



End to End Software Quality

Quality through the eyes of a Quality Architect

Mark Wilding
mwilding@ca.ibm.com

© 2006 IBM Corporation

Agenda

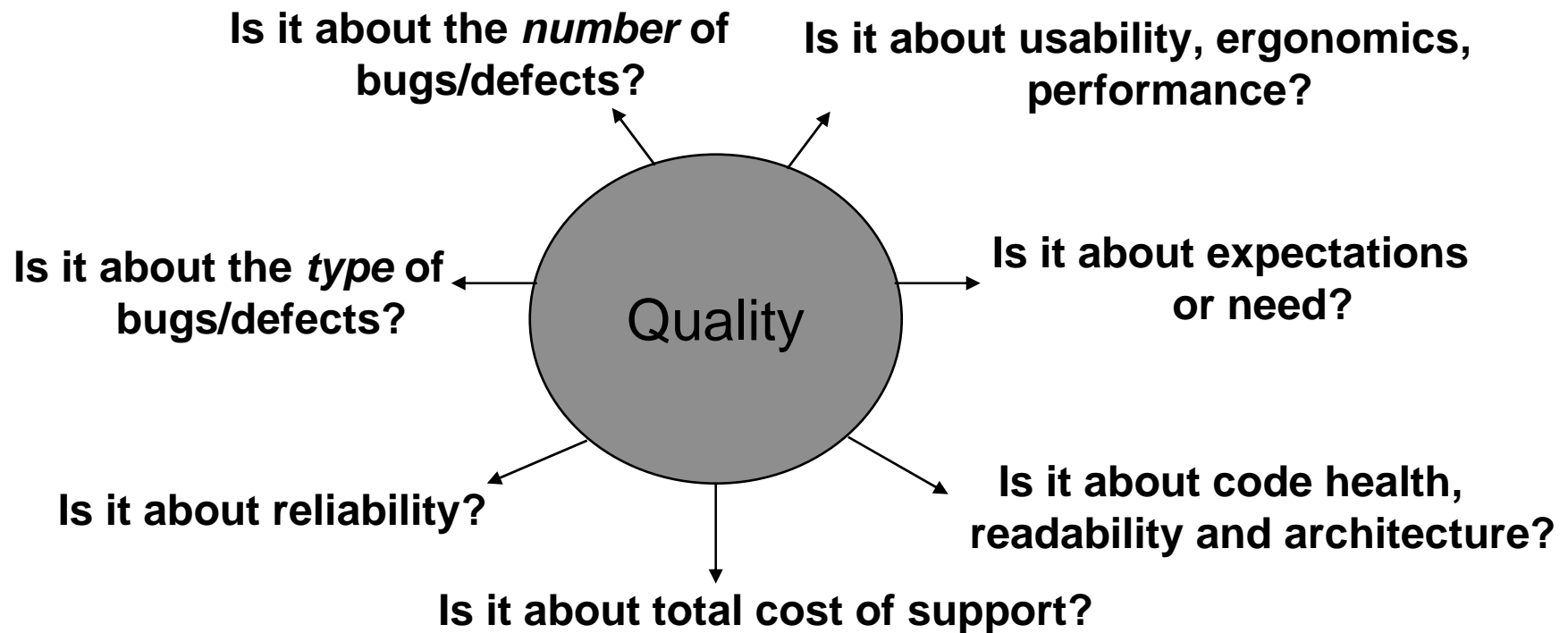
- **Quality Themes**
- **What is software Quality?**
- **Why is software Quality important?**
- **The 6 steps for Quality Software**
- **Improving Quality of an existing product**
- **Summary**

Themes

- 1. Quality has many aspects, all are important**
- 2. Think negatively**
- 3. Find defects early**
- 4. The 80/20 rule**
- 5. Simplicity!**

What is Quality?

What is Quality?



Quality is all of the above!

Software Quality is...

- **External Quality: *Perception***
 - Perception is reality for external software quality
 - It is about meeting and exceeding expectations
 - Ayn Rand: "We can avoid reality, but we cannot avoid the consequences of avoiding reality"

- **Internal Quality: *Innately clean and manageable software***
 - Purist: Clean architecture, design and code
 - Business: The total cost of supporting and maintaining a software product

- **Fortunately, perception of quality and internal quality are not necessarily orthogonal**

Why is Quality Important?

Famous software bugs

- **The Mars Climate Orbiter crashed in September 1999 due to a software defect**
 - "The 'root cause' of the loss of the spacecraft was the failed translation of English units into metric units in a segment of ground-based, navigation-related mission software, as NASA has previously announced,"
 - <http://mars.jpl.nasa.gov/msp98/news/mco991110.html>

- **The nine-hour breakdown of AT&T's long-distance telephone network in Jan. 1990, caused by an untested code patch, dramatized the vulnerability of complex computer systems everywhere. "Ghost in the Machine," *Time Magazine*, Jan. 29, 1990. p. 58.**
 - **The nine-hour breakdown of AT&T's long-distance telephone network dramatizes the vulnerability of complex computer systems everywhere**
 - <http://www.cs.berkeley.edu/~nikitab/courses/cs294-8/hw1.html>

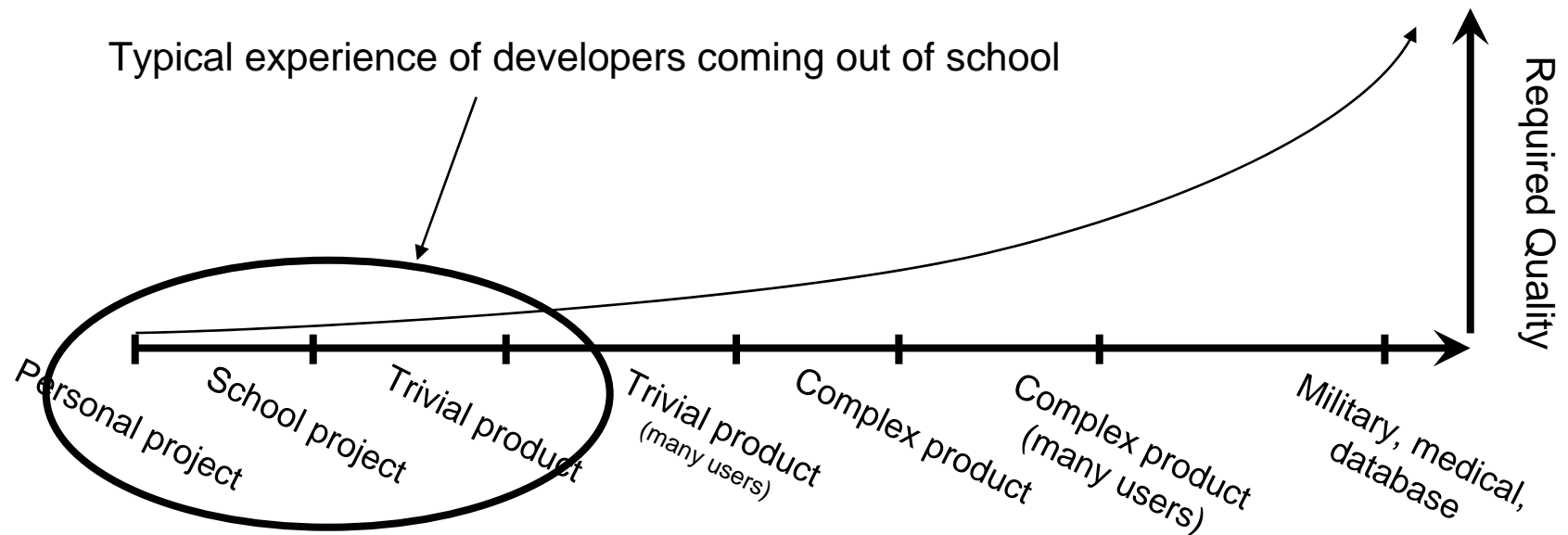
Famous software bugs

- **Airbus A320 crashed at an air show in 1988**
 - Cause: "The pilot claims he was misled on the aircraft's true height by a bug in the software."
 - <http://catless.ncl.ac.uk/Risks/8.77.html#subj6>

