

Concepts in Statistical and Software Testing

COMP 598, Recitation #2

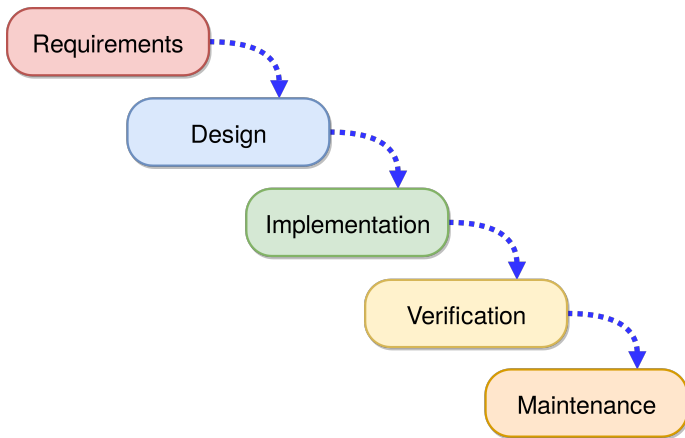
Breandan Considine

McGill University

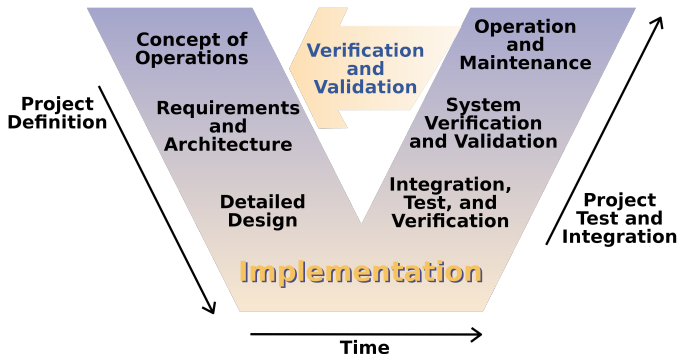
breandan.considine@mcgill.ca

October 29, 2020

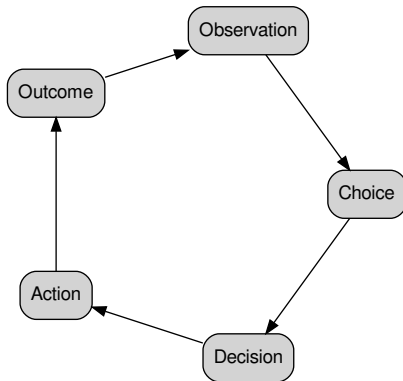
Royce's Waterfall Model



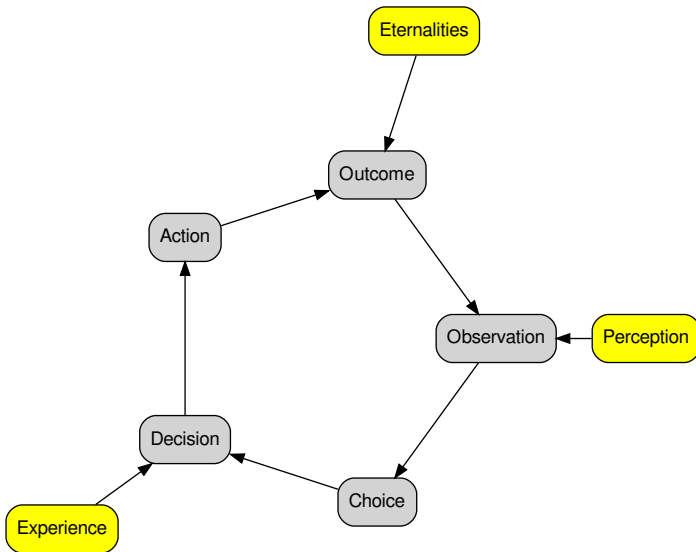
Software verification and validation



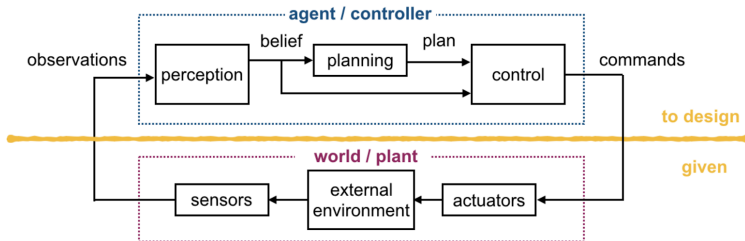
Simple feedback systems



Unmodeled dynamics

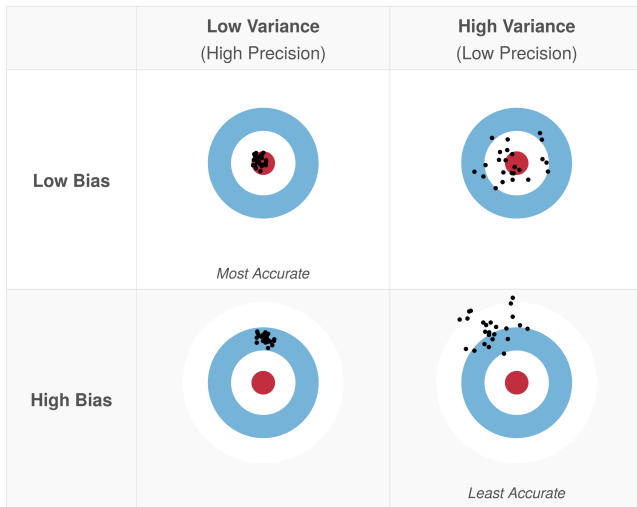


Closed loop control



Where are the boundaries between these components?
How do we safely and effectively test these systems?

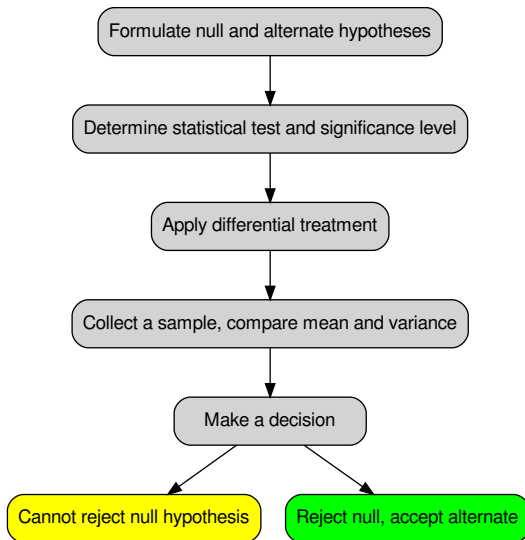
Bias/Variance and Sensitivity/Specificity



Validity and testing

- ① Are we really measuring what we want to measure? (Test validity)
 - ① Many advertisers try to maximize clicks. This is a poor metric.
 - ② Objectives may change over the production lifetime of a model.
