CS 4530: Fundamentals of Software Engineering
Module 2.2: Test-Driven Development

Adeel Bhutta, Rob Simmons, 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 Test-Driven Development
  - Explain the connection between conditions of satisfaction and testable behaviors
  - Begin developing simple applications using TypeScript and Vitest

Non-Goals for this Lesson

- This is *not* a tutorial for Typescript or for Vitest
- We will show you simple examples, but you will need to go through the tutorials to learn the details.

Test-Driven Development

PEOPLE PROCESSES PROGRAMS

PLANNING

ORGANIZING

IMPLEMENTING

Requirements Analysis

User Stories

Testing Conditions of Satisfaction

Review: User Stories

- As a College Administrator, I want to keep track of students, the courses they have taken, and the grades they received in those courses, so that I can advise them on their studies.

As a <role>
I want <capability>
so that I can <get some benefit>

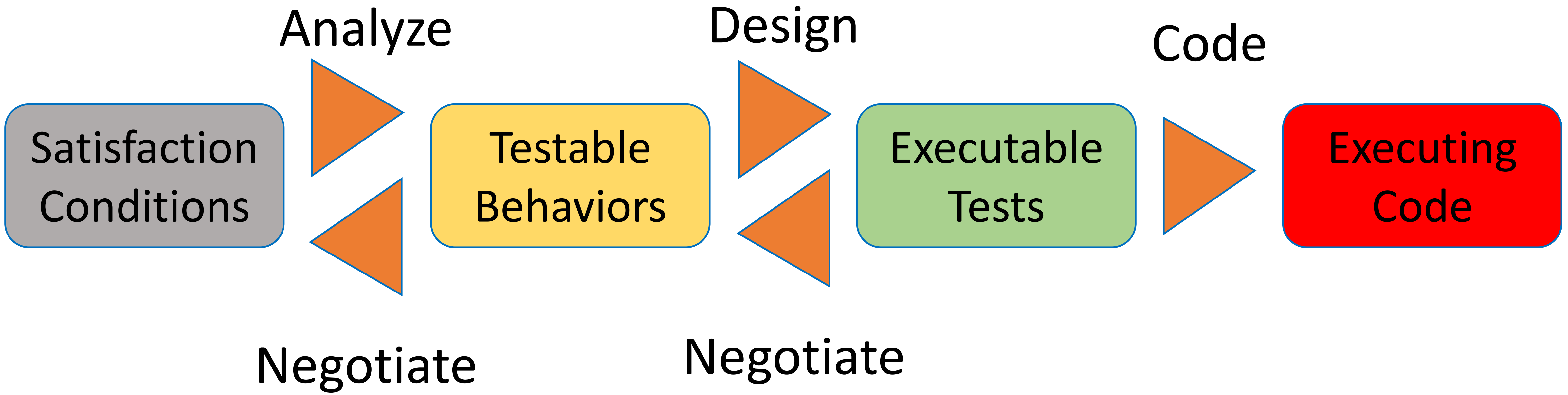
Review: Conditions of Satisfaction

- The college administrator can…
  - Access a persistent database of student records
  - Prevent unauthorized people from accessing or modifying the database
  - 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

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

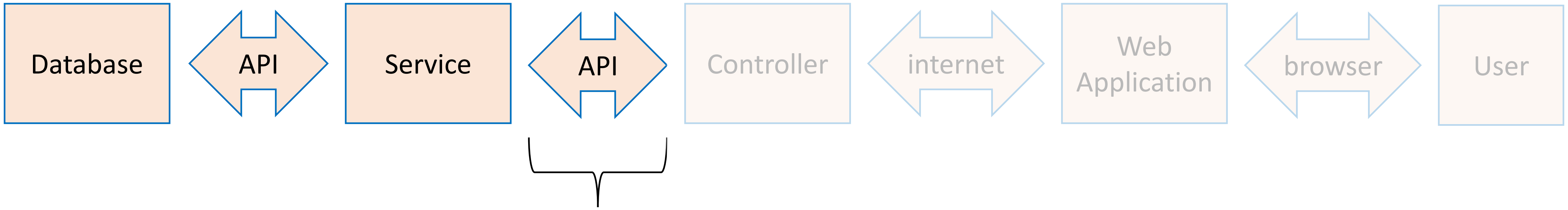
- Analyze:
  - how does the client want the functionality to be delivered?
  - what ambiguities can you find that might make a difference to the client?
- Negotiate
  - discuss these with the client and resolve the differences

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?

