

Testing, Verification, Debugging

CS577B Workshop

Spring 2002

Testing and your project, 1

- Test your stuff.
- Testing in MBASE:
 - The requirements in the SSRD, and their prioritizations in the LCP/FRD indicate the most important aspects of your project (which things need to be right first).
 - All those requirements have “measurable” fields: these should directly suggest possible tests.
 - The SSAD *is* useful:
 - Your component model gives you a perfect level of abstraction at which to apply black-box verification tests (are the behaviors correct?).
 - Your object model gives you a long list of things that need thorough white-box, black-box and integration testing (do it as you go along!).
 - The CTS documents are all about testing:
 - Use the Test Plan and Test Results Reports to document your tests.

Testing and your project, 2

- Who should test?
 - Testing is something you do as you program, often in an informal way.
 - Often when the one who programs something misses various mistakes, by being too close to the code. Thus, everything should be gone over by someone other than the person who wrote it.
 - A good software product requires many sorts of testing at various levels: probably your whole group can and should help out.
 - Some sorts of tests involve your customer or the TAs.
- Be sure to leave time for testing in your schedule.

Test Planning

- Do this NOW if haven't already !!
 - Follow the CTS guidelines for a Test Plan
- The testing plan should be an integrated part of the overall plan scheduled in the LCP
- The test plan must define how success will be measured
 - follows from MARS and scenarios in SSRD
- All requirements must have a test.
 - Many different kinds of tests, some trivial, some not
 - Degree of testing depends on risk considerations

Creating a Test Plan

- Step 1: Requirements Analysis
 - For each requirement in SSRD choose test types, and methodologies and document in the Test Plan (see CTS guidelines) - must be consistent with FRD, LCP
 - Each requirement may need multiple test types and methodologies
 - If you are having trouble choosing a test, the requirement may need to be more clearly specified
 - Project and Quality requirements have MARS
 - System Requirements have Use-Cases
 - Other Requirements (e.g. evolutionary SQs) may be “binary” tests or need additional specifications for test purposes
 - A test is defined as a combination of the test types, and methodologies chosen for the requirement

Creating a Test Plan (Cont'd)

- Step 2: Prioritize
 - For each test type, determine order (account for dependencies) and priority (critical, important, moderate, low) based on risk considerations
- Step 3: For each test, specify Test Identification details in Test Description
- Step 4: Account for test time/effort and other resource costs in LCP
 - Don't neglect the priorities and risks (especially in the FRD)