 - ③ A poorly chosen objective can have unintended consequences.
- ② Is the dataset free from confounds? (Internal validity)
 - ① People constantly forget (or conveniently overlook) confounds.
 - ② If the data generator is biased, the model will encode its bias.
 - ③ The method of sampling may have hidden biases.
- ③ Does the model generalize well in practice? (External validity)
 - ① Maybe the training data has grown stale over time.
 - ② Maybe the model is missing data on some key demographic.
 - ③ Maybe the true population is not the population we bargained for.

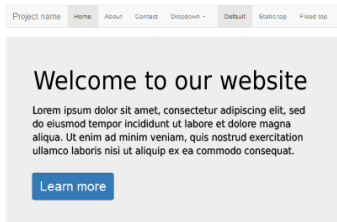
Hypothesis Testing



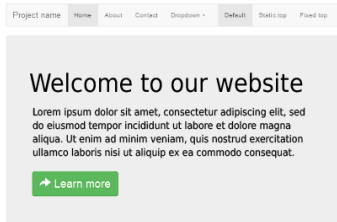
Hypothesis Testing

		Conclusion about null hypothesis from statistical test	
		Accept Null	Reject Null
Truth about null hypothesis in population	True	Correct	Type I error Observe difference when none exists
	False	Type II error Fail to observe difference when one exists	Correct

