Transforming Mutation Testing from the Technology of the *Future* into the Technology of the *Present*

Paul Ammann
2015 Mutation Workshop Keynote
Graz, Austria

April 13, 2015

GEORGE MASON UNIVERSITY
WHERE INNOVATION IS TRADITION

A Poll

Mutation analysis ought to be a standard part of the toolkit for the typical software engineer.

- **How many of you**
  - **Agree?**
  - **Disagree?**

- **Two goals for this talk**
  1. **Analyze why software engineers don't seem to be convinced**
  2. **Suggest what needs to be done to convince them**

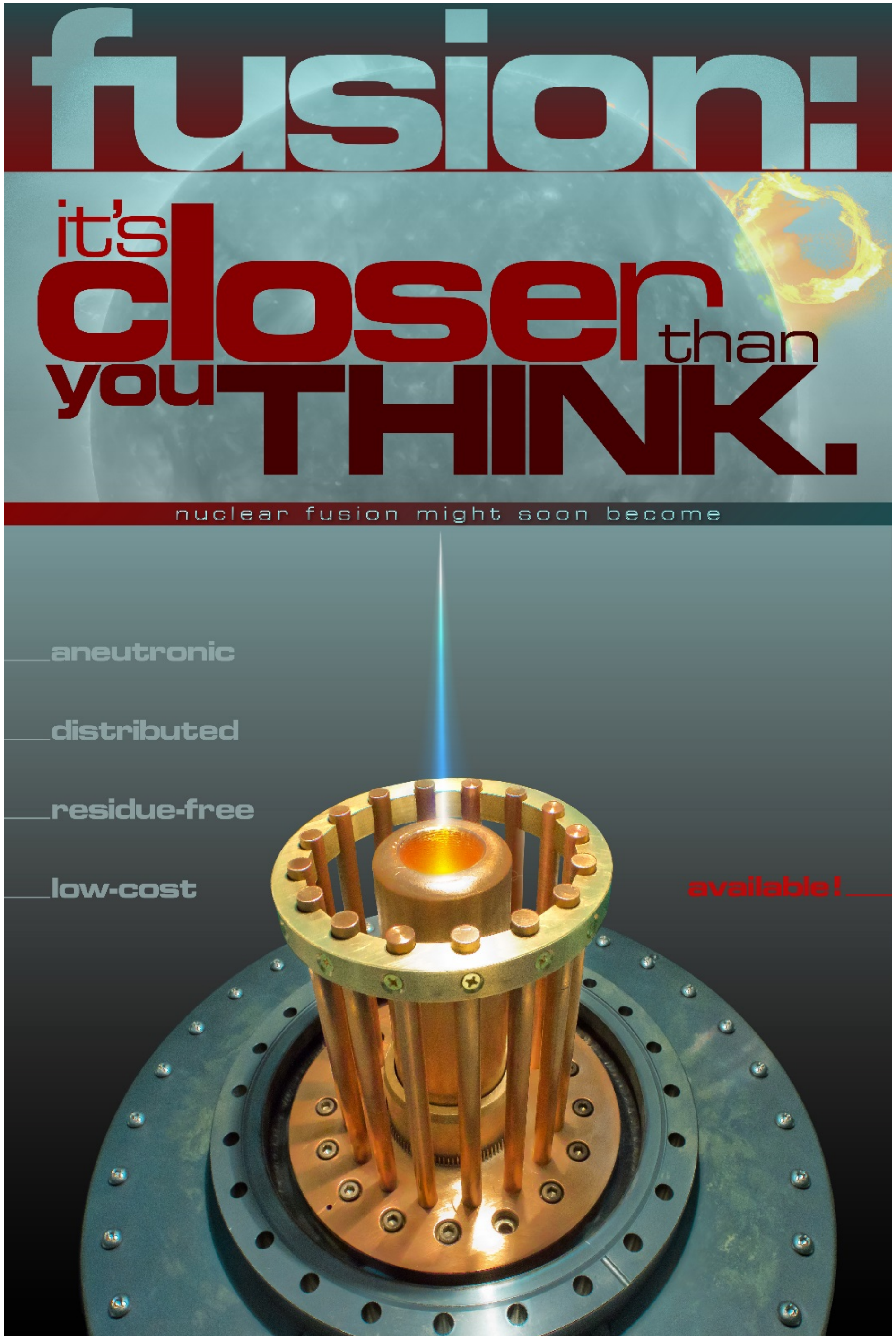
Personal Motivation

For decades, I've told my students:

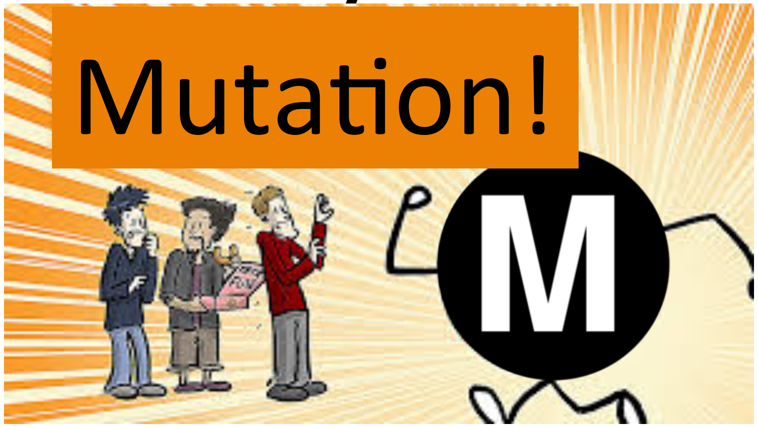
Soon, mutation is coming to your workplace!

Obviously, that hasn't happened

Today's talk: A reflection on why I think mutation is not commonplace



Reality Check!



Mutation!

Nuclear fusion is the energy of tomorrow!

…and always will be

Is mutation testing doomed always to be a technology of the future?

