# CS 4530: Fundamentals of Software Engineering

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

Adeel Bhutta and Mitch Wand

Khoury College of Computer Sciences

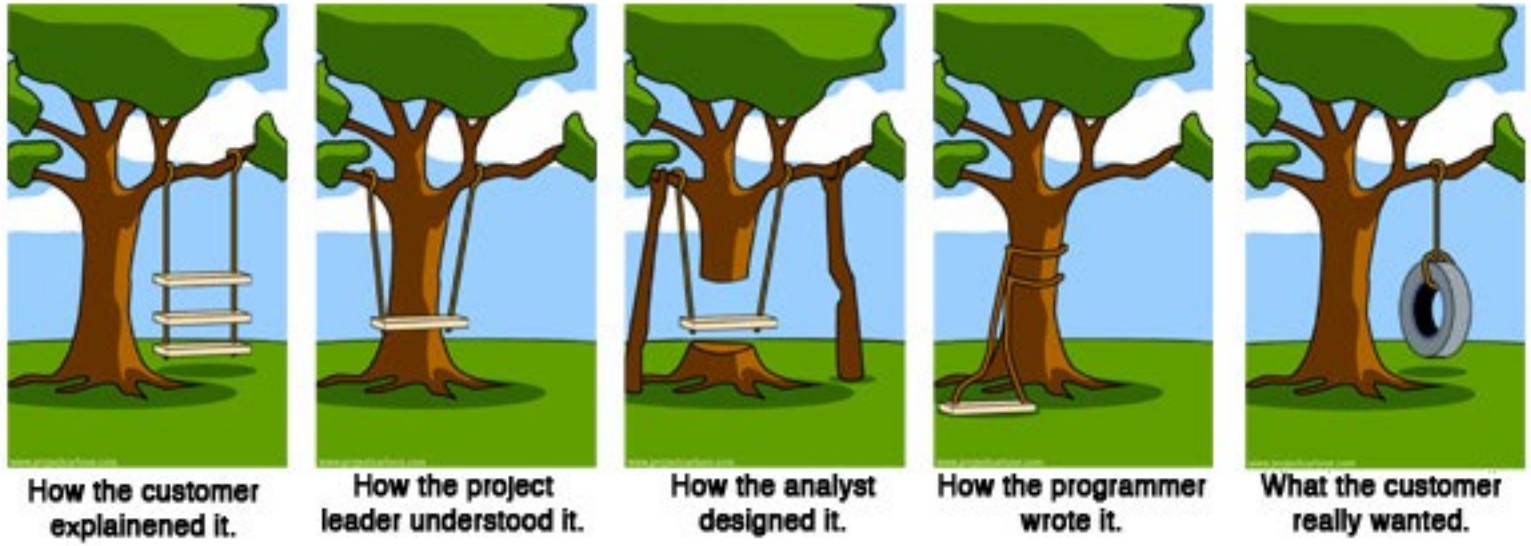
# Learning Goals for this Lesson

- At the end of this lesson, you should be prepared to
  - Explain the basics of the 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