- **Between June 1985 and January 1987, six known accidents involved massive overdoses by the Therac-25 -- with resultant deaths and serious injuries.**
 - Cause: race condition between concurrent tasks resulted in massive overdoses
 - As Frank Houston of the US Food and Drug Administration (FDA) said, "A significant amount of software for life-critical systems comes from small firms, especially in the medical device industry; firms that fit the profile of those resistant to or uninformed of the principles of either system safety or software engineering."
 - http://courses.cs.vt.edu/%7Ecs3604/lib/Therac_25/Therac_1.html

Levels of Software Quality

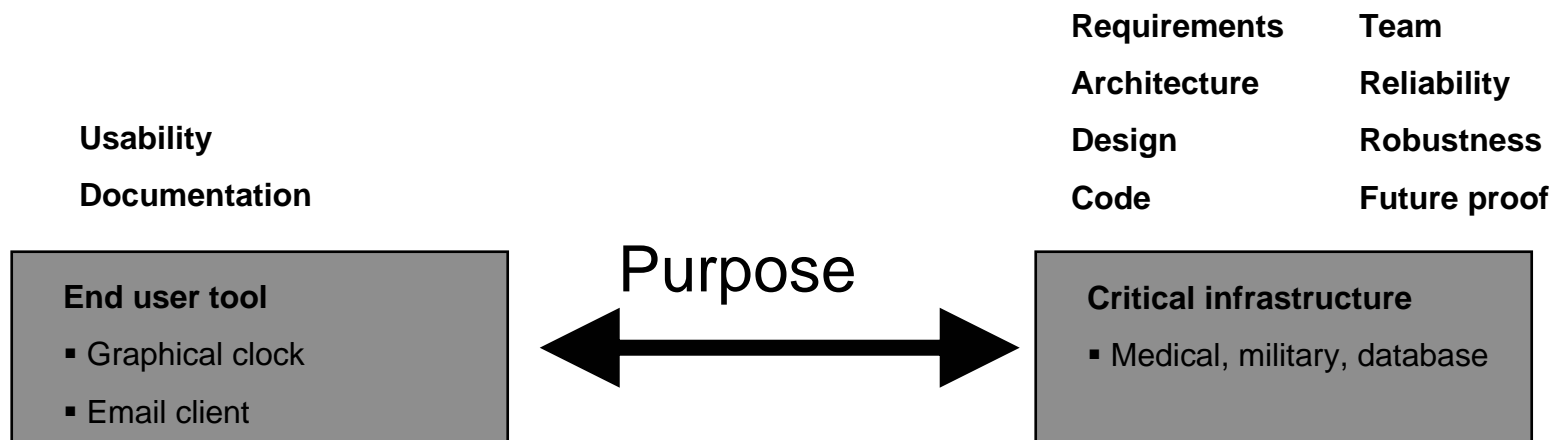
- **Software quality is not that important for the programs you write at school or for programs you write for yourself**
- **The level and type of Quality depends entirely on the purpose, size, complexity and number of end users**



Types of Software Quality: Purpose and Number of Users

- **A web browser needs Quality in the form of usability**
 - Crashing once a week may not be a big deal to the end user if everything else works well
 - Of course, not crashing at all would be better!

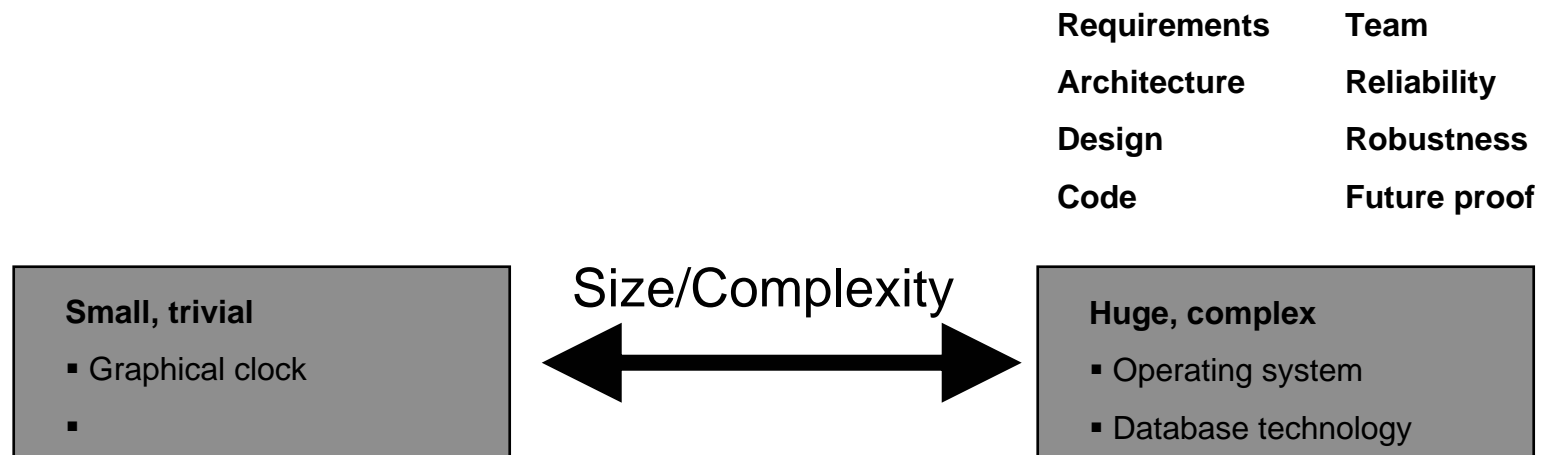
- **A database product needs Quality in the form of reliability**
 - Usability is less important than reliability
 - Crashing once a month is *not* okay for a database product



Types of Software Quality: Size and Complexity

- **A web browser needs Quality in the form of usability**
 - Crashing once a week may not be a big deal to the end user if everything else works well
 - Of course, not crashing at all would be better!

- **A database product needs Quality in the form of reliability**
 - Usability is less important than reliability
 - Crashing once a month is *not* okay for a database product



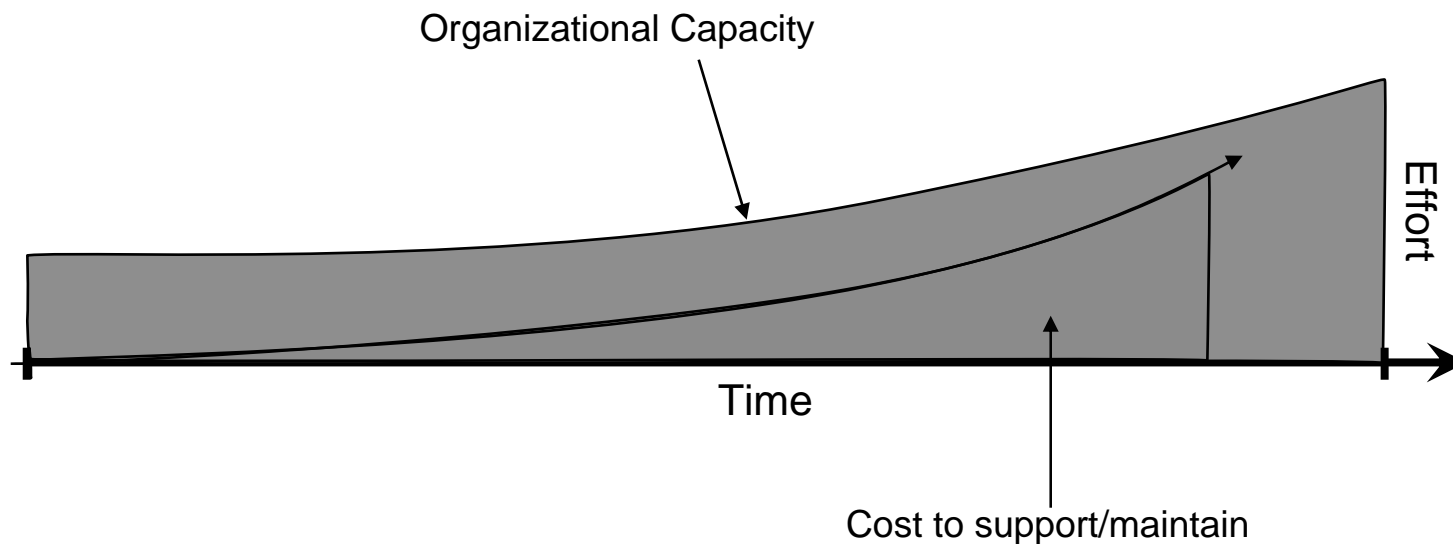
Importance of Quality to users

- **What is important:**
 - Usability
 - Expectations
 - Reliability
 - Depends on the purpose, size, complexity and number of end users