Outline

- What the researcher sees
  - Effectiveness, cost, and equivalent mutants
  - Broad status of current tools, research and otherwise

- What the engineer sees
  - "Mutation 1.0" incompatible with rational development!
  - No evidence the research community has grokked this

- What "Mutation 2.0" might look like
  - More progress in automated input generation
  - The "Personalized Medicine" analogy
  - Lessons from subsumption
  - Requirements for workable mutation

Outline

- What the researcher sees
  - Effectiveness, cost, and equivalent mutants
  - Broad status of current tools, research and otherwise

- What the engineer sees
  - "Mutation 1.0" incompatible with rational development!
  - No evidence the research community has grokked this
- What "Mutation 2.0" might look like
  - More progress in automated input generation
  - The "Personalized Medicine" analogy
  - Lessons from subsumption
  - Requirements for workable mutation

What the Researcher Sees

Good Stuff:
Lots and lots of papers!

Basic Questions:

- Effectiveness
  Are mutation tests
  good at finding faults?
- Cost
  Do faster
  Do smarter
  Do fewer
- Equivalent mutants
  Avoid
  Detect
  Mask

Research Efforts on Effectiveness

- Is mutant detection a good proxy for fault detection?
  - The evidence so far seems to be "Yes!"
    - Andrews *et al*, Duran/Thévenod-Fosse, Just *et al*
- This is a huge deal!
  - Underpins the assertion that mutation is a valid approach to developing high quality test sets
- Empirical results justify reliance on the RIP model:
  - Going all the way to propagation (mutation) yields better tests than just reachability (branch coverage)

Research Efforts on Cost

- Do Faster
  - Schema-based, test prioritization, state-checking
- Do Smarter
  - Weak, higher-order, clustering
- Do Fewer
  - Selective, delete, sampling, subsumption
- All good stuff, and we need more!
  - Execution cost matters
  - But not as much as human cost

Research Efforts on Equivalent Mutants

- Avoid
  - Analysis can identify many equivalent mutants
    - Mutations on dead variables
    - Mutations demanding unsatisfiable constraints
- Detect
  - Reduce percentage of equivalent mutants by analyzing "impact" on state.
    - Great idea; general reduction in equivalent incidence
- Mask
  - 2^{nd} order mutants, where only 1 has to be killed
    - We'll come back to the engineer's perspective on this
- Research success here as well, albeit less
  - But, after all, it is an undecidable problem…

Tool status

- Many mutations systems available online
  - Significant development effort involved
    - 36 tools as of 2009 (MT Repository)
  - Mothra, muJava, Javalanche, PIT, Jester, Jumble, mutate.py, Bacterio, Mutant, Judy, Heckle, Ninja Turtles, Nester, Humbug, MuCheck, Proteum, Mutator, Major…
- Not all mutations tools are equivalent
  - Let’s look at a few

muJava: A Tool Aimed at Researchers

We offer ***μJava*** on an "as-is" basis as a service to the community. We welcome comments and feedback, but do not guarantee support in the form of question answering, fault fixing, or improvements.

statement from muJava website (cs.gmu.edu/~offutt/muJava)

- **Widely used in research studies and classroom exercises.**
- **Implements selective mutation**
- **Has an Eclipse plug-in**
- **But clearly is not a commercial-grade tool**

PIT: A Tool Aimed at Practitioners

> PIT is a state of the art **mutation testing** system, providing **gold standard test coverage** for Java and the jvm. Its fast, scalable and integrates with modern test and build tooling.
>
> ***statement from PIT website (pitest.org)***

- **Mutation testing can yield "gold standard" test coverage**
  - **but only with appropriate mutation operators!**

Consider "if (a < b) S1 else S2"

PIT replaces "a < b" with

"a <= b" (CONDITIONALS_ BOUNDARY)
"a >=b" (NEGATE_CONDITIONALS)
"true" (REMOVE_CONDITIONALS)

All 3 of these mutants are killed with a test where a == b. Which means that the true branch is not tested!

Conclusion: PIT doesn't subsume branch coverage

Would be an easy fix

Javalanche: A Tool Aimed at Both

> With less than 3% of equivalent mutants, our approach provides a precise and fully automatic measure of the adequacy of a test suite -- making mutation testing, finally, applicable in practice
>
> ***statement from Javalanche website (javalanche.org)***

- **Javalanche widely used in research studies**
- **Javalanche recognizes the key hurdle posed by equivalent mutants**
  - **But is 3% good enough?**
    - **i.e is Javalanche really ready for prime time?**
  - **We'll return to this question shortly**

Outline

- What the researcher sees
  - Effectiveness, cost, and equivalent mutants
  - Broad status of current tools, research and otherwise
- What the engineer sees
  - "Mutation 1.0" incompatible with rational development!
  - No evidence the research community has grokked this

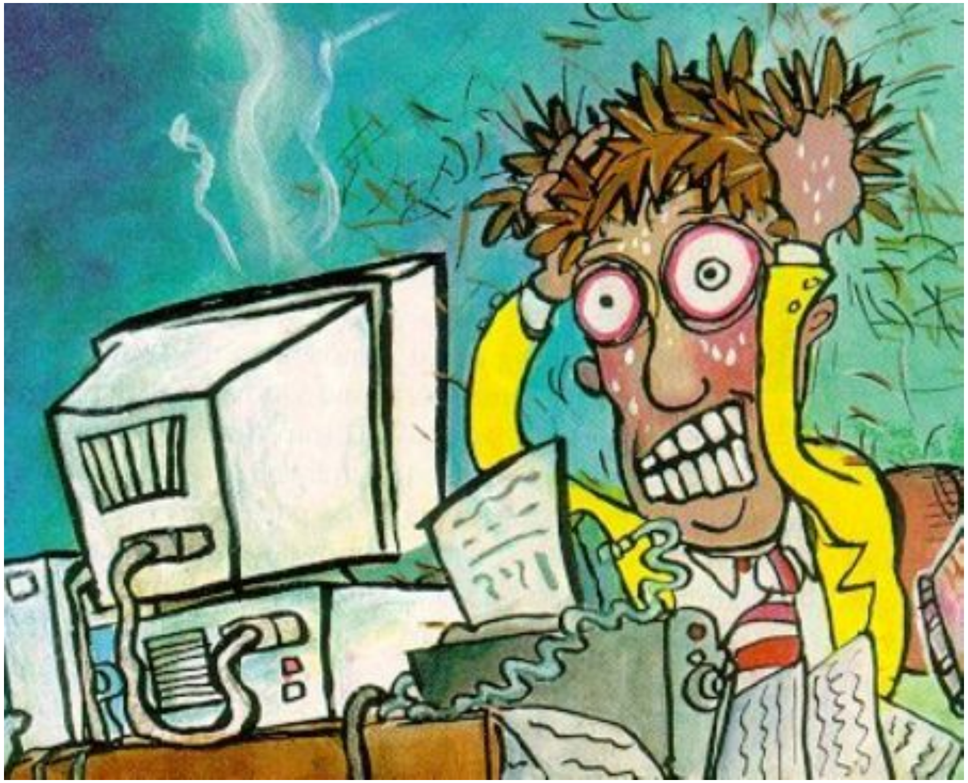
- What "Mutation 2.0" might look like
  - More progress in automated input generation
  - The "Personalized Medicine" analogy
  - Lessons from subsumption
  - Requirements for workable mutation

