CS 247

Software Engineering Principles

Chapter 17

The Model-View-Controller (MVC) Pattern

Victoria Sakhnini

Table of Contents

Introduction

Real software systems rarely use a single design pattern in isolation. As a system grows, multiple patterns begin to appear together, each solving a different coupling problem within the same design. When two or more patterns combine so consistently that the combination itself becomes a recognizable, reusable solution, it earns a name of its own.

This chapter introduces the most famous such combination: the Model-View-Controller pattern, universally abbreviated MVC. MVC has been reinvented dozens of times across decades of software history, in desktop GUI toolkits, web frameworks, mobile platforms, and game engines, because the problem it solves is universal: how do you cleanly separate an application's data from its presentation from its interaction logic, so that any of the three can change without breaking the others?

The answer is three patterns working together. Observer keeps the Model decoupled from the Views that watch it. Strategy lets the View delegate all behaviour to a swappable Controller. Composite lets the View be composed from a tree of nested display components, updated through a single root call.

Compound Patterns

A compound pattern combines two or more patterns into a solution that solves a recurring or general problem.

The keyword is recurring. A compound pattern is not just two patterns that happen to appear in the same codebase. It is a specific combination that recurs so consistently across problems, languages, and teams that naming it accelerates design conversations. When a team says "we're using MVC", they have communicated, in three letters, an entire architectural decision about how data, presentation, and logic will be separated.

> **Tip:**
>
> **Compound patterns vs. arbitrary pattern combinations.** Not every combination of patterns earns a name. MVC earns one because the same three-pattern combination (Observer + Strategy + Composite) appears independently in music players, web frameworks, document editors, and spreadsheets. The combination solves the same recurring problem everywhere: separate data, presentation, and interaction so each can vary independently.

MVC: The Big Picture

The overview diagram shows the three participants and their relationships at a high level. The user interacts with the View; the View communicates with the Controller; the Controller manipulates the Model; the Model notifies the View that something has changed; the View queries the Model for the new state.