Goal-Model-Question Metric Approach to Testing

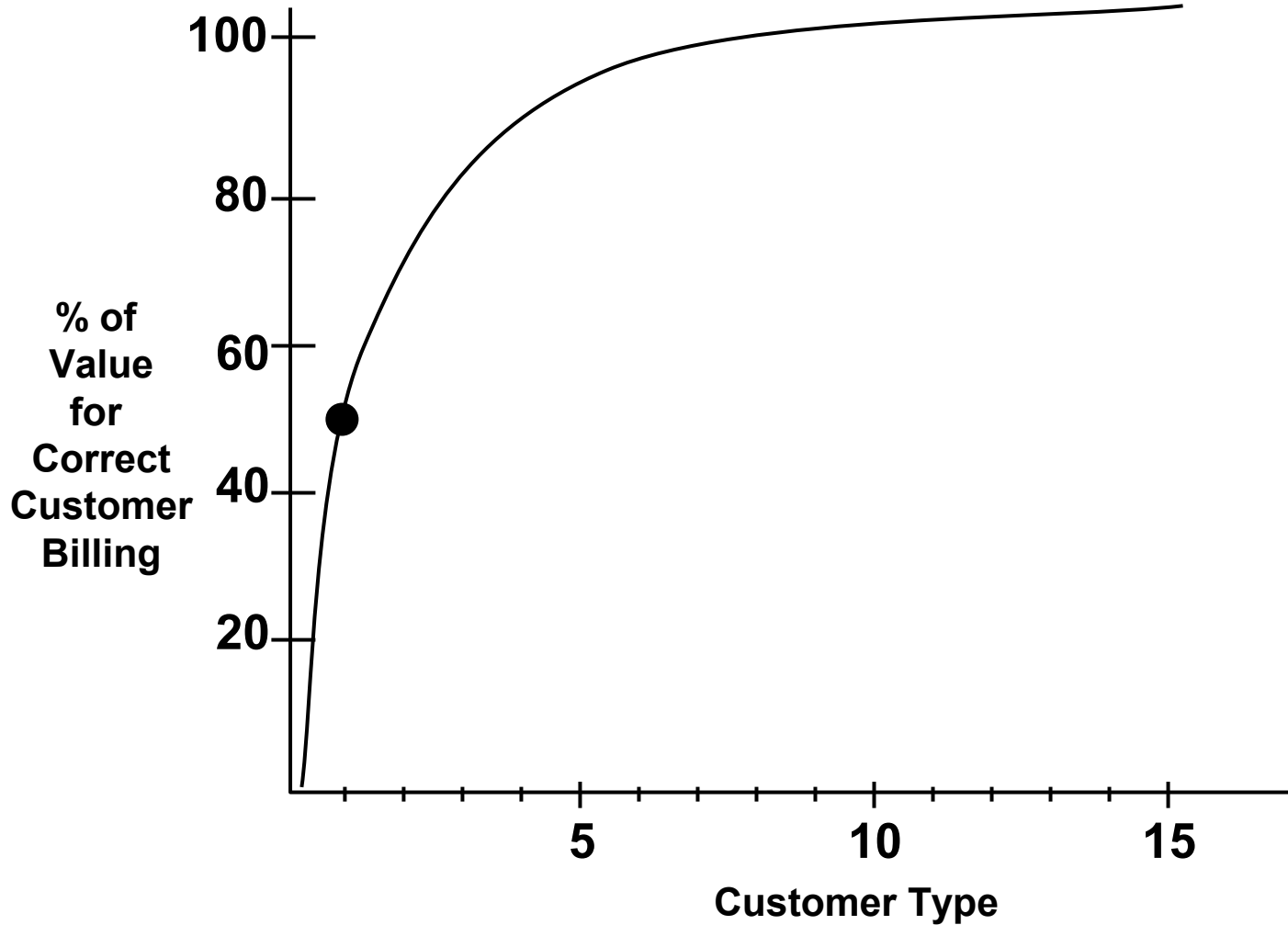
Goal	Models	Questions	Metrics
<ul style="list-style-type: none"> Accept product 	<ul style="list-style-type: none"> Product requirements 	<ul style="list-style-type: none"> What inputs, outputs verify requirements compliance? 	<ul style="list-style-type: none"> Percent of requirements tested
<ul style="list-style-type: none"> Cost-Effective Defect Detection 	<ul style="list-style-type: none"> Defect source models <ul style="list-style-type: none"> Orthogonal defect Classification (ODC) Defect-prone modules 	<ul style="list-style-type: none"> How best to mix and sequence automated analysis, reviewing, and testing? 	<ul style="list-style-type: none"> Defect detection yield by defect class Defect detection costs, ROI Expected vs. actual defect detection by class
<ul style="list-style-type: none"> No known delivered defects 	<ul style="list-style-type: none"> Defect closure rate models 	<ul style="list-style-type: none"> How many outstanding defects? What is their closure status? 	<ul style="list-style-type: none"> Percent of defect reports closed <ul style="list-style-type: none"> Expected; actual
<ul style="list-style-type: none"> Meeting target reliability, delivered defect density (DDD) 	<ul style="list-style-type: none"> Operational profiles Reliability estimation models DDD estimation models 	<ul style="list-style-type: none"> How well are we progressing towards targets? How long will it take to reach targets? 	<ul style="list-style-type: none"> Estimated vs. target reliability, DDD Estimated time to achieve targets
<ul style="list-style-type: none"> Minimize adverse effects of defects 	<ul style="list-style-type: none"> Business-case or mission models of value by feature, quality attribute, delivery time <ul style="list-style-type: none"> Tradeoff relationships 	<ul style="list-style-type: none"> What HDC techniques best address high-value elements? How well are project techniques minimizing adverse effects? 	<ul style="list-style-type: none"> Risk exposure reduction vs. time <ul style="list-style-type: none"> Business-case based "earned value" Value-weighted defect detection yields for each technique