Diversion: The Goofball Engineer

Mutation testing won't help Dilbert at this company

Good News!
I believe fewer companies think this way these days.

What the Serious Engineer Sees



Mutation testing certainly yields lots of mutants

Does mutation testing help the engineer?

But…

- Many are redundant
- Many are just weird
- Many are equivalent
- The engineer wants tests!

Many Mutants are Redundant (1)

- The research community has known this for years
  - What exactly are all those extra mutants for?
- Redundant mutants are expensive
  - Computational cost
  - Mental overhead
  - Meaningless metrics
    - If my mutation score is 50% after 1 simple test
    - Then am I half done?
- Current approaches generate absurdly many redundant mutants
  - Selective mutation has not solved this problem

Many Mutants are Redundant (2)

- Bad news from (dynamic) minimal mutation
  - SELECT produces many redundant mutants
  - SELECT misses many minimal mutants
- Good news for researchers! More work to do!

Program	Total Mutants	Killed Mutants	SELECT Mutants	Minimal Mutants	SELECT Min MS
print_tokens	4336	3711	138	28	81%
print_tokens2	4746	4042	244	30	57%
replace	11101	8783	499	58	48%
schedule	2109	1838	84	42	65%
schedule2	2627	2132	121	46	72%
tcas	2384	1957	113	61	44%
totinfo	6698	5821	332	19	60%

SELECT produced more mutants than minimal by factor of 5

But SELECT tests only killed about 60% of the minimal mutants

Many Mutants Just Weird (1)

- Consider a “2^{nd} order” mutant designed to reduce equivalent mutants
  - 2 unrelated mutants installed together
  - Developer just has to kill one
- Possible sequence of events
  - Analyze mutant A; give up
  - Analyze mutant B, kill it, declare victory
  - The engineer’s perspective:
    - Wait! Why did I waste time on the “A” part?
    - This isn’t helping me get better tests

Many Mutants Just Weird (2)

- Consider ASR (Assignment Replacement)
  - Original statement: x += 3;
  - One mutant: x >>>= 3;
- From the developer's perspective
  - My code never uses ">>>"!
  - I don't even remember what ">>>" does!
  - Why do I need this mutant?
- Observation: "Mutation 1.0" is insensitive to the specifics of the developer's overall program

Many Mutants are Equivalent (1)

- Consider the engineer's interaction
  - Attempt to kill mutant
    - Invest effort failing
  - Discover mutant is equivalent
    - Remove from "to do" list
  - Produce nothing as a result of this process
    - Negative ROI
- This is, literally, a complete waste of time
  - And hence it is unreasonable to expect engineers to do this
- Conclusion
  - Equivalent mutant analysis breaks the engineering process

Many Mutants are Equivalent (2)

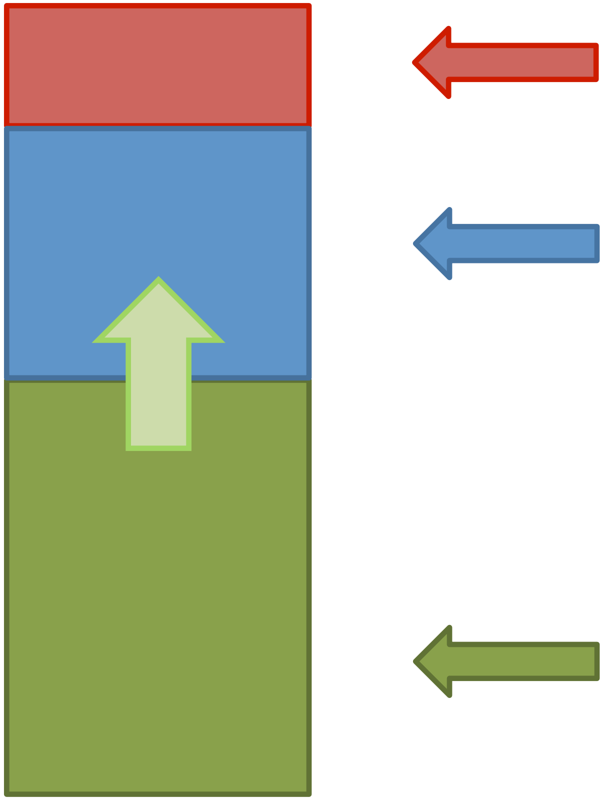
- How much do equivalent mutants cost?
  - Research answer: 15 minutes (or 30 seconds)
    - Doesn’t sound *too* bad…
  - But engineers modify code all the time
    - It’s what they do!
    - Agile processes have simply codified this fact
  - How many times does the engineer need to re-analyze the same equivalent mutant?
    - It may not be equivalent anymore after the next build…
    - So, “15 minutes” is not the answer
- Equivalent mutants are far worse in practice than they are in research!

