EECS4302 Compilers and Interpreters
Fall 2026

Digest of Lecture 3, September 17

Cohesion, Single Choice, and Compiler Infrastructure

Jackie Wang

How to use this Digest

Use this digest as a **roadmap and checklist**. It highlights the main ideas, important examples, common pitfalls, practice targets, and where to return in the course materials. It does **not** replace attending or reviewing the session, using the recording when needed, studying Jackie's annotated iPad notes and other official course materials, completing assigned work, or practising independently. Use it to identify what you understand, what you need to revisit, and what you should be able to do on your own.

If time is limited, start with the sections most relevant to your current review and return to the rest later; selective use is better than skipping the digest because it looks long.

This is an AI-assisted study roadmap reviewed by Jackie and grounded in the session transcript and Jackie's annotated iPad notes. Like any course document, it may still contain an error or omission. If something seems technically doubtful or inconsistent with the original course materials, return to those materials and ask Jackie for clarification rather than treating the digest as final authority.

Table of Contents

Main Ideas 2

Roadmap 2

Key Terms, Notation, and Structures 5

Key Examples and Demonstrations 6

Common Pitfalls and Misconceptions 7

What You Should Be Able to Do 8

Suggested Practice 9

Where to Review 9

Closing Reminder 10

Main Ideas

- **Cohesion** asks whether the attributes and methods of one class serve the same purpose or fit under a unifying theme. **Single Choice Principle (SCP)** asks whether a change can be localized to the minimum number of places rather than being duplicated throughout the software.
- The `Student` case study showed why a type-code design can be troublesome. Keeping both `premiumRate` and `discountRate` in every `Student` weakens cohesion, while repeatedly branching on `kind` duplicates change points and violates SCP.
- An inheritance-based design separates common student behavior from resident/non-resident variation. Subclasses keep only the relevant rate and reuse common tuition logic through the parent class, removing repeated type checks and giving a single place to add a new student kind.
- **Regression testing** means re-running all existing tests after the model changes. Passing tests establishes correctness only with respect to the tests actually run; coverage and test quality still matter.
- A typical compiler can be organized around a **front end**, an **Intermediate Representation (IR)**, and a **back end**. Source and target programs use concrete syntax, while the IR is an abstract runtime structure used inside the compiler.
- Shared IRs can make a multi-language compiler architecture additive rather than multiplicative. Jackie contrasted building 20 directed language-to-language compilers for five languages with building five front ends plus five back ends around shared IRs.
- The session ended by locating **scanner** and **parser** in the compiler front end: scanning performs lexical analysis, parsing performs syntactic analysis with respect to a grammar, and ANTLR can generate these components from suitable language specifications.

Roadmap

1. Using the lecture digest and preparing for assessment

Jackie began by showing where the Lecture 2 digest appears on the lecture site and how its sections are meant to be used. The digest is a textual roadmap: it summarizes main ideas, terms, examples, pitfalls, mastery checks, practice, and cross-references, but students should return to Jackie's annotated iPad notes and the recording for full diagrams and reasoning. Jackie noted that the written test and final exam are closed-book. He may not announce every posting, so students should check the lecture site proactively. The syllabus was also still to be updated to reflect the class-attendance bonus.

2. Cohesion: one class, one unifying purpose

On page 3 of Jackie's annotated iPad notes, cohesion is stated for an individual class: its attributes and methods should **serve the same purpose** or be **defined under a unifying theme**. Jackie emphasized that design judgments can contain gray areas, so a claim that cohesion is satisfied or violated should be justified rather than asserted mechanically.

The Unix principle "do one thing and do it well" supplied the analogy. Commands such as `ls`, `cd`, and `pwd` each have a focused purpose; multiple operations can be composed when needed rather than being fused into one oversized command.

3. Single Choice Principle: localize the effect of change

Jackie defined SCP as requiring a change to affect a **minimum number of places** in the software. Minimum does not necessarily mean one; the important point is that the number of affected classes or methods should be no larger than necessary and should be defensible.

A common warning sign is duplication. If the same logic occurs in several places, a change must be repeated and may be performed inconsistently. Jackie noted that helper methods are one common way to factor duplicated logic into a single reusable place.

4. First `Student` design

Pages 4–6 of Jackie's annotated iPad notes develop a first design for `Student`. The class contains `kind`, `premiumRate`, and `discountRate`. A `kind` value of 1 represents a resident student; a value of 2 represents a non-resident student. Both `getTuition()` and `register(...)` branch on that type code.