Note: No one-size-fits-all solution

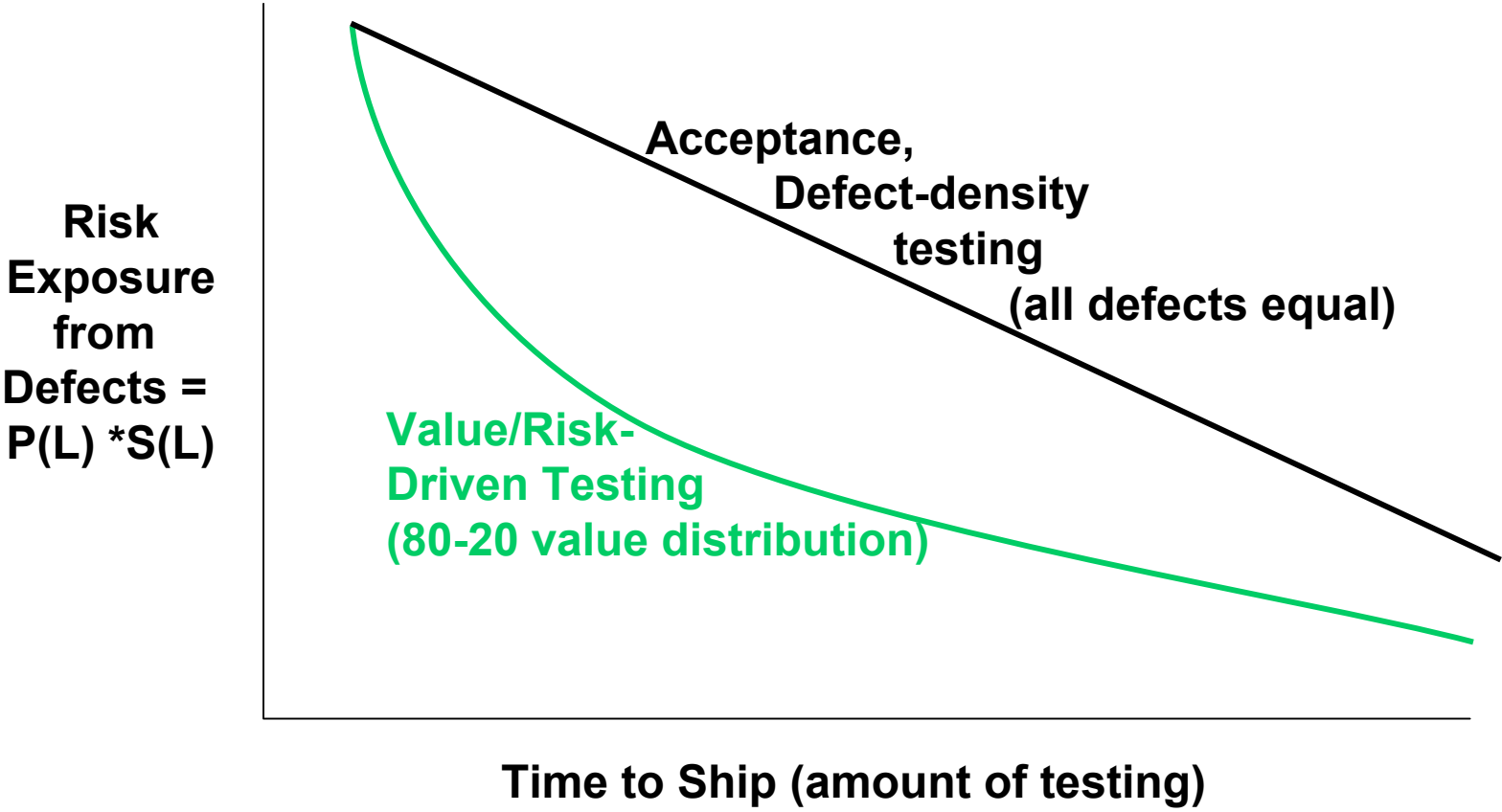
GMQM Paradigm: Value/Risk - Driven Testing

