Software Testing

How to Write Untestable Code

Or

How to Be a Horrible Unfriendly Dev

Writing Terrible Code

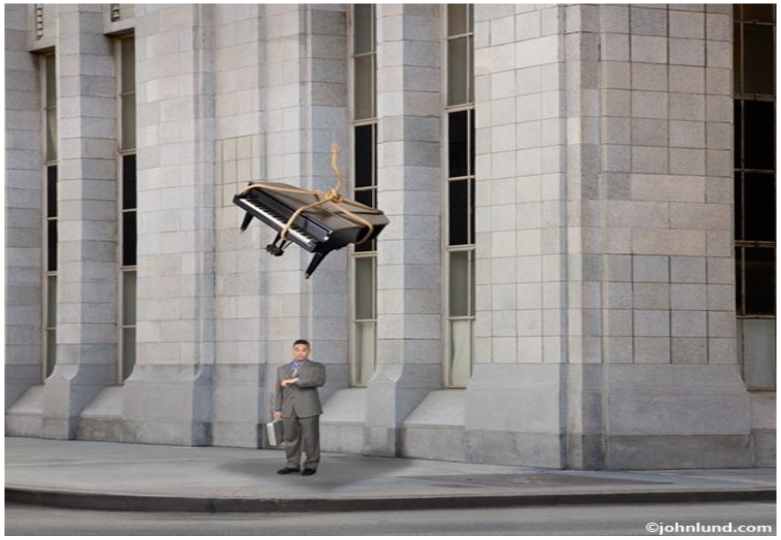
- Or, how to:
  - Write code that's difficult to test
  - Write documentation that's unhelpful for testing
- Which:
  - Makes the lives of others difficult, painful and expensive.
  - May keep you employed for life.
  - Won't win you friends.
  - Will enrage and sadden your professors.

Don’t Do What Donny Don’t Does

- An excerpt from “The 10 Do’s and 500 Don’ts of Knife Safety”.

General Principles

- Understand your audience: the maintenance programmer, tester.
  - They're relying on your code and documentation to evolve the system or test what you've done.
- They must be stopped.

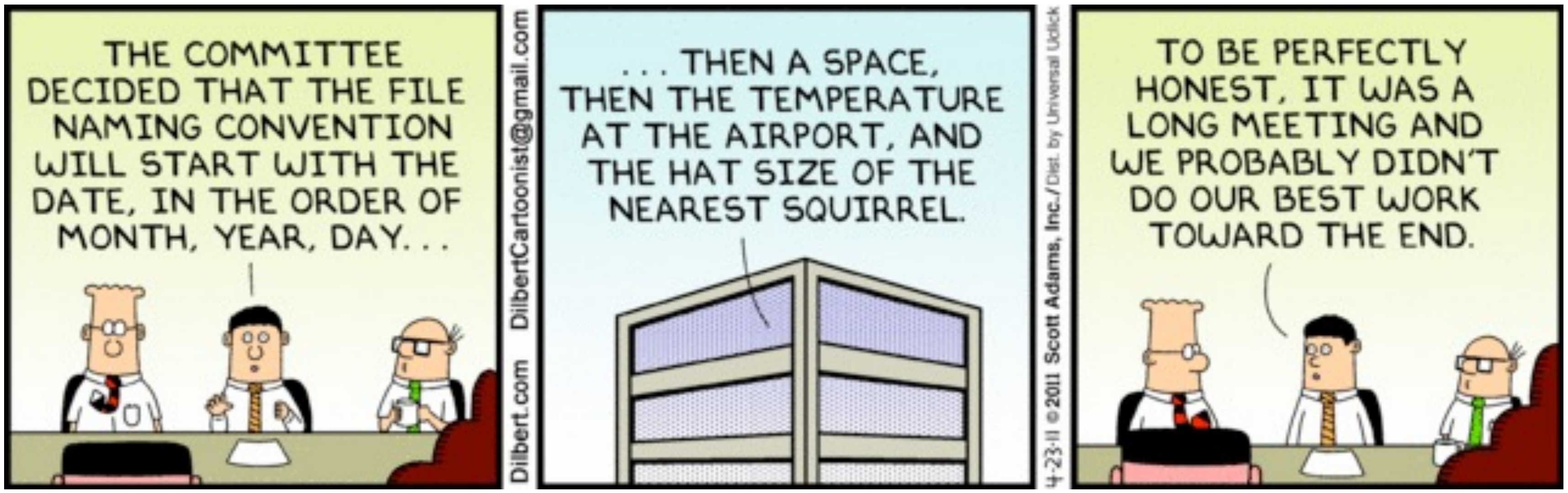


General Principles

- Your audience (the “victim”):
  - Has minimal time to read your code and documentation.
  - Needs to quickly find where to make changes.
  - Wants no unexpected side effects as a result of making changes or running tests.