- **Impacts of poor external quality**
 - Users will ask questions
 - Users will find defects
 - Users will become irate
 - Users will buy someone else's product

Importance of Quality to business costs

- **Cost of support/maintenance can cripple a product**
- **Poor Quality means high/increasing costs in the long run**



6 Steps for good quality software

6 Steps for Quality Software

- 1. Understand your users**
 - Roles, personas, environments, scenarios
 - Requirements/pain points
- 2. Spend enough time on architecture and design**
 - Think ahead (somewhat)
 - Positive and **negative** scenarios
 - Prototype as/if needed
- 3. Thoroughly validate your solution before coding**
 - Does it address the original requirements, roles, scenarios and environments?
- 4. Write the code, carefully.**
- 5. Test**
 - Unit test
 - Externals (positive and negative)
 - Create automated regression
 - Stress the code (drive timing conditions and error conditions)
 - Test out error paths
- 6. Learn from your mistakes and your customers' mistakes, repeat**

Step #1: Understand your users

Step #1: Understand your users

- **“users” are often very different than developers**
 - Each have different roles in life
 - Each will see things from a different point of view (e.g., they are probably not software developers)
 - The “users” may use the software for a living... developers probably don’t

- **Understanding users and their requirements means understanding:**
 - What they need to do and why
 - Personas: who are these user?
 - Environment
 - Pain points
 - Adverse conditions
 - Related pain points

- **Adverse conditions can break an otherwise good design!**

- **Be sure to validate the requirements, environments, adverse conditions, use cases and pain points with the users!**

Example: Mobile Computing

- **Consider two very different individuals that have similar high level requirements:**
 - To check email every 30 minutes or so
 - To look through live inventory and online documentation.
 - To check live schedules during the day
- **We would need more details on these requirements but let's focus on who these people are!**

Toronto Business Professional



Railway Conductor



Mobile computing example #1

Toronto Business Professional

- **Requirements**
 - To check email every 30 minutes or so
 - To look through live inventory and online documentation.
 - To check live schedules during the day
- **Pain points**
 - Cannot easily carry a heavy laptop in busy subway and around Toronto
 - Connectivity is not always available when needed (depends on location)
- **Environment**
 - Subway, restaurants, sidewalk, outdoors
 - Small work space (lap, coffee table)
- **Adverse conditions**
 - Weather (rain, snow, heat)
 - Walking/running
 - Coffee spills
 - Cramped spaces
- **Persona**
 - Very busy, always on the move

Mobile computing example #2

Railway Conductor

- **Requirements**
 - To check email every 30 minutes or so
 - To look through live inventory and online documentation.
 - To check live schedules during the day
 - **Pain points**
 - No WiFi connectivity in the country (away from any city)
 - No cellphone connectivity in the country side
 - Vibrations of the train damages laptops
 - **Persona**
 - Not much experience with computers
 - **Environment**
 - Driving usually at 50-90 km/h in various directions
 - Inside a radio frequency blocking steel box (train car)
 - Indoors (in train)
 - **Adverse conditions**
 - Moving at high speeds, vibrations
 - Different directions
 - Low hanging trees can damage antennas
 - Coffee/drink spills
 - Greasy/dirty hands
- Think about the perception of Quality if the laptop gets dirty and breaks

Review the problem space and user information

- **This step is the most important part of building high quality software**
- **A mistake in requirements gathering can make an entire solution useless**

- **Look for ways to reduce the scope of the problem without reducing the customer satisfaction.**
 - Example: is the customer okay with connectivity at every train stop?
 - Use the 80/20 rule to reduce the number of key requirements if possible

- **It is often required to iterate again and again to fully understand the problem space and user information.**

Step #2: Architecture and Design

Step #2: Spend enough time on architecture and design

- **There is often a tendency to “get something done”... that is, to write some code**
 - Resist!
 - Coding should be about 1/5th of the total effort
- **The software architecture will form the basis for all the product features**
 - It is expensive to build on top of a messy architecture
 - It is difficult to debug a messy architecture

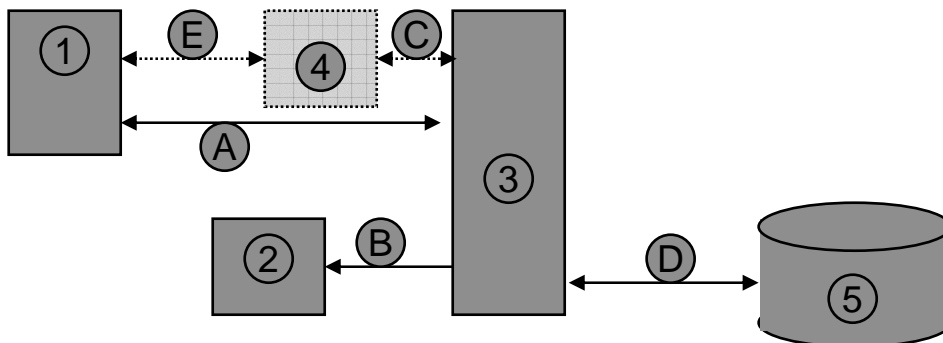


I'd like that now...



Creating a clean architecture

- **Start with a “component diagram”**
 - Define the “components” and “interfaces” of the architecture
- **Define specs for the overall architecture as well as the general purpose of the components and interfaces**
 - Focus on the usage scenarios (how the architecture will be used by the users)
 - Focus on the negative scenarios (how the architecture will react to them)
- **Think strategically, act tactically**
 - Define enough of the details to make the architecture work and to make it future proof, and no more
 - Use potential scenarios that may be valid in the future
- **If unsure about the required architecture or design, prototype and learn**
- **A good architecture is clean and simple!**

