C	Y	

# Combination Testing

- The 4th column went in easily (note that we started by making sure we hit all pairs of values of column 4 and column 2, then all pairs of column 4 and column 3).
- Watch this first attempt on column 5. We achieve all pairs of GH with columns 1, 2, and 3, but miss it for column 4.
- The most recent arbitrary choice was HG in the 2nd section. (Once that was determined, we picked HG for the third in order to pair H with a 1 in the third column.)
- So we will erase the last choice and try again:

A	X	1	E	G
A	Y	0	F	H
B	X	0	F	H
B	Y	1	E	G
C	X	1	F	H
C	Y	0	E	G

# Combination Testing

- We flipped the last arbitrary choice (column 5, section 2, to GH from HG) and erased section 3. We then fill in section 3 by checking for missing pairs. GH, GH gives us two XG, XG pairs, so we flip to HP for the third section and have a column 2 X with a column 5 H and a column 2 Y with a column 5 G as needed to obtain all pairs.

A	X	1	E	G
A	Y	0	F	H
B	X	0	F	G
B	Y	1	E	H
C	X	1	F	H
C	Y	0	E	G

# Combination Testing

- But when we add the next column, we see that we just can't achieve all pairs with 6 values. The first one works up to column 4 but then fails to get pair EJ or FI. The next fails on GJ, HI

A	X	1	E	G	I
A	Y	0	F	H	J
B	X	0	F	G	J
B	Y	1	E	H	I
C	X	1	F	H	J
C	Y	0	E	G	I

A	X	1	E	G	I
A	Y	0	F	H	J
B	X	0	F	G	I
B	Y	1	E	H	J
C	X	1	F	H	J
C	Y	0	E	G	I

# Combination Testing

- When all else fails, add rows. We need one for GJ and one for HI, so add two rows. In general, we would need as many rows as the last column has values.
- The other values in the two rows are arbitrary, leave them blank and fill them in as needed when you add new columns. At the very end, fill the remaining blank ones with arbitrary values

A	X	1	E	G	I
A	Y	0	F	H	J
				G	J
B	X	0	F	G	I
B	Y	1	E	H	J
				H	I
C	X	1	F	H	J
C	Y	0	E	G	I

# Combination Testing

- If a variable is continuous but maps to a number line, partition and use boundaries as the distinct values under test. If all variables are continuous, we end up with all pairs of all boundary tests of all variables. We don't achieve all triples, all quadruples, etc.
- If some combinations are of independent interest, add them to the list of n-tuples to test.
  - With the six columns of the example, we reduced 96 tests to 8. Give a few back (make it 12 or 15 tests) and you still get enormous reduction.
  - Examples of “independent interest” are known (from tech support) high risk cases, cases that jointly stress memory, configuration combinations (Var 1 is operating systems, Var 2 is printers, etc.) that are prevalent in the market, etc.

# *Many Additional Key Testing Tasks*

Here are just a few more to stimulate your thinking:

Questioning	Creating models
Keeping lab notes	Estimating tasks
Designing tests that make failures obvious	Facilitating and recording meetings (such as technical reviews)
Status reporting	Regression automation
Bug advocacy	Editing
Usability testing	Systematic exploration
Troubleshooting	Programming
Predicting errors	Training staff
Packaging info for tech support	Software architecture (just try doing a big test automation without doing good architecture...) (then try again)

# *Many Additional Key Skills or Attributes*

---

- Here are some examples of skills or attributes that I look for when recruiting testers (obviously, I can't have all of them in any one person)
- We can improve some of these.
- In other cases, we might not know how to train for improvement but we might discover that a specific tester has other strengths (rather than this one)



# *Some characteristics of great testers*

Alert	Attentive to detail	Analytical problem solver
Architect	Arrogance (usually, less is better)	Artistic (knowledgeably critique esthetic issues)
Assertive	Auditor	Author
Commitment (keep promises, stick around)	Commitment to a task (do what it takes)	Commitment to quality
Copes with difficult circumstances	Courageous	Creative
Credible	Curious (inquisitive)	Customer focused
Decision maker (good judgment)	Decisive	Not very defensive (able to take criticism)
Diplomatic	Editor (criticize / improve printed materials)	Effective with junior testers
Effective with senior testers	Effective with test managers	Effective with programmers

# *Some characteristics of great testers*

Effective with non-testing managers	Empathetic	Empirical frame of reference
Empowering	Energizing	Fast abstraction skills
Financially aware and sophisticated	Finds bugs (intuitive tester)	Flexible
Goal setting	Glue (promotes group cohesiveness)	Humility
Integrity (honest; keeps commitments)	Interpersonally perceptive	Interviewer
Investigative reader	Leadership	Long term thinker
Meeting manager	Mentor	Multi-tasking
Organizer and planner	Persuasive	Politically perceptive
Policy and procedure developer	Pragmatic	Programmer

# *Some characteristics of great testers*

Protective (stands behind his staff)	Punctual	Scholarly (collects information, can back up or evaluate arguments)
Sense of humor	Spoken communication	Strength of character
Subject matter expert	Substance abuser (not)	Team builder
Tolerant of ambiguity	Tolerant of different development approaches	UI design
Versatile (many abilities)	Warm (makes the human environment more pleasant)	Written communication
Zealot (Rarely desirable in large quantities.)	Catalyst	

# *Closing Notes*

---

- Given a group of motivated, bright people who have appropriate background knowledge and belong to your testing group or your testing class:
  - Only some testers will learn well in lectures
  - Only some testers will learn well by being thrown into a project and being told to figure it out
  - Only some testers will learn well from books
  - Only some testers will learn well or work well on their own and some will only learn well or work well on their own
- Build training strategies to achieve the learning you want to convey, accommodate multiple learning styles, and push for mastery, not memory.

# *Appendix: Sample Exam Questions*

---

From Florida Tech Testing Class  
Spring 2001

# *Sample Exam Questions*

---

- The following are examples of the review questions that my first semester testing students work from. They get the questions during the term and work together to create answers.
- The tests and exams are closed book, and present students with a subset of these questions.
- I teach the course using a reference product. Last year it was TI's interactive calculator. We work many of our examples and tests around that one example product.

# Reference Information

## TI InterActive! toolbar

The buttons on the TI InterActive! toolbar give you quick access to the program's main features.



Performs calculations and defines variables and functions.



Performs typical spreadsheet operations.



Specifies the mode settings for each object.



Performs statistics regression calculations on lists of data.



Graphs functions and plots statistical data.



Transfers data to/from a connected calculator.



Generates a table of values for defined functions.



Captures the screen of a connected calculator (TI-83 or TI-83 Plus).



Enters and/or edits lists of data.



Browses the Web and extracts data directly from Web pages.



Enters and/or edits matrices.



Sends e-mail attachments of your current document.

# Definitions

1. Domain testing (as defined by Clarke et al.)
2. Equivalence class
3. Boundary condition
4. Best representative
5. Fault vs. failure vs. defect
6. Function testing
7. Regression testing
8. Specification-based testing
9. Finite state machine
10. Stochastic testing
11. Oracle
12. Risk factors (as described by Amland)
13. Exploratory testing
14. Dumb monkey
15. State
16. Line (or statement) coverage
17. Boundary chart
18. Software quality
19. Black box testing
20. Glass box / white box testing
21. Risk-based testing
22. All-pairs combination testing
23. Combination testing



# Short Answers

---

1. Describe the characteristics of a good scenario test.
2. Describe two difficulties and two advantages of state-machine-model based testing.
3. Give two examples of defects you are likely to discover and five examples of defects that you are unlikely to discover if you focus your testing on line and branch coverage.
4. Give three different definitions of “software error.” Which do you prefer? Why?
5. Ostrand & Balcer described the category-partition method for designing tests. Their first three steps are:
  - (a) Analyze
  - (b) Partition, and
  - (c) Determine constraints

Describe and explain these steps.

