COURSE OUTLINE

Composable Software Construction

The mathematical foundations of simple, reusable, maintainable and dependable code in Typescript.

INSTRUCTOR

Lakin Wecker

lakin@structuredabstraction.com

structuredabstraction.com

Introduction

Traditionally, schools teach students how to write small pieces of code and how to solve algorithmic challenges, but do not prepare them for writing effective software as part of a larger team. Typically, the primary experience they will have is from a course or book they have read on "Software Engineering". The term Software Engineering frequently refers to a list of 4 important, but incomplete, aspects of producing software :

1. Working in groups.
2. Requirements/project management.
3. Design patterns
4. System Architecture

While these are important subjects, it skips over the most fundamental aspect of software development: actually writing the code. In Code Complete, McConnel [1] uses the term Software Construction to label what comes after the design and before the deployment. It is the act of writing code to satisfy project requirements.

However, when the student begins their journey as a professional software developer, they quickly realize that their schooling has not prepared them to bridge the quality-gap between their code and the code found in top-tier libraries
In the introduction to Rust for Rustaceans, David Tolnay explains how a deep comfort with fundamental language and programming concepts is the most important building block to crossing this skill-gap. The gap may seem insurmountably large and choosing where to start is confusing. This course gives the students the tools they need to begin their journey across the gap. It takes them on an in depth tour of the mathematical foundations that power the patterns and structures in effective software development. This course provides the students the ability to take a well-defined task and identify the appropriate mathematical techniques which will allow them to construct software that will be praised by their colleagues for its clarity, succinctness, dependability and reusability.

Overview

The course takes 40 hours with 16 classes over an 8-10 week timeline. Each class takes 2.5 hours and will cover into three aspects of software development::

1. **Theory**
   Introduction to each of the concepts in a mathematical, diagrammatic way using a functional style notation. Motivated by a real-world problem, I will show the students how they can use existing theory to solve these problems. At the end of these sections, a simple in-class quiz will be given to assess learning outcomes.
2. **Typescript**
   I cover typescript features which are directly related to the previously introduced theory in detail. I develop code on-screen during the session, with feedback and interaction from the students. Care will be taken to show how the typescript code is compiled to javascript. A simple online in-class quiz format is used to assess learning outcomes. A laptop or access to a computer for development will be necessary to complete some sections of the quiz.
3. **Industry examples + Group coding session.**
   An example from real projects (typically a project from the company) will be presented. These examples will use imperative coding paradigms that are taught in many courses and are common in industry among new programmers. The examples will not use the concepts taught during the class and students will be challenged to improve the code into something which does use these concepts with guidance from the instructor.

Logistics & Scheduling

This course is provided on a per-client scheduled basis for a minimum of 2 students and maximum of 6 students at a time. Tuition is due in full before the start of the first class. The course can be held over zoom, or in person at a client-provided location within Calgary, AB.

The course includes 16 classes, which are scheduled according to the availability of the students and instructor within a 10 week period.

Cost

The cost is based on the time spent by the instructor. Additional students incur additional time spent preparing, evaluating and tutoring outside of in class hours. The base cost of the course is $12,000 and includes 2 students, each additional student costs an additional $1500.

- 2 Students - $12,000 total
- 3 Students - $13,500 total
- 4 Students - $15,000 total
- 5 Students - $16,500 total
- 6 Students - $18,000 total

Outcomes:

Students of this course are expected to gain a measurable elevation of their software development abilities. The primary outcome is an increased ability to read, design and write high quality software. The particulars of this are:

- Observable decrease in run time errors with a goal of 0
- Increased ability to construct maintainable software through an improved ability to identify, design and construct code with well-known composable design-patterns
- Increase in ability to write self-documenting code, expressing ideas, constraints and domain-specific concepts in types and letting the compiler enforce them
- Increased ability to approach software modelling based on mathematical concepts and notation.
- Knowledge and implementation of algebraic data types
- Knowledge and implementation of functional programming best practises

These outcomes will be materialised by the students through the course requirements of in class discussion, hands on coding, and industry examples. Retention and proper implementation of course material will be evaluated on a continuous basis to ensure the acquired skills persist beyond the completion of the course.

