

# CS 4530: Fundamentals of Software Engineering

## Module 2, Lesson 3

### Testing Conditions of Satisfaction

---

Rob Simmons

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
- Derive testable behaviors and tests from conditions of satisfaction
- Begin developing simple applications using TypeScript and Vitest
- Learn more about TypeScript and Vitest from tutorials, blog posts, and documentation

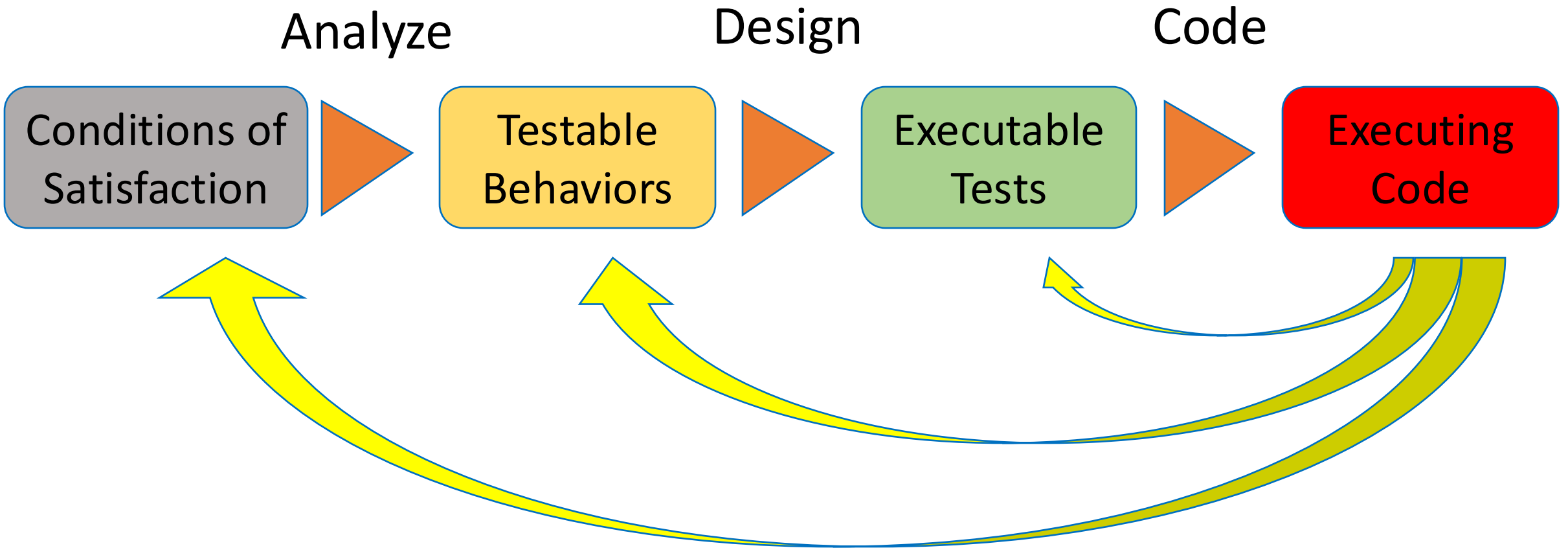
# Test Driven Development (TDD)

---

- 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

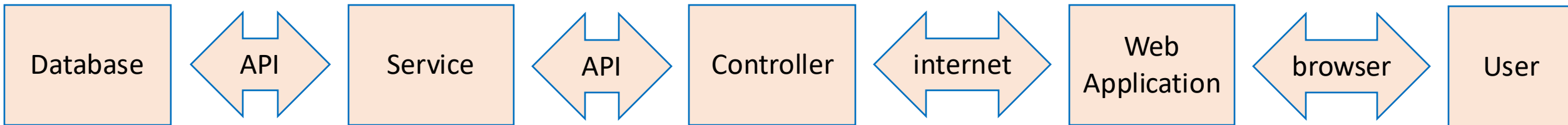
---



# CoS are ultimately about the user

---

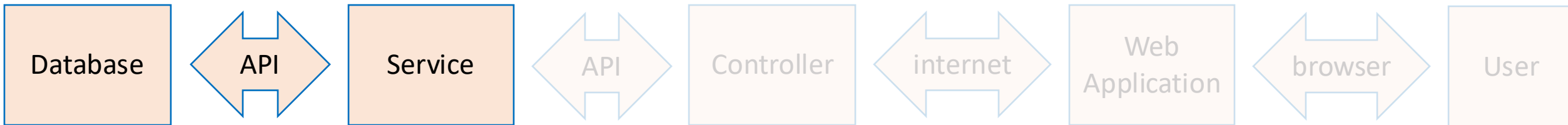
We will build a secure web application backed by a persistent database that allows an authenticated administrator to add a new student to the database



# CoS are ultimately about the user

---

We will build a secure web application backed by a persistent database that allows an authenticated administrator to add a new student to the database



The `addStudent` service function should add a student to the database

# Analyzing CoS to get testable behaviors

---

```
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: Student, course: Course, courseGrade: CourseGrade): void;  
  getGrade(id: Student, course: Course): CourseGrade;  
  nameToIDs(studentName: string): StudentID[];  
}
```

# Analyzing CoS to get testable behaviors

---

## CoS: The user 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
- Given the ID of a student, getTranscript should return the student's transcript.
- Given an ID that is not the ID of any student, getTranscript should ...????...