# Long Answers

---

1. Imagine testing a date field. The field is of the form MM/DD/YYYY (two digit month, two digit day, 4 digit year). Do an equivalence class analysis and identify the boundary tests that you would run in order to test the field.
- 2 I, J, and K are integers. The program calculates  $K = I * J$ . For this question, consider only cases in which you enter integer values into I and J. Do an equivalence class analysis from the point of view of the effects of I and J (jointly) on the variable K. Identify the boundary tests that you would run (the values you would enter into I and J) if
  - I, J, K are unsigned integers
  - I, J, K are signed integers

# Long Answers

---

3. (The following statement about TI is not true, but pretend it is for exam purposes.) TI has just announced that they will include database support in Release 2.0 of the Interactive product, which they will ship in November, 2001. They announce that the first application that uses the database will be a feature that allows teachers to store the students' grades, calculate percentages, final grade percentages, class averages, etc., and print associated reports.
- List and briefly explain 5 risk factors that you would expect to find associated with the database part of the 2.0 project. (Refer to Amland's paper for discussion of risk factors.)
  - For each risk factor, predict 2 defects that could arise in the database part of the 2.0 project. By “predict”, I mean name and describe the potential defect, and explain why that particular risk factor makes this defect more likely.

# Long Answers

---

4. Ostrand & Balcer described the category partition method for designing tests. Their first three steps are:
1. Analyze
  2. Partition, and
  3. Determine constraints

Apply their method as follows:

I, J, and K are integers. For each function,  $F$ ,  $K = F(I, J)$ . The program is a simple calculator and handles the usual, basic arithmetic functions.

Use Ostrand & Balcer's category-partition method to design a series of tests for four of the basic arithmetic functions.

# *Long Answers*

---

5. The Spring and Fall changes between Standard and Daylight Savings time creates an interesting problem for telephone bills. Create a table that shows risks, equivalence classes, boundary cases, and expected results for a long distance telephone service that bills calls at a flat rate of \$0.05 per minute. Assume that the chargeable time of a call begins when the called party answers, and ends when the calling party disconnects.

## **6. Describe a traceability matrix.**

- How would you build a traceability matrix for the TI Interactive product?
- What is the traceability matrix used for?
- What are the advantages and risks associated with driving your testing using a traceability matrix?
- Give examples of advantages and risks that you would expect to deal with if you used a traceability matrix for the TI Interactive product. Answer this in terms of two of the main features of the TI Interactive product. You can choose which two features.

# *Long Answers*

---

7. Define a scenario test and describe the characteristics of a good scenario test. Imagine developing a set of scenario tests for the TI interactive product, that involved matrix handling.
- What research would you do in order to develop a series of scenario tests?
  - Describe two scenario tests that you would use and
  - Explain why each is a good example of a scenario test.

# *Long Answers*

---

8. Define a scenario test and describe the characteristics of a good scenario test. Imagine developing a set of scenario tests for the TI interactive product, that were focused on configuration-related issues (compatibility of hardware/software with the product).
- What research would you do in order to develop a series of scenario tests?
  - Describe two scenario tests that you would use and
  - Explain why each is a good example of a scenario test.



# Long Answers

---

9. Define a scenario test and describe the characteristics of a good scenario test. Imagine developing a set of scenario tests for a C compiler. What research would you do in order to develop a series of scenario tests? (*NOTE: I am not asking for tests of the user interface to the compiler. I'm asking for tests of what we typically think of as compiler functionality.*) Describe two scenario tests that you would use and explain why each is a good test.

# *Long Answers*

---

10. What is regression testing? What are some benefits and some risks associated with regression testing? Under what circumstances would you use regression tests?

# Long Answers

11. Consider the tools in TI interactive. For each tool, identify the two testing strategies that you would consider most useful, and explain why. (Across the different tools, please describe a variety of *different* testing strategies.)

## TI InterActive! toolbar

The buttons on the TI InterActive! toolbar give you quick access to the program's main features.



Performs calculations and defines variables and functions.



Performs typical spreadsheet operations.



Specifies the mode settings for each object.



Performs statistics regression calculations on lists of data.



Graphs functions and plots statistical data.



Transfers data to/from a connected calculator.



Generates a table of values for defined functions.



Captures the screen of a connected calculator (TI-83 or TI-83 Plus).



Enters and/or edits lists of data.



Browses the Web and extracts data directly from Web pages.



Enters and/or edits matrices.



Sends e-mail attachments of your current document.

# Long Answers

---

12. Suppose that you had access to the TI source code and the time / opportunity to revise it. Suppose that you decided to do a *diagnostics-based* high volume automated test strategy to test TI Interactive's treatment of lists.

- What diagnostics would you add to the code, and why?
- Describe 3 potential defects, *defects that you could reasonably imagine would be in the list handling software*, that would be easier to find using a diagnostics-based strategy than by using a lower-volume strategy such as exploratory testing, spec-based testing, or domain testing.

# Long Answers

---

13. You are using a high volume random testing strategy for the TI Interactive calculator and will evaluate results by using an oracle.
- For the functional area, “Generates a table of values for defined functions”, how would you create an oracle (or group of oracles)? What would the oracle(s) do?
  - For the functional area, “Graphs functions and plots statistical data,” how would you create an oracle (or group of oracles)? What would the oracle(s) do?
  - Which oracle would be more challenging to create or use, and why?

# Long Answers

---

14. Consider testing the TI Interactive function, “Transfers data to or from a connected calculator.”

- How would you develop a list of risks for this capability? (If you are talking to people, who would you ask and what would you ask them?) (If you are consulting books or records or databases, what are you consulting and what information are you looking for in it?)
- Why is this a good approach for building a list of risks?
- List 10 risks associated with this function.
- For each risk, briefly (very briefly) describe a test that could determine whether there was an actual defect.

# *Long Answers*

---

15. Why is it important to design maintainability into automated regression tests? Describe some design (of the test code) choices that will usually make automated regression tests more maintainable.

# *Long Answers*

---

16. Suppose that you find a reproducible failure that doesn't look very serious.
- Describe three tactics for testing whether the defect is more serious than it first appeared.
  - As a particular example, suppose that the display got a little corrupted (stray dots on the screen, an unexpected font change, that kind of stuff) in the TI program when you entered data into a matrix. Describe some follow-up tests that you would run.



# Long Answers

---

17. The *oracle problem* is the problem of finding a method that lets you determine whether a program passed or failed a test.

Suppose that you were doing automated testing of the TI Interactive function, “Graphs functions and plots statistical data.”

Describe three different oracles that you could use or create to determine whether this feature was working. For each of these oracles,

- identify a bug that would be easy to detect using the oracle and
- identify another bug that your oracle would be more likely to miss.

# Long Answers

---

18. Imagine that you were testing the feature, “Enters and/or edits matrices.”

Describe four examples of each of the following types of attacks that you could make on this feature, and for each one, explain why your example is a good attack of that kind.

- Input constraint attacks
- Output constraint attacks
- Storage constraint attacks
- Computation constraint attacks.

(Notes for you while you study. Refer to Jorgensen / Whittaker’s paper on how to break software. Don’t give me two examples of what is essentially the same attack. In the exam, I will not ask for all 16 examples, but I might ask for 4 examples of one type or two examples of two types, etc.)

# *Long Answers*

---

- 19 Imagine that you were testing the feature, “Stat calculation tool.”
- Describe a scenario test that you would use to test this feature.
  - Explain why this is a particularly good scenario test.

# *Long Answers*

---

- 20 Imagine that you were testing the feature, “Stat calculation tool.”
- Explain how you would develop a set of soap operas to test this feature.
  - Describe one test that might qualify as a soap opera.
  - Explain why this is a good soap opera test.