Many Mutants are Equivalent (3)

- How to reduce equivalent mutant cost?
  - Research Answer
    - Reduce equivalent mutants to small %
- Practical reality
  - In terms of minimal mutants, the % isn’t small!
    - Impressive research result (Javalanche):
      - Reduce equivalent mutant percentage to 3%
    - But what if only 20% of mutants are minimal?
      - 3% of all mutants translates into 15% of minimal mutants
  - The engineer learns nothing from equivalent mutants
    - Yet another undesirable way to refactor my code

Many Mutants are Equivalent (4)

- Identifying equivalent mutants requires both *ability* and *willingness*



Can't identify
Simply too many to analyze, no matter how dedicated

Could identify but won't
More effort than perceived ROI

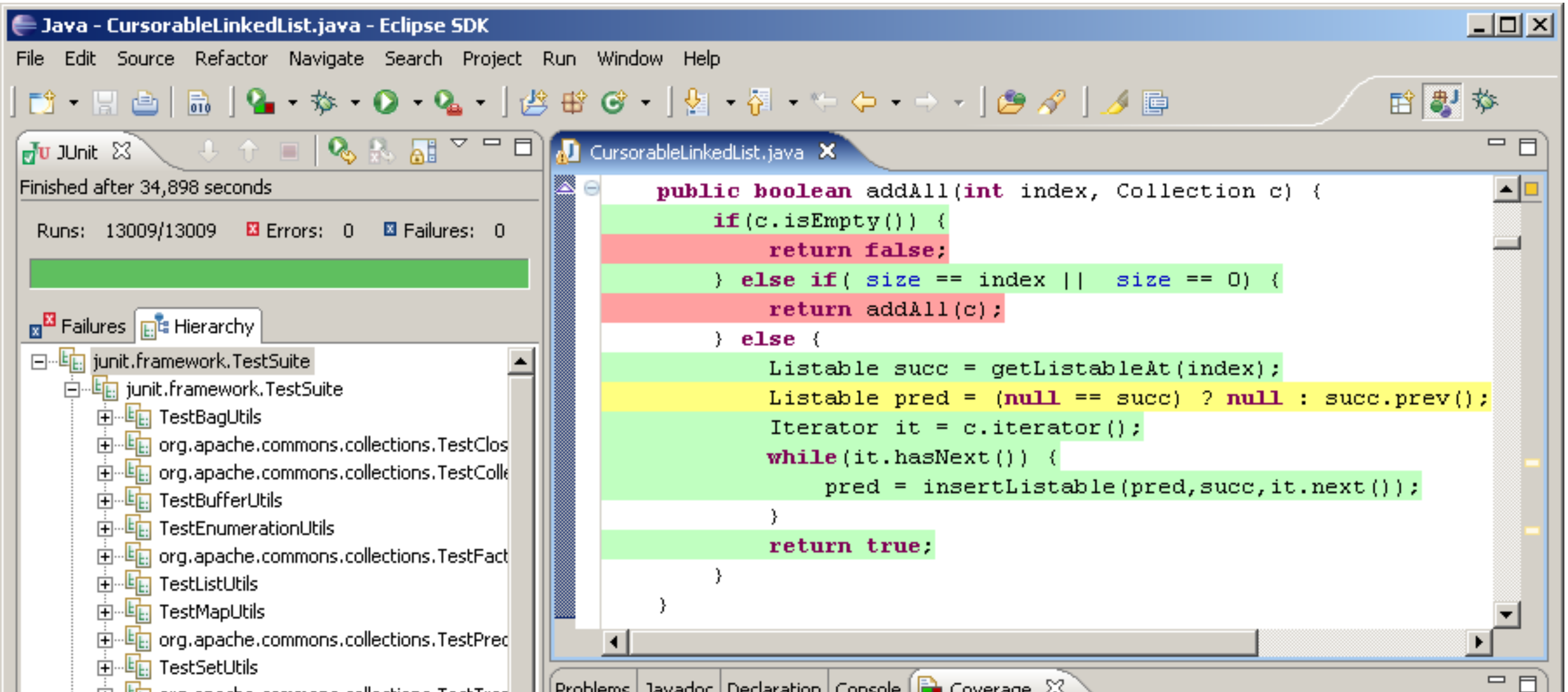
Can identify and does
Sweet spot!
But not clear this set is nonempty

The Engineer Wants Tests! (1)

- This is basic usability
  - Mutants by themselves are uninteresting
  - It's the resulting tests that matter
- Is mutation analysis really part of the engineer's mental model?
  - And, if not, how can it be integrated?
  - Let's look at how other coverage criteria do this

The Engineer Wants Tests! (2)

- Code coverage tools visually display test requirements with color
  - Test/requirement relation very clear in Eclemma
  - How do you display mutants in a meaningful way?



Finished after 34,898 seconds

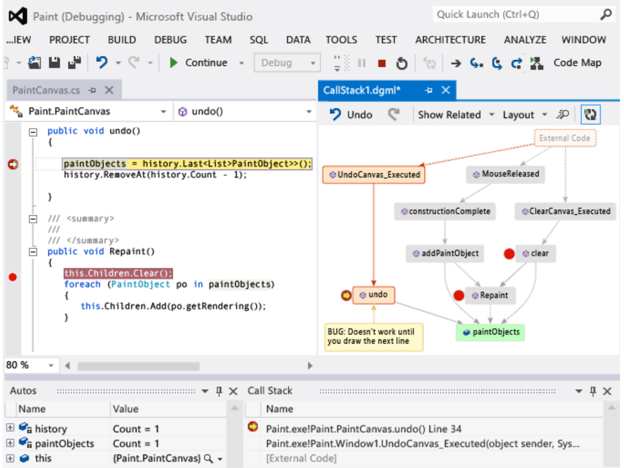
Runs: 13009/13009 Errors: 0 Failures: 0

- junit.framework.TestSuite
  - junit.framework.TestSuite
    - TestBagUtils
    - org.apache.commons.collections.TestClos
    - org.apache.commons.collections.TestColle
    - TestBufferUtils
    - TestEnumerationUtils
    - org.apache.commons.collections.TestFact
    - TestListUtils
    - TestMapUtils
    - org.apache.commons.collections.TestPred
    - TestSetUtils

```
public boolean addAll(int index, Collection c) {
    if(c.isEmpty()) {
        return false;
    } else if( size == index ||  size == 0) {
        return addAll(c);
    } else {
        Listable succ = getListableAt(index);
        Listable pred = (null == succ) ? null : succ.prev();
        Iterator it = c.iterator();
        while(it.hasNext()) {
            pred = insertListable(pred,succ,it.next());
        }
        return true;
    }
}
```

