CLEMSON UNIVERSITY

CPSC 3720
Lesson 35

Connie Taylor
Professor of Practice

School of COMPUTING

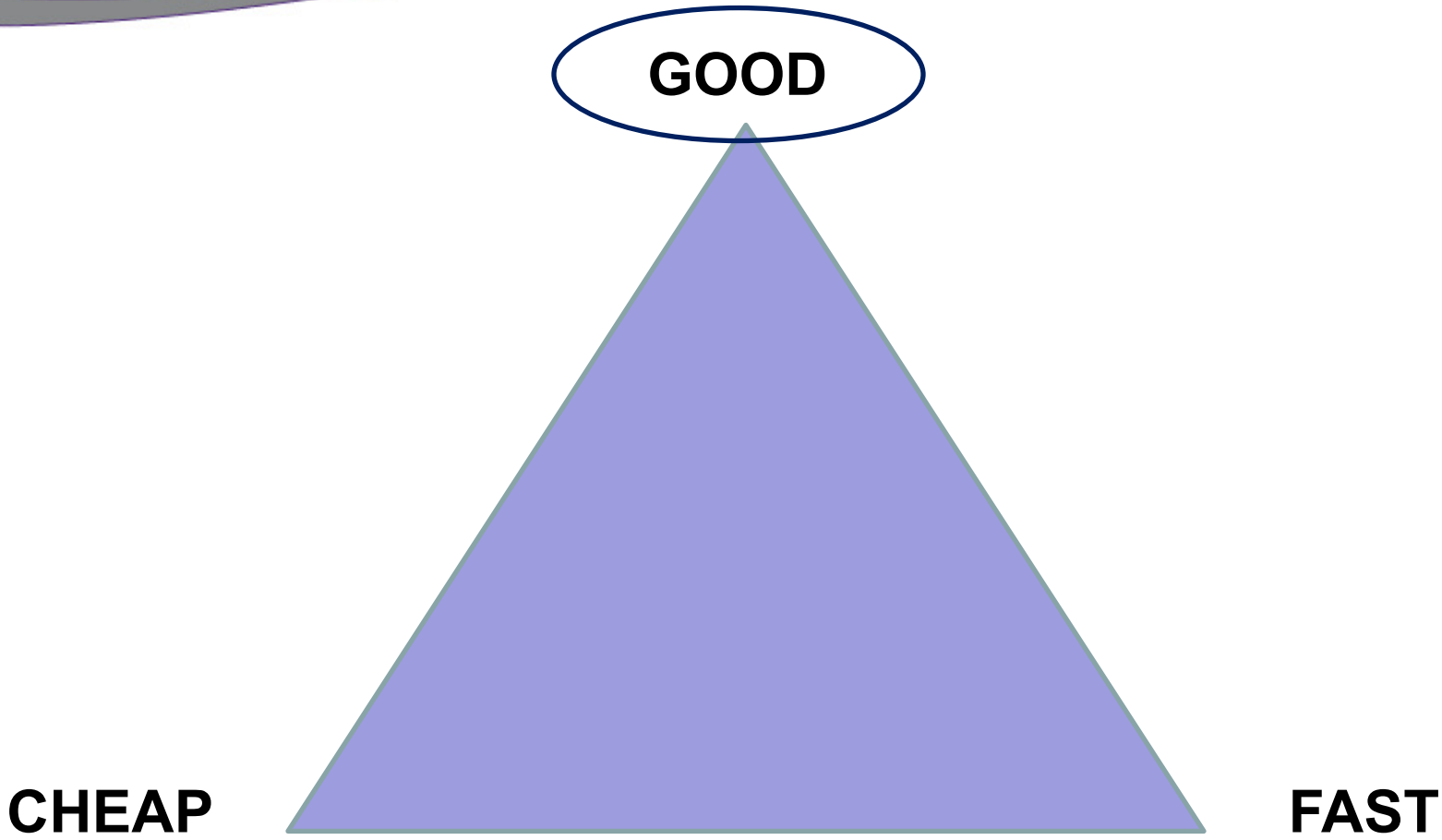
Class Updates

- **Half of the teams did not submit anything for Sprint 2**
- Quiz 4 due on Tuesday night (open now)
- Sprint 3 kickoff on Wed
- Sprint 2 Reviews will be this Fri and Mon week (will send out schedule)