First negotiation: what are we going to test?

A user story is about a person in a specific role, who will need to access a complex application



We test specific parts of the application, still thinking about the user story and conditions of satisfaction

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!!

Now we can design the interface we are going to test.

src/transcript.service.ts

```
import { StudentID, Student, Course, CourseGrade, Transcript }
from './types.ts’;

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

Analyzing CoS to get testable behaviors

CoS: The college administrator can…

- …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

Testable behaviors:

- addStudent should add a student to the database and return their ID
- addStudent should return an ID distinct from any ID in the database
- addStudent should permit adding a student with the same name as an existing student
- getTranscript, given the ID of a student, should return the student’s transcript.
- getTranscript, given an ID that is not the ID of any student, should …*????...*

To run the tests, we'll need to design the data in a little more detail

```
// types.ts - types for the transcript service
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;
```

src/types.ts

The tiniest introduction to Vitest

```
// types.spec.ts
import { describe, expect, it } from 'vitest';
import { type Student } from './types.ts';

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

describe('the Student type', () => {
  it('should allow extraction of id', () => {
    expect(alvin.studentID).toEqual(37);
    expect(bryn.studentID).toEqual(38);
  });
  it('should allow extraction of name', () => {
    expect(alvin.studentName).toEqual('Alvin’);
    expect(bryn.studentName).toEqual('Jazzhands’); // will fail
  });
});
```

You can run tests from the command line

```
% npx vitest --run src/types.spec.ts

 RUN  v4.0.16 /Users/rjsimmon/r/transcript-server

 ❯ src/types.spec.ts (2 tests | 1 failed) 4ms
   ❯ the Student type (2)
     ✓ should allow extraction of id 1ms
     × should allow extraction of name 3ms

─────────────────────────────────────── Failed Tests 1 ───────────────────────────────────────

 FAIL  src/types.spec.ts > the Student type > should allow extraction of name
AssertionError: expected 'Bronwyn' to deeply equal 'Jazzhands'

Expected: "Jazzhands"
Received: "Bronwyn"

 ❯ src/types.spec.ts:13:30
     11|   it('should allow extraction of name', () => {
     12|     expect(alvin.studentName).toEqual('Alvin');
     13|     expect(bryn.studentName).toEqual('Jazzhands'); // will fail
       |                              ^
     14|   });
     15| });


 Test Files  1 failed (1)
      Tests  1 failed | 1 passed (2)
```

or better yet, use an npm script to automate this…

Now we can start writing tests

```
// transcript.service.spec.ts
import { beforeEach, describe, expect, it } from 'vitest';
import { TranscriptDB, type TranscriptService } from ‘./transcript.service.ts';

let db: TranscriptService;
beforeEach(() => {
  db = new TranscriptDB();
});

describe('addStudent', () => {
  it('should add a student to the database and return their id', () => {
    expect(db.nameToIDs('blair')).toStrictEqual([]);
    const id1 = db.addStudent('blair’);
    expect(db.nameToIDs('blair')).toStrictEqual([id1]);
  });
});
```

Start each test with a new empty database

Most tests are written in AAA Pattern: Assemble/Act/Assess

```
describe('addStudent', () => {
  it('should add a student to the database and return their id', () => {

    expect(db.nameToIDs('blair')).toStrictEqual([]);

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

    expect(db.nameToIDs('blair')).toStrictEqual([id1]);

  });
});
```

Assemble (and verify)

Act

Assess

Turn each testable behavior into one or more tests