- You:
  - Must prevent them from understanding the `big picture’.
  - Make it impossible to safely ignore anything!
  - Conceal the innovative ideas in your code.

Conventions and Naming


THE COMMITTEE DECIDED THAT THE FILE NAMING CONVENTION WILL START WITH THE DATE, IN THE ORDER OF MONTH, YEAR, DAY. . .
. . . THEN A SPACE, THEN THE TEMPERATURE AT THE AIRPORT, AND THE HAT SIZE OF THE NEAREST SQUIRREL.
TO BE PERFECTLY HONEST, IT WAS A LONG MEETING AND WE PROBABLY DIDN'T DO OUR BEST WORK TOWARD THE END.
Dilbert.com DilbertCartoonist@gmail.com
4-23-11 © 2011 Scott Adams, Inc./Dist. by Universal Uclick

Conventions

- Coding and documentation conventions are a programmer and tester's friend.
  - They help programmers focus; ignore irrelevant details; and optimize code and documentation reading.
- So screw your victims over by, every now and then:
  - Subtly violating conventions.
  - E.g., slight change to in-house naming conventions (name a field using a type convention)
  - E.g., Violating information hiding principles (random public attribute)
- It forces testers to read everything.

Naming

- You can use naming of entities to confuse, mislead and slow down testers/victims.
- Proper use of names leads to self-documenting code, and better messages from testing failures
  - *This must not happen!*
- Some suggestions:
  - Use single letter variable names; it makes it impossible to easily search for them and test failures are hard to parse.
  - Buy a baby name book and choose your names from there.
  - Pick easy to type variable names: **`fred, adsf, aoeu,`**…
  - Mis-spell creatively! **`Set_pront, honour, honor`**…

More naming evil

- Use generic and abstract names.
  - `It, everything, data, stuff, model, doStuff(), test`
- Use acronyms. Don’t bother to define them – that slows you down.
- Your thesaurus can give you hours of fun.
  - Use **`display, show, view`** variously in your code; they’re all subtly different – but why?
- Reuse names in different scopes for different purposes.
  - Inner functions/inner classes give you plenty of opportunity to reuse names and be a bastard.
- Choose names that masquerade as math:
  - **`openParen = (slash+asterisk)/equals`**;

Self-Documenting methods()

- The name of a method should not provide any kind of documentation.
- You can use artificial names; you can also be evil.
  - **`isValid(int x)`** is just too meaningful
  - **`isValid(int x)`** should:
    - Test that **`x`** is valid
    - And then wrap **`x`** in an object, serialize it and store it in a database.
    - Testing for side effects is fun!
- Some annoying languages enhance the documentation of methods with pre and postconditions.
  - These should be avoided as you can give too much away with pre/post, and it makes it easier to test!

Funny Method Names (all true…)

- `isServerOn()` and `isServerOnFire()`
- `ICantBelieveIUsedGoto`
- `TestBashThatBadBoy()`
- `die_you_gravy_sucking_pigdog()`
- `IDontKnowWhatThisCodeDoesButItIsUsedTwentyTimesInThisClass(int i)`
- `_setBasesHoldOnToYourButts()`
- `ZombifyActivationContext`
- `pleaseRefrainFromDoingSoInTheFutureOkayThanksBye`

Hiding Your Evil

Camouflage!

- A compiler can process code and documentation at a much more fine-grained level than paltry humans.
- This can lead to hours of fun.

```
for(j=0; j<a.length(); j+=8){
   total += array[j+0];
   total += array[j+1];
   total += array[j+2]; /* Main body of
   total += array[j+3]; * loop unrolled
   total += array[j+4]; * for greater speed
   total += array[j+5]; */
   total += array[j+6];
   total += array[j+7];
}
```

Camouflage!

