EXCLUSIVE
Advanced Software Design
Dojo

As an alum of **Mirdin's** introductory code design course, you've witnessed first hand the game-changing value and efficiency benefits of designing and executing excellently-designed code.

But now you need more. You KNOW there's still a world of skills, techniques and models to learn and want to accelerate your path to peak proficiency. This is your opportunity for greatness.

The **Advanced Software Design Dojo** is a *group coaching* program designed to help you and four other pros learn and practice the most advanced topics in contemporary software design. You'll study together while also benefiting from premiere individualized instruction and live feedback.

You've laid the roots. Now it's time to help it **grow**. And grow BIG!

Format of Program

The program objective is to engage and upskill a cohort of **5 participants** focusing on the most defining and critically transformative topics related to practical software design.

You will emerge far better equipped to drive clearer, faster, more business-enhancing results in your teams, departments and companies.

The program consists of twenty-one (21) 90-minute calls spread over a manageable 6 month timeframe. Always interactive and dynamic, some calls will be delivered without customary student prep, others will require moderate prework for maximum learning.

The program will be delivered across the following three formats:

- Expository sessions cover some new topic. Around half of each session will involve solving practice problems.
- In design-exercise sessions, you are asked to design and not-implement a solution to some problem.
  The coach will review your solution and realistically "criticize" design mistakes with an adversarially-oriented feature-request.
  These sessions occur at the end of a unit, and have problems meant to highlight the skills learned in that unit. If you took the web course, the design exercise problems were a pre-fabricated version of this format.
- *Wilderness sessions* involve looking at examples of discussed concepts in open-source code, or going into detail about a solution to a previous problem.

Sessions will be scheduled 2-8 weeks in advance, with the goal of maximizing the number of sessions the collective cohort can attend. Generally sessions will be at the same time every week, but some weeks will be skipped.

All calls will be recorded, except for design-exercise sessions.

Also included is a private one hour ***1-on-1 coaching call*** with the instructor to be completed in the middle of the program.

In comparison to the web course, topics may be more *advanced* and/or more niche.

For example, the unit "The Truth about OO" was originally designed for the web course, but was removed when most of the test audiences didn't understand. Meanwhile, the design exercises in this program are *highly* challenging. The exact topics covered will shift based on group preferences.

The coaching group places a higher emphasis on 1-1 interaction and live feedback with a secondary emphasis on polished lectures.

Pricing

$4990 for the full six month program, due two weeks before the first session

Pricing

- Option 1: $4990, due before the first session
- Option 2: $875/month for 6 months
- Option 3: $455/month for 12 months

Example Syllabus

It is guaranteed that topics here will not match the schedule of your group. Topics can be done in different levels of depth, taking different amounts of time, and will be selected based on group interest. But this should give you an idea of the already-prepared material available.

Unit: The Truth About Objects

Session 1: What is an object?

- You've all heard very strong claims against OO, from it being synonymous with modularity and good design, to it being spaghetti code. We'll learn about language-independent definitions of objects and begin to build an understanding on which we can evaluate these claims objectively.

Session 2: Object encodings and inheritance.

- A hardcore version of the previous lecture. We will learn how to translate specific design proposals and language features involving objects into type theory, where their costs and benefits become easier to evaluate.

Session 3: Design exercise: Deck-building Game

- Our first design exercise: building a digital deck-building game inspired by Dominion and its online implementation. This one will require some major objects.

Session 4: Visitor combinators & freestyle

- We will learn visitor combinators, a little-known yet highly useful programming technique for modular traversals, which is best viewed as an advanced form of inheritance. Extra time is allotted to wrapping up previous topics.

Unit: Construction and Destruction

Programs are structured differently depending on if they're focused on producing outputs or consuming inputs. Ever needed to rewrite a desktop program to be client/server and found you needed to redo everything, even though the logic was the same? This material explains why, and what the exact correspondence is between both styles. This theme explores three different instantiations of this duality.

Session 1: Data and Codata

- Why is "stream-based" programming a thing, while "list-based" programming is not, when they seem to represent the same thing? It's because lists are data, which we're all used to. But streams are codata. Come learn why entire ecosystems need to be built separately around each.

