

# EECS4302 Compilers and Interpreters

## Fall 2026

### Digest of Lecture 9, October 8

#### Composite Design Refinement and Visitor Motivation

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                             | 3  |
| Key Terms, Notation, and Structures | 6  |
| Key Examples and Demonstrations     | 7  |
| Common Pitfalls and Misconceptions  | 8  |
| What You Should Be Able to Do       | 9  |
| Suggested Practice                  | 10 |
| Where to Review                     | 10 |
| Closing Reminder                    | 11 |

## Main Ideas

- **Design diagrams encode both inheritance and object relationships.** Jackie distinguished inheritance arrows from client–supplier arrows and emphasized that a project design diagram should be selective rather than an exhaustive picture of every class and relationship.
- **The Composite pattern emerged through a sequence of design attempts.** Putting children and add at the root Equipment type makes leaf objects expose meaningless operations; moving them to CompositeEquipment fixes that type-safety problem but duplicates collection logic across recursive domains; extracting a generic Composite<E> removes the duplication but initially creates an illegal multiple-inheritance requirement in Java.
- **The final architecture separates the reusable implementation from the domain contract.** In Jackie’s final diagram, Equipment is an interface, Composite<E> is an abstract generic class carrying the reusable collection behaviour, and CompositeEquipment both extends Composite<Equipment> and implements Equipment.
- **Uniform access plus dynamic binding drives recursive aggregation.** Both base and composite equipment respond to price(), but the base case returns a stored price while the composite case iterates through its children and calls child.price(). The child’s dynamic type determines whether that call terminates at a base case or recurses into another composite.
- **Tracing the runtime tree matters as much as reading the code.** Jackie built a cabinet–chassis–bus tree with a video card priced at 200 and a disk drive priced at 500, then traced cabinet.price() through the nested calls to obtain 700. He emphasized being able to move both from code to tree and from tree to code.
- **Visitor was motivated by a different kind of change.** A naive expression hierarchy that embeds evaluation, printing, type checking, code generation, and other operations directly into every expression class works functionally, but it spreads changes across the hierarchy and creates poorly cohesive “Superman” classes. This set up the later Visitor design.
- **Three design principles frame the Visitor discussion.** Cohesion asks whether a class contains closely related responsibilities; Single Choice asks that a change be localized to as few places as possible; and the Open–Closed discussion distinguishes a system with open structure and closed operations from one with closed structure and open operations. Jackie’s final annotated page highlights the latter direction for the Visitor discussion, while double dispatch was previewed as the key mechanism still to be developed.

# Roadmap

## 1. Reading design diagrams and deciding what to show

Jackie began by standardizing the visual notation used in the design diagrams. In the diagrams for this lecture, inheritance is shown with an empty arrowhead, usually arranged vertically. A **client–supplier** relation uses a solid arrowhead from the client class to the supplier class and is labelled with the attribute name that creates the relationship. Page 2 of Jackie’s annotated iPad notes illustrates this with a Student object using a Card as its yuCard.

The practical point is not merely notation. Jackie said that a project design model does not need to display every class and every relationship. The final report should select the important parts needed to communicate the design. He also warned that assessment diagrams may be printed without color, so arrowhead shape and direction matter more than color. He suggested reviewing the posted Composite and Visitor design diagrams as preparation for questions that ask why an arbitrary design is good or bad.

## 2. Composite Attempt 1: recursion at the root creates invalid leaf operations

Page 3 places the recursive collection directly in the abstract Equipment type: it has a List<Equipment> of children and an add(Equipment) operation. The self-loop client–supplier relation is a visual cue that Equipment is recursively defined in terms of itself. Cabinet, chassis, and bus are intended to be composite equipment, while VideoCard and DiskDrive are leaves.

The flaw is static, not dynamic. Because add lives in the root type, every subclass inherits it. Calls such as a video card adding a disk drive would therefore compile even though they make no sense in the modeled domain. Jackie’s criterion was that the design itself should prevent these meaningless operations, not merely rely on programmers to avoid writing them.