- A compiler can process code and documentation at a much more fine-grained level than paltry humans.
- This can lead to hours of fun.

```
for(j=0; j<a.length(); j+=8){
   total += array[j+0];
   total += array[j+1];
   total += array[j+2]; /* Main body of
   total += array[j+3]; * loop unrolled
   total += array[j+4]; * for greater speed
   total += array[j+5]; */
   total += array[j+6];
   total += array[j+7];
}
```

Camouflage! (2)

- Many languages permit macro definitions.
  - These often get mistaken for documentation.
  - Use this to your advantage!

```
#define   a=b  a=0-b
```

- Make sure your variable names have no relation to labels used in GUI/display.
  - E.g., attribute named `postcode`, on-screen label zip.
  - It makes the program's conceptual model more complicated (and testing harder).

Documentation-Specific Evil

"Incorrect documentation is often worse than no documentation."

- Bertrand Meyer, Chair of Software Engineering, ETH Zurich.

- Because a compiler ignores your internal documentation, you can screw your victims over outrageously by lying and confusing them.

  – It's fun and a profitable way to spend your time.

Key Principles

- Lie in the comments.
  - Well, maybe not actively, but at least don’t keep your comments up to date with the code.
- Document the obvious.
  - `/* add 1 to i */`
  - Never go so far as to document the intent of a method or package.
- Document how things work rather than why.
  - If there’s a bug, the tester will have no clue!
  - Enormous victory!