Session 2: Defunctionalization: The Best Refactoring You've Never Heard Of

- What do these design changes have in common?
    - Letting a search procedure take an arbitrary filter function, instead of a fixed set of options
    - Changing a program using blocking I/O to non-blocking I/O with an event loop
    - Letting a user stop in the middle of an action — and resume it after a server reboot
- Answer: They're all instances of a transformation called "defunctionalization" or its inverse, "refunctionalization." From the design of web backends to permission systems, many design problems can be reduced to this handy technique.

Session 3: Existential and Universal Modularity

- "Modularity" as you've normally heard it is hiding the internals of a module from the rest of the program. There is another kind of modularity though: having the rest of the system is hidden from a module. Come learn this inverted way of thinking about generic code.

Session 4: Design exercise: Fraud Detection

- Our next design exercise will put everything you've learned to the test. Can you outrace the expanding requirements?

Unit: Directed Testing

Test more, but don't write brittle tests. Unit tests are key, except they don't really tell you anything and you need integration tests. This unit will explore more

Session 1: Testing is about properties

- We'll learn about how to break the correctness of code into a handful of discrete properties, and then write tests to check that each property holds. This leads to the idea that "good tests compose like proofs," and helps you understand what tests should not check, and what assurances certain tests actually give you. We'll also explore property-based testing, a technique which uses this idea to reduce test burden while increasing the promises given by tests.

Session 2: Tests vs. Simplicity

- We'll explore the analogy between tests and curve fitting, and learn about the tradeoff between test specificity and model complexity. We'll explore using a program synthesis tool to discover when a certain class of programs contain examples which pass all tests, but are incorrect.
- This is a long version of material discussed in the latter part of "You are a Program Synthesizer"

Session 3: What is a "case?" Tests and proofs

- We'll explore the connection between testing and proving, and learn an objective, albeit difficult, measure of what is the exact set of tests needed to show some code is correct.

Session 4: In Soviet Russia, Tests Fail You!

- In this design exercise, participants will both design a system and explain what tests they'd write. Then they'll turn on each other to discover both incorrect implementations that pass all tests, and innocuous changes that break tests.

Unit: Speed-Reading Code

Part of your power as a developer comes from the number of codebases you can competently read, reason about, and modify. And that comes from your ability to rapidly learn new codebases.

In the web course, we taught you the magic trick (data-structure centric reading). But there's more to learn.

Session 1: Dataflow patterns

- Given a web app in an unfamiliar framework, how would you find the routing code that maps URLs to their handlers? In this lesson, we'll discuss how to identify concepts in code by searching for data flow patterns.

Session 2: Pushouts and transformations

- This lesson covers in detail two concepts that we've previously covered in passing.
- In the first part of this lesson, we'll cover recognizing pushouts. In software, pushouts explain how a generic interface for e.g.: lists combines with a generic interface for e.g.: packet handlers to yield an API for adding/removing packet handlers. Because many interfaces are implicitly constructed in this fashion, recognizing pushouts can help you more rapidly understand a large API.
- In the second part, we'll study a few common transformations that people manually do to code, and hence learn to look at complicated code that addresses many concerns (e.g.: a single loop that reads, transforms, and writes data), and recognize the simple pieces for each concern (e.g.: it was created by fusing separate operations for reading, transforming, and writing).

Session 3: Practice

- In this session, we'll take turns each attempting a different 15-minute code-reading challenge. After each one, we'll debrief about what strategies they could have used to answer the question.

FAQ

Q: How much work is involved in the Dojo?

A:

Short answer: Much, much less than the web course.

Longer answer: The program will be divided into units (each lasting 4-6 weeks), each of which comes with a handful of readings. For weeks with design exercises (once every 4-6 sessions), you will be asked to prepare the first round of the design exercise ahead of time. Further, you may choose to read about things mentioned in the sessions, chat with the other members, or rewatch recordings. But, otherwise, there is no work outside of attending the sessions themselves.

Q: What aspects of the Dojo have proved stressful or troublesome for past participants?

A:

At least one: There is some live coding in C. (Hand-coding objects in a language that does not support them natively.)