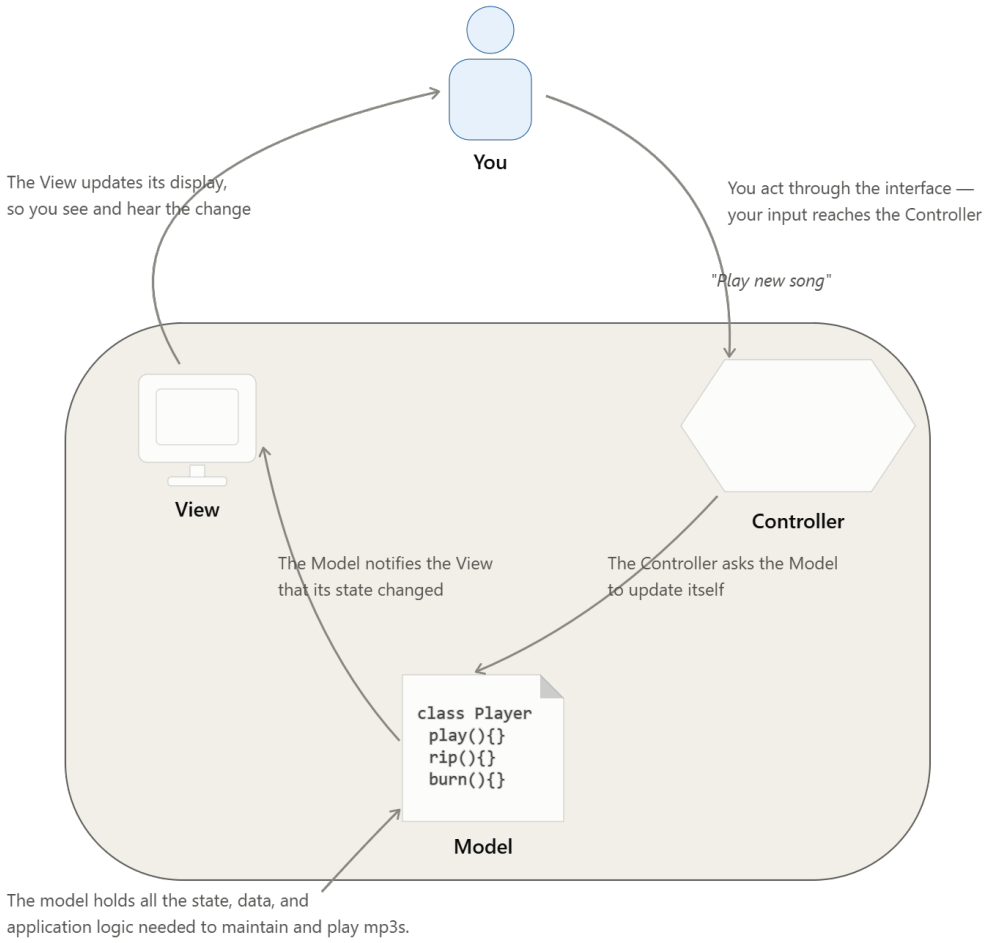

Figure 1. The MVC overview.

Notes:

1. The user does something, clicks a button, types in a field, presses a key. This event reaches the View.
2. The View passes the event to its Controller. The Controller interprets the action and asks the Model to change its state accordingly.

3. The Controller may optionally ask the View to update its display directly, for example, to show a loading spinner while the Model works.
4. The Model notifies all registered observers that its state has changed.
5. The View, responding to the notification, queries the Model for its new state and redraws itself.

Steps ④ and ⑤ are the Observer Pattern in action. The Model does not call the View directly; it fires a generic notification, and any observer that cares can respond. This keeps the Model completely ignorant of the View's existence.

The Three Roles

The Model

The Model is the core of the application. It holds all the data, all the state, and all the application logic. In a music player, the Model knows the current track, playlist, volume, and playback position. In a student information system, the Model holds student names, roll numbers, scores, and the grading algorithm.

The defining property of the Model is that it is completely oblivious to how it is displayed. It does not know whether there is a graphical interface, a web interface, a REST API, or a command-line interface. All it knows is that when its state changes, it should notify its observers, and it does so by calling notifyObservers() at the end of every mutating method.

This ignorance is the source of the Model's power. Because the Model knows nothing about the View, you can attach multiple different Views to the same Model, or swap one View for another, without touching a single line of Model code.

The View

The View presents the Model to the user. It is a window into the Model's state. The View gets its data directly from the Model; it asks the Model questions, receives answers, and renders them on screen. The View does not compute application logic, store application data, or decide what happens when the user clicks a button. All of that is delegated elsewhere.

In a real GUI application, the View is not a single flat object; it is a tree of nested display components: a window containing panels containing buttons and text fields. This hierarchical structure is exactly what the Composite Pattern models, and it is why Composite is one of the three constituent patterns of MVC.

The View registers itself with the Model as an Observer, so it receives notifications whenever the Model changes. Upon receiving a notification, the View re-queries the Model and redraws.

The Controller

The Controller sits between the user and the Model. It is responsible for interpreting user actions and translating them into Model mutations. When the user clicks "play", the View captures the click event and passes it to the Controller; the Controller interprets it and calls the appropriate Model methods.

The key insight about the Controller is that it is the Strategy for the View. The View is configured with a specific Controller object; swapping the Controller yields completely different behaviour from the same View. A read-only view might be configured with a Controller that ignores all mutation events. An admin view might be configured with a Controller that accepts all of them. The View itself does not change; only its strategy changes.

> **Tip:**
>
> **The Controller is not a traffic cop.** A common misunderstanding is that the Controller directs all traffic between the Model and the View, and that all communication must pass through it. That is not MVC. The View queries the Model directly for the state it needs to render (step ⑤). The Controller only handles user-initiated events. Routing all data through the Controller would create unnecessary coupling and make the system more rigid, not less.

The Patterns Inside MVC

Observer: Model notifies View

The Model extends Subject. The View implements Observer (and in our implementation, so does StudentGroup, which is how a change to a student also notifies every group on its path to the root). Whenever a Model setter changes data, it calls notifyObservers(), which iterates the registered observer list and calls update() on each one. The Model never references a View directly; it does not know the View exists.