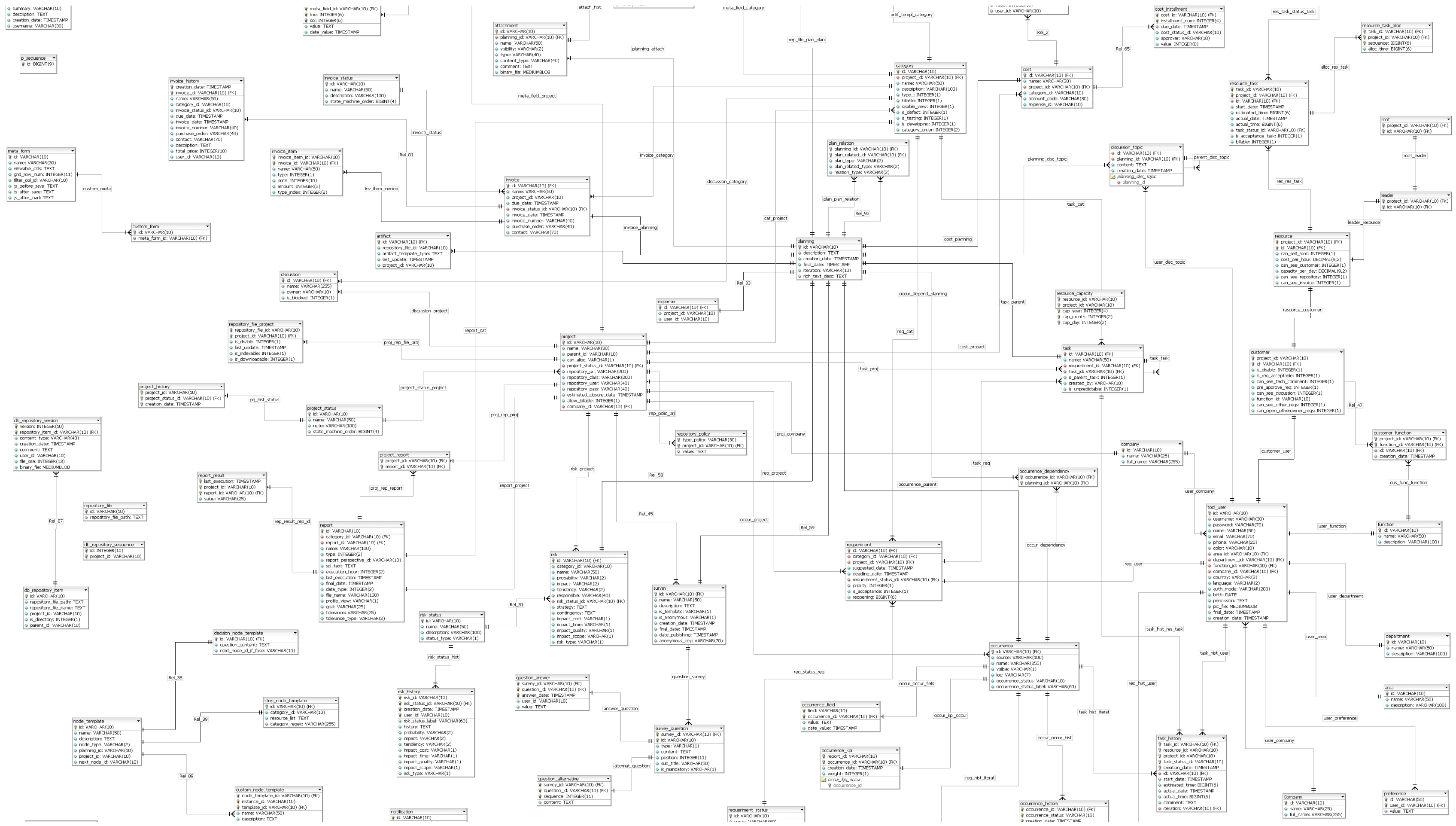
Avoid Documenting the Obvious

- Suppose you are building a game.
- Suppose you wanted to add a new command to the game.
- Make sure there are at least 100 places in the code that need to be modified in order to add a new command.
- AND DON’T DOCUMENT WHERE THESE ARE!
  - Only a genius who thoroughly understands your code should have the right to change it.
  - And just think about the additional testing needed in this case!

Design Documents

- Write an extremely detailed design document, with incredibly complicated and detailed UML diagrams.
- It should describe each and every step in all the very complicated algorithms.
  - Preferably using sequence diagrams or statecharts, which are very easy to test.
- The more detailed the document is, the better.
- The ideal situation is one where the detailed design is so detailed, you have at least 1000 numbered sections.
  - This will be exquisitely difficult to maintain.
- Your maintainers should only be able to find two contradictory out-of-date documents.

Do it like this!



Units of Measure

- We heard about a famous failure – the Mars Polar Lander.
  - The failure came about as a result of two different teams using different units of measure (Imperial vs Metric).
  - This was good evil.
- So: never document the units of measure of any variable, parameter, input or output.
- Never document the units of measure of any conversion constants, or how values were derived.
  - It may even be fun to document some incorrect units of measure.
  - Or make up your own, and name it after yourself
  - Try testing that!

Insult People in the Code

- Do this to discourage the maintenance folks.

```
/* Optimized algorithm for data analysis.
    This is way too complex for those idiots at Stupid
    Maintenance, Inc., who would probably do it 100
    times slower and use 100x as much CPU. Those stupid
    idiots.
*/
```

- If you are feeling particularly malicious, put the offensive comments in the critical parts of the code.
  - ➢ Then, management may break the code while sanitizing it before sending it out for maintenance

Maintenance-Specific Evil

- Put data that you know will definitely grow too large to store in memory into an array for now.
  - It’s not your problem what happens in the future.
  - The maintenance programmer will have the delightful task of converting the array to database tables later.
- Similarly, put really small files into databases.
  - The maintenance programmer will have the delightful task of converting them into arrays or hashmaps later, for better performance.
- Consequence: your test case become unnecessarily complicated as a result!

Never Test Inputs

- Assume that the user who enters data is perfect.
  - They’ve already shown good taste by using your program.
- Never test input at source.
  - It shows you are a team player who fully trusts everyone.
  - It also means you believe in the equipment of your employer.
- Also, it means that your input handling code is short, simple, and blindingly fast.
- It also gives the programmers and maintainers who use the input something else to do.
  - We wouldn’t want them to get bored.

Never Assert()

- Asserts are a form of documentation and help with writing test cases.
  - Of method interfaces.
  - Of code.
- Get rid of them!
- Obviously they slow down your code.
- They might also give some hints to the maintainers.
- They could, at worst, turn a three-day debugging party into a ten minute one.

Information Hiding Sucks

- A class’s public API is claimed to be a form of documentation.
  - More documentation is always better, right?
- So make every attribute and method public!
  - After all, someone might want to use it.
- Once it’s public, it’s tricky to make it private.
  - Fun, too: you break lots of code!
- Also, a class with only public features has the pleasing side-effect of obscuring what it’s for.