# Long Answers

---

21. Imagine that you were testing the feature, “Graphs functions and plots statistical data.”

Describe four examples of each of the following types of attacks that you could make on this feature, and for each one, explain why your example is a good attack of that kind.

- Input constraint attacks
- Output constraint attacks
- Storage constraint attacks
- Computation constraint attacks.

(Notes for you while you study. Refer to Jorgensen / Whittaker’s paper on how to break software. Don’t give me two examples of what is essentially the same attack. In the exam, I will not ask for all 16 examples, but I might ask for 4 examples of one type or two examples of two types, etc.)

# Long Answers

---

22. Imagine that you were testing the feature, “Transfers data to or from a connected calculator.”

Describe four examples of each of the following types of attacks that you could make on this feature, and for each one, explain why your example is a good attack of that kind.

- Input constraint attacks
- Output constraint attacks
- Storage constraint attacks
- Computation constraint attacks.

(Notes for you while you study. Refer to Jorgensen / Whittaker’s paper on how to break software. Don’t give me two examples of what is essentially the same attack. In the exam, I will not ask for all 16 examples, but I might ask for 4 examples of one type or two examples of two types, etc.)

# Long Answers

---

- 23 We are going to do some configuration testing on the TI Interactive product. We want to test it on
- Windows 95, 98, and 2000 (the latest service pack level of each)
  - Printing to an HP inkjet, a LexMark inkjet, and a Xerox laser printer
  - Connected to the web with a dial-up modem (28k), a DSL modem, and a cable modem
  - With a 640x480 display and a 1024x768 display
    - How many combinations are there of these variables?
    - Explain what an all-pairs combinations table is
    - Create an all-pairs combinations table
    - Explain why you think this table is correct.

# *Long Answers*

---

24. Imagine that you are an external test lab, and the TI company comes to you with TI interactive. They want you to test the product. How will you decide what test documentation to give them? (Suppose that when you ask them what test documentation they want, they say that they want something appropriate but they are relying on your expertise.) What questions would you ask (up to 7 questions) and how would the answers to those questions guide you in deciding what to do?



# *Long Answers*

---


25 The following group of slides are from Windows Paint 95.

Please don't spend your time replicating the steps or the bug. (You're welcome to do so if you are curious, but I will design my marking scheme to not give extra credit for that extra work.)

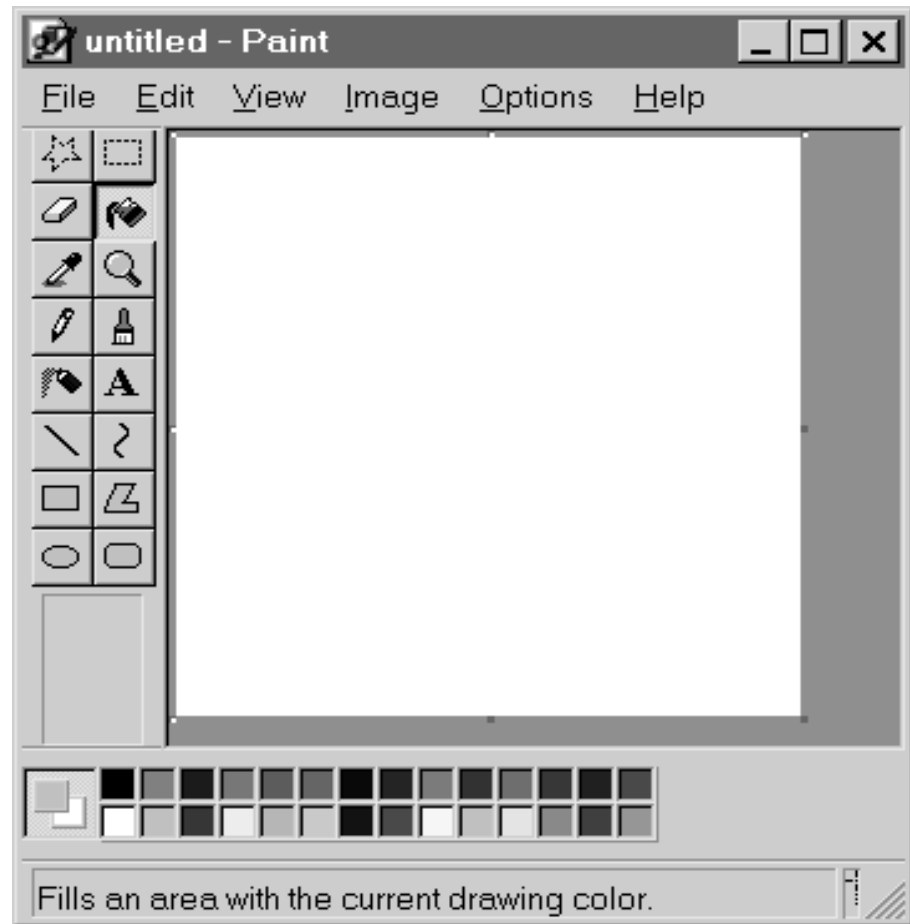
Treat the steps that follow as fully reproducible. If you go back to ANY step, you can reproduce it.

For those of you who aren't familiar with paint programs, the essential idea is that you lay down dots. For example, when you draw a circle, the result is a set of dots, not an object. If you were using a draw program, you could draw the circle and then later select the circle, move it, cut it, etc. In a paint program, you cannot select the circle once you've drawn it. You can select an area that includes the dots that make up the circle, but that area is simply a bitmap and none of the dots in it have any relationship to any of the others.

# Bug Question Continued

Here's the opening screen. The background is white. The first thing that we'll do is select the Paint Can 

We'll use this to lay down a layer of grey paint on top of the background. Then, when we cut or move an area, we'll see the white background behind what was moved.

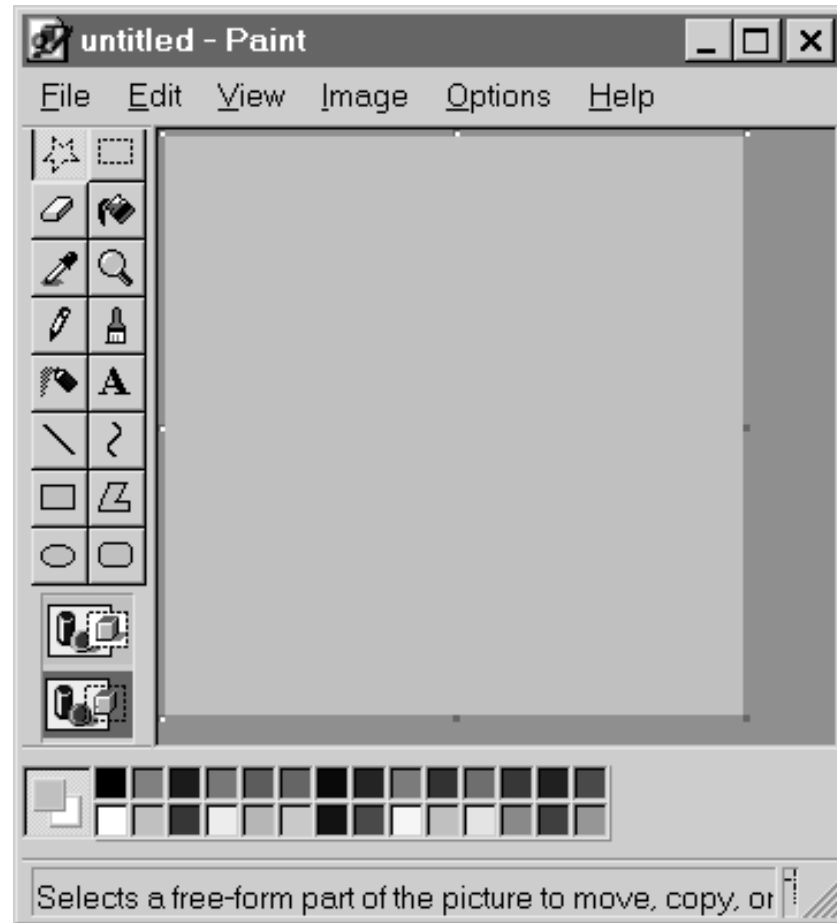


# Bug Question Continued

Here's the screen again, but the background has been painted gray.