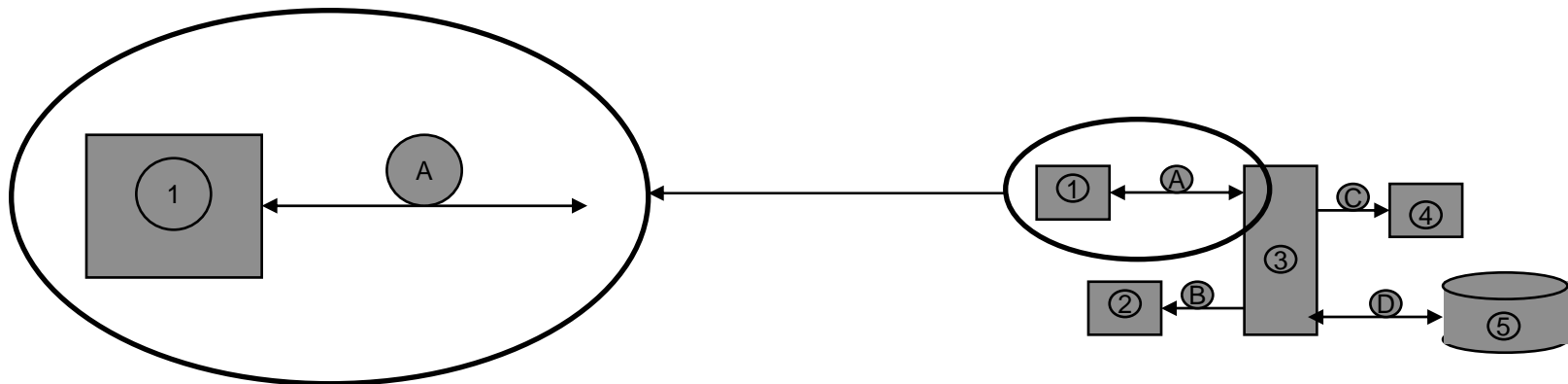


▪ **Components:** the logical units of functionality

▪ **Interfaces:** the APIs or protocols that connect the components

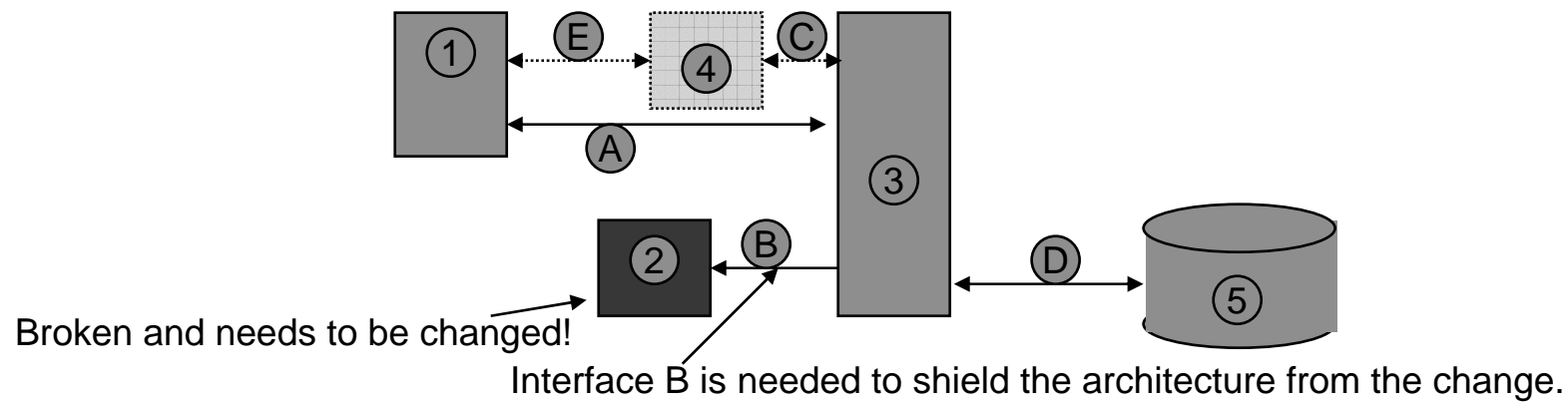
Creating a clean design

- **Design the functionality of each required component in detail. Only design enough details to:**
 1. Make the solution work now
 2. Make the solution future proof
- **Design the interfaces in detail. Again, only design enough details to:**
 1. Make the solution work now
 2. Make the solution future proof
- **Save the rest of the details for the future!**



Creating clean interfaces!

- **These will be boundary points in the product**
 - They need to be well defined so they can be well tested
 - They need to be well defined to hide the details of the underlying component (in case the component needs to be replaced)
- **Future proof interfaces.**
 - Interfaces can “shield” the details of the underlying component incase it needs to be replaced in the future
 - This is a key part to a future proof architecture since there is a good chance that a component will need to be re-written or replaced
 - Interfaces are more difficult to change in the future than components!

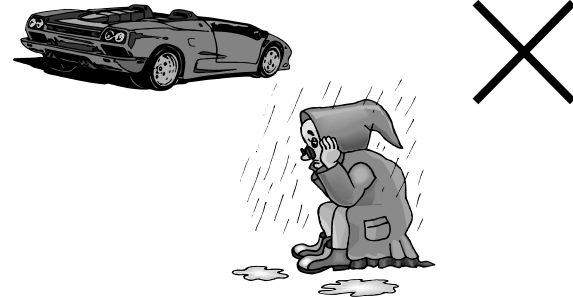


“Good Enough” Software

- **80/20 Rule: Only build what is needed... and nothing more**
- **Scenario: customer wants a car and a house... there is a fixed amount of resources**
- **If we build a super fancy car, and not a house: our efforts are completely wasted!**
 - We have an unhappy customer!
- **A future proof architecture and design will let us build that fancy car if/when it is needed.**



Happy Customer



Unhappy Customer

A quick note about graphical interfaces

- **Graphical interfaces require a more iterative approach.**
 - Get the users involved early and work directly with them if possible
 - It may take 10 iterations or more to create a useful user interface!

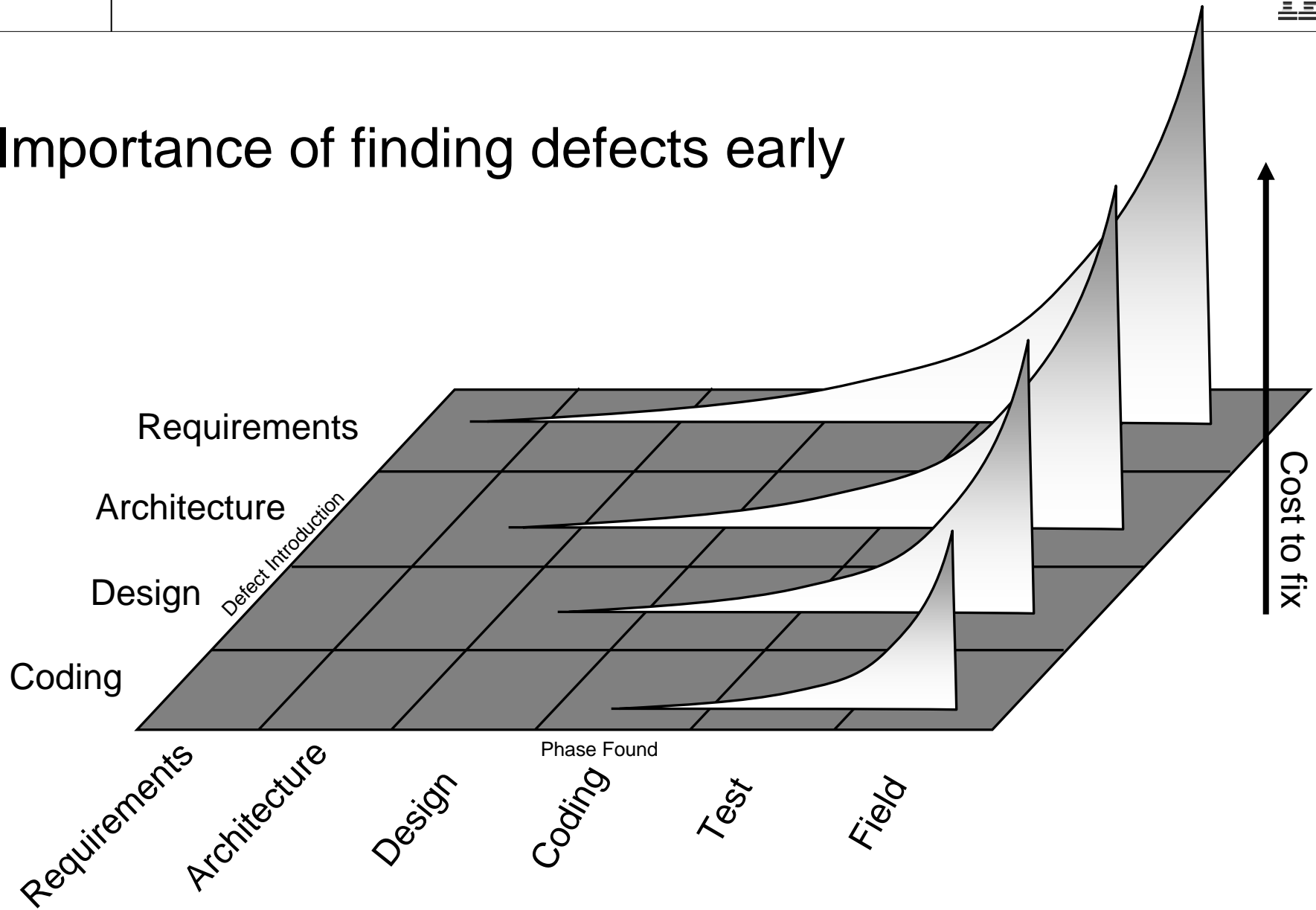
- **Why are graphical interfaces different wrt Quality?**
 - Because their quality is much more about usability and ergonomics

