CS 4530: Fundamentals of Software Engineering

Module 2: From Requirements to Code: Test-Driven Development

Adeel Bhutta, Mitch Wand, Joydeep Mitra

Khoury College of Computer Sciences

Learning Goals for this Lesson

- At the end of this lesson, you should be prepared to
  - Explain the basics of Test-Driven Design
  - Develop simple applications using Typescript and Jest
  - Learn more about Typescript and Jest from tutorials, blog posts, and documentation
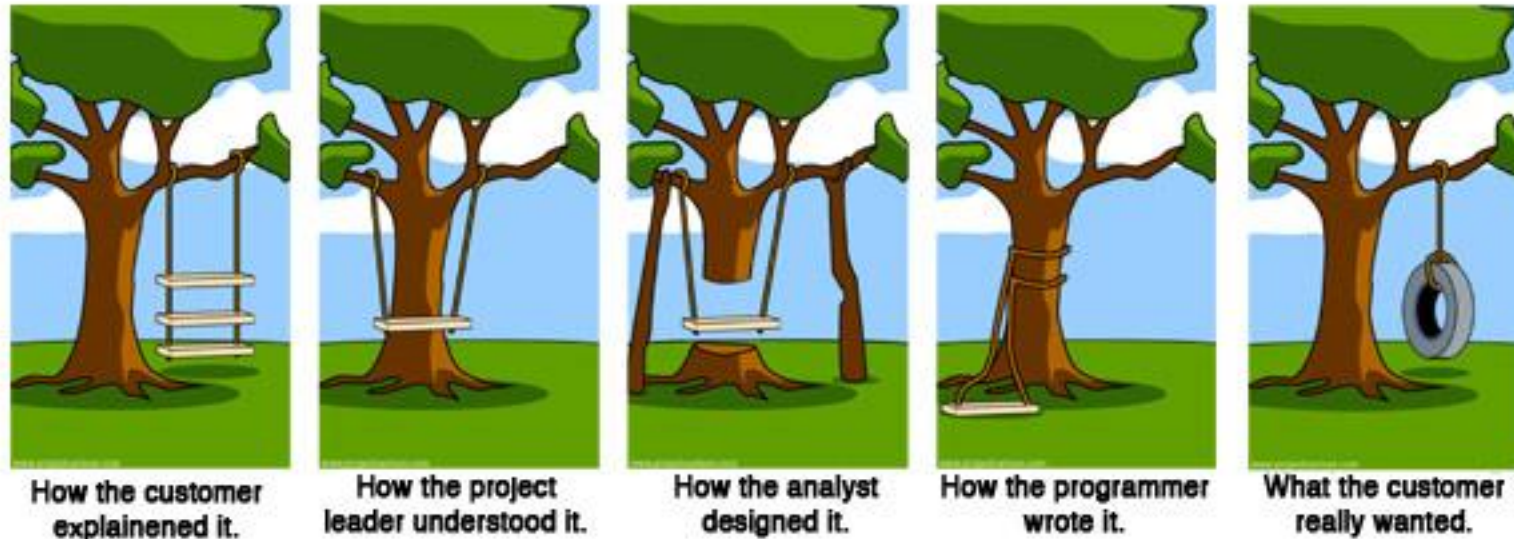
Non-Goals for this Lesson

- This is *not* a tutorial for Typescript or for Jest

- We will show you simple examples, but you will need to go through the tutorials to learn the details.

Review:
How to make sure we are building the right thing



Requirements Analysis

Planning & Design

Implementation

Test Driven Development (TDD)

- Puts test specification as the critical design activity
  - Understands that deployment comes when the system passes testing

- The act of defining tests requires a deep understanding of the problem

- Clearly defines what success means
  - No more guesswork as to what "complete" means

