# EECS4302 Compilers and Interpreters
# Fall 2026

## Digest of Lecture 10, October 20

Visitor Architecture, Double Dispatch, and Design Trade-offs

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 4

Key Examples and Demonstrations 5

Common Pitfalls and Misconceptions 6

What You Should Be Able to Do 7

Suggested Practice 7

Where to Review 8

Closing Reminder 8

# Main Ideas

- The Visitor pattern separates a recursive language structure from the operations applied to it. In Jackie's architecture, the `structures` package contains the Composite-style expression hierarchy, while the `operations` package contains the `Visitor` hierarchy.
- The architecture depends on two correspondences: every concrete structure class implements `accept(Visitor v)`, and every structure class has a corresponding `visit*` method in every concrete visitor.
- Runtime execution uses **double dispatch**. First, the dynamic type of the structure object selects the concrete `accept` method. Second, the dynamic type of the visitor selects the concrete `visit*` method.
- Recursive operations such as evaluation visit subtrees independently and combine their results. The session traced this with an `Addition` whose two children are constants with values 1 and 2.
- Visitor is a good fit when the language structure is relatively stable but new operations are expected. Adding a new language construct forces coordinated changes across the visitor hierarchy, while adding a new operation can be localized in one new visitor class.
- Assessment planning matters now: Assignment 2 is direct preparation for the November 5 Programming Test, and the self-guided regular-language material is part of preparation for the November 19 Written Test.

# Roadmap

## 1. Post-reading-week assessment roadmap

Jackie began by reconnecting the class to the semester calendar. Assignment 2 is the ANTLR4 tutorial, and Jackie recommended completing it alone so that it genuinely prepares you for the **Programming Test on November 5**. He also described an optional mock lab-test session that may be scheduled separately. The **Written Test is on November 19**, in class, on paper, for **80 minutes**. The self-guided lecture on formulating regular-language tools should be completed before that written test. Jackie also reminded project teams to work continuously rather than waiting until a milestone; Milestone 1 is a low-stakes checkpoint in Week 9, around November 9, intended to expose design problems early.

## 2. Visitor architecture: separating structures from operations

The technical development resumes the Composite and Open-Closed discussions from before Reading Week. Pages 3–4 of Jackie's annotated iPad notes show two packages. The `structures` side contains `Expression`, `CompositeExpression`, `Constant`, `Addition`, and `Subtraction`; the `operations` side contains the `Visitor` interface and concrete operations such as `Evaluator`, `PrettyPrinter`, and `TypeChecker`.

Jackie emphasized four architectural observations:

1. each structure class can accept a visitor;
2. each concrete structure class has a corresponding `visit*` method in every operation class;
3. each operation class represents one recursive operation over the Composite structure; and
4. an operation class may introduce operation-specific query methods, such as an integer evaluation result or string pretty-printing result.

The parameter of `accept` is deliberately typed as the general `Visitor` interface. That static type allows different concrete visitors to be supplied polymorphically at runtime.

## 3. Implementing `accept` and using a visitor

The standard implementation of `accept` is a one-line delegation that depends on the enclosing structure class. For example, inside `Constant`, the method delegates to the corresponding constant visit method. Inside `Addition`, it delegates to the corresponding addition visit method. Jackie explicitly warned against memorizing these lines without understanding how to derive them from the current class and the visitor parameter.

Page 4 then shows the client-side setup. A variable may have static type `Visitor` while the created object has dynamic type `Evaluator`. The Composite tree is built first, the desired operation object is created, and the root is asked to `accept` that visitor. Because a variable with static type `Visitor` does not expose evaluator-specific methods such as `result()`, directly calling such a method through that reference does not compile; the example uses an explicit cast to the concrete visitor type.

## 4. Building the concrete expression tree

Page 5 visualizes the object structure created by the test code. `c1` refers to `Constant(1)`, `c2` refers to `Constant(2)`, and `add` refers to an `Addition` whose left and right references lead to those two constants. Jackie used this simple tree because the dynamic type of each referenced object becomes crucial during the subsequent dispatch trace.

## 5. Double dispatch: first the structure, then the visitor

Page 6 contains the central runtime trace for `add.accept(v)`. The order is fixed.

1. **First dispatch:** identify the context object to the left of the dot. Its dynamic type is `Addition`, so dynamic binding enters `Addition.accept(Visitor v)`.
2. Inside that method, execution reaches `v.visitAddition(this)`.
3. **Second dispatch:** now the context object is `v`. Its dynamic type is `Evaluator`, so dynamic binding enters `Evaluator.visitAddition(Addition e)`.

Jackie stressed that a written-test explanation should name the context object, state its dynamic type, and identify the method/class selected by dynamic binding. Saying only that one object "points to" another is too informal for the intended explanation.