## 3. Composite Attempt 2: type safety improves, but duplication appears

On page 4, children and add move down into CompositeEquipment. Leaf types no longer expose add, so the first design’s type-safety problem is fixed.

Jackie then introduced a second recursive domain, furniture, on page 5. A second composite hierarchy needs the same kind of collection management as the equipment hierarchy: adding, removing, or inserting child objects. Repeating that logic independently in each recursive domain is a design smell. The next step is therefore to factor the common collection behaviour into one reusable abstraction.

## 4. Generic Composite and the multiple-inheritance obstacle

Page 6 extracts the shared collection logic into an abstract generic class Composite<E>. The generic parameter can later be instantiated with a domain type such as equipment or

furniture, allowing the same collection code to be reused.

That first refactoring is still illegal in Java. The pictured design would need the class for composite equipment to extend both the existing equipment class and the new `Composite` class. Jackie used the page-7 diamond example to explain the general difficulty. If a subclass inherited two distinct implementations of the same method, a call could be ambiguous. Java avoids this class-level multiple-inheritance problem rather than providing a renaming mechanism. Jackie mentioned Eiffel's renaming idea only as background and explicitly said that aside was not exam-covered.

## 5. Final Composite architecture: one abstract superclass plus one interface

The final design on pages 8–10 removes the illegal double extension. The domain contract becomes an `Equipment` interface containing the common `price()` operation. Reusable collection behaviour remains in the abstract generic class `Composite<E>`. Then `Composite Equipment` can legally

```
extend Composite<Equipment>
implement Equipment
```

while `BaseEquipment` implements `Equipment` without inheriting the composite collection machinery.

This also preserves the cross-domain reuse that motivated the refactoring: `Composite<Equipment>` and `Composite<Furniture>` are different instantiations of the same generic abstraction. The final architecture therefore addresses all three earlier concerns: leaves do not expose composite-only operations, collection behaviour is not duplicated across domains, and the Java inheritance structure is legal.

## 6. Uniform access and dynamic binding in the implementation

Page 11 shows how the design works in code. The `Equipment` interface presents one uniform operation, `price()`. A `BaseEquipment` implementation stores a numeric price and returns that value directly. A `CompositeEquipment` instead iterates through its children and accumulates

```
result = result + child.price();
```

for each child.

The expression `child.price()` is where dynamic binding matters. A child whose runtime type is a leaf executes the base implementation and terminates that branch. A child whose runtime type is another composite executes the recursive implementation and descends further into the tree. Jackie described this as uniform access: the caller uses the same operation even though the underlying behaviour differs by dynamic type.

## 7. Building and tracing the concrete tree

Page 12 turns the architecture into a test. The code creates a video card with price 200, a disk drive with price 500, a bus, a chassis, and a cabinet. The calls build the tree bottom-up:

```

bus.add(card);
chassis.add(bus);
chassis.add(drive);
cabinet.add(chassis);

```

Thus the cabinet has one chassis child; the chassis has a bus and a drive; and the bus has the video card.

Tracing the `price()` call on `cabinet` follows that same structure recursively. The bus branch reaches the card and returns 200; the drive branch returns 500; those results combine at the chassis and then propagate to the cabinet, producing 700. Jackie stressed that simply spotting the two numbers is not enough for an assessment: students should be able to reconstruct the tree and explain the nested calls and loop iterations. For debugging recursive project code, he recommended starting with small trees and reproducing this kind of trace.

## 8. From Composite to Visitor: why operations become the next design problem

Pages 13–14 switch from a hardware tree to an expression AST. The structure contains an `Expression` interface, a composite expression abstraction, `Constant`, and `Addition`. Jackie used a small expression such as “341 + 2” to keep the focus on the recursive shape rather than language complexity.

The next question is how to support operations over that tree: evaluation, prefix/postfix printing, type checking, and potentially code generation. The naive design places every operation in the expression hierarchy and requires every concrete expression class to implement them. It works, but adding one new operation means editing many classes, and each expression class accumulates responsibilities of very different kinds.