# Analyzing CoS to get testable behaviors

---

- The user-centric satisfaction conditions didn't give us any guidance on the exceptional condition "not an ID of any student"
- We need to elaborate: what should **getTranscript** do?
- Possibilities:
  - return an error value (undefined, -1, etc.)
  - Throw an exception
- Decision: throw an exception

# The tiniest introduction to Vitest

---

```
// transcript.service.ts
import {
  StudentID,
  Student,
  Course,
  CourseGrade,
  Transcript,
} from './types.ts';
export interface TranscriptService {
  addStudent(studentName: string): StudentID;
  getTranscript(id: StudentID): Transcript; // throws Error if id invalid
  deleteStudent(id: StudentID): void; // throws Error if id invalid
  addGrade(id: Student, course: Course, courseGrade: CourseGrade): void;
  getGrade(id: Student, course: Course): CourseGrade;
  nameToIDs(studentName: string): StudentID[];
}
```

# The tiniest introduction to Vitest

---

```
// 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;
```

# 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
  });
});
```

# The tiniest introduction to Vitest

---

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

```
RUN v3.1.2 /Users/rjsimmon/r/strategytown-su25/client
```

```
> types.spec.ts (2 tests | 1 failed) 4ms
✓ the Student type > should allow extraction of id 1ms
× the Student type > should allow extraction of name 3ms
  → expected 'Bronwyn' to deeply equal 'Jazzhands'
```

---

## Failed Tests 1

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

```
Expected: "Jazzhands"
Received: "Bronwyn"
```

```
> types.spec.ts:16:30
14|   it('should allow extraction of name', () => {
15|     expect(alvin.studentName).toEqual('Alvin');
16|     expect(bryn.studentName).toEqual('Jazzhands');
   |                               ^
17|   });
18| });
```

---

[1/1]—


# Turning testable behaviors into Vitest 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

# Turning testable behaviors into Vitest tests

---

```
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

# Turning testable behaviors into Vitest 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);  
  });  
});
```



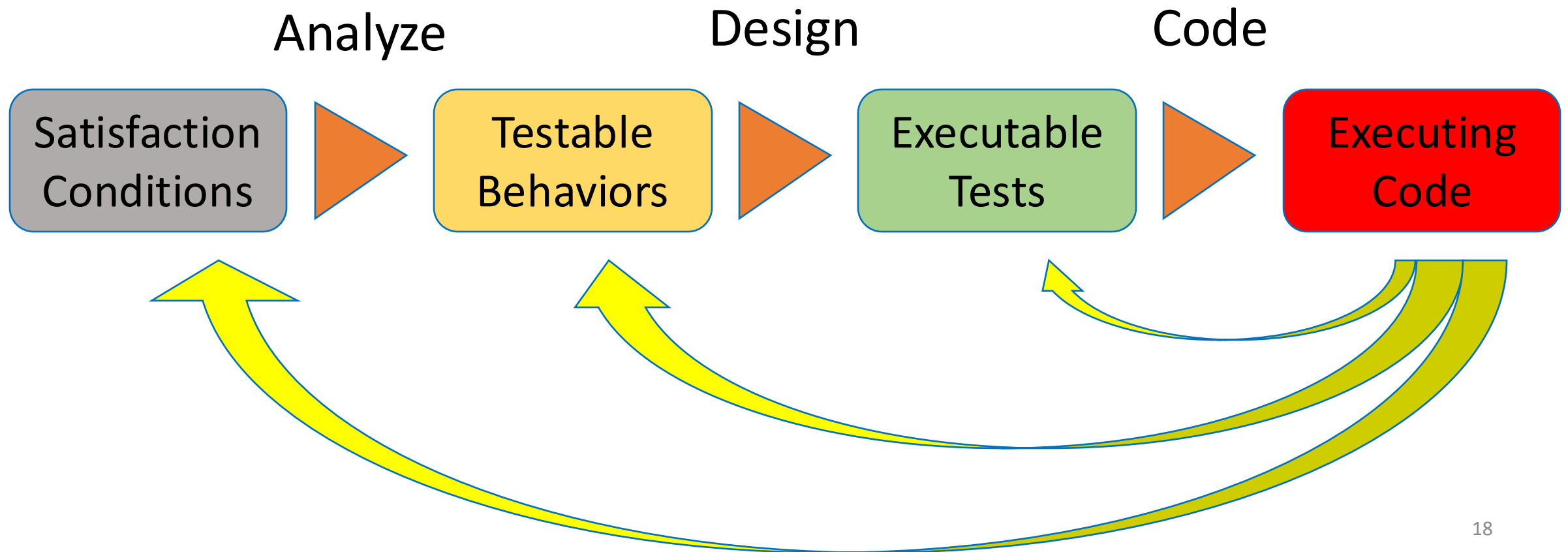
# Turning testable behaviors into Vitest tests

---

```
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);  
  });  
});
```

# ...now TDD lets us implement addStudent!

Implementing the TranscriptDB according to the TranscriptService spec will let us turn our testable behaviors into fully executable tests.



# Review

---

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

- Explain the basics of Test-Driven Development
- Derive testable behaviors and tests from conditions of satisfaction
- Begin developing simple applications using TypeScript and Vitest
- Learn more about TypeScript and Vitest from tutorials, blog posts, and documentation