- **Goal: Minimize adverse effects of defects**
- **Models: Business-case or mission models of value**
 - By feature, quality attribute, or delivery time
 - Attribute tradeoff relationships
- **Questions: What testing techniques best address high-value elements?**
 - How well are project techniques minimizing adverse effects?
- **Metrics: Risk exposure reduction vs. time**
 - Business-case based “earned value”
 - Value-weighted defect detection yields by technique

20% of Features Provide 80% of Value: Focus Testing on These (Bullock, 2000)



Resulting Reduction in Risk Exposure



Kinds of Tests

- There are many “types” of tests that can be performed:
 - Acceptance
 - Unit and Validation
 - white-box
 - black-box
 - Integration
 - interface
 - Path

Acceptance tests

- An *acceptance test* is any where some person or people decide whether an aspect of the software is acceptable, or not.
- Use acceptance tests when you have something concrete to test:
 - At the ends of development cycles.
 - With prototypes.
 - With “beta” versions of your system, or components.
- Acceptance tests return a boolean value: watch out for “that’s nice, but....” responses from your customer (ie, feature creep).

Unit and Validation testing

- A *unit test* is any that is specific to some component of your system (ie, not the whole system).
- *Validation tests* determine how well a component corresponds to its specification in the design.

white-box tests

- A *white-box test* (or *glass-box*) uses knowledge of component's implementation to test it.
 - A tester needs to be very familiar with the language and technology.
 - The tester reads the code, looking for mistakes or ways to break it (boundary conditions, errors, limits).
 - White-box testing is time-consuming, and requires much skill. Thus it is often only done thoroughly on a per-component basis (ie, as a unit test), and only for critical components.

black box tests

- *Validation tests* determine how well a component corresponds to its specification in the design.
- *Black-box testing* isn't concerned with the component's implementation, but gauges whether it accepts the correct inputs and produces correct outputs.
- We can use *equivalence partitioning* to guide black-box/validation testing:
 - Since we rarely have time to try all the possible inputs, group them, and test representatives from each group.

black box testing (cont'd)

- Use *Equivalence Partitioning* to test the system
 - group valid inputs together
 - group invalid inputs together
 - group questionable inputs together
 - group outputs and identify erroneous groupings
- Focus on *boundary values*, and inputs which may be strange or unexpected
 - For a search routine, search for the first and last items
 - Search for the middle item
 - Search for a non-existent item
 - Search an empty database

Integration tests

- Not just getting all of it to compile!
- Use Black Box methods, and a limited number of white box methods to confirm that the sub-systems are interacting correctly
- Tests must be thorough. They must be designed to exercise all of the use cases and error conditions

Integration and interface tests

- *Integration tests* see whether components that work separately work together.
- *Interface tests* focus on the points of contact between two components. Interfaces include:
 - Method invocations.
 - Network or other stream-based communications.
 - Shared memory, files and other system resources (and therefore, resource locking).

Paths tests

- A *paths test* identifies some number of paths through a system, usually from the user's point of view.
- It's attractive to try an *all-paths test* of the system, where each possible interaction with the system is tried at least once. However, we can show by a combinatorial argument that this takes far too long for a non-trivial program. Some useful alternatives:
 - Identify *critical paths*, the most important, frequent and useful manipulations of the system.
 - Identify *invariants* in the system state; then we show that each path returns us to the home state.