Syllabus

Week 1 > Class 1 > Introduction & Theory

- Goals and ideas of course.
- Starting from basics (functions/expressions/statements/purity)
- Types, and type safety
    - Using the compiler to help us
- Error Handling
- Get to a point where you can understand and use fp-ts
- Functions revisited:
    - Basics:
        - Notation
        - Inputs / Outputs
        - Purity and data immutability
        - Side Effects
        - Signature
        - Examples from Elm
    - Composition
        - What does this mean?
        - bWhy do we care
        - Where does composition show up?
            - Objects
            - Functions
            - Examples from scala

Week 1 > Class 1 > Typescript

- Functions in typescript:
    - Typical Functions & Alternate function syntax (function keyword vs =>)
    - IIFE (Expressions vs Statements)
    - Typing functions
    - Mutability
    - Immutability

Week 1 > Class 2 > Theory

- Functions Again:
    - Functions as sets
        - Set Theory
    - Multiple Arguments
    - Function Application
        - Elm
        - Typescript
        - Scala
    - Partial Application
    - Currying
    - Why do we use this? Why do we care?
    - Example in Elm
- Higher Order Functions
    - Identifying the parts of a function
    - Which parts are static?
    - Which parts are dynamic?
    - How can we customise these parts?
    - Variable naming convention
    - Argument order

Week 1 > Class 2 > Typescript

- Typescript:
    - When do we use these in typescript?
    - Application
        - .call
        - .apply
        - .bind
    - Partial Application
    - Currying
- Higher Order functions in typescript
    - Implementing a function purely from its type.

Week 2 > Class 3 > Theory

- Other functions we can implement directly from their type.
    - map
        - Where does the name 'map' come from?
    - filter
    - apply
    - identity
    - compose
    - reduce
    - ramda-js - searching by "Type"

- Function Customization
    - Closures and Scope
    - Higher Order Functions in other languages
        - Scala

Week 2 > Class 3 > Typescript

- Functions as variables in typescript
    - How to store them
        - What is their scope?
    - Arity
        - function length

Week 2 > Class 4 > Theory

- Types revisited
    - What are they?
        - Representation vs Behaviour vs Domain
        - primitive types vs builtin types vs user-defined types
            - typeof
    - Composite types (composition again!):
        - Product types
            - Records.
            - What number of valid states are there?
            - What implications does this have for maintenance?
            - Examples in Elm
        - Sum types
            - Variants
            - Enums
            - Wrapping Data
            - What number of valid states are there?
            - Examples in Elm
            - Matching
                - Sum types
                - Elm
                - What about product types?

Week 2 > Class 4 > Typescript

- Types in typescript:
    - Records
    - Types
    - Interfaces
    - Mappings
    - Variants
    - Matching
    - Making the compiler work for you.

Week 3 > Class 5 > Theory

- Generics & Types
    - Similar to dynamic portions of a function, how do we get dynamic portions of a type?
    - Where is this useful?
    - Constraints on types
    - monomorphization
    - polymorphization
- Discriminated Unions

Week 3 > Class 5 > Typescript

- Build an Option in typescript
    - Discriminated Unions in typescript
    - Type Guards
    - Matching
    - Generics
    - How does the compiler implement these
- Let's implement something together
    - We'll make something that represents an optimization result
    - We'll implement type guards (again)
    - We'll implement a match
    - Can we implement a partial match?
    - What is the usefulness of both?
    - Can we implement a better one?

Week 3 > Class 6 > Theory

- Builder Pattern
    - How it's used in other languages
- What it's used for.

Week 3 > Class 6 > Typescript

- Using the builder pattern to implement a match
- Let's build it up with type safety
    - Type Algebra
    - Exhaustivity
    - Using utility types
- Using them with record types

Week 4 > Class 7 > Theory

- Making your tools work for you.
- How can we do less work, but write better code?
- Continuous integration
- Continuous testing
- Continuous delivery
- Editors
- Refactoring
- Docs on hover
- Git
- Learning from colleagues
    - Typescript compiler configuration.
    - Configuring VS Code.
    - Turning off "any".
    - Using linters
    - Using hinters
    - Using CI
    - Using CD
    - Refactoring tools built into your editor.
    - Type-info on hover.
    - Using type information to navigate the code base.s

Week 4 > Class 7 > Typescript

- Review of Utility types in Typescript
    - Partial<Type>
    - Required<Type>
    - Readonly<Type>
    - Record<Keys, Type>
    - Pick<Type, Keys>
    - Omit<Type, Keys>
    - Exclude<UnionType, ExcludedMembers>
    - Extract<Type, Union>
    - NonNullable<Type>