The TDD Cycle
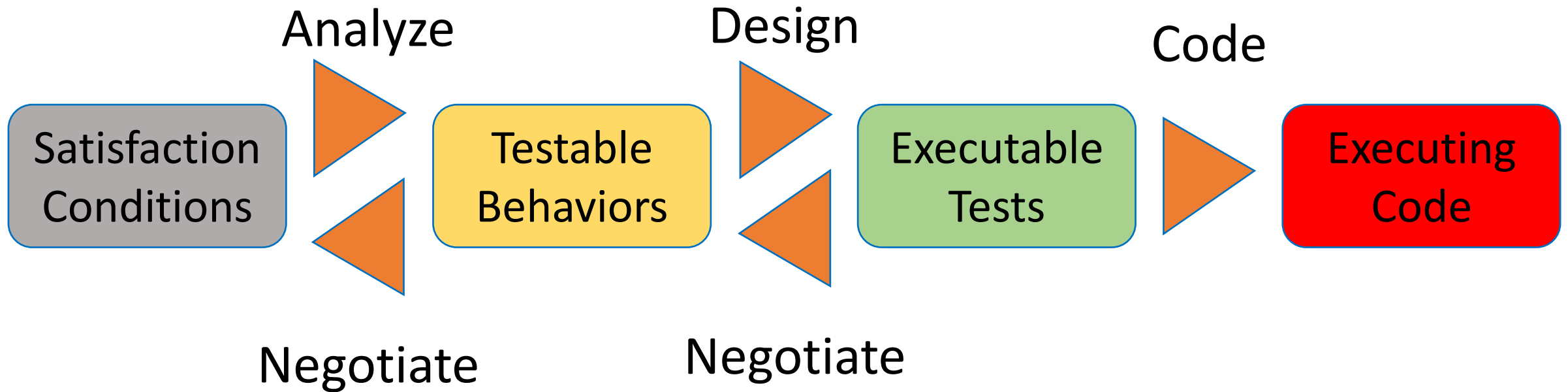
Analyze and Negotiate

- Analyze:
  - how does the client want the functionality to be delivered?
  - what ambiguities can you find that might make a difference to the client?

- Negotiate
  - discuss these with the client and resolve the differences

Design and Negotiate

- Design the least fragment of the system that could possibly deliver the COS. (YAGNI!)
  - What data would the system need?
  - What operations are needed on the data?
  - Give names to the testable things

- Negotiate
  - Does the data include everything that the client wants?
  - Do the operations include enough to support the COS?
  - Did the client forget something?
  - Are the COS realistic and achievable?

Example: a Transcript database User Story

- User story: tells what the user wants to do, and why.

- Example:

> **As a College Administrator, I want a database to keep track of students, the courses they have taken, and the grades they received in those courses.**

Conditions of Satisfaction

- Satisfaction Conditions list the capabilities the user expects, in the user's terms.

- Example:

  My database should allow me to do the following:
  - **Add a new student to the database**
  - **Add a new student with the same name as an existing student.**
  - **Retrieve the transcript for a student**
  - **Delete a student from the database**
  - **Add a new grade for an existing student**
  - **Find out the grade that a student got in a course that they took**

Analyze/Negotiate: what data do we need to worry about?

- We agreed with the client that for each student we will need to save:
  - a student ID
  - the student's name
  - a list of the student's courses and grades
  - for each course the student has taken, the name of the course and the student's grade.
- The client agreed that we don't have to keep track of when the student took the course
  - Keeping it simple for now!!

Design: Types.ts

```typescript
export type StudentID = number; // Unique identifier for a student
export type StudentName = string; // Name of the student
export type CourseID = string; // Unique identifier for a course
export type Grade = number; // Grade for a course, typically between 0 and 100

 // Transcript containing student ID, student name, and their courses with grades
export type Transcript = {
    studentID: StudentID,
    studentName: StudentName,
    courses: { courseID: CourseID, grade: Grade }[]
};
```

Analyze/Negotiate: How are these behaviors to be delivered?

- Result of negotiation:
  - they are to be delivered as a class with methods that provide the specified behaviors.

More analysis/design/negotiation

- We've agreed with the client:
    1. We'll need some way of getting from a student name to a set of possible IDs (analysis)
    2. That the student ID will be a positive integer (design/negotiation)
    3. That identifying a request with just the studentID is too error prone, so we'll label all requests with a studentID and a student name
    4. The service should deal with illegal requests by throwing an error. (design/negotiation)

And still more analysis/negotiation

In our discussions with the client, we started talking about the list of courses and grades. Here are some of the issues that came up, and how they were resolved:
1. Can a student take a course more than once? (no)

2. Must the grades on the transcript be listed in the same order that the course was taken (ie, the order that the grade was added) (Yes)

3. Do we care about misspelled course names? (no)

Our next step is to turn these satisfaction conditions into testable behaviors

- To do this, we will have to design our program at least enough to give names to the things we want to test.

- For our example, we need to design the external interface for our database.

- We document this in a file we will call **transcriptService.interface.ts**

transcriptService.interface.ts

```typescript
import { StudentID, StudentName, Transcript, CourseID, Grade } from "./Types"

export interface ITranscriptService {

    initialize(arg0:Transcript[]): void; // Initializes the database with an array of transcripts

    getAll(): Transcript[]; // Returns all transcripts in the database

    addStudent(name: StudentName): StudentID;

    deleteStudent(studentID: StudentID, name: StudentName): void;

    getTranscript(studentID: StudentID, name:StudentName): Transcript;

    addGrade(studentID: StudentID, studentName: StudentName, courseID: CourseID, grade: Grade): void

    getGrade(studentID: StudentID, studentName: StudentName, courseID: CourseID): Grade

    getStudentIDsByName(name: StudentName): StudentID[]
}
```

Now we can write down some testable behaviors.