Test Methodologies

Ways of performing and planning tests:

- Ad-hoc (sometimes called “agile”)
 - “Big Bang”
- Tactical
 - Build-Test-Debug
 - Back-to-Back
- Strategic
 - Top-down
 - Bottom-up
 - Sandwich
 - Defect risk driven
 - stress

Methodology: “Big Bang”

- The ‘Let ‘er rip!!’ methodology that you are all familiar with
- Useful for small systems that contain few program paths
- Difficult to manage on a large scale
- May not identify all of the problems with a system

Methodology: B-T-D

1. *Build something*
2. *Test it*
3. *Debug it*
4. *Goto step 1*

- Very common approach
- Slow, uncontrolled, unpredictable
- Hard to make risk driven
- Good for very small, non-critical components
- Very locally focused, not end-to-end

Methodology: Back-to-back

- Used when a previous system existed
- Run the old system and compare it to the new system
- Compare the response times
- Compare the data outputs

Methodology: Top-down, bottom-up

- *Top-down testing* starts with the most abstract view of the system, tests it at that level, then partitions it into components and recursively tests each.
- *Bottom-up testing* starts with the most basic software units: individual code bits (each method or “distinct” part of a program). It works up through the software, testing things at an increasingly more abstract level.
- The bottom-up method is more complete, but takes longer than the top-down method. The top-down method is more user-centered (it finds the most annoying bugs first).

Methodology: Sandwich

- One team tests from the bottom-up
- One-team tests from the top-down
- This guarantees that the base components will work
- This guarantees that the High level integration will work
- Question: Does this guarantee that the system will work as a whole?

Methodology: defect risk driven

- Identify which cases will cause a system to keep a user from performing a desired task
- Use one, or more of the previously discussed methodologies to evaluate the key cases
- Good for critical sub-systems
- Integration of techniques is cost effective and relatively efficient

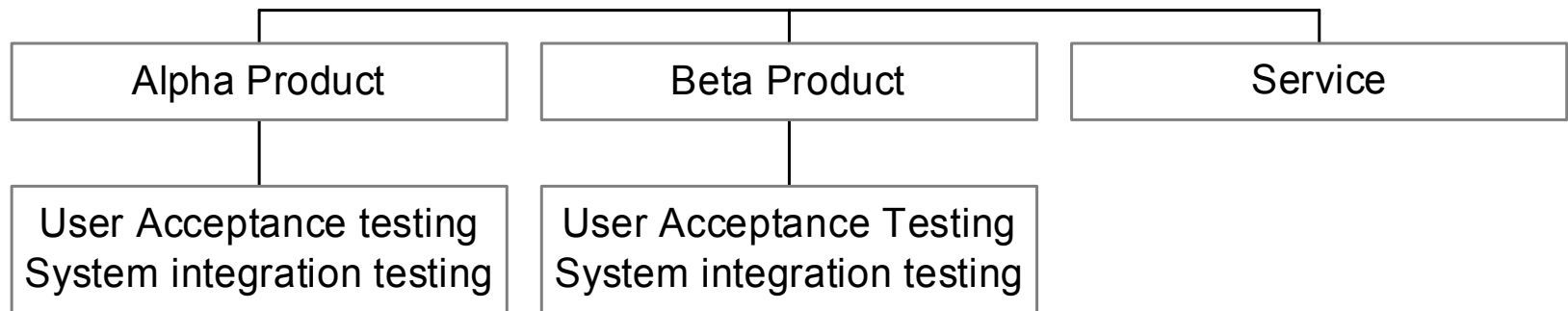
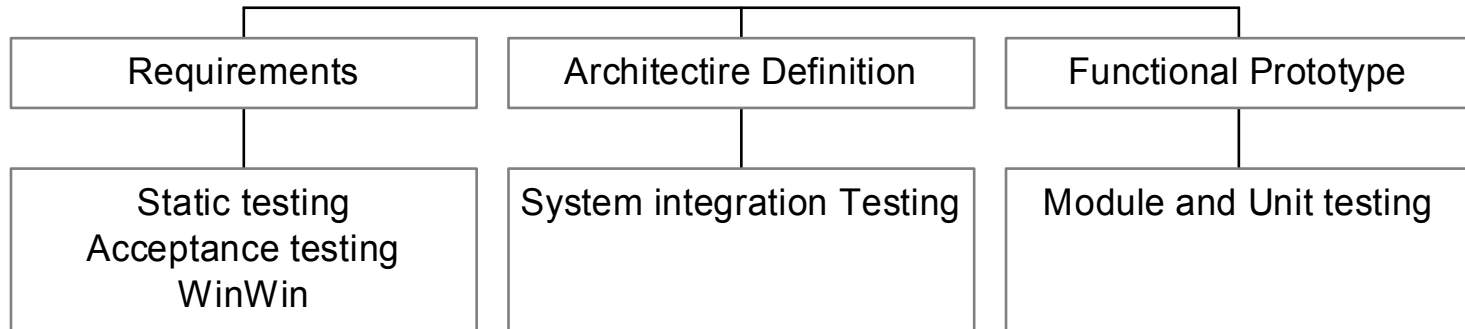
Methodology: System and stress