    - Parameters<Type>
    - ConstructorParameters<Type>
    - ReturnType<Type>
    - InstanceType<Type>
    - ThisParameterType<Type>
    - OmitThisParameter<Type>
    - ThisType<Type>
    - Intrinsic String Manipulation Types
    - Uppercase<StringType>
    - Lowercase<StringType>
    - Capitalize<StringType>
    - Uncapitalize<StringType>
- TSConfig Settings
- strict
    - noImplicitAny
    - noImplicitThis
    - alwaysStrict
    - strictBindCallApply
    - strictNullChecks
    - strictFunctionTypes
    - strictPropertyInitialization
    - forceConsistentCasingInFileNames
    - noFallThroughCasesInSwitch
    - noUncheckedIndexedAccess
    - isolatedModules
    - noUnusedLocals

Week 4 > Class 8 > Theory

- Recursion
    - Sum
    - Product
    - Base case
- Recursive data types
    - Building this with a discriminated union

Week 4 > Class 8 > Typescript

- Using this to implement a recursive list data type
- Implement basic methods:
    - tail
    - head
    - length
    - drop
    - dropWhile
    - append

Week 5 > Class 9 > Theory

- Making impossible states a compiler error
    - Let's build a questionnaire
        - State knows which question they're on, and information about their answers.
        - No empty questionnaires, never beyond the limits.
    - Let's make it so that we literally can't represent invalid states

Week 5 > Class 9 > Typescript

- Let's build a complete zippered list in typescript
    - Functions for going backwards and forwards and answering questions.

Week 5 > Class 10 > Theory

- SOLID Design principles
- Contravariance / Covariance
- Consumer / Producer

Week 5 > Class 10 > Typescript

- Refactor the zipper list to use the single responsibility
- Show how we can also use "mapping" to make certain problems easier to solve with a different type
- Discuss how this affects our "impossible states as compiler errors"

Week 6 > Class 11 > Theory

- How can we deal with functions that may have an error condition?
- Exceptions?
- What about languages that don't have exceptions?
- Result<T>
- Either<L, R>
- Other uses of Either

Week 6 > Class 11 > Typescript

- Implementing a Result type in typescript.
- fp-ts Either
- Rust's result type
- How rust works compared to

Week 6 > Class 12 > Theory

- Making impossible states impossible
- Fetching data from a remote data store
- What are the various valid states?
- How can we represent them?
- RemoteData<T>

Week 6 > Class 12 > Typescript

- How can we implement Remote data in typescript?
- Let's implement a simple one.
- What about match?

Week 7 > Class 13 > Theory

- Advanced functions
- Thinking of them as a pipeline
- Thinking about what a function does
- Not how it does it.
- Advanced operations available
- pipe, fold, match, etc. Other more advanced algorithms.
    - partition, etc.
    - monoid
    - semi-group
    - group
    - merge

Week 7 > Class 13 > Typescript

- Implementing these advanced function types in typescript
- Identifying where they are used.
- fp-ts.
    - Groups, Monoids and Semi-groups
    - Using this to implement well known patterns (like merge)
    - How these compositional techniques eliminate errors

Week 7 > Class 14 > Theory

- Refactoring code to be readable.
- What is readable code?
- Why do we care?

Week 7 > Class 14 > Typescript

- More group sessions refactoring code from existing code base

Week 8 > Class 15 > Theory

- Software architecture
- How do we make software that works well with each other?
- How do you design interfaces
- What code goes where?
- Single-responsibility

Week 8 > Class 15 > Typescript

- Implementing all of this in typescript
- Making certain things private
- Testing
- Linting

Week 8 > Class 16 > Theory

- The onion architecture
- Dependency injection
- Domain specific design

Week 8 > Class 16 > Typescript

- Implementing all of this in typescript.

The course includes 8 weeks of material. Depending on the level of the students, and their progress, the following "Bonus" classes may be taught near the end of the course.

Week 9 > Class 17 > Theory

- Understanding Javascript Module Formats
    - Commonjs
    - Require
    - AMD
    - UMD
    - ESM
    - Transpiling
    - What works in a browser/node and what doesn't

Week 9 > Class 17 > Typescript

- Transpiling Javascript with babel
- Transpiling compiled typescript with babel
- package.json/tsconfig.json inclusions/modifications for facilitating different scenarios

Week 10 > Class 18 > Theory

- Domain Driven Design
    - Modeling software and data for a business domain
    - Placing uncertainties at the edge
    - Comparison to OOP, Object type, and FP
    - Why is this important?

Week 10 > Class 18 > Typescript

- Types revisited
    - User defined types
    - Type composition
    - Required vs optional

References

[1] - https://www.oreilly.com/library/view/code-complete-second/0735619670/