The Engineer Wants Tests! (3)

- Consider the visualization problem a bit more
- Reachability is easy (color works)
- Infection requires visualization of state
  - Can we leverage experience from debuggers?
  - Which variables are affected by mutation on a given test case (or set?)
- Propagation seems harder
  - No idea how to visualize this!
  - Research opportunity!



```
public void undo()
{

    paintObjects = history.Last<List>PaintObject>>();
    history.RemoveAt(history.Count - 1);

}

/// <summary>
///
/// </summary>
public void Repaint()
{
    this.Children.Clear();
    foreach (PaintObject po in paintObjects)
    {
        this.Children.Add(po.getRendering());
    }
```

BUG: Doesn't work until you draw the next line

Name	Value
history	Count = 1
paintObjects	Count = 1
this	{Paint.PaintCanvas}

Name
Paint.exe!Paint.PaintCanvas.undo() Line 34
Paint.exe!Paint.Window1.UndoCanvas_Executed(object sender, Sys...
[External Code]

Summary: What the engineer sees

- Given large set of mutants to kill
  - Most are unimportant
    - But this information is hidden
  - Many unrelated to programmer's mental model
    - Confusing to analyze
  - Many are equivalent
    - Demands continuous manual effort with zero ROI
    - In practice, no idea when the last "real" mutant is killed
  - Connection to tests less direct than other coverage
- Conclusion: Given limited resources, it is rational for the engineer to focus effort elsewhere

Outline

- What the researcher sees
  - Effectiveness, cost, and equivalent mutants
  - Broad status of current tools, research and otherwise
- What the engineer sees
  - “Mutation 1.0” incompatible with rational development!
  - No evidence the research community has grokked this
- What “Mutation 2.0” might look like
  - More progress in automated input generation
  - The “Personalized Medicine” analogy
  - Lessons from subsumption
  - Requirements for workable mutation

More Progress in Input Generation

- Constraint solvers and search mechanisms don't care how weird test requirements are
- They don't get tired of analyzing the same constraints over and over
- They handle equivalent mutants well from an engineering perspective
  - Try for N cycles and then give up!
- Some input generation tools are being used in industry
- Caution: This is harder than it looks
  - Integrating automated input generation with development processes is still a work in progress

Caution: Input Generation Obstacles

- Research community focuses on technical challenges
  - Generating "good" tests (eg Jamrozik et al)
  - Making constraint solvers scale better
  - Making test inputs intelligible to the engineers
- But practitioners face many process issues as well
  - The engineer needs more than inputs
    - Wait! I'm supposed to write assertions?
  - How does this work as code evolves?
    - Tests need to evolve as well
  - How to integrate automated and "real" tests?

The Personalized Medicine Analogy

Contrast:

Selective mutation is effective on a benchmark set of programs. Your program appears to fit the benchmark.

Therefore, use selective mutation operators on your program.

vs.

Here are mutants specifically engineered for your program.

"BUT IF YOU WANT THE REAL LOWDOWN, WE'LL NEED SOME OF YOUR DNA."

Question: What does it mean to engineer mutants for a specific program?

Lessons from APR

- The Automated Program Repair (APR) problem
  - Given a failing program
  - Search a large space of variants
  - Find one that passes all the tests
    - and possibly more has other desirable properties…
- These variants are just mutants
  - Which variants are generated depends on the program
  - Example: replace suspect code with similar code that appears elsewhere in the program
- Observation: APR is already practicing "personalized medicine" for mutation analysis

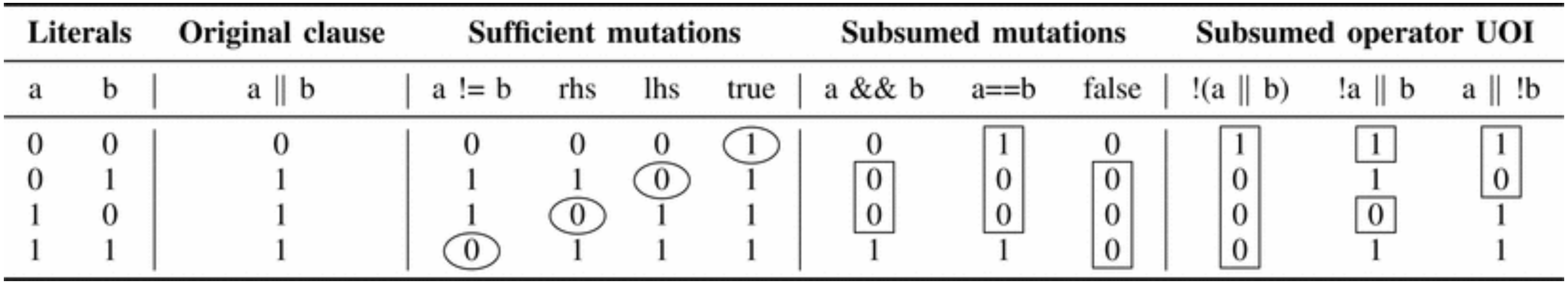
Subsumption: An approach to personalizing mutation analysis

- Personalizing mutation requires understanding the relationship between different mutants
  - Subsumption captures this relationship
- Subsumption comes in two basic varieties
  - Subsumption relations invariant across all programs
  - Subsumption relations specific to individual programs

Invariant Subsumption Relations

- Kuhn noticed that detecting some faults might guarantee detecting other faults
  - Just et al: COR Mutations
  - Lau/Yu Fault Hierarchy: Faults in DNF
  - Kaminski et al: ROR Mutations
- Lots of other possibilities
  - Just need some mathematical structure
  - Some other candidates: regular expressions, SQL
- Let's take a look at COR, DNF, and ROR