## 9. Cohesion, Single Choice, and Open–Closed alternatives

Jackie used the naive design to motivate three principles. **Cohesion** asks that a class contain features with closely related responsibilities rather than becoming a “Superman” class. **Single Choice** asks that a change be localized to one place when possible, or at least to the minimum justified number of places. The repeated operation methods in every expression class make both principles relevant.

Pages 15–16 then separate the system into two dimensions: the **structure** (the syntax classes) and the **operations** applied to that structure. The Open–Closed discussion presents two choices:

|               | Structure | Operations |
|---------------|-----------|------------|
| Alternative 1 | Open      | Closed     |
| Alternative 2 | Closed    | Open       |

“Open” means expected to change or be extended; “closed” means intended to remain comparatively stable. Jackie’s final annotation circles the second alternative—no new syntax is expected, but new operations may continue to be added—as the direction to carry forward.

He previewed **Visitor** for this recursive setting and said that **double dispatch** is the central mechanism still to be learned in the continuation.

## Key Terms, Notation, and Structures

|                                  |                                                                                                                                                                                                       |
|----------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Inheritance arrow</b>         | In the lecture diagrams, an empty arrowhead denotes inheritance. Jackie warned students not to rely on color when reading an assessment diagram.                                                      |
| <b>Client–supplier relation</b>  | A solid-arrow relation from the client to a supplier class, typically labelled by the attribute name that creates the relationship. Page 2 uses Student → Card labelled by <code>yuCard</code> .      |
| <b>Recursive type</b>            | A type that refers to objects of the same general family through child references. The self-loop in the first Composite design visualizes this idea.                                                  |
| <b>Base component</b>            | A leaf object that cannot contain further components. <code>VideoCard</code> and <code>DiskDrive</code> are the equipment examples.                                                                   |
| <b>Composite component</b>       | An internal node that owns child components and can therefore contain leaves or other composites.                                                                                                     |
| <b>Design smell: duplication</b> | Repeated collection-management logic across independent recursive domains. Jackie used equipment and furniture to show why the common code should be factored into one abstraction.                   |
| <code>Composite&lt;E&gt;</code>  | The abstract generic class in the final pattern that stores reusable child-collection behaviour such as <code>children</code> and <code>add</code> .                                                  |
| <b>Diamond problem</b>           | The ambiguity that can arise when a class inherits multiple implementations of the same operation from two parent classes. Jackie used it to motivate Java's rejection of multiple class inheritance. |
| <b>Uniform access</b>            | The same operation, such as <code>price()</code> , is used for both base and composite equipment even though one returns a stored value and the other computes recursively.                           |
| <b>Dynamic binding</b>           | Runtime selection of the <code>price()</code> implementation according to the child's dynamic type during the recursive traversal.                                                                    |
| <b>Cohesion</b>                  | Keep a class focused on related responsibilities rather than accumulating unrelated operations.                                                                                                       |
| <b>Single Choice Principle</b>   | When a change is required, localize it to one place when possible, or to the minimum justified number of places.                                                                                      |

- Open–Closed alternatives** A design choice about which dimension is expected to vary: either the structure is open and operations are closed, or the structure is closed and operations are open.
- Visitor / double dispatch** Visitor was introduced as the next pattern for operations on a recursive structure. Double dispatch was identified as the important mechanism to learn next; its full architecture was not yet developed in this session.

## Key Examples and Demonstrations

### 1. Reading a client–supplier relation

**Setup:** Page 2 starts from an attribute of type Card inside Student.

**Main point:** The class declaring the attribute is the client; the attribute's type is the supplier. The design diagram therefore draws a solid arrow from Student to Card and labels it yuCard.

**What to practise:** Given a small class fragment with one or two object attributes, draw only the corresponding relations that are important to the design you want to communicate.

### 2. Why Composite Attempt 1 is invalid as a design

**Setup:** Page 3 puts children and add in the root Equipment class.

**Main point:** Leaves inherit add. A call such as a video card adding a disk drive therefore passes the static interface even though it is meaningless in the modeled system.