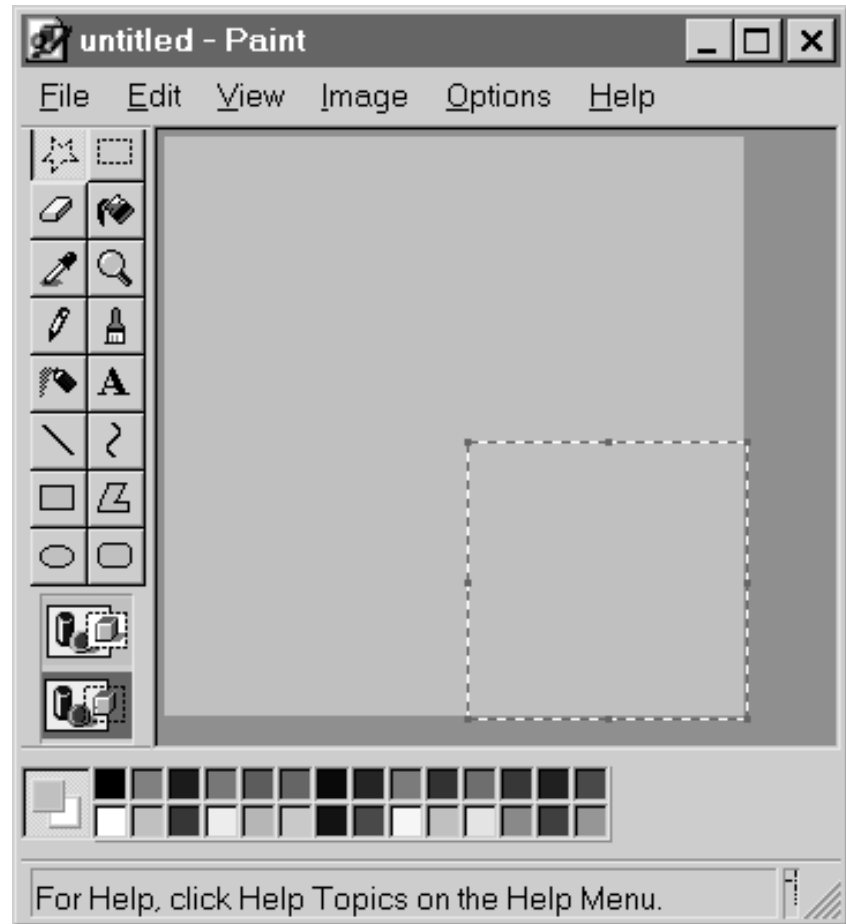
The star in the upper left corner is a freehand selection tool. After you click on it, you can trace around any part of the picture. The tracing selects that part of the picture. Then you can cut it, copy it, move it, etc.



# Bug Question Continued

This shows an area selected with the freehand selection tool. The bottom right corner is selected. (The dashed line surrounds the selected area.)

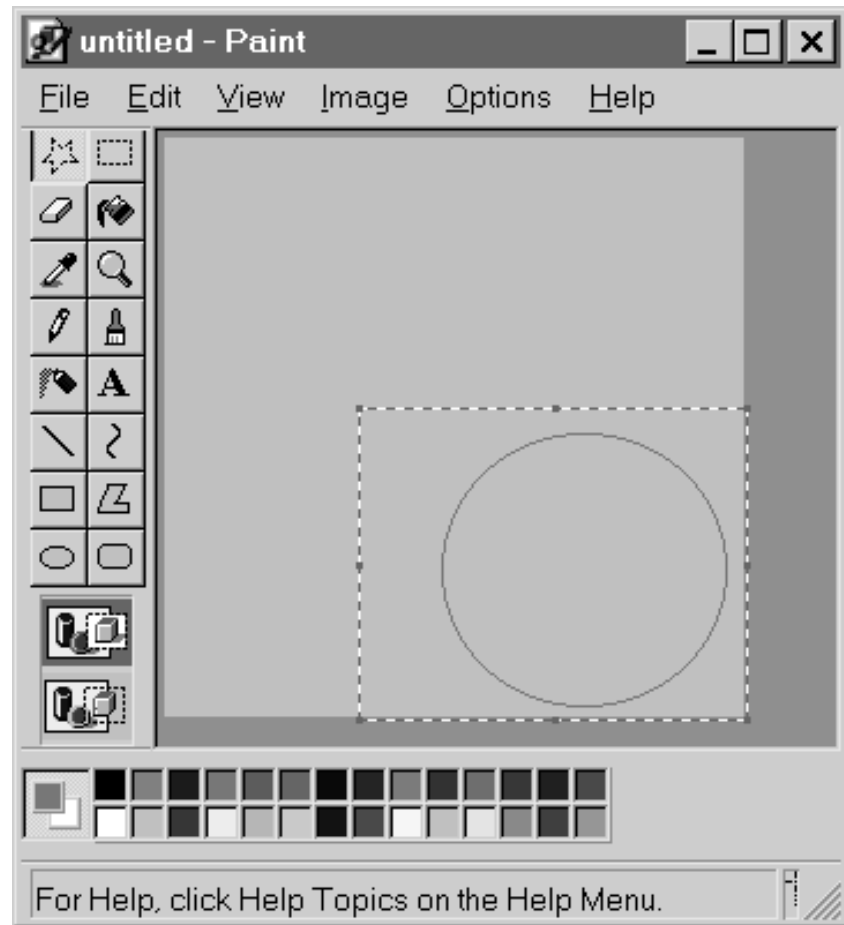
***NOTE: The actual area selected might not be perfectly rectangular. The freehand tool shows a rectangle that is just big enough to enclose the selected area. For our purposes, this is not a bug. This is a design decision by Microsoft.***



# Bug Question Continued

Next, we'll draw a circle (so you can see what's selected), then use the freehand select tool to select the area around it.

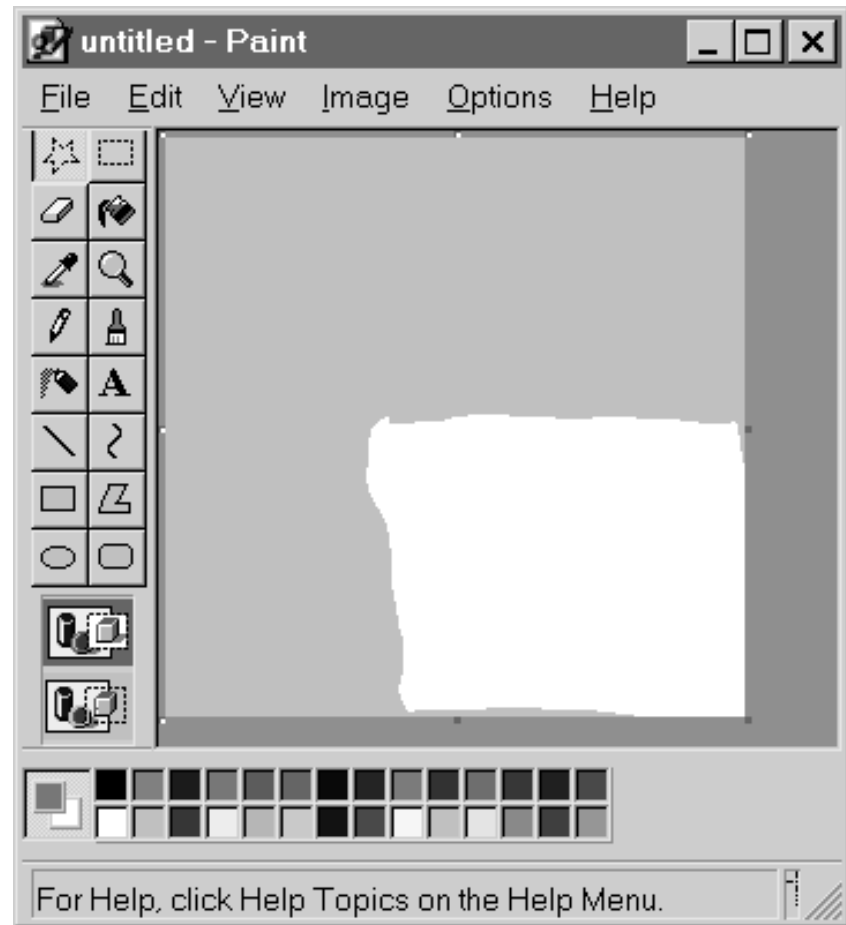
When you use the freehand selection tool, you select an area by moving the mouse. The real area selected is not a perfect rectangle. The rectangle just shows us where the selected area is.