COR Mutations

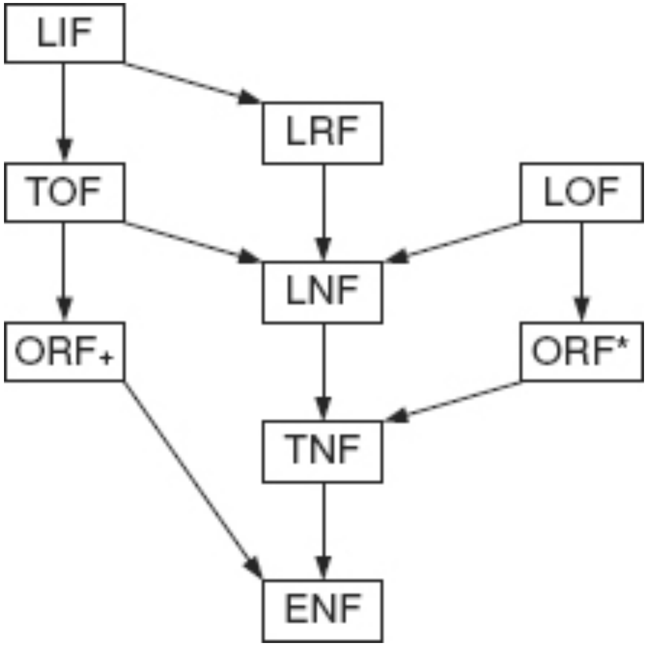


Literals		Original clause	Sufficient mutations				Subsumed mutations			Subsumed operator UOI		
a	b	a || b	a != b	rhs	lhs	true	a && b	a==b	false	!(a || b)	!a || b	a || !b
0	0	0	0	0	0	1	0	1	0	1	1	1
0	1	1	1	1	0	1	0	0	0	0	1	0
1	0	1	1	0	1	1	0	0	0	0	0	1
1	1	1	0	1	1	1	1	1	0	0	1	1

Source: Just *et al*

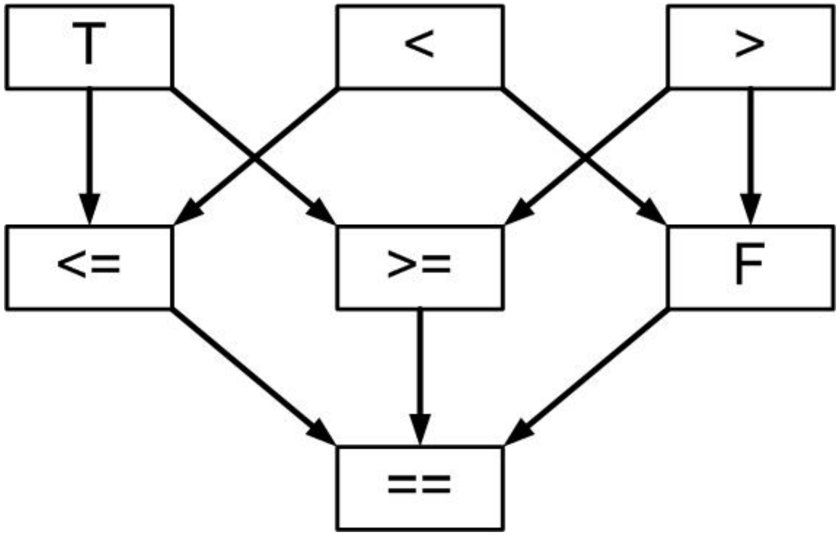
- COR generates 7 mutants
- But there are only 4 possible tests!
- Hence 4 mutants are enough
  - These 4 also subsume 3 UOI mutants
- Conclusion: There is never a reason to generate 10 mutants
  - Just generate 4!
- There is a slightly different table for the && operator
  - Hence, you need a different set of 4 mutants

Lau/Yu Fault Hierarchy



- 9 Fault classes defined on minimal DNF
- MUMCUT tests detect all possible faults
  - MCDC tests only guarantee ENF/TNF detection…
- For mutation implementation, only need a subset
  - Some complexity due to infeasible test requirements
  - Some mutants can be combined in subsuming higher order mutants

ROR Mutations (!= operator)