**What to practise:** Explain the flaw from the class hierarchy alone, then show exactly how Attempt 2 prevents the same call from compiling.

### 3. The refactoring sequence across pages 4–10

**Setup:** Attempt 2 fixes leaf operations but duplicates composite logic. Attempt 3 extracts a generic Composite class but creates a multiple-inheritance problem. The final architecture turns Equipment into an interface and uses one abstract generic superclass.

**Main point:** Each design is best understood by the specific problem it solves and the new problem it reveals. The final design is not an isolated pattern to memorize; it is the result of this sequence of constraints.

**What to practise:** For each attempt, write one sentence for the design improvement and one sentence for the remaining flaw.

### 4. Diamond-problem ambiguity

**Setup:** Page 7 has superclass S, subclasses A and B, and a hypothetical class C that inherits both. Both A and B provide doSomething().

**Main point:** A call through a C object has two inherited method candidates. Without an additional conflict-resolution mechanism, the dispatch is ambiguous.

**What to practise:** Use the diagram to explain why the intermediate Composite refactoring cannot simply extend two Java classes. Treat the Eiffel renaming aside as background only; Jackie explicitly excluded it from exam coverage.

## 5. Composite price trace

**Setup:** Page 12 builds cabinet → chassis → {bus, drive} with bus → card. The card has price 200 and the drive has price 500.

**Main point:** cabinet.price() repeatedly invokes the composite loop until dynamic binding reaches base objects, where price() returns the stored values. The recursive sum is 700.

**What to practise:** Reconstruct the tree from the four add calls, then trace the calls in the order produced by the child lists. Also try the reverse direction: given the tree, write the corresponding construction calls.

## 6. Expression operations and the Visitor motivation

**Setup:** Pages 13–14 use a small AST with Constant and Addition and consider operations such as evaluate, print\_prefix, print\_postfix, and type\_check.

**Main point:** Adding all of these operations directly to every expression class makes the structure responsible for unrelated behaviours and forces many classes to change when one new operation is added.

**What to practise:** Use cohesion and Single Choice to explain why the naive design is undesirable, then use pages 15–16 to state the two possible Open–Closed choices without yet inventing the final Visitor implementation.

## Common Pitfalls and Misconceptions

- **Relying on diagram color instead of notation.** Assessment copies may be grayscale. Use arrowhead shape, direction, labels, and class stereotypes to distinguish inheritance from client–supplier relations.
- **Trying to make a design diagram exhaustive.** Jackie explicitly encouraged selective models. Include the classes and relationships that are important to explaining the design rather than every implementation detail.
- **Putting composite-only operations in the root type.** If add is declared on Equipment, leaf types inherit an operation that should not exist for them.
- **Stopping after Attempt 2 because it is type-safe.** Attempt 2 prevents invalid leaf additions, but the equipment and furniture examples reveal duplicated collection logic that should be factored out.
- **Fixing duplication by extending two Java classes.** The page-6 intermediate design would require multiple class inheritance and therefore would not compile in Java. The final design uses one abstract superclass plus an interface instead.

- **Treating `price()` as a flat arithmetic expression.** In the Composite implementation, the result follows the recursive tree and repeated dynamic dispatch. Build the object structure before tracing the numeric aggregation.
- **Choosing the `price()` implementation from the variable's static type.** The common interface makes the call legal, but the implementation executed at runtime depends on whether the actual child is a base or a composite object.
- **Turning the expression hierarchy into a “Superman” class family.** Evaluation, printing, type checking, and code generation are all operations on the same structure, but they are not the same responsibility. Jackie used this to motivate cohesion and the later Visitor refactoring.
- **Assuming Open–Closed means everything should stay open or everything should stay closed.** Jackie separated the design into structure and operations. The question is which dimension should be stable and which should remain extensible.