Step #3: Validate the solution

Validate the solution

- **With the details of the solution in hand:**
 - Go back to the requirements and information about the users of the solution and confirm that the solution will work
- **Go through end to end scenarios with the solution in mind**
 - Expand or redefine the component architecture as needed to meet the requirements, environment and pain points.
 - Focus again on the adverse conditions
 - Spend some time on the related pain points
 - Example: if WiFi is used inside the train for the Railway Conductor, it would mean passengers could use it as well (and they may be willing to pay!)
- **Key point: fixing defects at this point is still relatively cheap**
 - May even be worth while to revalidate the requirements
- **Validate the new scenarios with the users and test team!**

Importance of finding defects early



Fixing requirements or architecture defects later...

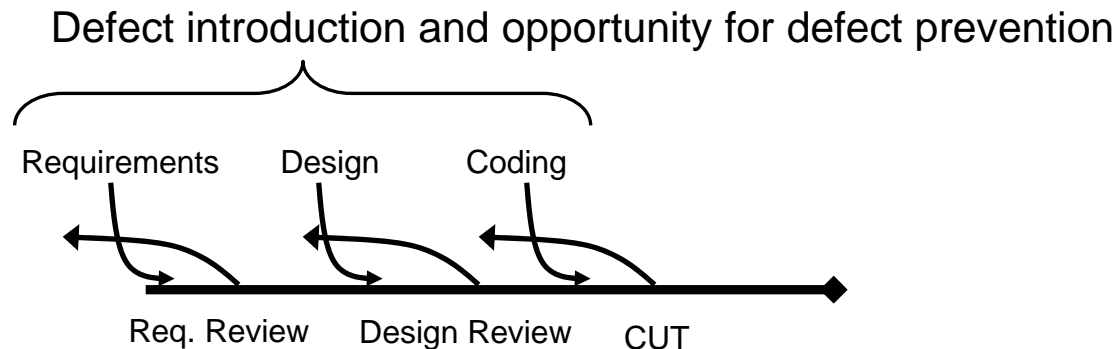
- **Imagine trying to move brick walls after they are build**
- **Worse: imagine trying to move brick walls after they are holding up offices where people are now working!**
- **Software is very similar. Fixing an architecture flaw can mean:**
 - Changing the design of the interfaces and components
 - Rewriting an enormous amount of code
 - Rewriting test cases and test drivers
 - Changing externals or behavior of the product (customer may not be relying on the externals/behavior!)



“Hmm... I think we'll need to move the walls around a bit”

Quick Note on the Importance of Review and Iterations

- **Each of the steps produces a real result that is valuable and can be reviewed.**
- **It is okay (and often required) to revalidate the result of each step (e.g., the requirements document)**
- **Multiple changes and revalidations are okay!**
- **Get the test or QA teams involved early to ensure the architecture and design will be easily tested**
 - QA will understand what broke in the past and what can be easily tested
 - The cheapest way to fix a defect is to identify it before it is written in code



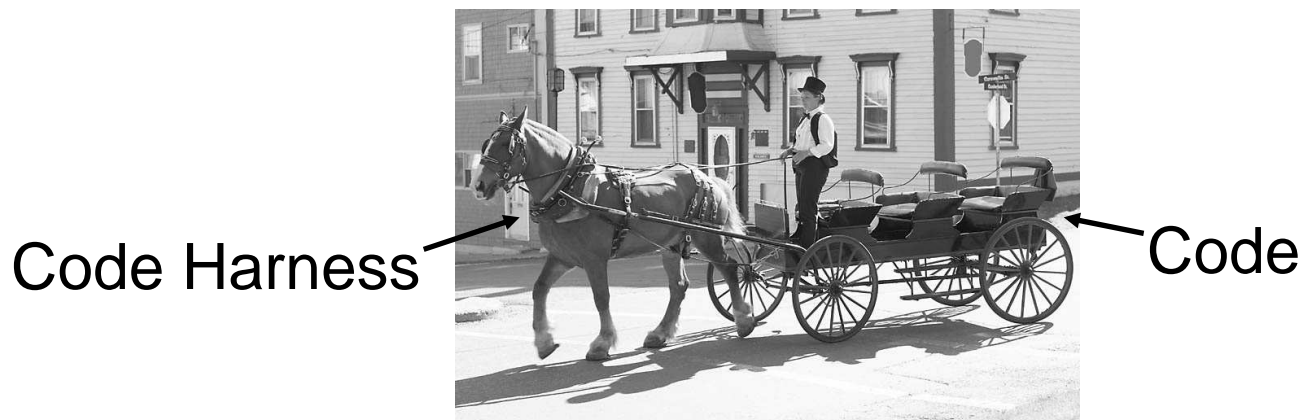
Step #4: Write the code, carefully

Write the code

- **Hire passionate, well educated developers**
 - Developers that are not passionate and educated about quality will not write quality code
 - Example: do your developers understand what ANSI Aliasing Compliance means?
 - Do they care enough to find out?
- **Make sure the developers are aware of the original requirements and the information about the users**
 - Again... developers are not the typical end users
- **Make sure the developers are aware of the overall architecture and design**
 - Developers need to know where and how their code will fit into the larger architecture
- **Make sure the developers are well informed of changes and that they communicate often**

Test the Code as it is Written

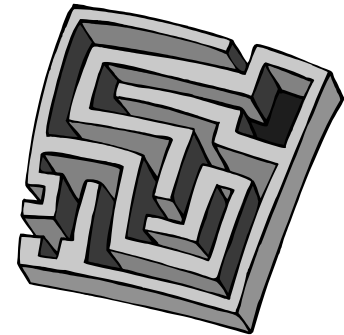
- **Ideally, the code should be tested as it is written**
 - The code can often be tested on its own and without a major infrastructure or a specific state
 - The code can be tested early on when it is still cheap to find/fix defects
 - A “code harness” is often needed to drive the code, might as well write it with the code itself!
- **Integrate the code harness into the functional tests and regression tests**
 - Don't throw away a good thing!