This decoupling has a powerful consequence: you can have multiple Views observing the same Model simultaneously. All of them receive the same notification, and all of them redraw. A student management system might have a table view and a statistics view, both observing the same Student models. Adding or modifying a student automatically updates both views, with no extra code.

Strategy: Controller is the View's strategy

The View holds a reference to its Controller and delegates all user-interaction logic to it. The Controller is the View's strategy for behaviour. In our implementation, StudentView holds a shared_ptr<ControllerInterface> and forwards every user event to it; swapping in a ReadOnlyController turns the very same View read-only, no View or Model code changes. You can swap the Controller to get different behaviour from the same View, which is extremely powerful for testing: in a unit test, you can configure the View with a mock Controller that simply records which methods were called, without actually touching the Model.

The controller is the strategy for the view. The view can use different controllers to get different behaviour.

Composite: The View is a tree

Real-world Views are not flat objects. A window contains panels; panels contain buttons, labels, and text fields; those may themselves contain sub-components. This is the Composite Pattern: every component in the tree, whether a leaf (a single button) or a composite (a panel containing several buttons), implements the same interface. When the Controller tells the root to update, the Composite structure automatically propagates the update call to every descendant.

Pattern	Role in MVC	Problem it solves
Observer	Model to View (and Controller)	Keeps Model ignorant of View; supports multiple simultaneous Views; View registers and deregisters at runtime
Strategy	View to Controller	Makes Controller swappable; View delegates behaviour without knowing details; enables testing with mock controllers
Composite	Inside the View	View is built from nested components; Controller only calls the root; tree handles propagation automatically

A Complete C++ Example: Student Information System

A student management system tracks student names, roll numbers, and scores; computes letter grades using a pluggable grading strategy; organizes students into a tree of nested groups (sections within courses within a department); and automatically notifies all interested parties whenever any student's data changes.

The following classes participate in the design:

- **Observer and Subject,** the Observer Pattern infrastructure.
- **GradeStrategy, SimpleGradeStrategy, CurvedGradeStrategy,** the Strategy Pattern for grade calculation.
- **StudentComponent, StudentLeaf, and StudentGroup,** the Composite Pattern: a tree of nested groups with students at the leaves.
- **Student,** the Model (inherits Subject).
- **StudentView,** the View (inherits Observer).
- **ControllerInterface, StudentController, and ReadOnlyController,** the Controller, implemented as the View's swappable Strategy.

Observer and Subject base classes

```
// Observer Pattern


class Observer {

public:
    virtual ~Observer() = default;
    virtual void update() = 0;

protected:
    Observer() = default;
};



class Subject {

public:
    virtual ~Subject() = default;
    void addObserver(Observer* observer) {
        if (observer && std::find(observers_.begin(),
                                  observers_.end(), observer)
                        == observers_.end()) {
            observers_.push_back(observer);
        }
    }
```

```
    void removeObserver(Observer* observer) {
        observers_.erase(
            std::remove(observers_.begin(), observers_.end(), observer),
            observers_.end());
    }
    void notifyObservers() {
        for (auto* observer : observers_) {
            observer->update();
        }
    }

protected:
    Subject() = default;

private:
    std::vector<Observer*> observers_;

};
```

Subject manages the observer list with three operations: addObserver (with duplicate-detection using std::find), removeObserver (using the erase-remove idiom), and notifyObservers (a range-based for loop over the list). Both constructors are protected to prevent direct instantiation.

GradeStrategy: the Strategy Pattern

```
// Strategy Pattern

class GradeStrategy {

public:
    virtual ~GradeStrategy() = default;
    virtual char calculateGrade(double score) const = 0;

protected:
    GradeStrategy() = default;
};

class SimpleGradeStrategy final: public GradeStrategy {

public:
    char calculateGrade(double score) const override {
        if (score >= 90) return 'A';
        if (score >= 80) return 'B';
        if (score >= 70) return 'C';
        if (score >= 60) return 'D';
        return 'F';
    }
};
```

```
class CurvedGradeStrategy final: public GradeStrategy {

public:
    explicit CurvedGradeStrategy(double curveAmount)
        : curveAmount_(curveAmount) {}

     char calculateGrade(double score) const override {
        double curved = std::min(100.0, score + curveAmount_);
        if (curved >= 90) return 'A';
        if (curved >= 80) return 'B';
        if (curved >= 70) return 'C';
        if (curved >= 60) return 'D';
        return 'F';
    }

private:
    double curveAmount_;
};
```

GradeStrategy is the abstract algorithm for converting a numeric score to a letter grade. SimpleGradeStrategy applies a fixed scale. CurvedGradeStrategy adds a curve amount (capped at 100) before applying the same scale. The Student Model holds a shared_ptr<GradeStrategy> and delegates getGrade() to it, so the grading algorithm can be changed at runtime without modifying the Student class.