```
describe('addStudent', () => {
  it(‘should return an ID distinct from any ID in the database',
() => {
    // 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 from Testable Behaviors (2)

```
describe('addStudent', () => {
  it('should permit adding a student w/ same name as an 
existing student’, () => {
    const id1 = db.addStudent('blair’);
    const id2 = db.addStudent('blair’);
    expect(id1).not.toEqual(id2);
  });
});
```

Tests from testable behaviors (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 from testable behaviors (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

src/transcript.service.ts

```
import { type StudentID, type Student, type Course, type CourseGrade,
         type Transcript } from './types.ts';

export interface TranscriptService { ... }

export class TranscriptDB implements TranscriptService {
  /** the list of transcripts in the database */
  private _transcripts: Transcript[] = [];

  /** the last assigned student ID
   * @note Assumes studentID is Number
   */
  private _lastID: number = 0;

  constructor() {}

  // etc
```

Code (2)

src/transcript.service.ts

```
/** Adds a new student to the database
 * @param {string} newName - the name of the student
 * @returns {StudentID} - 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;
  }
```

A quick word about cleanup

Start each test with a new empty database

```
let db: TranscriptService;
beforeEach(() => {
  db = new TranscriptDB();
});
```

OR

Create one database at the very start

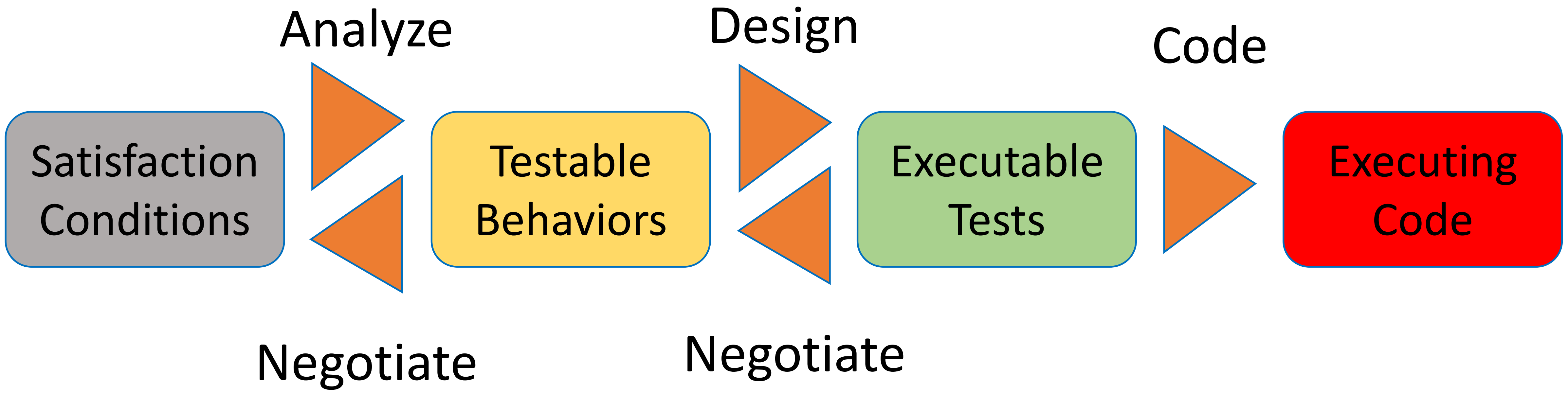
```
let db: TranscriptService;
beforeAll(() => {
  db = new TranscriptDB();
});
```

Start every test with the database cleared out

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

- Use `afterEach()` if needed.

Review: Here is the process we followed



Review: CoS to testable behaviors to TDD

CoS: The college administrator can…

- …add a new student to the database

Testable behaviors:

- addStudent should add a student to the database and return their ID
- addStudent should return an ID distinct from any ID in the database

It's the end of the lesson, so you should be prepared to:

- Explain the basics of Test-Driven Development
- Explain the connection between conditions of satisfaction and testable behaviors
- Begin developing simple applications using TypeScript and Vitest