Assertions

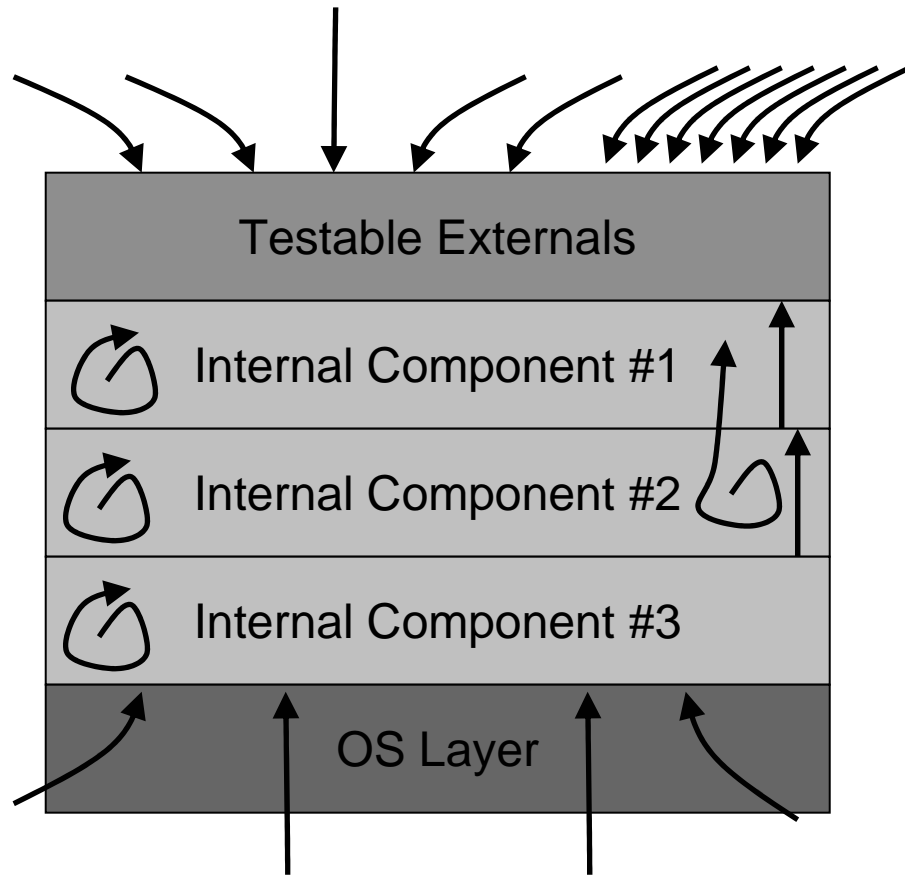
- **Are assertions considered “unit test”? Well...**
 - They can ensure the proper execution of the code for internal test and development environments
 - Assertions can set strict boundaries for how the code should run
 - They are easy to add and can be very effective
 - Add an assertion to ensure a loop boundary is correct (and will stay correct)
 - The conditions will be automatically “tested” when the code paths are run
 - In theory, you could design an entire software product based only on assertions!

- **Assertions however, do not:**
 - Force specific conditions or code paths!
 - Ensure the code is tested up front!



Step #5: Test

7 Types of Testing



- Stress testing
 - Positive Externals
 - Negative Externals
- Inter-function Logic
 - Intra-function Logic
 - Internal Error Paths
 - OS Errors (errors from below)

“Internal Tests”

6 Types of Testing (excluding stress)

Positive Externals

- Standard FVT and External Testing
- Includes integration testing, migration testing, customer scenario testing, etc
- Does the function work as designed?
- Easy to automate

Negative Externals

- How does it react to incorrect externals?
- How does it react to a failure of a needed external product/component?
- Does the function *fail* as designed?
- Is it externally robust?
- Easy to automate

Inter-Function Logic

- Logic that requires more than one function
- Can test with debuggers but it is time consuming
- Race conditions (timing holes)
- Often includes complex logic

OS Errors

- Errors that come from the operating system
- Examples: Out of memory, out of disk space, etc
- Expensive to test manually
 - Need to fill up file system for every potential code path that could be affected
 - Exhausting memory for every code path would effect others on the system

Intra-Function Logic

- Logic that is limited in scope to a single function
 - Loop boundary conditions
 - Missing “break” in a switch statement
 - Local error handling
 - etc

Internal Errors

- Errors that are internally generated
 - `if (foo > 5) rc = ERROR_CODE ;`
- Internal errors are not “OS errors” since they originate from the product’s internal code

Stress testing

- **The goal of stress testing is to drive:**
 - Boundary conditions, complexity, timing conditions
- **Make sure you are finding these types of defects or the previous test phases may not be finished!**
 - Finding unit test or function test type defects during stress testing is not cost effective



Big, Expensive Hardware is Required

The importance of testing in order

1. Unit test

- At the same time as coding if possible
- Unit testing is inexpensive since it requires only one person and the current development machine

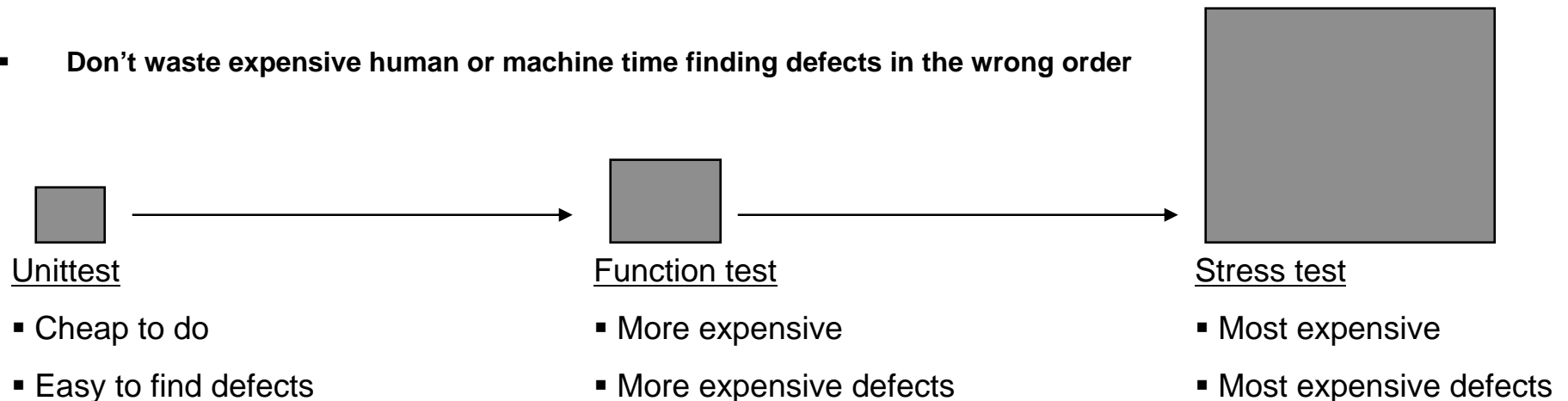
2. Externals and internals testing (positive and negative). A.K.A. Function testing.

- Unit testing should be complete
- Functional tests are important next to ensure the components and interfaces work (and fail!) as designed
- More expensive than unit test since more machines and other people are involved in finding defects

3. Solution level and stress testing

- Function test should be complete (or at least far along)
- Most expensive since large machines and many dedicated people are involved.

- **Don't waste expensive human or machine time finding defects in the wrong order**



Cheating: Flushing out Defects with Tools

- **Static code analysis**
- **Run time analysis tools**
- **Other tools!**
- **Tools like these can be a cheat and effective way to flush out defects**

Static Code Example:

```
-- ERROR2 /*operating on NULL*/ >>>ERROR2_sqlolngs_8cb55ba8010e02
"acsu.C", line 457: invalid operation involving NULL pointer `last_write_ptr'
ONE POSSIBLE PATH LEADING TO THE ERROR:
"acsu.C ", line 394: the if-condition is false
"acsu.C ", line 417: conjunct is true
"acsu.C ", line 417: entering the loop for the 1st time
"acsu.C ", line 420: the if-condition is true
"acsu.C ", line 423: the if-condition is true
"acsu.C ", line 447: the if-condition is false
"acsu.C ", line 455: the if-condition is true
"acsu.C ", line 457: invalid operation involving NULL pointer `last_write_ptr'
```

Limit testing and find the right defects

- **You have:**
 - 15 Operating systems
 - 30 Levels of patches on average for each OS
 - There is 4-6 types of computer systems for each operating system
 - There are 100s of disk, network and memory combinations possible
 - Each hardware module has its own set of firmware patches

- **You have one test team**
 - What do you do?

- **Validate and test specific solution combinations!**
 - Pick a combination and test/validate and advertise it!
 - This is the Dell model for reliability!