Coming in for a landing…

- 10 more classes including today
  - 2 for Project work
  - 2 for Project reviews
  - Remaining topics – continuous delivery, tooling, metrics
- Quiz 5 and Exam 2 will be before the final class on Dec 4.
- Final Exam will be optional for everyone
  - I will provide your class grade by Dec 4 for you to make this decision.
  - If you want to opt out of the exam, I will need an email from you.

Software Planning Triangle



Pick any two…

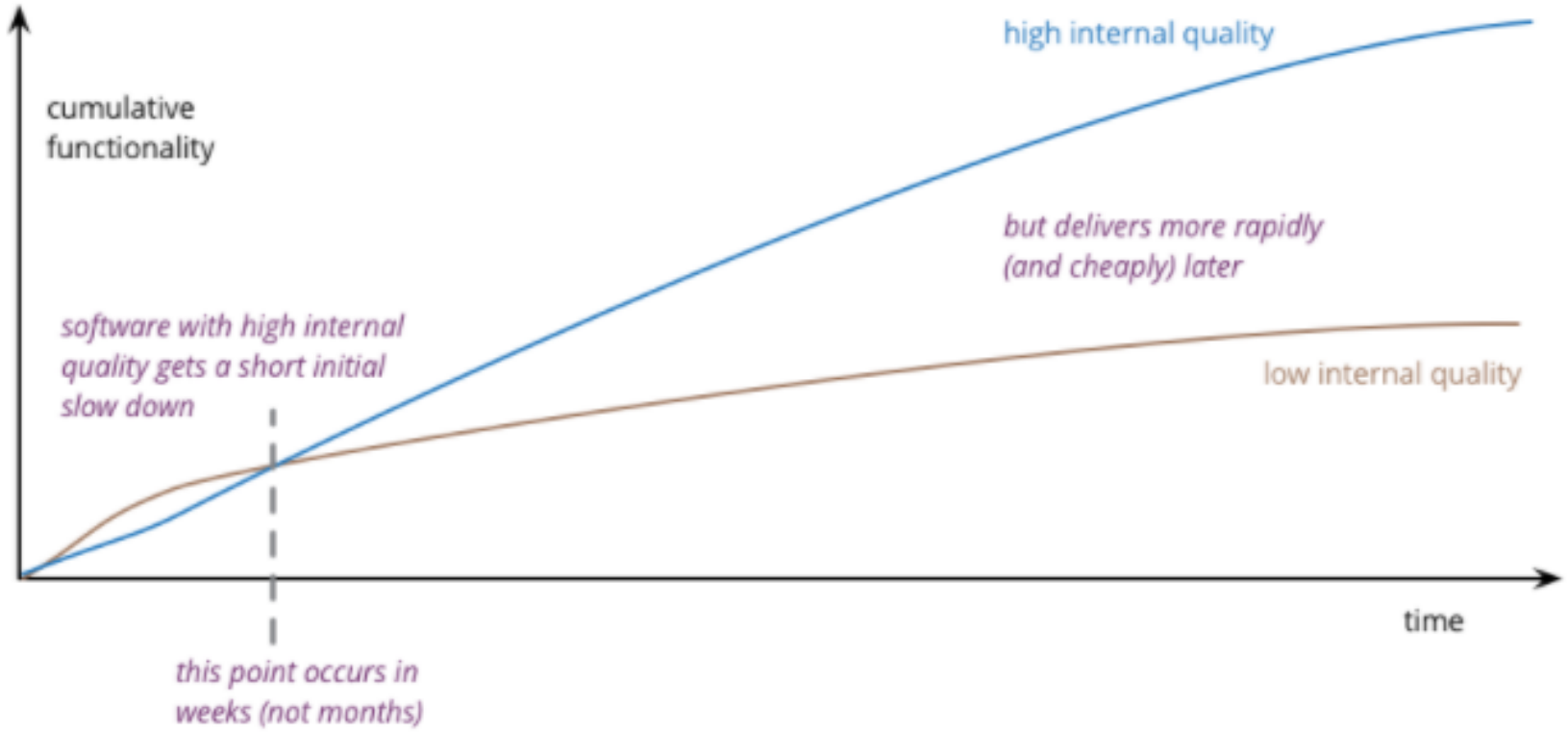
Is Quality Worth the Cost?