- These could serve as titles for our tests

Testable Behaviors:
- **addStudent should add a student to the database**
- **addStudent should return an ID that is distinct from any ID in the database**
- **addStudent should permit adding a student with the same name as an existing student**
- **Given the ID of a student, getTranscript should return the transcript for that student**
- **Given an ID that is not the ID of any student, getTranscript should throw an error**

A tiny example of Jest: jestSample.test.ts

```typescript
const alvin = {studentID: 37, studentName: "Alvin"}
const bryn  = {studentID: 38, studentName: "Bronwyn"}

describe("simple Jest examples", () => {

    test("extracting a studentID should give the ID", () => {
        expect(alvin.studentID).toEqual(37)
        expect(bryn.studentID).toEqual(38)
    })

    // this illustrates what Jest shows when a test fails
    test("extracting a studentID should give the name", () => {
        expect(alvin.studentName).toEqual("Alvin")
        expect(bryn.studentName).toEqual("Jazzhands")
    })

})
```

Now we can start writing tests

```javascript
import TranscriptService from "./transcriptService";

const db = new TranscriptService();

// start each test with a fresh empty database.
beforeEach(() => {
    db.initialize([]);
});

// illustrative tests for lecture slides
describe('tests for addStudent', () => {

  test('addStudent should add a student to the database', () => {
    expect(db.getStudentIDsByName('blair')).toEqual([])
    const id1 = db.addStudent('blair');
    expect(db.getStudentIDsByName('blair')).toEqual([id1])
  });
```
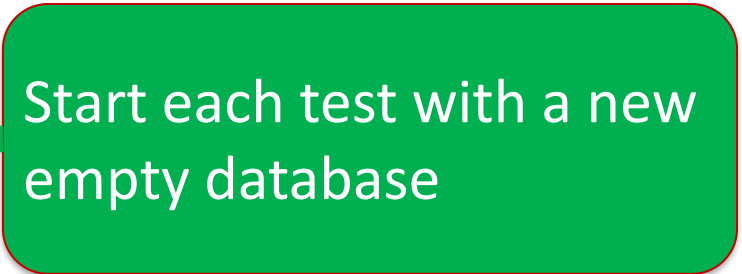
Start each test with a new empty database

Most tests are in AAA form: Assemble/Act/Assess

```
test('addStudent should add a student to the database', () => {

    expect(db.getStudentIDsByName('blair')).toEq...


    const id1 = db.addStudent('blair');


    expect(db.getStudentIDsByName('blair')).toEq...

});
```

Assemble (and check that you've assembled it correctly)

Act (do the action that you are trying to test)

Assess: check to see that the response is correct

Tests (2)

```
test('addStudent should return a unique ID for the new student',
    () => {
        // we'll add 3 students and check to see that their IDs
        // are all different.
        const id1 = db.addStudent('blair');
        const id2 = db.addStudent('corey');
        const id3 = db.addStudent('del');
        expect(id1).not.toEqual(id2)
        expect(id1).not.toEqual(id3)
        expect(id2).not.toEqual(id3)
    });
```

Tests (3)

```
test('the db can have more than one student with the same name',
    () => {
        const id1 = db.addStudent('blair');
        const id2 = db.addStudent('blair');
        expect(id1).not.toEqual(id2)
    })
```

Tests (4)

```
test('getTranscript should return the right transcript',
    () => {
        // add a student, getting an ID
        // add some grades for that student
        // retrieve the transcript for that ID
        // check to see that the retrieved grades are
        // exactly the ones you added.
    });
```

Tests (5)

```
test('getTranscript should throw an error when given a
bad ID',
    () => {
        // in an empty database, all IDs are bad :)
        // Note: the expression you expect to throw
        // must be wrapped in a (() => ...)
        expect(() => db.getTranscript(1)).toThrowError()
    });
```

Now we can write some code

```typescript
import { StudentID, StudentName, CourseID, Grade, Transcript } from "./Types";
import { ITranscriptService } from "./transcriptService.interface";

export default class TranscriptService implements ITranscriptService {

    /** the list of transcripts in the database */
    private transcripts: Transcript[] = []

    /** the last assigned student ID
     * @note Assumes studentID is Number
     */
    private lastID: StudentID = 0  // this should be private to nextID().
    private nextID(): StudentID { return this.lastID++ }

  constructor() { }
```

Code (2)

```
/** Adds a new student to the database
    * @param {StudentName} newName - the name of the student
    * @returns {StudentID} - the newly-assigned ID for the new student
    */
addStudent(newName: StudentName): StudentID {
    const newID: StudentID = this.nextID()
    const newTranscript: Transcript = {
        studentID: newID,
        studentName: newName,
        courses: [] // initially no courses
    };
    this.transcripts.push(newTranscript)
    return newID
}
```

A quick word about cleanup

```
beforeEach(() => {
  db.initialize([]);
});

describe('tests for addStudent', () => {

  test('addStudent should add a student to the database', () => {
    expect(db.nameToIDs('blair')).toEqual([])
    const id1 = db.addStudent('blair');
    expect(db.nameToIDs('blair')).toEqual([id1])
  });
});
```

Start each test with a new empty database

- Our test left a student in our database at the end of the test.

- Use afterEach() if needed.

Learning Goals for this Lesson

- At the end of this lesson, you should be prepared to
  - Explain the basics of the Test-Driven Design
  - Develop simple applications using Typescript and Jest
  - Learn more about Typescript and Jest from tutorials, blog posts, and documentation