The cohesion problem appears at the object level: a resident student uses the premium rate but not the discount rate, while a non-resident student uses the discount rate but not the premium rate. The class therefore contains state that serves different kinds of objects rather than one unified purpose.

The SCP problem appears in the repeated `kind` checks. If a third kind is introduced, every method containing the duplicated conditional may need another branch; removing a kind creates the same maintenance problem in reverse. The change is spread across multiple places even though the variation conceptually belongs together.

5. Inheritance-based redesign

Page 7 refactors the design around a parent `Student` class. The resident variant is `ResidentStudent`; the non-resident variant is `NonResidentStudent`. Common student data and common tuition computation remain in the parent. Each subclass contains only its relevant rate and can call `super.getTuition()` to reuse the common base calculation before applying its own variation.

This arrangement improves cohesion because unrelated rates are no longer stored in the same kind of object. It also improves SCP because adding a new student kind can be localized by adding a new subclass instead of editing repeated type-code branches throughout existing methods. Jackie pointed ahead to the Visitor design pattern, where the idea of controlling where change occurs will reappear.

6. Regression testing and debugging discipline

Page 8 shows the regression-testing loop. Start from the Java model, derive tests that cover normal and abnormal cases, run them, and investigate failures with a debugger and breakpoints. Jackie discouraged relying on `System.out.println` as the main debugging technique at this level.

A green test run does not prove absolute correctness. Jackie stressed that the software is correct **with respect to the tests run**; therefore, test quality and coverage matter. He used Dijkstra's testing principle to reinforce the distinction: testing can reveal bugs but cannot establish that no bugs remain. Regression testing then means running all existing tests again whenever a change is introduced.

7. Compiler infrastructure: concrete syntax, IR, front end, and back end

Page 9 introduces a simple compiler architecture. The source program is written in **concrete syntax**, such as an ASCII/plain-text Java file. The front end transforms that source program into an **Intermediate Representation (IR)**. The IR is the compiler's **abstract syntax**: an internal runtime structure, often tree-like, rather than the source text seen by the external user. The back end transforms the IR into the concrete syntax of the target program.

This decomposition allows later stages such as optimization to operate on an internal representation without changing the external source and target forms.

8. Multi-language compilation: multiplicative versus additive architecture

Page 10 asks how to support translation among Java, C, C#, C++, and Python. Building every directed pair separately gives five input choices times four other output choices, or 20 compilers. Jackie called this the **multiplicative** approach.

He then grouped language features into shared semantic domains/IRs: basic procedural features, advanced procedural features such as pointer arithmetic, and object-oriented features. With those IRs acting as a hub, each source language needs a front end into the appropriate IR combination and each target language needs a back end from it. For five languages, the schematic count is five front ends plus five back ends. Jackie called this the **additive** approach. Chaining one compiler into another can sometimes reduce work too, but he presented the shared-IR architecture as the more general pattern.

9. Scanner and parser in a conventional compiler pipeline

Page 11 moves to a more conventional high-level-to-machine-oriented compiler. The **scanner** performs lexical analysis; Jackie initially described this as checking whether tokens are formed correctly. The **parser** performs syntactic analysis with respect to a context-free grammar. Jackie noted that ANTLR can generate scanner/parser infrastructure once the relevant grammar and regular-language patterns are specified.

The lecture stopped at this overview. Jackie said the compilation introduction would continue next week before the course moves into scanning. He also announced that Assignment

1 would be released next week, with two weeks to complete it; it will involve implementing DFA/NFA-related work in Java, with relevant slides to be identified for students.

Key Terms, Notation, and Structures

Cohesion For an individual class, its attributes and methods should serve the same purpose or fit under a unifying theme.

Single Choice Principle (SCP) A change should affect a minimum number of places in the software. "Minimum" must be justified and is not necessarily one.

Type code In the first design, `Student.kind` uses an integer value to distinguish resident and non-resident students. Repeated branching on such a code is the duplication highlighted in the SCP discussion.

Inheritance / subclass The second design places common behavior in `Student` and specializes it in `ResidentStudent` and `NonResidentStudent`.

Parent-call reuse In the notes, a subclass calls `super.getTuition()` to reuse the parent's base tuition computation before applying its subclass-specific rate.

Regression testing Re-running all existing tests whenever a change is introduced to the model or implementation.

Concrete syntax The externally visible textual form of a source or target program, such as a plain-text program file.