A/B Testing

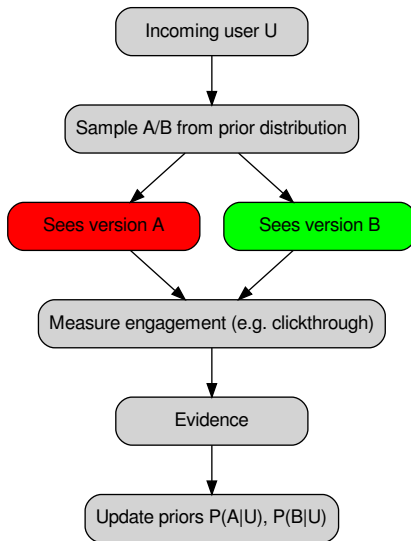


Click rate: 52 %

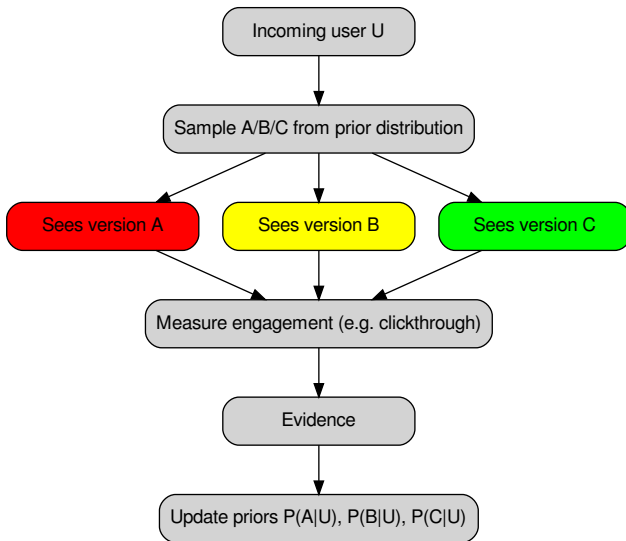


72 %

A/B Testing



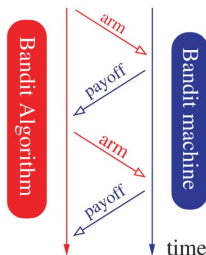
A/B/C Testing



Multi-armed bandits

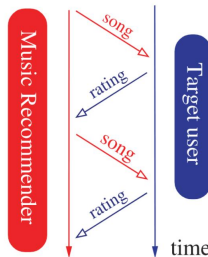
Interactive Process

Multi-armed bandit



Objective maximize the sum of payoffs

Music Recommendation



maximize the sum of ratings

<https://dl.acm.org/doi/pdf/10.1145/2623372#page=8>

Unit Testing

Checks a small “unit” of code on a small number of input/outputs.



```
fun test(subroutine: (Input) -> Output) {  
    val input = Input()           // Construct an input  
    val expectedOutput = Output() // Construct an output  
    val actualOutput = subroutine(input)  
    assert(expectedOutput == actualOutput) { // Evaluate  
        "Expected $expectedOutput, got $actualOutput"  
    }  
}
```

Pros: Simple to understand and helps document design assumptions.

Cons: High coverage is cumbersome to write, grows stale over time.

Integration Testing

Checks for terminating exceptions and postconditions on a large program.



```
fun <I, O> test(program: (I) -> O, inputs: Set<I>) =
    inputs.forEach { input: I ->
        try {
            val output: O = program(input) // Whole program
            assert(postcondition(output)) {
                "Postcondition failed on $input, $output"
            }
        } catch (exception: Exception) {
            assert(false) { exception } //
        }
    }
```

Pros: Easier to write, checks for high-level exceptions and postconditions.

Cons: Only covers the “happy path”, can often be too coarse grained.

Fuzz Testing

Generates many random inputs and compare outputs with an oracle.



```
@Repeat(LARGE_NUMBER)
fun <I, O> test(program: (I) -> O, // System under test
               oracle : (I) -> O, // Source of Truth
               sample : (Random) -> I) {
    val input: I = sample(Random())
    assert(program(input) == oracle(input)) {
        "Oracle and program disagree on $input"
    }
}
```

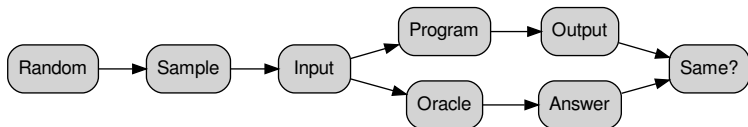
Pros: Lower effort required to write test cases, higher test coverage.

Cons: Input space often too large to brute force, “test oracle problem”.

Fuzzing Dataflow




```
@Repeat(LARGE_NUMBER)
fun <I, O> test(program: (I) -> O, // System under test
               oracle : (I) -> O, // Source of Truth
               sample : (Random) -> I) {
    val input: I = sample(Random())
    assert(program(input) == oracle(input)) {
        "Oracle and program disagree on $input"
    }
}
```



Property Based Testing

Specify a universal output property and implement a sampler and shrinker.



