Closing the Risk Gap:

A “Case” Study

How do we know when a ticket is ready? What does it mean to be "done?"

1. Requirements are a reasonable solution to the problem
2. Requirements are consistent with the current body of requirements
3. Implementation satisfies the requirements
4. Implementation is consistent with the rest of the system
5. Implementation is otherwise robust

Each question represents a gap in our knowledge. As we develop the product we seek to fill that gap with evidence. We gather evidence by analyzing, interacting, and observing what the product does and how it does it under a variety of cases. This paper seeks to provide some structure around the implicit processes we use to evaluate questions 2, 3, and 4. Question 1, while important to think about during development, is too highly subjective to be treated with the rigor we attempt to apply to the other questions. Question 5 is already well treated in other works.

Cases

A case is a single combination of trigger, conditions, and result. As you interact with the product you're observing an uncountable number of cases. They're like the coastline of Britain: their number depends on the scale of your measuring stick. But when you notice something unexpected, you intuitively boil it down to a reproducible combination of trigger, conditions, and result.

Testing seeks the cases necessary to prove the Risk Gap questions false, not true. We deem the answers sufficiently true through our failure to falsify them with effort proportionate to the project. There are three kinds of cases: deducible, predictable, and unforeseeable.

Deducible Cases

Deducible cases can be deduced from the information contained in the requirements. Decent requirements contain the same ingredients as cases: triggers, conditions, and results. You can deduce that if all explicit conditions are met and the trigger is run, the result should happen. You can also deduce that if any single condition is false while all others are true, the result should not happen. Testing two or more false conditions at the same time doesn't tell you which condition falsified the result. You need to isolate the variables in the same way as any scientific experiment. It's important to note that deducible cases assume all other possible values within the valid and invalid classes are equivalent. In science this would be known as a very coarse grain.

For the condition "Value must be greater than 100" there are two deducible cases:

1. Greater than 100 (positive)
2. Less than or equal to 100. (negative)

Within those cases 101 is equivalent to 2,234 and 99 is equivalent to 1.

If we add another condition, "Color must be red," we've only added one more case, for a total of three:

1. Greater than 100 and red (positive)
2. Great than 100 and blue (negative)
3. Less than or equal to 100 and blue (negative)

We might also create a fork in the logic by creating a different outcome for green: "If green, any value." Our full logic is now "Red color and values over 100, or any value if green." The deducible cases are:

1. Greater than 100 and red (positive)
2. Greater than 100 and blue (negative)
3. Less than or equal to 100 and red (negative)
4. Any value and green (positive)

Note we didn't add a negative case for green. Cases 2 and 3 already cover that. However, we have a strong urge to specify two positive cases for green, "Greater than 100" and "Less than 100." Otherwise, how could we know that both value classes (>100 and <=100) will work for green? This is an example of a predictable case.

Predictable Cases

Predictable cases draw upon domain knowledge, test skill, and human experience to divide the huge equivalence classes of deducible cases into a finer grain. We use our knowledge to predict cases which we estimate have a higher risk. Besides checking both value classes for green, our experience says we should check:

1. A value of exactly 100
2. A decimal value like 100.001
3. A negative value like -10
4. A null value
5. An empty value
6. An alpha value
7. An enormous value
8. Color = pink (contains red but not equal to red)
9. Color = purple (contains red but also blue)
10. Color = yellow (partial green)
11. Color = white (contains all colors)
12. Etc.

Many bugs are in predictable cases, but they often aren't discovered until halfway through development or after release. Requirements should be refined early in the process to define the deducible and predictable cases that would show the answer to any of our Risk Gap questions is false.

Predictable cases also include checking other parts of the system which could be related to or affected by the new requirements or their implementation. Previously existing automated checks help here, but previous checks cannot always account for new information. They may not be sensitive to a new variable or output. Basically, any variation of triggers, conditions, or methods of evaluating results that we associate with risk can be included as predictable cases. Ideally, with full system knowledge, we could predict all relevant cases. Unfortunately we neither have full system knowledge or are capable of predicting all outcomes given the knowledge we have. We can't foresee all the consequences.

Unforeseeable Cases

Unforeseeable cases are discovered only as we interact with the product in the wild or run deep simulations. Unforeseen variables emerge through the deployed system, its environment, and our dynamic use and abuse of the product. Even accidentally changing the order of our clicks because we looked away for a minute can unwittingly change the state enough to change a result. Picking random data values often triggers unexpected effects that are tied to those values from a ticket in the distant past. Each discovery of an unforeseen case becomes ammo for designing predictable cases in the future.

Finding unforeseeable cases is an indeterminate, creative, and grueling process, but often uncovers critical issues. It's the last line of defense in a software team. You can maximize the probability of finding unforeseen cases by automating deducible and predictable cases as part of the development process. Since deducible cases are deducible, you can know how many of them any ticket needs and when they're done. The number of deducible cases required to cover all the logic gives a decent indication of complexity. The number of predictable cases thought of by the team gives a decent indication of risk. It says there are a lot of ways a requirement could go wrong. Checking that the requirements can work with deducible and predictable cases before asking a tester to evaluate a deployed product maximizes the return on the experiential test investment. That investment is wasted if testers discover bugs within clearly deducible or predictable cases.

Since testers interact with and experience the product more and more deeply than anyone else, they're prime sources of information for designing predictable cases. The process of brainstorming predictable cases early in the development lifecycle can yield massive returns on the investment by preventing days of testing, reporting, fixing, reviewing, and retesting avoidable bugs and changes.

Experiential testing also allows us to connect more closely with end users by sharing a similar experience in using the application. Without this intimate interaction, any number of automated checks which perfectly prove correctness may detect that some wouldn't want to use the application.

What's in a Case

So far we've glossed over a lot of important detail. What is meant by conditions, triggers, and results?

These terms are specifically designed to be agnostic of technical implementation details. They use the language of the requirements to be accessible to any team member, but follow the rigorous three part structure to be directly translatable into atomic automated checks.

Trigger

A "trigger" is any action which triggers the software to perform its functions and calculations. They can be internal or external to the system. Unit and integration automation can surgically trigger many parts of the system.

Conditions

The "conditions" are the current values of all variables which could influence the result. These can also be internal or external to the system, for instance, a setting in the app, the time of day, location, and temperature. One of the biggest, if not the biggest challenge in software development is accounting for unforeseen conditions that negatively influence results. Finding those conditions is one of the main focuses of testing.

Result

The "result" is the sum of effects that happen as the system reacts to the trigger and conditions.

The result can also be literally anything, from the bits on harddrives to rendering the UI to the physical motion of a device. We usually only observe a fraction of the whole result, or a sample. We try to take samples that represent the whole, while keeping an eye out for results that might go unobserved. Deciding what and how to sample is a crucial element of automated check design.

Conclusion

Combining automated deducible and predictable cases with proportionate time to discover unforeseeable cases is a responsible and practical heuristic to close the Risk Gap around questions 2, 3, and 4. If the answers to all three questions is still "yes" after we did our best to find cases which would falsify them, then we're done.

The library of deducible and predictable cases we build over time provides rapid feedback to questions 2 and 4. The case's business oriented language allows people to learn how the system is supposed to behave in any given case. Finally, the unforeseen conditions we discover with experiential testing improve our predictable cases in a beneficial feedback loop.