# Bug Question Continued

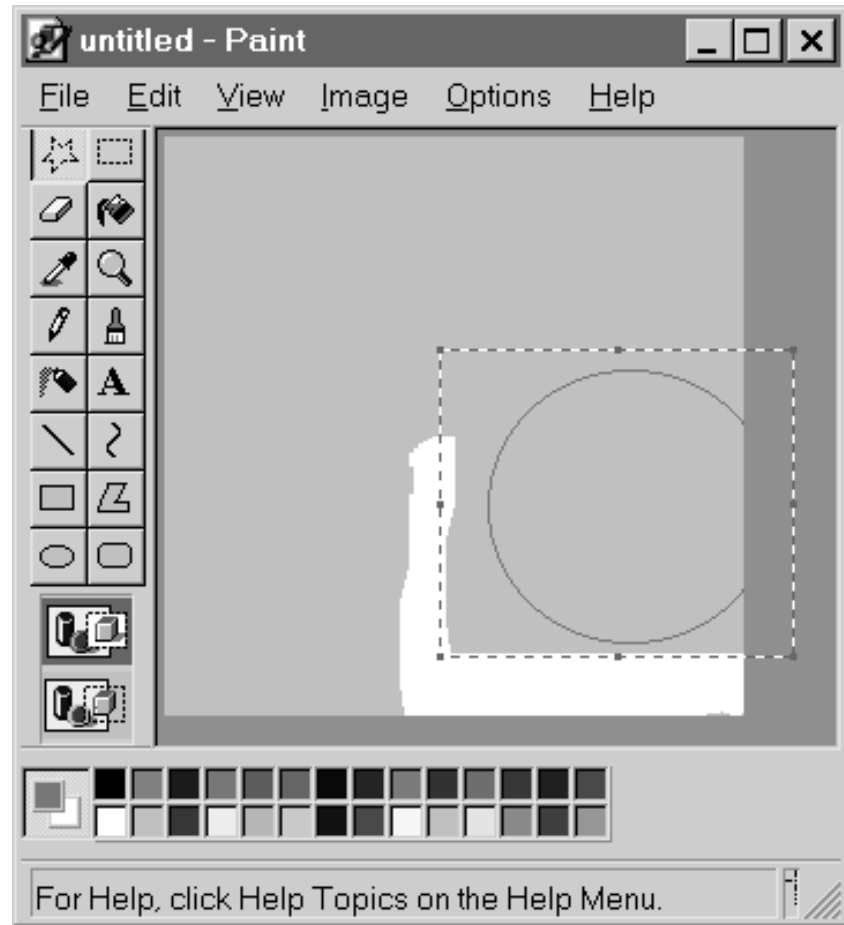
Now we cut the selection. (To do this, press **Ctrl + X**)

The jagged border shows exactly the area that was selected.



# Bug Question Continued

Next, exit the program, restart it, color the background grey, draw the circle, select the area around the circle and drag it up and to the right. This works.



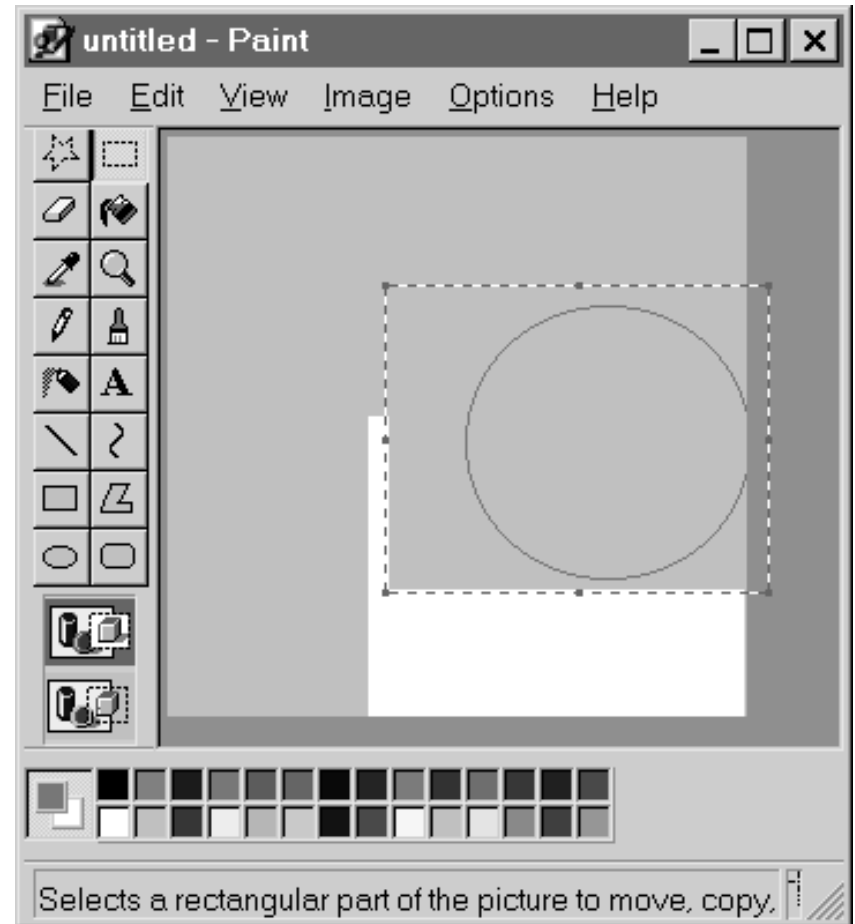
# Bug Question Continued

This time, we'll try the Rectangular Selection tool.



With this one, if you move the mouse to select an area, the area that is actually selected is the smallest rectangle that encloses the path that your mouse drew.

So, exit the program, start it up, color the background, draw a circle, click the Rectangular Selection tool, select the area around the circle and move it up. It works.