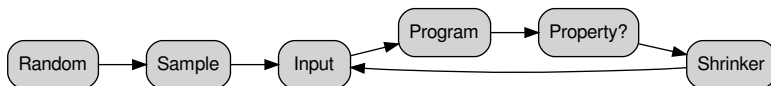
```
@Repeat(LARGE_NUMBER)
fun <I, O> test(program : (I) -> O,
               property: (O) -> Boolean,
               sample  : (Random) -> I,
               shrinker: (I, (I)->O, (O)->Boolean) -> I) {
    val randomInput: I = sample(Random())
    assert(property(program(randomInput))) {
        "Error detected on: $randomInput!"
    }
    val min = shrinker(randomInput, program, property)
    "Minimal counterexample of property: $min!"
}
```

Pros: No oracle required, can detect and minimize counterexamples.

Cons: Property specification can be brittle and requires domain expertise.



```
@Repeat(LARGE_NUMBER)
fun <I, O> test(program : (I) -> O,
               property: (O) -> Boolean,
               sample  : (Random) -> I,
               shrinker: (I, (I)->O, (O)->Boolean) -> I) {
    val randomInput: I = sample(Random())
    assert(property(program(randomInput))) {
        "Error detected on: $randomInput!"
    }
    val min = shrinker(randomInput, program, property)
    "Minimal counterexample of property: $min!"
}
}
```



Metamorphic Testing

Specify a relation (e.g. \approx), transform inputs and compare with labels.



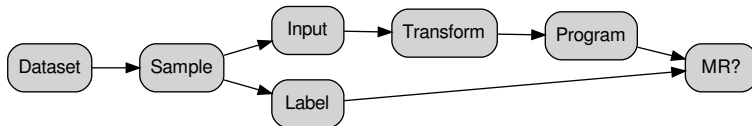
```
@Repeat(LARGE_NUMBER)
fun <I, O> test(program: (I)          -> O,
               inputTx: (I)          -> I,
               metaRel: (O, O)       -> Boolean,
               sample : (Set<Pair<I, O>>) -> Pair<I, O>,
               dataset: Set<Pair<I, O>>) {
    val (trueInput: I, trueOutput: O) = sample(dataset)
    val transformedInput : I = inputTx(input)
    val transformedOutput: O = program(transformedInput)
    assert(metaRel(trueOutput, transformedOutput)) {
        "<$trueOutput> not $metaRel <$transformedOutput>!"
    }
}
```

Pros: No domain expertise required, just a small labeled dataset.

Cons: How to design TX to detect errors efficiently? Open research.



```
@Repeat(LARGE_NUMBER)
fun <I, O> test(program: (I)          -> O,
               inputTx: (I)          -> I,
               metaRel: (O, O)       -> Boolean,
               sample : (Set<Pair<I, O>>) -> Pair<I, O>,
               dataset: Set<Pair<I, O>>) {
    val (trueInput: I, trueOutput: O) = sample(dataset)
    val transformedInput : I = inputTx(input)
    val transformedOutput: O = program(transformedInput)
    assert(metaRel(trueOutput, transformedOutput)) {
        "<$trueOutput> not $metaRel <$transformedOutput>!"
    }
}
```



Hypothesis testing vs. Software testing

Statistical Testing

- We're not sure what the correct answer is, want to find out
- Tries to minimize empirical risk by increasing sample size
- Often assumes IID randomness for accurate results
- Deliberately injects noise into sampling process
- Results are never exactly the same twice
- Requires non-determinism to demonstrate reproducibility

Software Testing

- We know what the correct behavior is, need to validate it!
- Tries to minimize operational risk, increase test coverage
- Uses randomness as a substitute for formal verification
- Tries to reduce noise to prevent false alarms
- Requires determinism for software reproducibility

Further resources

- ① Aleatoric and Epistemic Uncertainty in Machine Learning, Eyke Hüllermeier and Willem Waegeman (2020)
- ② The Multi-Armed Bandit Problem and Its Solutions, Weng (2017)
- ③ Exploration in Interactive Music Recommendation, Wang (2014)
- ④ Intro to property-based testing, Alex Chan (2016)
- ⑤ Property based testing, Pierre Felgines (2019)
- ⑥ Coverage Guided, Property Based Testing, Lampropoulos (2019)
- ⑦ Metamorphic testing, Wayne (2019)
- ⑧ Survey of Trends in Oracles for Software Testing, Harman (2014)
- ⑨ American Fuzzy Lop
- ⑩ TensorFuzz: Debugging Neural Networks with Coverage-Guided Fuzzing, Odena et al. (2019)