Abstract syntax The internal structural form used during compilation; in this lecture it was associated with a runtime tree-like representation.

Intermediate Representation (IR) The internal representation between front end and back end. Multiple compiler stages can operate on IR rather than directly on source text.

Front end The compiler component that transforms the source program into IR.

Back end The compiler component that transforms IR into the target program.

Multiplicative / additive architecture Jackie's contrast between building separate compiler paths for every source-target pair and using shared IRs so source and target support can be added around a common hub.

Scanner / lexical analysis
: The front-end stage responsible for lexical structure and token formation.

Parser / syntactic analysis
: The front-end stage that analyzes syntax with respect to a grammar.

Key Examples and Demonstrations

Diagnosing the first `Student` design

Setup. Pages 4–6 show one `Student` class that stores both rate fields and uses `kind` branches in multiple methods.

Main point. The same code exhibits two different design problems. The irrelevant rate field in each kind of object is a cohesion problem; the repeated `kind` branching is an SCP problem because future changes propagate to multiple methods.

What to practise. Look at the code without the annotations first. Mark which fields apply to each object kind and circle every repeated type check. Then state separately why cohesion and SCP are violated.

Refactoring with inheritance

Setup. Page 7 moves common state/behavior into `Student` and puts resident/non-resident differences in subclasses.

Main point. The parent provides reusable common behavior; each subclass owns only its relevant variation. Calling `super.getTuition()` avoids copying the shared tuition computation.

What to practise. Trace what changes when a new student kind is added. Explain why the new design localizes that change more effectively than editing several `if/else if` chains.

Regression-testing loop

Setup. Page 8 shows Java model classes, derived JUnit tests, pass/fail results, debugging, and re-running tests after a change.

Main point. Test success is relative to the chosen tests. A test suite should include normal and abnormal scenarios, and regression testing protects existing behavior as the implementation evolves.

What to practise. Be able to redraw the loop and explain what action follows a failing test versus an all-passing run, and why an all-passing run is not a proof that the program is bug-free.

Front end–IR–back end

Setup. Page 9 places IR between a source program and a target program.

Main point. The compiler translates from concrete source text into an internal abstract representation, then from that representation to target text. This separation provides a common structure on which additional compiler stages can operate.

What to practise. Redraw the architecture and label which forms are concrete syntax and which form is abstract syntax. Explain the direction of the front-end and back-end transformations.

Five languages: 20 direct compilers versus a shared-IR hub

Setup. Page 10 uses Java, C, C#, C++, and Python.

Main point. Five directed source languages each targeting the other four produce 20 pairwise compiler paths. A hub architecture instead creates source-to- IR and IR-to-target components, giving an additive schematic of five plus five.

What to practise. Reproduce the count and then explain why the IR design changes the architecture rather than merely changing the arithmetic.

Scanner versus parser

Setup. Page 11 places scanner and parser in the compiler front end.

Main point. Lexical analysis handles token formation; syntactic analysis checks larger language structure against a grammar. Jackie used the misspelling of `class` as a simple lexical example.

What to practise. Point to the scanner and parser in the pipeline and state which kind of language rule each one is responsible for at the level introduced in this lecture.

Common Pitfalls and Misconceptions

- **Treating cohesion and SCP as the same principle.** Cohesion asks whether one class contains members serving a unified purpose; SCP asks how broadly a change propagates through the software.
- **Assuming "minimum" in SCP always means exactly one place.** Jackie explicitly said that the minimum may be greater than one. The design claim has to be justified.
- **Looking only for literal duplicate text.** In the `Student` example, repeated decisions about `kind` are the important duplicated choice: adding or removing a kind forces coordinated edits in multiple methods.
- **Keeping variation flags and all variation-specific data in one class.** The first design stores fields that are irrelevant to some objects and spreads type checks through behavior. The inheritance refactoring separates those concerns.
- **Believing that all tests passing proves absence of bugs.** Passing establishes correctness only with respect to the tests that were run. Missing cases can still hide defects.

- **Using print statements as the main debugging technique.** Jackie expected students at this level to use a debugger and breakpoints for serious debugging, with office-hour help available if needed.
- **Confusing concrete syntax with IR.** Source and target text are concrete syntax visible outside the compiler; the IR is an internal abstract runtime structure.
- **Building every source-target pair independently when shared structure is available.** The 20-path example was meant to motivate an additive IR-centered architecture rather than a multiplicative collection of pairwise compilers.
- **Conflating scanning and parsing.** The scanner handles lexical analysis; the parser handles syntactic analysis with respect to a grammar.