The Composite: StudentComponent, StudentLeaf, StudentGroup

In the flat version of this system, a group was nothing more than a list of students. Real academic structures are trees: a department contains courses, a course contains sections, and a section contains students. This is exactly the shape the Composite Pattern is built for. Three classes play the three Composite roles: **StudentComponent** is the common Component interface, **StudentLeaf** is the Leaf that wraps a single Student, and **StudentGroup** is the Composite whose children may be leaves or other groups. Client code treats a single student and an entire department uniformly through the same interface.

```
// Composite Pattern: the Component interface

class StudentComponent: public Subject, public Observer {

public:
    ~StudentComponent() override = default;
    virtual void display(int indent) const = 0;

    // Default behaviour for a leaf: groups override these.
    virtual void add(std::shared_ptr<StudentComponent>) {
        std::cerr << "add() is not supported on a leaf\n";
    }
    virtual void remove(const std::shared_ptr<StudentComponent>&) {
        std::cerr << "remove() is not supported on a leaf\n";
    }
```

```
protected:
    StudentComponent() = default;

    // Shared print helper: indent 0 is the root (no connector), every
    // deeper level gets a "|-- " branch connector.
    static void printLine(int indent, const std::string& label) {
        if (indent > 0)
            std::cout << std::string((indent - 1) * 4, ' ') << "|-- ";
        std::cout << label << '\n';
    }
};
```

StudentComponent declares the one operation every node in the tree supports: display(indent), which recurses downward and prints the node (and, for a group, its children) with an indentation-based branch connector. add() and remove() default to rejecting the call (“not supported on a leaf”); StudentGroup overrides both to manage its children. StudentComponent itself inherits from both Subject and Observer, so every node in the tree, leaf or group, gets addObserver()/removeObserver() for free — there is no separate attach()/detach() indirection layer.

```
// Composite Pattern: the Leaf

class StudentLeaf final: public StudentComponent {

public:
    explicit StudentLeaf(std::shared_ptr<Student> student)
        : student_(std::move(student)) {
        if (student_) student_->addObserver(this);   // leaf observes the
Model
    }
    ~StudentLeaf() override {
        if (student_) student_->removeObserver(this);
    }

    void display(int indent) const override {
        printLine(indent,
                  std::string(student_->getName()) + " (Roll No: "
                      + std::string(student_->getRollNo()) + ", Grade: "
                      + student_->getGrade() + ")");
    }

    // Student changed -> tell whichever StudentGroup holds this leaf.
    void update() override { notifyObservers(); }

private:
    std::shared_ptr<Student> student_;
};
```

StudentLeaf terminates the recursion. It wraps a shared_ptr to a Student model and registers itself as an Observer of that Student in its constructor, deregistering in its destructor, so a change to the underlying

Student flows straight into the leaf. Its display() prints a single indented line: name, roll number, and grade. Its update() (inherited from Observer) simply calls notifyObservers() again, forwarding the change to whatever is observing the leaf itself, ordinarily its parent group.

```
// Composite Pattern: the Composite

class StudentGroup final: public StudentComponent {

public:
    explicit StudentGroup(std::string name) : name_(std::move(name)) {}
    ~StudentGroup() override {
        for (const auto& child : children_) child->removeObserver(this);
    }

    // Delete copy; a copied group would double-deregister on destruction
    StudentGroup(const StudentGroup&) = delete;
    StudentGroup& operator=(const StudentGroup&) = delete;

    void add(std::shared_ptr<StudentComponent> child) override {
        if (!child) return;
        child->addObserver(this);      // this group observes the child
        children_.push_back(std::move(child));
    }
    void remove(const std::shared_ptr<StudentComponent>& child) override {
        if (!child) return;
        child->removeObserver(this);
        children_.erase(
            std::remove(children_.begin(), children_.end(), child),
            children_.end());
    }

    void display(int indent) const override {
        printLine(indent, name_);
        for (const auto& c : children_) c->display(indent + 1);
    }

    // A child changed -> announce it, then keep climbing toward the root.
    void update() override {
        std::cout << "Group " << name_ << " notified\n";
        notifyObservers();
    }

    std::string_view getName() const noexcept { return name_; }

private:
    std::string name_;
    std::vector<std::shared_ptr<StudentComponent>> children_;
};
```