Is it worth it?

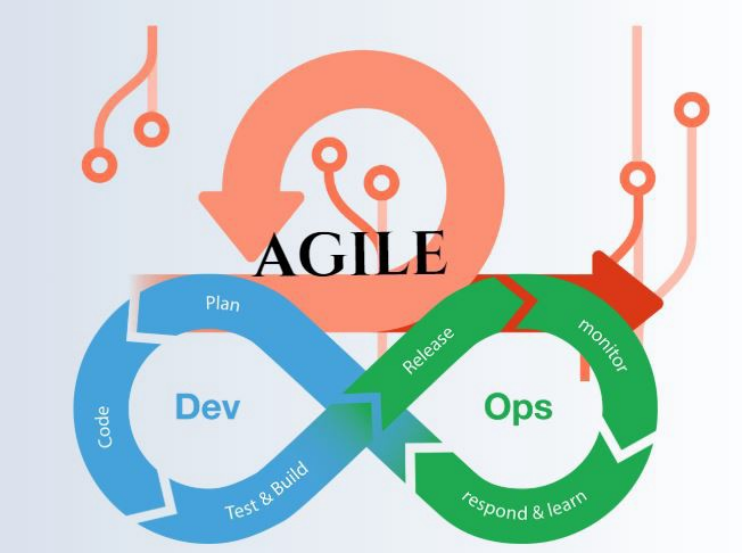
Key Takeaways

1. Cruft

2.



Testing – DevOps



- "Shift Left" and Test-Driven Development (TDD)
- More blur between Dev and Test roles
- Automation is ESSENTIAL

Shift Left: TDD

- Benefits of TDD:
  - Writing clear requirements
  - Development in small steps. This will make debugging easier since we will have small code chunks to debug.
  - Minimalistic code and enforce the YAGNI principle – “You Ain’t Gonna Need It”

Sound familiar??

Shift Left: TDD Steps*

1. **Add a test:** Each new feature begins with writing a test. This is a differentiating feature of test-driven development versus writing unit tests after the code is written: it makes the developer focus on the requirements before writing the code, a subtle but important difference.

2. **Run all tests and see if the new test fails:** This validates that the test harness is working correctly, shows that the new test does not pass without requiring new.

3. **Write the code:** The next step is to write some code that causes the test to pass. The new code written at this stage is not perfect and may, for example, pass the test in an inelegant way. That is acceptable because it will be improved in Step 5. At this point, the only purpose of the written code is to pass the test.

4. **Run tests:** If all test cases now pass, the programmer can be confident that the new code meets the test requirements, and does not break or degrade any existing features. If they do not, the new code must be adjusted until they do.

5. **Refactor code:** The growing code base must be cleaned up regularly during test-driven development.

*Beck, Kent (2002-11-08). Test-Driven Development by Example. Vaseem: Addison Wesley. ISBN 978-0-321-14653-3.

Shift Left: TDD Key Concepts

- **Test Fixtures:** A set of objects that we have instantiated for our tests to use

- **Test Doubles:**
  - Dummy – A dummy is the simplest form of a test double. It facilitates linker time substitution by providing a default return value where required.
  - Stub – A stub adds simplistic logic to a dummy, providing different outputs.
  - Mock – A mock is specified by an individual test case to validate test-specific behavior, checking parameter values and call sequencing.
  - Simulator – A simulator is a comprehensive component providing a higher-fidelity approximation of the target capability (the thing being doubled). A simulator typically requires significant additional development effort.