## What You Should Be Able to Do

After this session, you should be able to:

- distinguish inheritance and client–supplier relations in Jackie’s design-diagram notation and explain the direction of a client–supplier arrow;
- decide which classes and relationships are important enough to include in a selective project design diagram;
- explain the flaw in Composite Attempt 1 using a concrete leaf add call;
- explain how Attempt 2 fixes that flaw and why cross-domain duplication still motivates another refactoring;
- explain why the first generic-Composite refactoring would require illegal multiple class inheritance in Java and relate that problem to the diamond example;
- reconstruct the final architecture involving `Equipment`, `BaseEquipment`, `Composite<E>`, and `CompositeEquipment`;
- explain uniform access for `price()` and identify where dynamic binding occurs in the recursive composite implementation;
- build the page-12 equipment tree from the test code and trace `cabinet.price()` to its base cases and final result;
- evaluate a simple design with respect to cohesion and the Single Choice Principle;
- distinguish the two Open–Closed alternatives for structure versus operations; and
- explain why a recursive AST with a stable structure but an expanding set of operations motivates the forthcoming Visitor discussion, while recognizing that the full double-dispatch mechanism was only previewed here.

## Suggested Practice

1. **Review the posted design diagrams as designs, not pictures.** Jackie explicitly suggested using the Composite and Visitor attempts on the lecture site to practise explaining why a design is good or bad. For each Composite attempt, identify the intended improvement and the remaining limitation.
2. **Reconstruct the final Composite architecture from memory.** Start with the reusable generic collection abstraction, then add the Equipment interface, BaseEquipment, and CompositeEquipment. Check that your arrows correspond to exactly one class extension plus the required interface implementation.
3. **Redo the page-12 tree trace in both directions.** First use the four add calls to draw the runtime tree and trace `cabinet.price()`. Then hide the code, use the tree alone, and write the construction calls that rebuild it. Jackie specifically said this kind of tree-building knowledge is relevant to quizzes/tests.
4. **Use small recursive trees when debugging project code.** Jackie's practical advice was to reduce a suspected recursion problem to a small structure and manually reproduce the method trace before returning to a large tree.
5. **Apply the design principles to the naive expression hierarchy.** Explain separately how it challenges cohesion and Single Choice. Then write the two Open–Closed alternatives from pages 15–16 and state what “open” and “closed” mean for the structure and operation dimensions.
6. **Connect this to Reading Week.** Revisit the expression/composite material used by the practice test and tutorial, and keep notes on places where the ANTLR4 material introduces Visitor. The detailed Visitor/double-dispatch architecture is the next step rather than something to invent from this digest.

## Where to Review

- **Jackie's annotated iPad notes, page 2:** client–supplier notation, the Student–Card example, and the attribute label on the relation.
- **Jackie's annotated iPad notes, page 3:** Attempt 1, the recursive self-reference at Equipment, and the invalid leaf add consequence.
- **Jackie's annotated iPad notes, pages 4–6:** Attempt 2, the equipment/furniture duplication, extraction of generic `Composite<E>`, and the multiple-inheritance problem in Attempt 3.
- **Jackie's annotated iPad notes, page 7:** the diamond problem and the ambiguity between two inherited implementations.

- **Jackie's annotated iPad notes, pages 8–10:** the final Composite architecture, the distinction between implements and extends, and reuse of the generic Composite abstraction across equipment and furniture.
- **Jackie's annotated iPad notes, page 11:** the Java implementation, uniform `price()` access, the base/composite split, and the `child.price()` dynamic-binding point.
- **Jackie's annotated iPad notes, page 12:** the test code, the cabinet/chassis/bus tree, the 200 and 500 leaf prices, and the recursive `cabinet.price()` trace.
- **Jackie's annotated iPad notes, pages 13–14:** the expression AST and the naive design that distributes multiple operations across the expression hierarchy.
- **Jackie's annotated iPad notes, pages 15–16:** the Open–Closed alternatives for structure versus operations and the highlighted structure-closed/operations-open direction.
- **Session recording:** return to the spoken discussion for the assessment guidance on reading/evaluating design diagrams, the rationale behind each Composite refactoring, the detailed runtime trace, and Jackie's explanations of cohesion, Single Choice, and the Visitor motivation.

## 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.