# *Bug Question Continued*

---

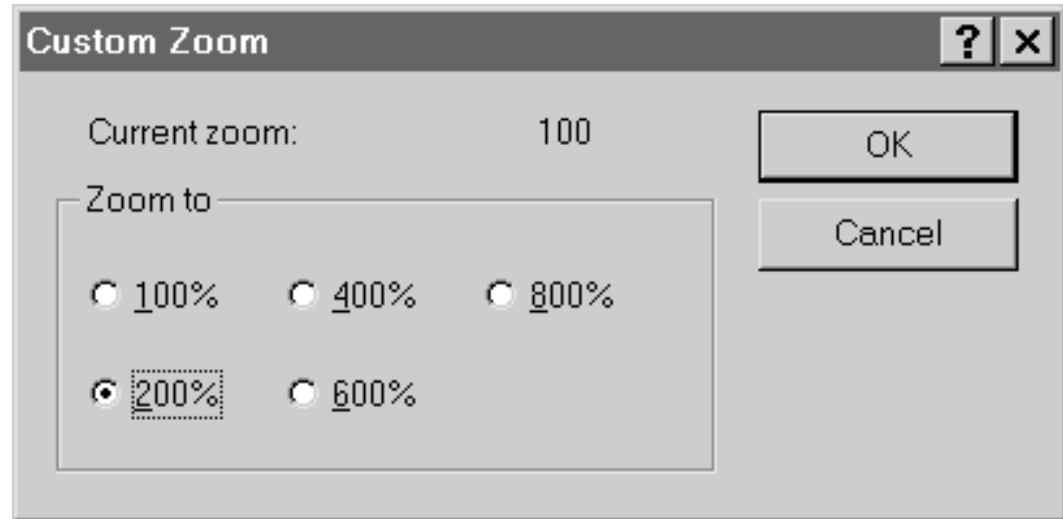
Well, this was just too boring, because everything is working. When you don't find a bug while testing a feature, one tactic is to keep testing the feature but combine it with some other test.

In this case, we'll try Zooming the image. When you zoom 200%, the picture itself doesn't change size, but the display doubles in size. Every dot is displayed as twice as tall and twice as wide.

So we exit the program, start it up, color the background grey, draw the circle, and then . . .

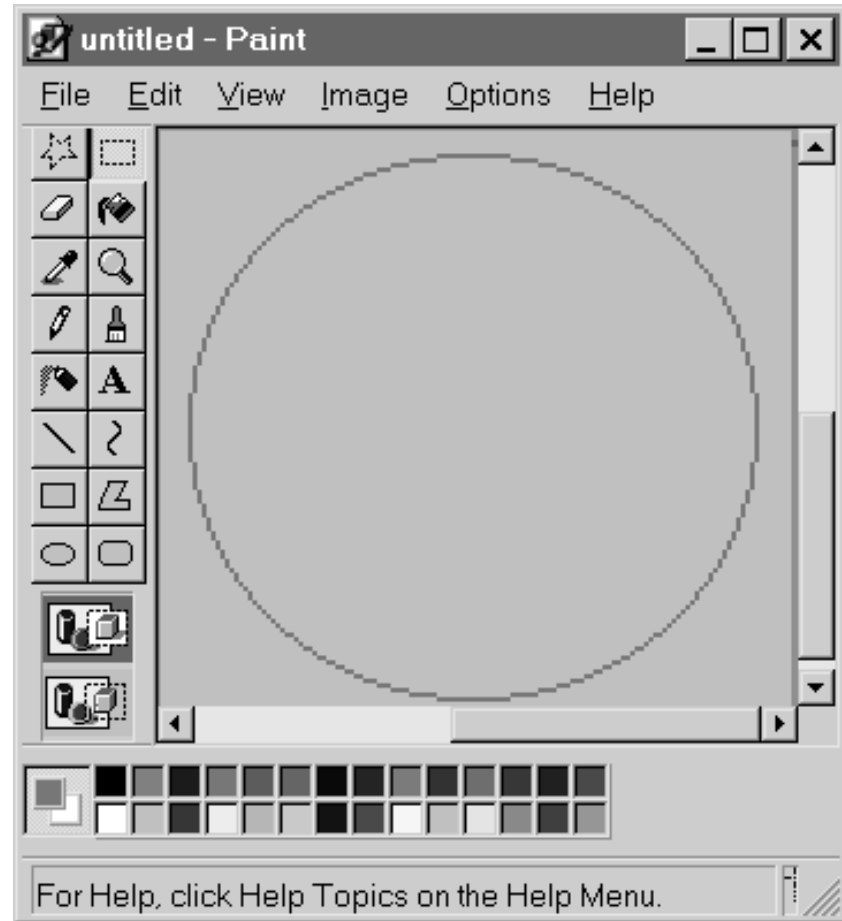
# *Bug Question Continued*

Bring up the Custom Zoom dialog, and select 200% zoom, click OK.



# Bug Question Continued

It worked. The paint area is displayed twice as tall and twice as wide. We're looking at the bottom right corner. To see the rest, we could move the scroll bars up or left.

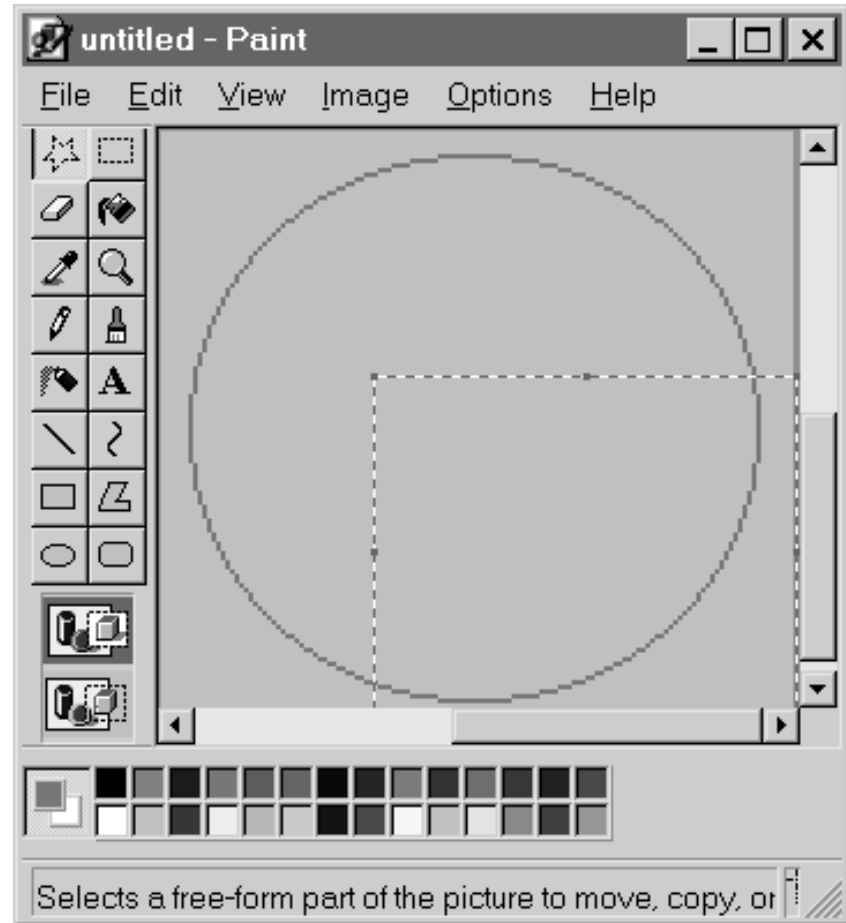


# Bug Question Continued

So, we select part of the circle using the freehand selection tool. We'll try the move and cut features.

Cutting fails.