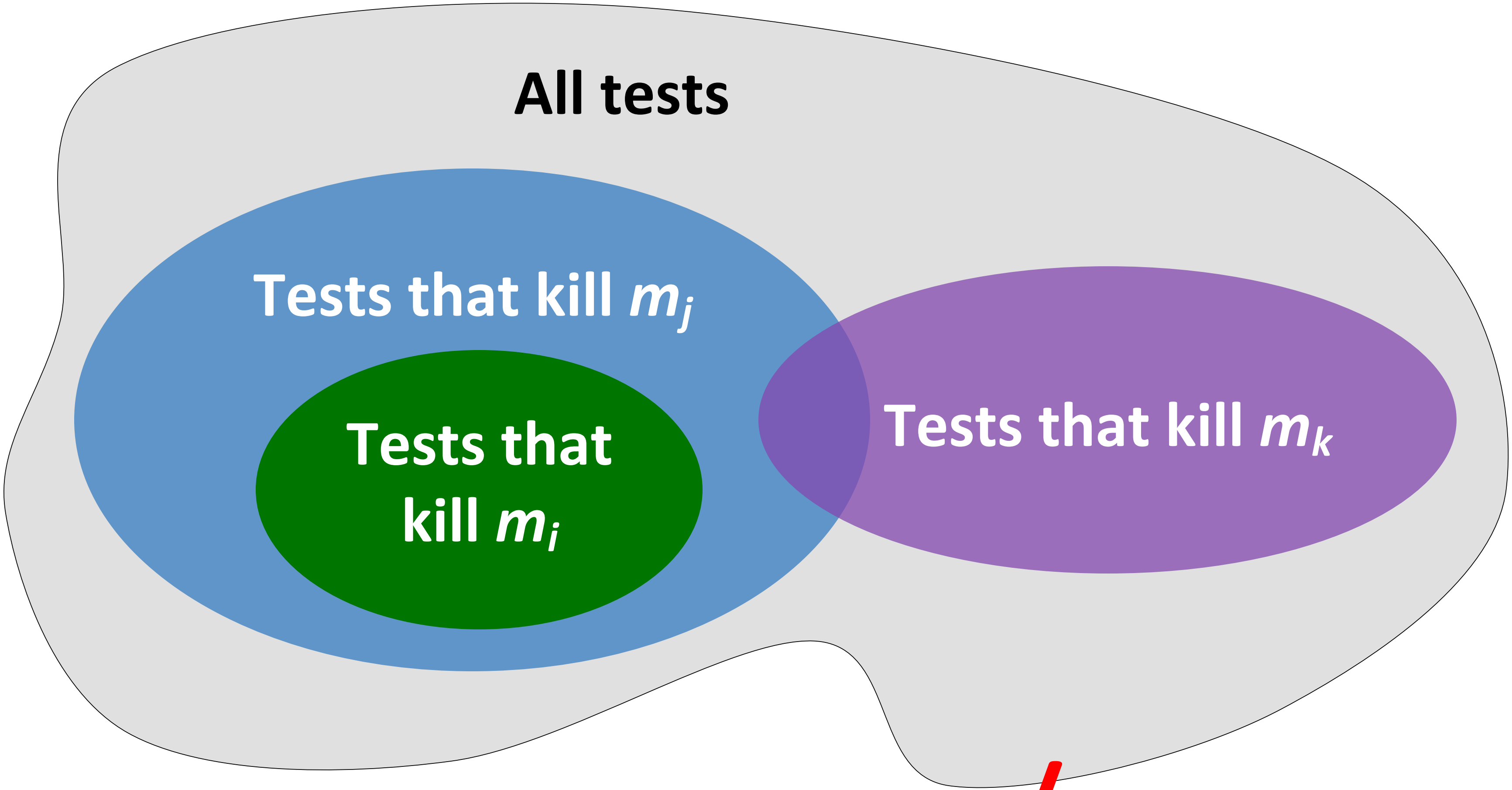


- ROR (SELECTIVE) requires 7 mutants
- But there are only 3 possible tests: <, ==, and >
- Hence 3 mutants are enough
  - Which 3 depends on the original operator
  - Note that the T and F mutants are critical
    - They also yield MCDC Coverage….

Program Specific Subsumption Relations

- Given a set of mutants *M* on artifact *A*, mutant m_i subsumes mutant m_j ($m_i \rightarrow m_j$) iff:
  - Some test kills m_i
  - All tests that kill m_i also kill m_j
- Subsumption relations naturally form graphs
- Subsumption is the key to constructing "useful" Higher Order Mutants (HOMs)
  - Given N mutants, construct a HOM that is only killed by a test that kills all N mutants