### 6. Recursive subtree visits

Inside `Evaluator.visitAddition`, separate evaluator objects are created for the left and right subtrees. Each subtree root calls `accept` again, producing another complete instance of double dispatch. In the worked tree, `e.getLeft()` has dynamic type `Constant`, so its first dispatch enters `Constant.accept`; the evaluator supplied to that call then produces the second dispatch into `Evaluator.visitConstant`. The right subtree follows the same pattern. The two results are then combined to compute the value of the addition.

Jackie also generalized the idea beyond binary trees: if a structure contains a list or array of subexpressions, iterate through the collection, apply a visitor to each element, and aggregate the results.

### 7. Visitor under Open-Closed and Single-Choice

Pages 7–8 use two extension scenarios to explain when Visitor is appropriate. If a new `Multiplication` construct is added, the structure hierarchy changes, the `Visitor` interface needs a corresponding visit method, and every concrete visitor must implement it. Jackie identified this ripple across multiple places as a **Single-Choice Principle violation**. Therefore, if Visitor is adopted, the language structure should be relatively **closed and stable**.

By contrast, adding a new operation such as assembly-code generation requires one new visitor class that implements the existing set of `visit*` methods. The change is localized to one new unit, which fits the Single-Choice Principle. Therefore, under Visitor, the **operations side should remain open** for extension.

## Key Terms, Notation, and Structures

`structures` **package**
: The recursive language-structure side of the design, built from the Composite pattern.

`operations` **package**
: The Visitor hierarchy containing the recursive operations to apply to the structure.

`accept(Visitor v)`
: The polymorphic hook implemented by each concrete structure class. Its parameter uses the general `Visitor` static type.

**Visit methods**
: Jackie's architecture uses separate visit methods for constants, additions, and subtractions. Jackie preferred explicit structure-specific names over relying only on overloaded `visit(...)` methods.

**First dispatch**
: Dynamic binding on the structure/context object selects the concrete `accept` implementation.

**Second dispatch**
: Dynamic binding on the visitor object selects the concrete `visit*` implementation.

**Double dispatch** The two dispatches occur in that order during a Visitor call.

**Static type vs. dynamic type**
The declared type determines what operations compile; the runtime type determines which overridden method is selected by dynamic binding.

**Single-Choice Principle**
A change should be concentrated in one place, or as few places as genuinely necessary. Jackie used the Visitor extension cases to judge whether this principle is satisfied.

**Open-Closed decision**
In the Visitor architecture developed here, language structures should be stable and closed to frequent change, while operations remain open to extension.

# Key Examples and Demonstrations

## The architecture diagram

**Setup:** Pages 3–4 of Jackie's annotated iPad notes place the Composite structure hierarchy and Visitor operation hierarchy side by side.
**Main point:** Trace the dependency through `accept(Visitor v)` and match every concrete structure class with a corresponding `visit*` method in each visitor.
**What to practise:** On a blank page, reproduce a small structure hierarchy and the matching visitor hierarchy, then check Jackie's four architecture observations against your design.

## The `Addition(Constant(1), Constant(2))` object graph

**Setup:** Page 5 builds two constant objects and one addition object whose left/right references point to them.
**Main point:** The dynamic type of each actual object, not just the declared variable type, determines the runtime dispatch path.
**What to practise:** Reconstruct the object graph from the Java setup, and then do the reverse: given the object graph, write the Java statements that build it. Jackie also suggested extending the example to something such as $1 + (2 + 3)$.

## Tracing `add.accept(v)`

**Setup:** `add` refers dynamically to an `Addition`; `v` is declared as `Visitor` but refers dynamically to an `Evaluator`.
**Trace skeleton:**

1. first dispatch on `add`: dynamic type `Addition` ⇒ `Addition.accept`;
2. execute `v.visitAddition(this)`;

3. second dispatch on `v`: dynamic type `Evaluator` ⇒ `Evaluator.visitAddition`;
4. recursively visit the left and right subtrees, each of which triggers another double dispatch;
5. combine the two evaluated subtree results.

**What to practise:** Write the explanation in full sentences using Jackie's required vocabulary: context object, dynamic type, invoked method, and class containing that method.

### Adding syntax versus adding an operation

**New construct:** Adding a multiplication construct requires a new structure class. It also requires a corresponding visitor method and implementations across the concrete visitors.
**New operation:** Adding an operation requires one new visitor class that implements the existing visit methods.
**Main point:** These cases justify the design rule developed in class: keep the structure stable and closed while operations remain open.