StudentGroup is the Composite and the heart of the tree. Its children are stored as StudentComponent pointers, so a child may be a StudentLeaf or another StudentGroup; nesting groups inside groups is what

produces a tree of arbitrary depth. display() loops over the children and recurses, indenting one level deeper each time, so a single call on the root walks and prints the entire tree.

StudentGroup also participates in the Observer Pattern from both sides. It is an **Observer**: add() calls child->addObserver(this), so the group hears about any change in a direct child, whether that child is a leaf or another group. It is also a **Subject** (inherited through StudentComponent): its update() prints a notification and calls notifyObservers(), re-broadcasting the change to whatever is observing this group, ordinarily its parent. A change to a leaf therefore climbs the tree, parent by parent, all the way to the root. When Robert's score changes, Section A is notified, then CS101, then the CS Department. Downward recursion for display, upward propagation for notifications: the two directions together are what make the Composite tree feel like a single living structure.

As with StudentView, copy operations are deleted: a copied group could leave two StudentGroup instances registered as the same observer, or double-deregister on destruction. The destructor calls removeObserver(this) on every child, so no dangling observer pointers are left behind when a group is destroyed.

Student: the Model

```
// Model

class Student: public Subject {

public:
    Student(std::string rollNo, std::string name, double score,
            std::shared_ptr<GradeStrategy> gradeStrategy)
        : rollNo_(std::move(rollNo))
        , name_(std::move(name))
        , score_(score)
        , gradeStrategy_(std::move(gradeStrategy)) {}

    std::string_view getRollNo() const noexcept { return rollNo_; }

    void setRollNo(std::string rollNo) {

        rollNo_ = std::move(rollNo);
        notifyObservers();
    }

    std::string_view getName() const noexcept { return name_; }
    void setName(std::string name) {
        name_ = std::move(name);
        notifyObservers();
    }

    double getScore() const noexcept { return score_; }
    void setScore(double score) {
        score_ = score;
        notifyObservers();
    }
```

```
    char getGrade() const {
        return gradeStrategy_->calculateGrade(score_);
    }

    void setGradeStrategy(std::shared_ptr<GradeStrategy> strategy) {
        gradeStrategy_ = std::move(strategy);
        notifyObservers();
    }

private:
    std::string rollNo_;
    std::string name_;
    double score_;
    std::shared_ptr<GradeStrategy> gradeStrategy_;
};
```

Student is the Model. It inherits from Subject, giving it the full observer-management infrastructure. Every setter ends with notifyObservers(), so any change to any field, name, roll number, score, or grading strategy immediately notifies every registered observer. The Student does not know who those observers are; it just broadcasts. Notice that getGrade() delegates to the gradeStrategy_, the Student itself has no grading logic, which is the Strategy Pattern at work inside the Model.