A Venn Diagram View of Subsumption



All tests

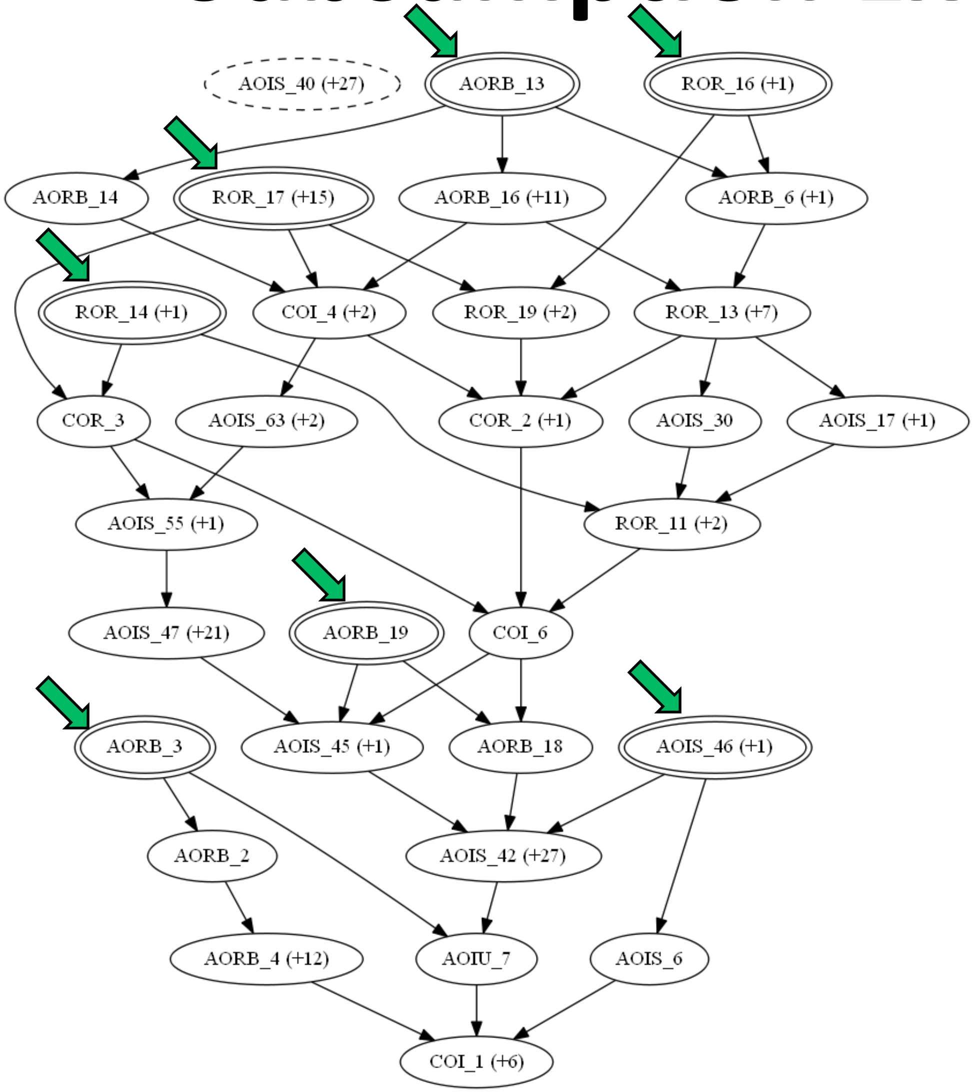
Tests that kill m_j

Tests that kill m_i

Tests that kill m_k

$$m_i \rightarrow m_j \qquad m_i \not\rightarrow m_k$$

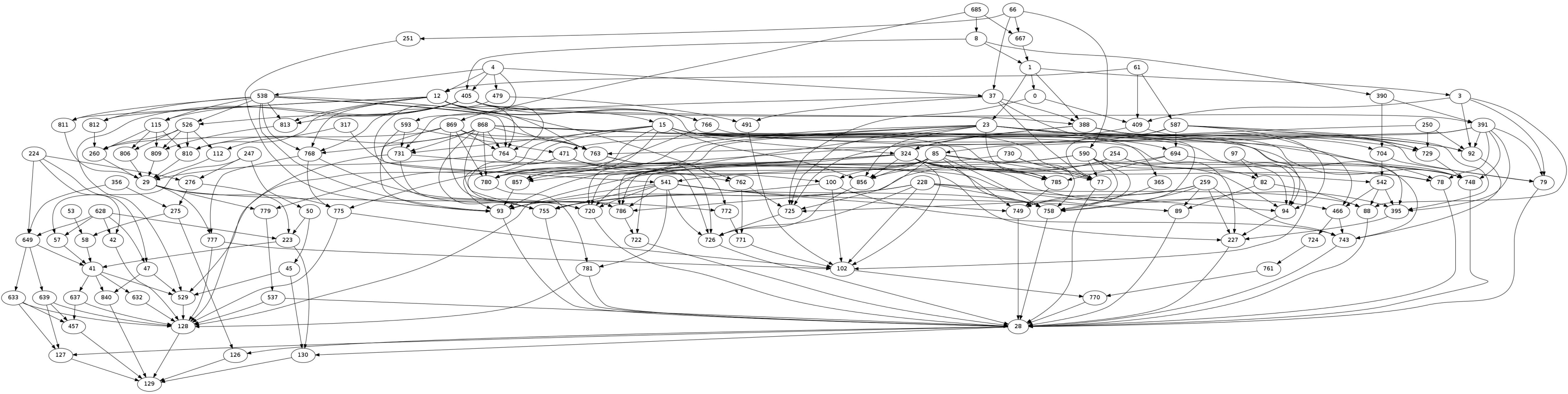
Subsumption Example: `cal()`



muJava generates 145 non-equivalent mutants

We only need 7 dominator mutants!

`cal()` With More Mutants



- Proteum generates many more mutants
  - Though nowhere near as many as possible
- Graph appears quite different than muJava graph
  - But clearly must be related
- Lots of research to be done yet

Subsumption

- Hypothesis:
  - Subsumption: *the* relation that describes mutation
  - Encodes we want to know about mutant behavior
    - and perhaps stuff we haven't thought of
  - Significant step towards "semantic mutation"
- Promises and Challenges
  - Subsumption relations are specific to artifact
  - Computing subsumption
  - Richer mutation systems

Subsumption Relations Specific to a Given Artifact

- Subsumption relation can give the engineer context
  - “These mutants all behave a certain way”
  - This can help the engineer find tests
- Subsumption approach can also structure equivalent mutants
  - If A equivalent implies B is equivalent
    - Then only A needs to be analyzed

Computing Subsumption

- For testing, dynamic subsumption is too late
  - – Dynamic approach uses tests to derive relations
  - – But it's tests the engineer wants!
- Static approaches to subsumption generation
  - – More on this later in the workshop!

(Much) Richer Mutation Systems

- The mutation community has spent years showing that first order mutants are “good enough”
  - The problem is that there are simply too many higher-order mutants (HOMs)
- Subsumption makes HOMs much more efficient
  - We don’t need all the HOMs
    - Subsumption tells us which ones are important
  - Can we compute subsuming HOMs “on-the-fly”?
  - Which ones do we need for testing?
    - The minimal ones
- One goal: HOMs that look more like “real” faults

Requirements for Workable Mutation (1)

- Workable mutation must use effective mutants
  - Let's not miss the easy cases
    - No reason not to subsume branch coverage
    - No reason for ROR not to subsume MCDC
- Mutation tools should leverage research
  - But not be a slave to it
  - Sometimes, other goals are more important
  - In particular, avoiding equivalent mutants is more important

Requirements for Workable Mutation (2)

- The engineer should not be aware of redundant mutants
  - If the tool hides them, that's fine
- The engineer should not see "weird" mutants
  - Everything the engineer sees should make sense, given the artifact
- The engineer should not see equivalent mutants
  - Better to ignore a mutant than give the engineer an equivalent one
  - Exception: offering the engineer a positive ROI
    - Capturing equivalent mutant analysis might work
    - Analogy is to @SuppressWarnings() annotations for Java "lint" function
      - Possible example: @Infeasible(a==b) as directive to ROR mutation
    - Captured analysis subject to subsequent falsification via tests!
- The engineer should work at the right level of abstraction
  - It's about the inputs, outputs, and program states, not mutants!

Design for the engineer, not the researcher!

A Modest Example: The Perfect Should Not Be the Enemy of the Good

- Consider “Delete” mutation
  - Systematically remove code (statements or otherwise)
- Desirable properties
  - Reasonable Effectiveness
    - Far more powerful than statement (or branch) coverage
      - Reachability is not the same as full RIP!
    - But definitely not as powerful as full mutation
  - Redundancy elimination
    - Should be possible to statically compute many subsumption relations
  - Nothing weird for developer, especially in TDD
    - If code is only written for failing tests, deleting code should make test set fail!
  - Equivalent mutants rare (although not eliminated)
    - Sometimes code really is “dead”
    - But could be annotated for a “clean” test (e.g. @Unreachable(…))
  - Engineers already work at the “branch” abstraction level
    - Simple color schemes might be adequate to convey infection, outputs

Questions?

- Acknowledgements
  - Many sources for the ideas in this presentation
    - Of course, I claim all the mistakes
  - Jeff Offutt, Bob Kurtz, Marcio Delamaro, Lin Deng
    - We've been working up to the "zero option" for equivalent mutants for quite a while now
  - Numerous prior students in my testing classes
- Contact:
  - pammann@gmu.edu