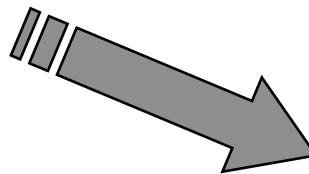
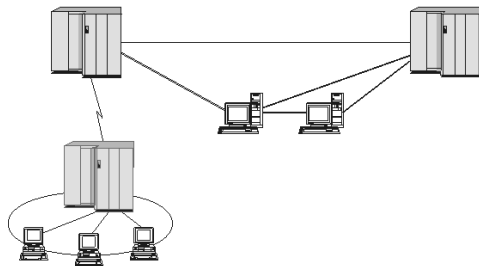
- **Let your users test!**
 - Release alpha or beta versions
 - For smaller projects, let them test parts of the solution

Automated Regression Tests

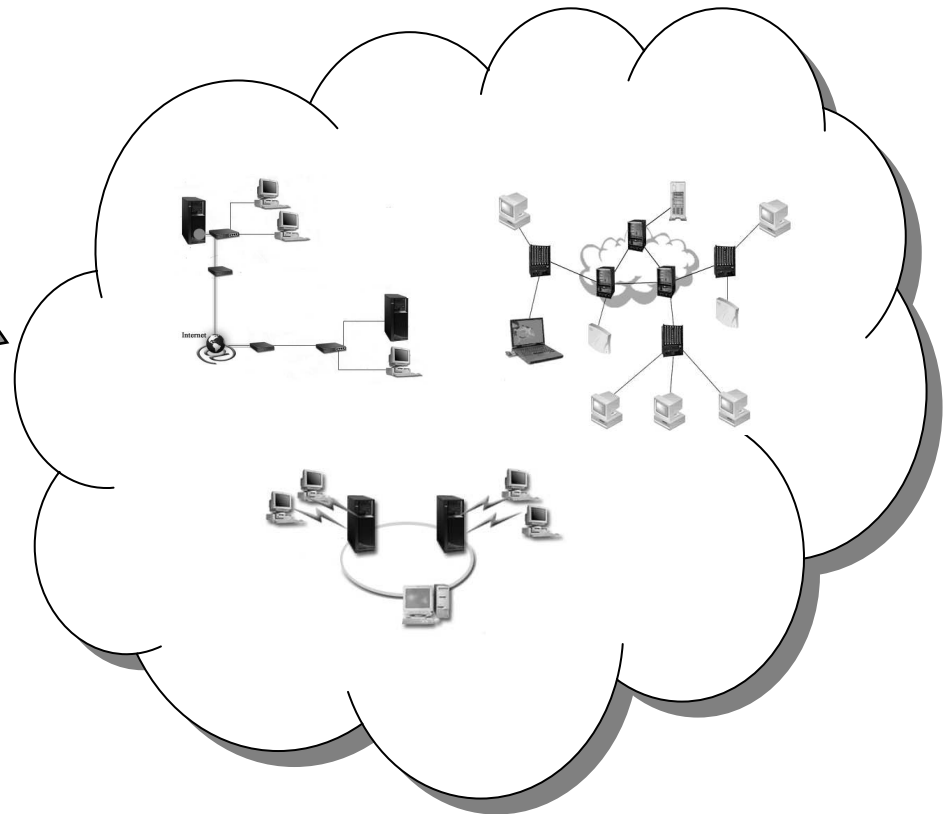
- **6 months (or a year, or 5 years) down the road, a change may break a feature that used to work**
 - Without automated regression tests, new defects would not be noticed until found by users!
- **This is extremely important for any long lived software**
 - Should be built using the function tests to ensure all function works as expected
- **It is also possible to automate unit testing in some cases**
- **The more automation the better!**

Client Environment/Workload Scenarios

Client Workloads



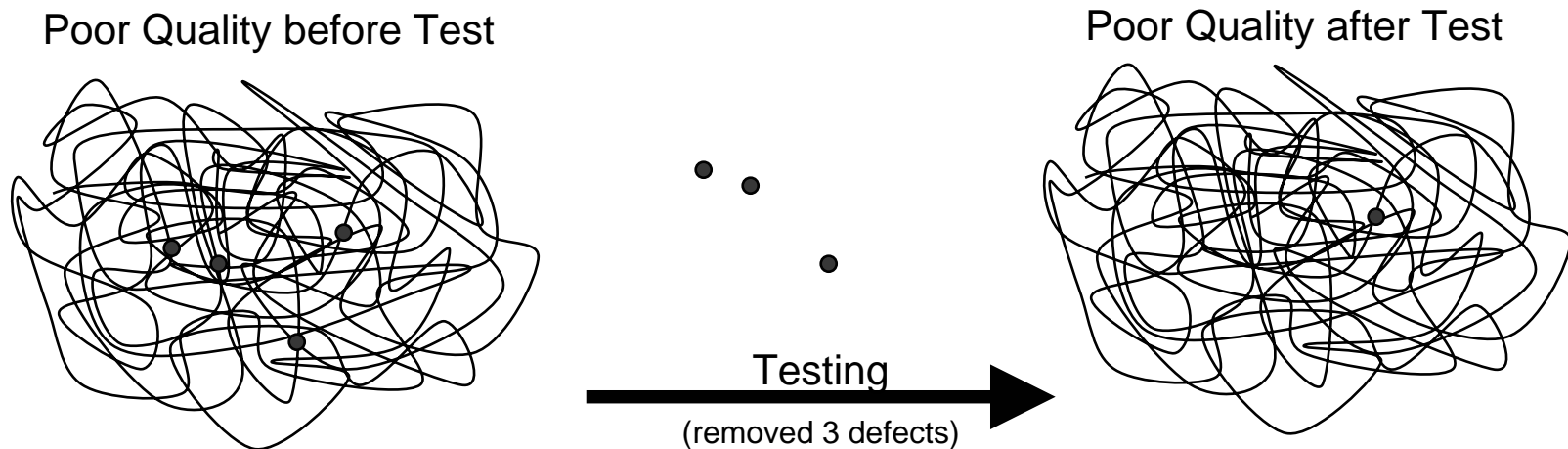
Test Environment



- Flush out defects based on real workloads:
 - Common systems, common usage, common environments
- In some cases, real customer data
- Move forward in different directions ahead of customers

Can you test in Quality?