Ultimate Evil Behaviour

- OO programs are meant to be self-documenting.
  - Class/object names, method names, private/public etc.
- "Any design problem can be solved by adding a further level of indirection."
- Decompose your Java code until it becomes next to impossible to find a method that actually updates program state.
- Arrange for all such methods to be activated as callbacks that come by traversing data structures containing pointers to every method in the program.
- You could even trigger the traversals as a side-effect of garbage collection!

Be a Hoarder

- You've written, tested and documented that method?
  - Why let it go to waste?
  - Someone might need it at some point.
  - Sure, requirements and the program have changed since you wrote the method, but they could change again, and you don't want to have to reinvent the wheel.
- Make sure the comments on these old, creaky methods are suitably cryptic.
  - `/* Touch this and you will be crushed into a cube */`
  - Anyone maintaining the code should be terrified.
  - You might even hesitate to test it!

Global Variables: Just Like Magic

- Don’t use exception handlers.
  - Idiots claim they also provide documentation and more maintainable code, and help structure tests.
  - Well, they’re idiots.
- When an error occurs, set a global variable.
- Then, make sure that every loop in the program checks this global variable and terminates if an error occurs.
- Single points of failure management are just so much easier to maintain and understand, right?

Code Layout Counts!

- So use semicolons whenever they are syntactically permitted, e.g.,

```
if(a);
else;
  {
     int d;
     d=c;
     }
     ;
```

- And why use one semicolon when nine will do?

Convert Indirectly!

- Conversion is complicated and needs proper documentation and testing.
  - Modern languages provide conversion mechanisms that do exactly what you need.
- To convert a double to a String, go circuitously.

  `new Double(d).toString()`
- Instead of **`Double.toString()`**
- Of course you can be far more circuitous than that (I wasn’t really trying…)

Numeric Literals

- If you have an array with 100 elements, hard-code the literal 100 in as many places as you can.
  - You can make it like a treasure hunt to find it and change it.
- Never, ever ever use a static final named constant.
- Also, try to disguise the fact that 100 is an important value, by (for example):
  - Testing a==101 instead of a>100
  - Testing a>99 instead of a>=100
- This works great if you have two or more arrays of length 100.
  - If the poor maintainer has to change the length of one of them, they will have to check everywhere and test everywhere

Exceptions

- Annoying.
- Properly written code never fails.
  - And you are awesome.
- Ergo, exceptions are unnecessary.
- Subclassing exceptions is for incompetents who admit their code will fail.
- So:
  - Greatly simplify your code by having a single try/catch (ideally in **`main`**) that calls **`System.exit()`**.
  - Stick a standard set of throws on every method header, whether they throw an exception or not.

Toilet Tubing

- Never allow the code from more than one method to appear on screen at once.
- Use the following handy tricks.
    1. Blank lines separate logical blocks of code. Each line is itself a logical block. So, put blank lines between each line of code.
    2. Never comment your code at the end of the line; put it above. If you’re forced to comment at the end of the line, pick the longest line of code in the entire file, add 10 spaces, and left-align all end-of-line comments.
    3. Comment blocks at the start of methods should use templates at least 15 lines long with lots of blank lines.

Dealing with Humans

- Don’t pass up an opportunity to show off everything you know.
    1. Create a class named **ArtifactFactory**, then populate it with comments related to GoF design patterns that have nothing to do with object creation.
    2. Create your own distinctive and undocumented EBNF. Be sure there’s no good way of distinguishing a terminal from a nonterminal!
    3. Write your own heap allocator and deallocator. Those idiots who wrote **malloc()**, etc, almost certainly got it wrong.

Miscellaneous Evil

- Standards are for losers.
  - Insist on using the wrong standard for your application.
- Reverse the usual true/false convention.
  - (My personal favourite).
- Ignore the presence of well documented and tested standard APIs; write your own string handlers yourself!
- Create a build order.
  - And don't document or test it.
  - Never reveal that you can treat javac as a class, nor that you can write little bits of Java that find all files and invoke javac.Main directly…

In Conclusion

- Don’t do what I just said.
- Please don’t be evil.
- Write well structured code, use design patterns, make your classes cohesive (do one thing), minimize coupling.
- Write code for testing and maintenance.
- Document your intentions and thinking.
- Document the non-obvious.
- Try to put yourselves in the shoes of the poor doomed person who will be working on and testing your code in two years.
  - That doomed person could be you.