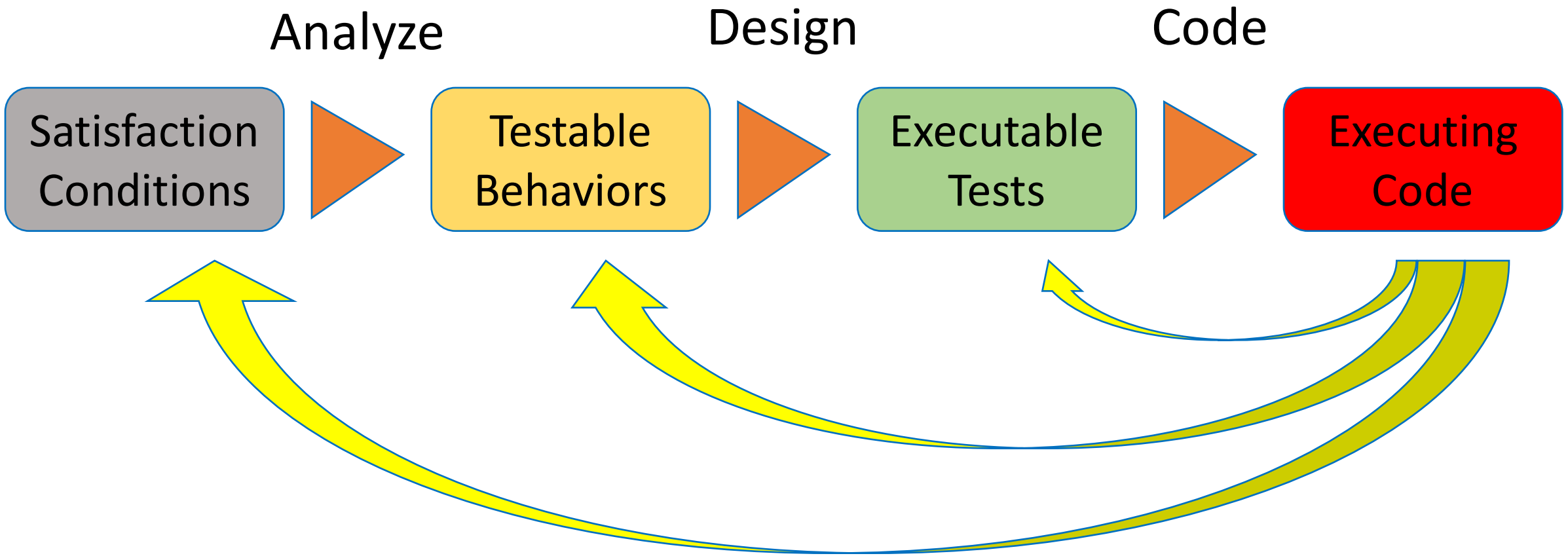
Planning & Design

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



# 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**

# 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 **IDataBase.ts**

# We start with the interface

```
import {StudentID, Student, Course, CourseGrade, Transcript} from './Types'

export interface IDataBase {
    addStudent (studentName: string): StudentID
    getTranscript (id: StudentID): Transcript
    deleteStudent (id: StudentID): void     // hmm, what to do about errors??
    addGrade (id: Student, course: Course, courseGrade: CourseGrade) : void
    getGrade (id: Student, course: Course) : CourseGrade
    nameToIDs (studentName: string) : StudentID[]
}
```

- The types are all *abstract*
- In the process of writing this down, we've discovered some more design decisions:
  - How to identify a student to the DB user
  - What to do about exceptional conditions in deleteStudent and elsewhere
  - We needed a new method to get from a student name to their ID.

# 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 <hmmm…. What *should* it do??????>**

# Writing down the testable behaviors may uncover more design decisions to make

- Here we realized that the user's satisfaction conditions didn't give us any guidance on the exceptional condition "not an ID of any student"
- What should **getTranscript** do?
- Possibilities:
  - return an error value (undefined, -1, etc.)
  - Throw an exception

# Testable Behaviors, revised

**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 exception**

# We still need to design some more before we can write some tests

- We wrote:
  - **Given the ID of a student, getTranscript should return the transcript for that student**
- But how can we test to see if the returned transcript is the right one?
- It must be time to elaborate the design of the type **Transcript**.

# Types.ts

```ts
// Types.ts
// Types for the transcript database.

export type StudentID = number;
export type Student = { studentID: number, studentName: StudentName };
export type Course = string;
export type CourseGrade = { course: Course, grade: number };
export type Transcript = { student: Student, grades: CourseGrade[] };
export type StudentName = string
```

# A tiny example of Jest: Types.test.ts is

```
import {StudentID, Student, Course, CourseGrade, Transcript} from './Types'

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

describe("exercise Types.ts", () => {

    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

```
import {StudentID, Student, Course, CourseGrade, Transcript} from './Types'
import { DataBase } from './dataBase';

let db: DataBase;

beforeEach(() => {
  db = new DataBase();
});

// this may look undefined in TSC until you do an npm install
// and possibly restart VSC.
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])
  });
```

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

```
test('addStudent should add a student to the data
    // const db = new DataBase ()
    expect(db.nameToIDs('blair')).toEqual([])

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

    expect(db.nameToIDs('blair')).toEqual([id1])
});
```

Assemble (and check that you’ve assembled it

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 an 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

```
import {StudentID, Student, Course, CourseGrade, Transcript} from './Types'
import { IDataBase } from './IDataBase'


export class DataBase implements IDataBase {

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

    /** the last assigned student ID; assumes studentID is Number */
    private lastID : number = 0
    constructor () {}
```

# Code (2)

```
/** Adds a new student to the database
 * @param newName - the name of the student
 * @returns the newly-assigned ID for the new student
 */
addStudent (newName: string): StudentID {
    const newID = this.lastID++
    const newStudent:Student = {studentID: newID, studentName: newName}
    this.transcripts.push({student: newStudent, grades: []})
    return newID
}
```

# Code (3)

```
/**
 * @param studentName
 * @returns list of studentIDs associated with that name
 */
nameToIDs (studentName: string) : StudentID[] {
    return this.transcripts
        .filter(t => t.student.studentName === studentName)
        .map(t => t.student.studentID)
}
```

# Activity

- Download and unpack the starter code
- Write down the testable behaviors for the satisfaction condition
    - **Add a new grade for an existing student**
- Identify at least two exceptional conditions or design decisions associated with these testable behaviors
- Write Jest tests for your testable behaviors
- Implement a method **addGrade** that passes your tests.

Your instructor will give you detailed instructions on where to get the starter code and how to submit your work.

# 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

# The TDD Cycle