- *System tests* test the entire assembled system as a unit.
 - The system should be tested in a realistic environment, on its actual hardware, by real users as well as the developers.
 - System testing is often done in two phases, “Alpha” and “Beta”. Beta release often go to many people outside the development team.
- A *stress test* is a type of system test, where the system’s performance limits are stretched (and broken, usually). Stress tests are interested in gathering performance statistics, and determining whether the system handles failure gracefully.

Methodology: Regression

- *Regression testing* tests a new version of software against a rigorously defined set of inputs and outputs, to ensure the new version has introduced no inconsistencies with old versions.
 - Regression testing is one reason why we keep old tests around (and document them): the old tests, passed once, should be applied again with every significant change to the system.
 - Continual Back-to-back testing is a specific form of regression testing.

The Testing Cycle



Debugging Techniques

Debugging

- Debugging is a black art. Some things to go over, though, so they'll be concrete in our brains:
 - relation to testing
 - why debugging is hard
 - types of bugs
 - process
 - techniques
 - tools
 - avoiding bugs

Debugging and testing

- Testing and debugging go together like peas in a pod:
 - Testing finds errors; debugging repairs them.
 - Together these form the “testing/debugging cycle”: we test, then debug, then repeat.
 - Any debugging should be followed by a reapplication of *all* relevant tests, particularly regression tests. This avoids (usually) the introduction of new bugs when debugging.
 - Testing and debugging need not be done by the same people (and often should not be).

Why debugging is hard

- There may be no obvious relationship between the external manifestation(s) of an error and its internal cause(s).
- Symptom and cause may be in remote parts of the program.
- Changes (new features, bug fixes) in program may mask (or modify) bugs.
- Symptom may be due to human mistake or misunderstanding that is difficult to trace.
- Bug may be triggered by rare or difficult to reproduce input sequence, program timing (threads) or other external causes.

Types of bugs

- Types of bugs (gotta love em):
 - Compile time: syntax, spelling, static type mismatch.
 - Usually caught with compiler
 - Design: flawed algorithm.
 - Incorrect outputs
 - Program logic (if/else, loop termination, select case, etc).
 - Incorrect outputs
 - Memory nonsense: null pointers, array bounds, bad types, leaks.
 - Runtime exceptions
 - Interface errors between modules, threads, programs (in particular, with shared resources: sockets, files, memory, etc).
 - Runtime Exceptions
 - Off-nominal conditions: failure of some part of software or underlying machinery (network, etc).
 - Incomplete functionality
 - Deadlocks: multiple processes fighting for a resource.
 - Freeze ups, never ending processes

The ideal debugging process

- A debugging algorithm for software engineers:
 - Identify test case(s) that reliably show existence of fault (when possible)
 - Isolate problem to small fragment(s) of program
 - Correlate incorrect behavior with program logic
 - Change the program (and check for other parts of program where same or similar program logic may also occur)
 - Regression test to verify that the error has really been removed - without inserting new errors
 - Update documentation when appropriate

(Not all these steps need be done by the same person!)