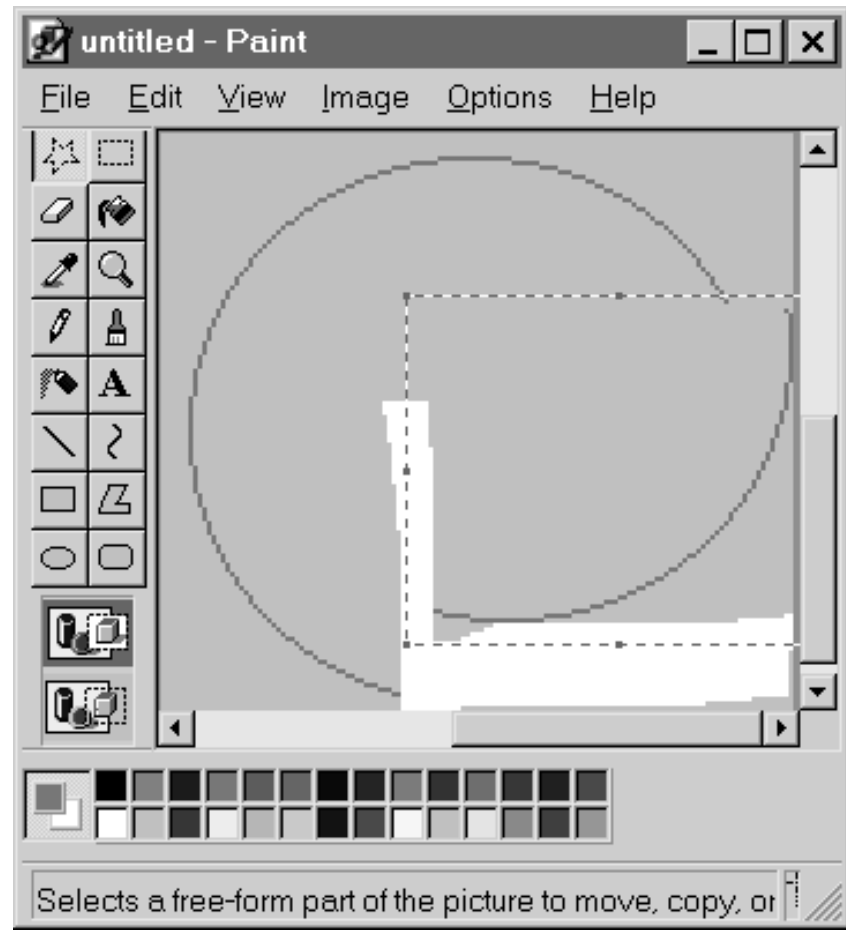
When we try to cut the selection, the dashed line disappears, but nothing goes away.



# Bug Question Continued

Exit the program, start again, color the background, draw the circle, zoom to 200%, select the area.

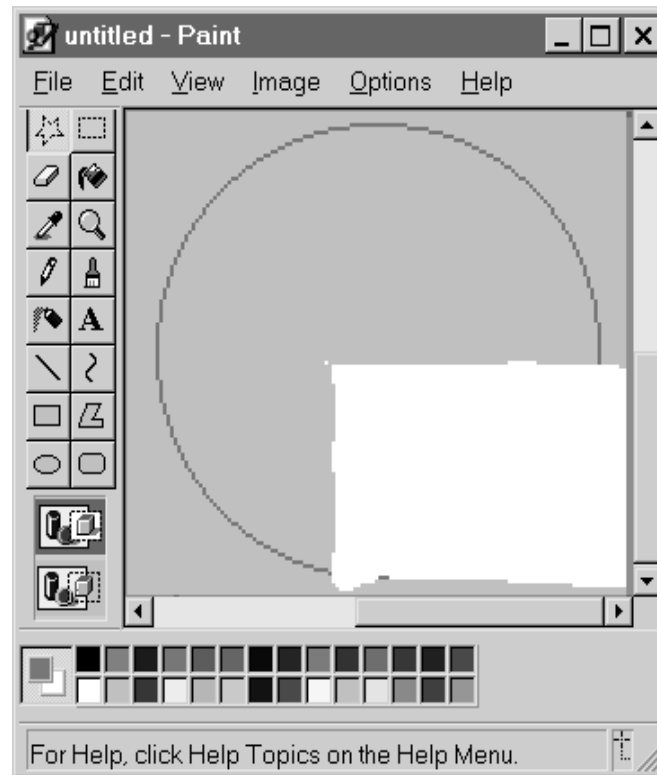
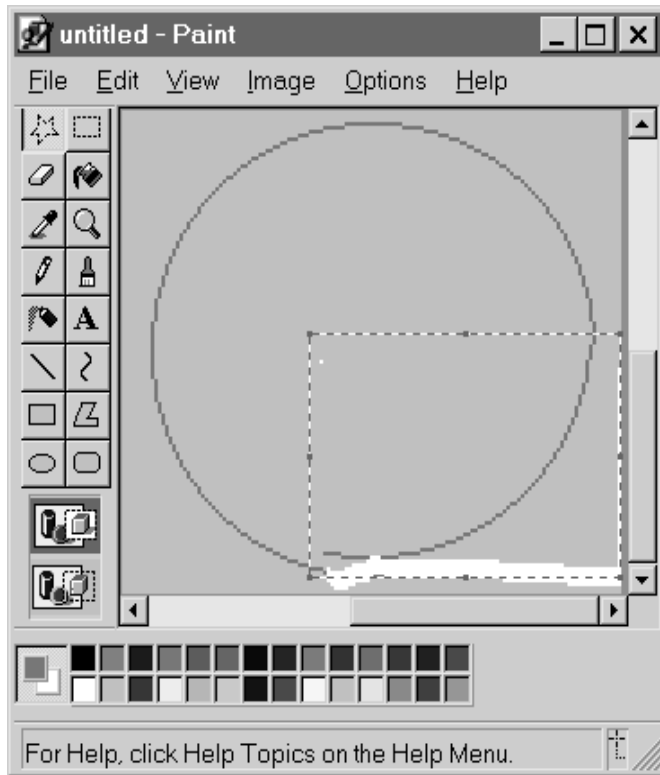
Drag the area up and to the right. It works.



# Bug Question Continued

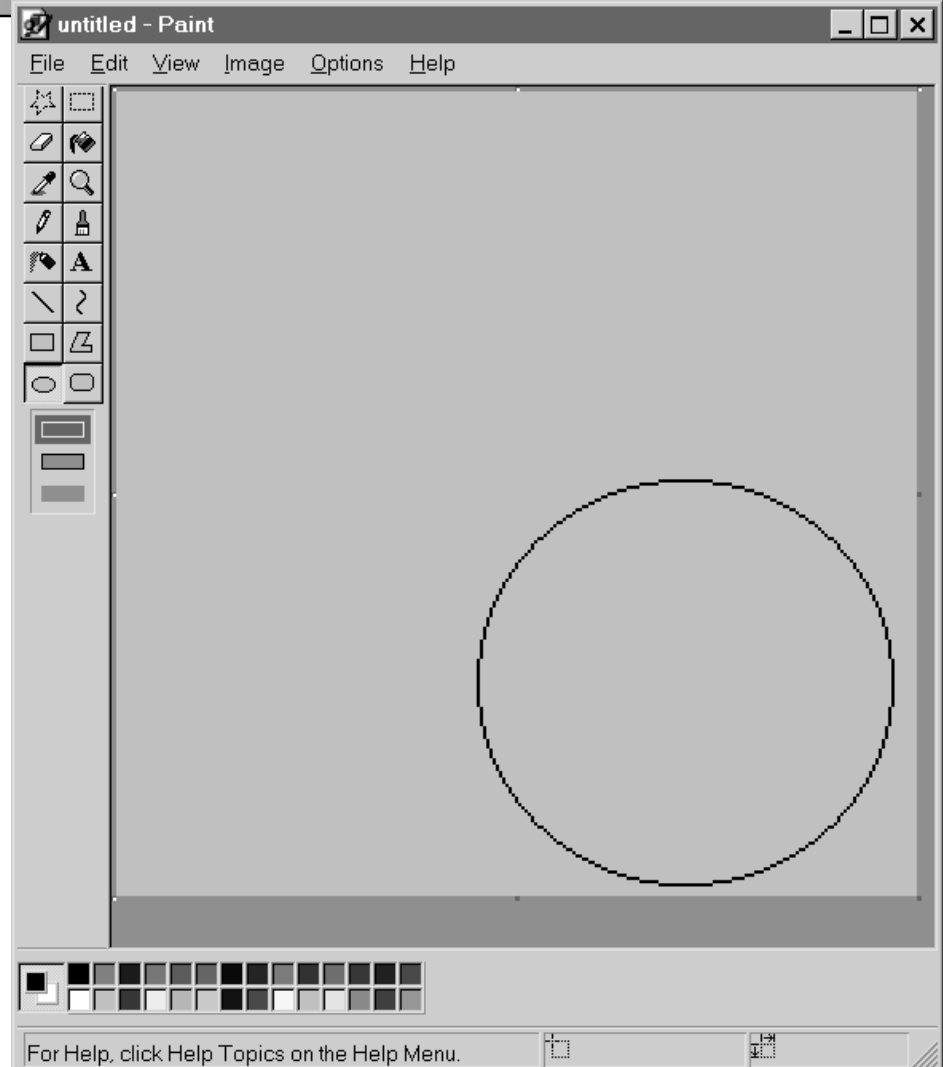
Exit the program, start again, color the background, draw the circle, zoom to 200%, select the area.