What You Should Be Able to Do

After this session, you should be able to:

- state Jackie's definitions of cohesion and the Single Choice Principle and distinguish the design question each principle asks;
- inspect a class and justify a cohesion or SCP violation using specific fields, methods, or duplicated decisions;
- explain why the first `Student` design spreads responsibility and change, and how the inheritance design localizes resident/non-resident variation;
- explain the role of `super.getTuition()` in reusing common behavior without duplicating the base tuition calculation;
- describe the regression-testing cycle and explain why test coverage matters even when every current test passes;
- distinguish concrete syntax, abstract syntax, and Intermediate Representation;
- draw and explain the source-program → front-end → IR → back-end → target-program architecture;
- reproduce the 20-versus-10 multi-language architecture calculation and explain the conceptual difference between multiplicative pairwise compilers and an additive shared-IR design;
- distinguish scanner/lexical analysis from parser/syntactic analysis at the level introduced in the session; and
- state the announced scope of Assignment 1: Java work involving DFA and NFA, with a two-week completion window after release.

Suggested Practice

1. Revisit pages 4–6 of Jackie's annotated iPad notes. Without reading the handwritten conclusions first, identify the cohesion problem and the SCP problem in the original `Student` design and justify each separately.
2. Reproduce the inheritance structure on page 7 from memory. Explain what remains in `Student`, what moves into each subclass, and why `super.getTuition()` is useful.
3. Redraw the regression-testing flow on page 8. Include normal/abnormal tests, the green/red outcomes, debugging after failures, and the requirement to re-run existing tests after a change.
4. Draw the page-9 compiler infrastructure from a blank page and label source concrete syntax, front end, IR/abstract syntax, back end, and target concrete syntax.
5. Rework the five-language architecture on page 10. First derive the 20 direct source-target paths, then reconstruct the additive shared-IR picture and explain why it needs five front-end and five back-end components in the schematic discussion.
6. Review the scanner/parser portion on page 11 and practise stating, in your own words, the boundary between lexical and syntactic analysis. This will prepare you for the upcoming scanning material.
7. Begin refreshing Java and the DFA/NFA material relevant to Assignment 1. Jackie said the assignment will be released next week, will run for two weeks, and will involve implementing DFA/NFA-related work in Java.

Where to Review

- **Jackie's annotated iPad notes, page 3:** definitions of cohesion and Single Choice Principle, including the Unix "do one thing and do it well" analogy and the minimum-change formulation.
- **Jackie's annotated iPad notes, pages 4–6:** the first `Student` design, object/state reasoning, cohesion diagnosis, and repeated `kind` conditions for the SCP diagnosis.
- **Jackie's annotated iPad notes, page 7:** inheritance-based refactoring with `Student`, `ResidentStudent`, `NonResidentStudent`, and reuse through `super.getTuition()`.
- **Jackie's annotated iPad notes, page 8:** regression-testing flowchart, JUnit pass/fail cycle, test coverage reminder, and Dijkstra's testing principle.
- **Jackie's annotated iPad notes, page 9:** front end, IR, back end, and concrete-versus-abstract syntax.
- **Jackie's annotated iPad notes, page 10:** five-language example, semantic-domain/IR grouping, and multiplicative-versus-additive architecture.

- **Jackie's annotated iPad notes, page 11:** conventional compiler infrastructure, scanner/lexical analysis, parser/syntactic analysis, and the ANTLR automation overview.
- **Lecture recording:** revisit the detailed Student refactoring, regression-testing reasoning, and multi-language IR architecture when the diagrams alone are not enough to reconstruct Jackie's explanation.

Closing Reminder

Use this digest as a roadmap for active review, not as a substitute for the learning process itself. Attend Jackie's in-person sessions and pay attention to how he develops ideas and works through the important reasoning step by step. If a transition, explanation, example, or reasoning step remains unclear afterwards, return to the recording to fill that gap.

For structured review, start with Jackie's annotated iPad notes because they most closely reflect the actual flow and development of the class. Work through them in manageable logical units. After each logical unit, study the corresponding part of the presentation slides alongside it. Jackie does not normally teach by reading the slides line by line, so the slides may contain relevant supporting material that was not explicitly read aloud.

Finally, choose a suitable selection of the important reasoning from this session and work through it again on your own. The goal is not to memorize Jackie's recorded steps, but to determine whether you can reproduce the reasoning from your own understanding and explain why the important steps work.