## Common Pitfalls and Misconceptions

- **Using a concrete visitor type in `accept`.** Jackie deliberately used `Visitor` as the parameter's static type so that different concrete visitors can be supplied polymorphically.
- **Trying to call an evaluator-specific result method through a variable statically typed as `Visitor`.** The static interface does not declare that operation-specific method, so the call does not compile without an appropriate concrete reference or cast.
- **Saying the second dispatch "goes to the Visitor interface."** The interface contains method declarations, not the runtime implementation. Dynamic binding selects the matching method in the visitor object's concrete dynamic type.
- **Reversing the dispatch order.** The first dispatch selects the structure's `accept`; the second selects the visitor's concrete `visit*`. Jackie explicitly said the order matters.
- **Misidentifying a subtree's dynamic type.** In the worked example, `e.getLeft()` refers to a `Constant`, not another `Addition`. Reconstruct the actual object graph before tracing.
- **Describing dynamic binding too casually on a test.** Jackie wants the explanation to identify the context object, its dynamic type, and the exact method/class selected.
- **Assuming Visitor makes every kind of extension cheap.** New operations are localized; new language constructs ripple through the visitor hierarchy and can violate Single-Choice.

## What You Should Be Able to Do

After this session, you should be able to:

- draw the two-package Visitor architecture for a small recursive language and explain the role of each side;
- derive the correct one-line `accept(Visitor v)` implementation for a concrete structure class instead of memorizing it;
- distinguish static type from dynamic type when a general visitor reference denotes a concrete operation object;
- explain why an operation-specific result method is not directly available through a reference whose static type is only `Visitor`;
- reconstruct the runtime object graph for the `Addition(Constant(1), Constant(2))` example;
- trace `add.accept(v)` through both dispatches and identify the context object, its dynamic type, and the invoked method at each stage;
- continue the trace through recursive left/right subtree visits and explain why each subtree call creates another instance of double dispatch;
- generalize the visitor traversal from binary left/right children to a collection of child expressions;
- compare the impact of adding a new language construct with the impact of adding a new language operation; and
- justify, using Open-Closed and Single-Choice reasoning, why Visitor is most suitable when language structure is stable and operations are expected to grow.

## Suggested Practice

1. **Rebuild the architecture from a blank page.** Jackie explicitly recommended practising this before the exam: create a small recursive expression language, then design the matching Visitor hierarchy and verify the four architecture observations from class.
2. **Construct a deeper tree.** Write Java code for a structure such as $1 + (2 + 3)$, draw the corresponding object/reference structure, and identify the dynamic type of every node.
3. **Write a formal double-dispatch trace.** Start from the root's `accept` call and explain the first and second dispatch in Jackie's preferred form: context object, dynamic type, then invoked method/class. Repeat for at least one subtree call.

4. **Compare two extension decisions.** Starting from the architecture in Jackie's notes, work out what must change when adding `Multiplication`, then contrast that with adding one new visitor operation. Use the result to justify which side should be open and which should be closed.

5. **Assessment preparation.** Try to complete Assignment 2 alone, as Jackie recommended, and use it to prepare for the November 5 Programming Test. Plan to complete the self-guided regular-language material before the November 19 Written Test.

## Where to Review

- **Jackie's annotated iPad notes, page 2:** assessment/reminder items, including Assignment 2, the Programming Test, and the self-guided lecture.
- **Pages 3–4:** Visitor package architecture, the four architecture observations, static and dynamic type setup, and client-side use of a visitor.
- **Pages 5–6:** the concrete object graph and the runtime trace of `add.accept(v)`, including recursive subtree double dispatch.
- **Pages 7–8:** Open-Closed and Single-Choice reasoning for adding a language construct versus adding an operation.
- **Recording and self-guided material:** revisit Jackie's precise dispatch explanation in the recording, and complete *Formulating Regular Language Tools* before the November 19 Written Test.

## Closing Reminder

Use this digest as a roadmap for active review, not as a substitute for the learning process. Attend Jackie's in-person sessions and follow how he develops architecture decisions and runtime reasoning step by step. If a dispatch step, type distinction, or design argument remains unclear, return to the recording to repair that gap.

For structured review, begin with Jackie's annotated iPad notes because they most closely reflect the class flow. Work through manageable units: the Visitor architecture, the object graph and double-dispatch trace, and the Open-Closed/ Single-Choice extension cases. After each unit, study the corresponding presentation-slide material alongside it. Jackie does not normally teach by reading slides line by line, so the slides may contain supporting material not explicitly read aloud.

Finally, reproduce selected work on your own: draw the structure and visitor hierarchies, rebuild the object/reference diagram, trace both dispatches with explicit static/dynamic type reasoning, and compare adding syntax with adding an operation. The goal is to reconstruct and justify the reasoning from your own understanding rather than memorize Jackie's recorded steps.