Now try this. Select the area and move it a bit. THEN press Ctrl-C to cut. This time, cutting works.



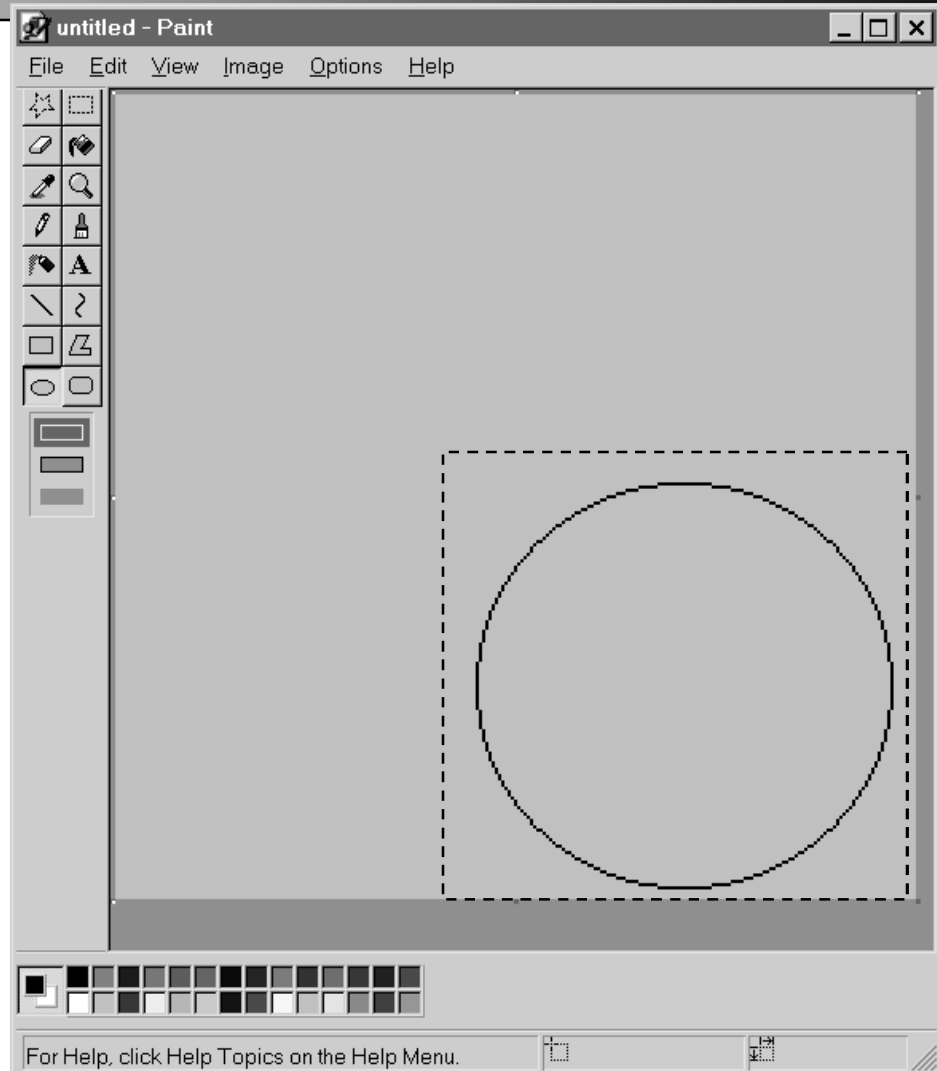
# Bug Question Continued

Exit the program, start again, color the background, draw the circle, zoom to 200%, and this time, *grow the window* so you can see the whole drawing area.



# Bug Question Continued

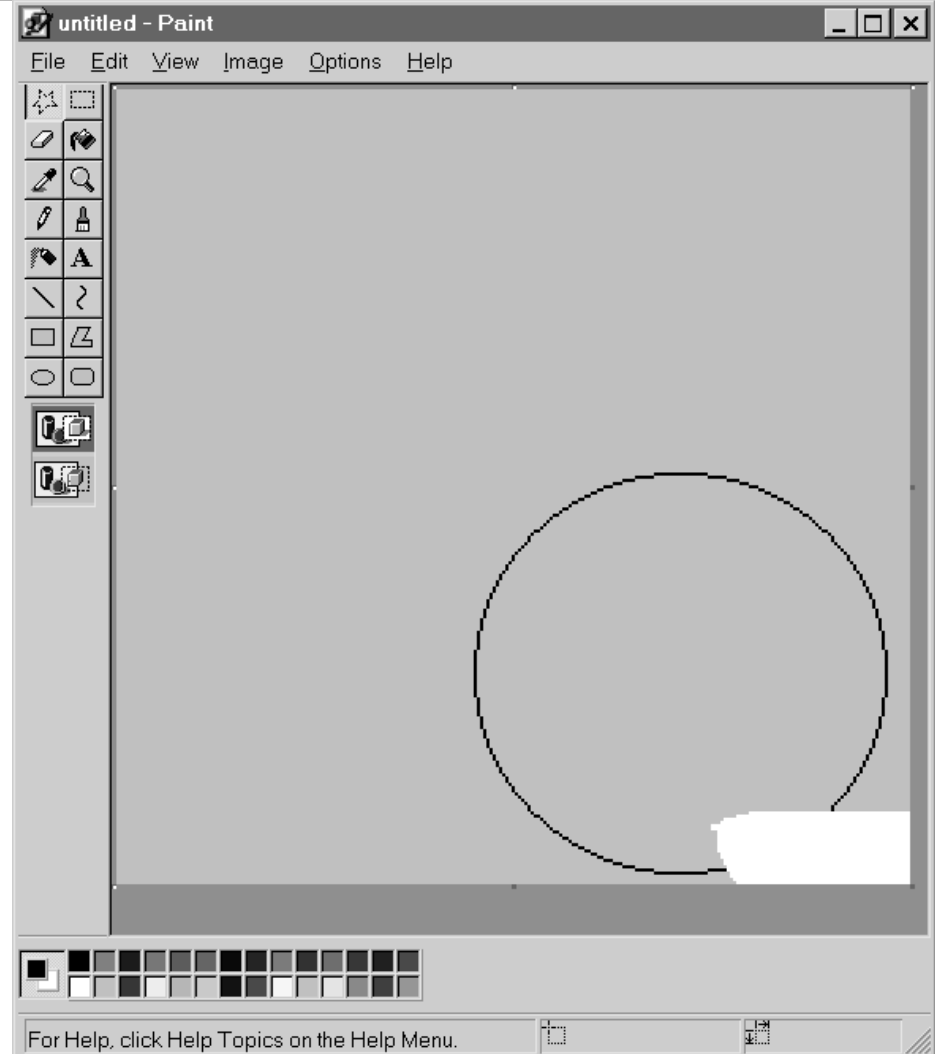
Now, select the circle.  
That seems to work.





# Bug Question Continued

But when you press Ctrl-X to cut the circle, the program cuts the wrong area.



# *Bug Question Continued*

---

Now, write a bug report. I want two sections:

- The Problem summary (or title)
- The Problem Description (how to reproduce the problem)

Additionally, please describe three follow up tests that you would run with this bug