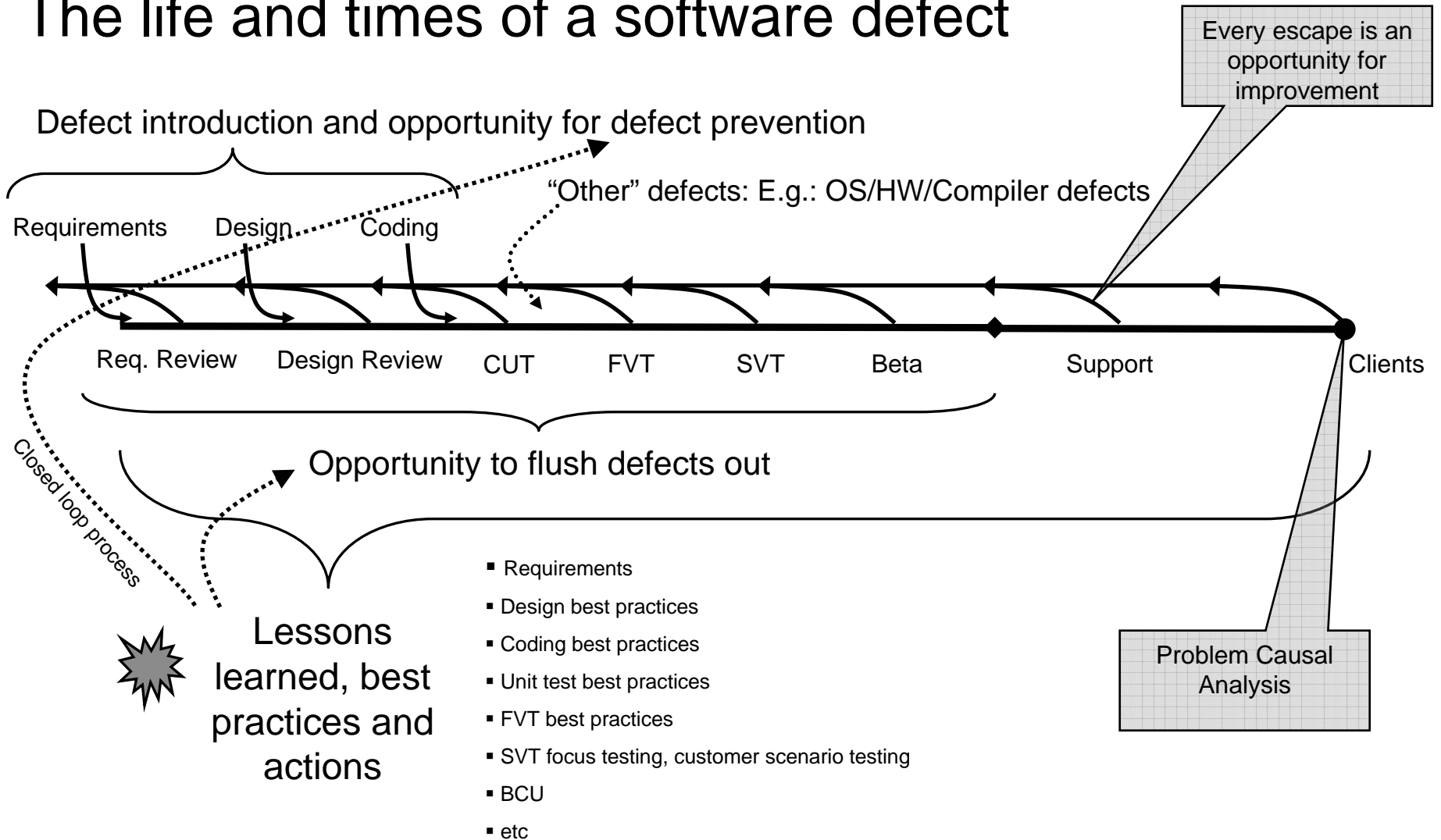
- **Testing removes defects which is different than adding in true quality**
 - Architecture, design and code may still be poor and difficult to maintain
- **Testing happens too late to have a major impact on architecture and design**
 - It also doesn't clean up poorly written code!



Step #6: Learn from mistakes and repeat

(Improving Quality of an Existing Product)

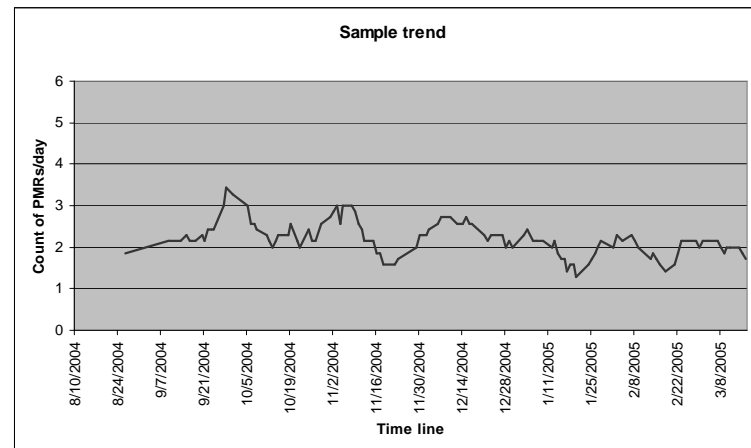
The life and times of a software defect



Measuring success for any problem type (trends for any problem type)

- Examples of trends
 - Memory leaks over time (cost, number, time to diagnose)
 - Error message type problems
 - Timing related problems

- **Work to improve and then measure results!**



**** Measurability is part of the closed loop of information ****

Improving Quality: Closed loop process

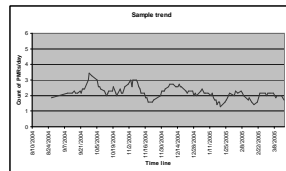
Track/Identify



Opportunity to flush defects out

- Lessons learned, best practices and findings**
- Design best practices
 - Coding best practices
 - Unit test best practices
 - SMT best practices
 - SMT focus testing, customer controls testing
 - RCU
 - etc.

Results/Measure



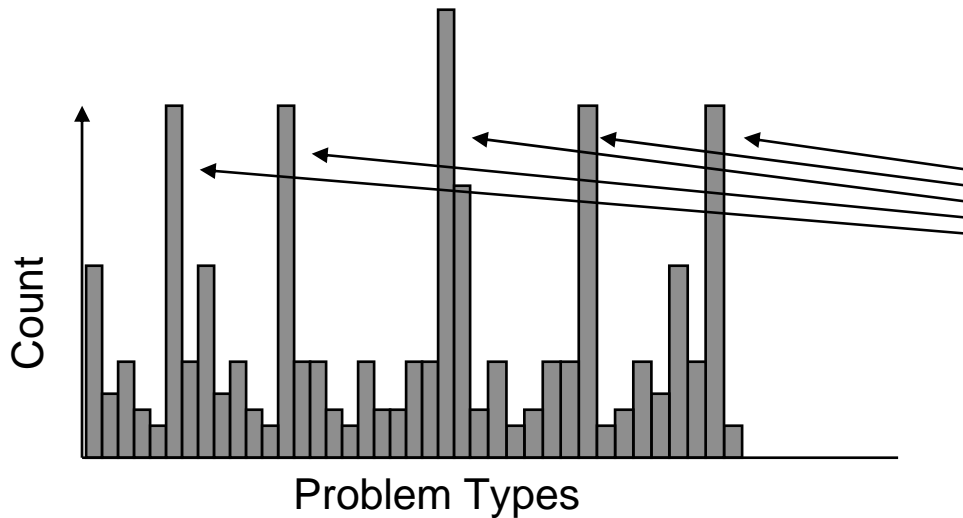
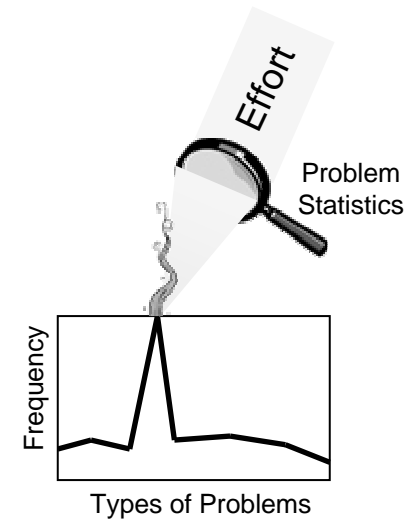
Analyze/Prioritize

Learn/Act



Focusing: 80/20 rule

- Graph of spread effort vs focused effort
- Find the right defects (there could be too many to fix!)
- Focus on the quality of the right areas
- Reduce to number of support configurations



Focus only on the top problem types

- By cause
- By product area
- etc

A Quality Architect Kit for Success

- **A good understanding of the “whys” of software quality**
- **Intimately understand the problem space**
 - What is Quality?
 - Customer problem analysis (tools and schema)
 - What, specifically, are the quality problems
- **Live by the 80/20 rule. You can only do so much... make sure it is focused on the right things**
- **Master the art of negative thinking!**
- **Learn to become a cultural leader**
 - A good book: “Leading Change”, Kotter
- **Persistence!**

Summary

- **Quality is critically important to software**
- **There are different types and levels of quality required for different applications and users**
- **The six steps are designed to**
 - Find defects early
 - Spend the least amount of effort on the right things for Quality (80/20 rule)
- **It is okay and usually required to iterate**
 - An iteration can be for one step
 - An iteration should definitely include learning from real field experience with a product

- **Lastly, Quality directly correlates to revenue growth and business costs!**

Thank
YOU




```
ERROR: undefined
OFFENDING COMMAND:

STACK:
```