Debugging techniques, 1

- Execution tracing
 - running the program
 - print
 - trace utilities
 - single stepping in debugger
 - hand simulation

Debugging techniques, 2

- Interface checking
 - check procedure parameter number/type (if not enforced by compiler) and value
 - *defensive programming*: check inputs/results from other modules
 - documents assumptions about caller/callee relationships in modules, communication protocols, etc
- Assertions: include range constraints or other information with data.
- Skipping code: comment out suspect code, then check if error remains.

Debugging with print, 1

- How debugging with print can be made more useful:
 - print variables other than just those you think suspect.
 - print valuable statements (not just “hi\n”).
 - use `exit()` to concentrate on a part of a program.
 - move print through a through program to track down a bug.

Debugging with print, 2

- Building debugging with print into a program:
 - print messages, variables in useful places throughout program.
 - use a ‘debug’ or ‘debug_level’ global flag to turn debugging messages on or off (compare perl’s ‘Carp’ module).
 - possibly use a source file preprocessor to insert/remove debug statements.

Tools

- Tracing programs:
 - strace, truss (print out system calls)
- Command-line debuggers:
 - gdb (C, C++), jdb (java), “perl -d”
- Random stuff:
 - electric fence (a specialized malloc() for finding memory leaks in C)
 - purify (purifies C programs)

Using gdb

- Compile debugging information into your program:

```
gcc -g <program>
```

- Read the manual:

- `man gdb`

- in emacs (tex-info):

- `M-x help <return> i <return>`, arrow to GDB, <return>

- online:

- http://www.cslab.vt.edu/manuals/gdb/gdb_toc.html

`gdb` commands

`run/continue/next/step`: start the program, continue running until break, next line (don't step into subroutines), next line (step into subroutines).

`break <name>`: set a break point on the named subroutine. Debugger will halt program execution when subroutine called.

`backtrace`: print a stack trace.

`print <expr>`: execute expression in the current program state, and print results (expression may contain assignments, function calls).

`help`: print the help menu. `help <subject>` prints a subject menu.

`quit`: exit `gdb`.

Avoiding bugs in the first place

- Coding style: use clear, consistent style and useful naming standards.
- Document everything, from architecture and interface specification documents to comments on code lines.
- Hold code reviews.
- Program defensively.
- Use/implement exception handling liberally; think constantly about anomalous conditions.
- Be suspicious of cut/paste.
- Consider using an integrated development environment (IDE) with dynamic syntax checking

Code reviews

- Primary programmer(s) for some piece of code presents and explains that code, line by line.
- Audience of programmers experienced in language, code's general domain. Audience may also contain designers, testers, customers and others less versed in code but concerned with quality and consistency.
- Review is a dialogue: audience pushes presenters to reevaluate and rationalize their implementation decisions.
- Extremely useful: reviews often turn up outright errors, inconsistencies, inefficiencies and unconsidered exceptional conditions.
- Also useful in familiarizing a project team with a member's code.

some favorite bugs

- Some o' my favorite bugz:

- In java (or whatever):

```
public void foo(int p, int q) {  
    int x = p;  
    int y = p;  
}
```

- In perl:

```
$foo = $cgi->param('foo');  
if (!$foo) {  
    webDie ("missing parameter foo!");  
}
```

- In C:

```
char *glue(char *left, char sep, char *right) {  
    char *buf = malloc(sizeof(char) *  
                        (strlen(left) + 1 +  
                        strlen(right)));  
    sprintf(buf, "%s%c%s", left, sep, right);  
    return buf;  
}
```