Shift Left: TDD Test "Doubles" Benefits

- Can test an object without the environment being available
- Allows stepwise implementation as we successively replace more and more doubles
- When something is wrong, we can be almost sure that the problem is in the new object and not in its environment

Sound familiar??

Shift Left: TDD
Assertions

- As we program, we make many assumptions about the state of the program at each point in the code
  - A variable's value is in a particular range
  - A file exists, is writable, is open, etc.
  - Some data is sorted
  - A network connection to another machine was successfully opened
  - …
- The correctness of our program depends on the validity of our assumptions
- Faulty assumptions result in buggy, unreliable code

Shift Left: TDD Assertions

```
int binarySearch(int[] data, int searchValue) {

   // What assumptions are we making about the parameter values?

   …
}
```

- data != null
- data is sorted

- What happens if these assumptions are wrong?

Shift Left: TDD
Assertions

- Assertions give us a way to make our assumptions explicit in the code

- `assert temperature > 32 && temperature < 212;`

- The parameter to assert is a boolean condition that should be true
- `assert condition;`

- If the condition is false, Java throws an AssertionError, which crashes the program

- Stack trace tells you where the failed assertion is in the code

Shift Left: TDD Assertions

```
int binarySearch(int[] data, int searchValue) {

   assert data != null;
   assert isSorted(data);

   …
}
String[] someMethod(int y, int z) {

   assert z != 0;
   int x = y / z;

   assert x > 0 && x < 1024;
   return new String[x];
}
```

Shift Left: TDD
Assertions

- Assertions are test cases throughout your code that alert you when one of your assumptions is wrong.
- Assertions are usually disabled in released software
- In Java, assertions are DISABLED by default
- To turn enable them, run the program with the `-enableassertions` (or `-ea`) option

  java –enableassertions MyApp

  java –ea MyApp
- In Eclipse, the `-enableassertions` option can be specified in the **VM arguments** section of the **Run Configuration** dialog

Shift Left: TDD Assertions

- Alternate form of assert
- `assert condition : expression;`
- If condition is false, expression is passed to the constructor of the thrown AssertionError

```
int binarySearch(int[] data, int searchValue) {

   assert data != null : ”binary search data is null”;
   assert isSorted(data) : ”binary search data is not sorted”;
   …
}
String[] someMethod(int y, int z) {

   assert z != 0 : ”invalid z value”;
   int x = y / z;

   assert x > 0 && x < 1024 : x;
   return new String[x];
}
```

Shift Left: TDD
Assertions

- *If one of my assumptions is wrong, shouldn't I throw an exception*?

Shift Left: TDD Assertions

- *If one of my assumptions is wrong, shouldn't I throw an exception?*

- No. You should fix the bug, not throw an exception.

Shift Left: TDD
Parameter Checking

- A method or function should always check its input parameters to ensure that they are valid
- If they are invalid, it should indicate that an error has occurred rather than proceeding
- This prevents errors from propagating through the code before they are detected
- By detecting the error close to the place in the code where it originally occurred, debugging is greatly simplified

Shift Left: TDD
Parameter Checking

- Two ways to check parameter values
  - assertions
  - if statement that throws exception if parameter is invalid

```
int binarySearch(int[] data, int searchValue) {
   assert data != null;
   assert isSorted(data);
   …
}

int binarySearch(int[] data, int searchValue) {
   if (data == null || !isSorted(data)) {
      throw new InvalidArgumentException();
   }
   …
}
```

Shift Left: TDD
Parameter Checking

- *When do you use assertions vs. if/throw to check parameters?*

Shift Left: TDD
Parameter Checking

- *When do you use assertions vs. if/throw to check parameters?*

- If you have control over the calling code, use assertions
  - If parameter is invalid, you can fix the calling code

- If you don't have control over the calling code, throw exceptions
  - e.g., your product might be a class library that is called by code you don’t control

Shift Left: Code Reviews are still very valuable!

- Inspections
- Walkthroughs