StudentView: the View

```
// Controller interface: the View's Strategy

class ControllerInterface {
public:
    virtual ~ControllerInterface() = default;
    // User-initiated events, forwarded by the View:
    virtual void scoreEntered(double score) = 0;
protected:
    ControllerInterface() = default;
};

// View
class StudentView final: public Observer {
public:
    StudentView(std::shared_ptr<Student> model,
                std::shared_ptr<ControllerInterface> controller)
        : model_(std::move(model)), controller_(std::move(controller)) {
        model_->addObserver(this);   // step 4: the View observes the Model
    }
    ~StudentView() override {
        if (model_) model_->removeObserver(this);
    }

    // Delete copy; a copied view would double-deregister on destruction
```

```
    StudentView(const StudentView&) = delete;
    StudentView& operator=(const StudentView&) = delete;

    // Strategy Pattern: swap the Controller -> same View, new behaviour
    void setController(std::shared_ptr<ControllerInterface> controller) {
        controller_ = std::move(controller);
    }

    // Simulated user input: the View captures the event and DELEGATES
    // it to its Controller; the View never interprets the event itself.
    void userTypedScore(double score) {
        if (controller_) controller_->scoreEntered(score);
    }

    // Observer callback: query the Model directly and redraw (step 5)
    void update() override { draw(); }

    void draw() const {
        std::cout << "Student:\n";
        std::cout << "Name: "    << model_->getName()   << '\n';
        std::cout << "Roll No: " << model_->getRollNo() << '\n';
        std::cout << "Grade: "   << model_->getGrade()  << '\n';
    }

private:
    std::shared_ptr<Student> model_;                    // queried, never
mutated
    std::shared_ptr<ControllerInterface> controller_;  // the View's
strategy
};
```

StudentView now plays both roles assigned by the theory. It is an Observer of the Model: it registers in its constructor, deregisters in its destructor, and its update() queries the Model directly and redraws, exactly steps ④ and ⑤ of the overview. And it delegates behaviour to its Strategy: every user event (simulated here by userTypedScore) is forwarded, uninterpreted, to whatever ControllerInterface it currently holds. The View reads the Model but never mutates it, nor does it decide what a user action means. If you wanted a web view or a JSON view, you would write a different Observer with its own draw(); the Controller would not change at all.

> ⚠ **Warning:**
>
> **Always deregister in the destructor.** If the View is destroyed while it is still registered as an observer, the Model's observer list contains a dangling pointer. The next call to notifyObservers() will dereference it, undefined behaviour, likely a crash. The destructor's model_->removeObserver(this) call is not optional.

StudentController: the Controller

```
// Controller: one concrete strategy for the View
class StudentController final: public ControllerInterface {
public:
    explicit StudentController(std::shared_ptr<Student> model)
        : model_(std::move(model)) {}

    void scoreEntered(double score) override {
        model_->setScore(score);   // mutate the Model; the Model itself
    }                              // notifies its observers
private:
    std::shared_ptr<Student> model_;
};

// A second strategy: the same View becomes read-only
class ReadOnlyController final: public ControllerInterface {
public:
    void scoreEntered(double) override {
        std::cout << "[read-only mode: change ignored]\n";
    }
};
```

The Controller does exactly one job: translate user events into Model mutations. It holds no View, ferries no data, and renders nothing. When it calls setScore(), the Model notifies the View directly, without the Controller in the loop. This is what the “not a traffic cop” tip means in code. ReadOnlyController demonstrates the Strategy relationship: hand it to the same StudentView, and every user event is ignored, with zero changes to the View or the Model.

main(): wiring it all together

main() builds a three-level tree: a department containing a course containing two sections, with students as the leaves. Note that the same add() call inserts a leaf into a section and a section into a course; that uniformity is the Composite Pattern’s whole point.

```
// Main function demonstrating MVC with Strategy, Observer,
// and a tree-structured Composite
int main() {
    // Create grading strategy (shared between students)
    auto gradeStrategy = std::make_shared<SimpleGradeStrategy>();

    // Create students (Models)
    auto student1 = std::make_shared<Student>("10", "Robert", 85.0,
gradeStrategy);
    auto student2 = std::make_shared<Student>("20", "John", 92.0,
gradeStrategy);
    auto student3 = std::make_shared<Student>("30", "Alice", 78.0,
gradeStrategy);
```

```
    // Create a controller (the View's strategy) and a View per student
    auto controller1 = std::make_shared<StudentController>(student1);
    auto controller2 = std::make_shared<StudentController>(student2);
    StudentView view1(student1, controller1);
    StudentView view2(student2, controller2);

    // Build the Composite TREE:
    //
    //   CS Department
    //   |-- CS101
    //   |   |-- Section A
    //   |   |   |-- Robert
    //   |   |   `-- John
    //   |   `-- Section B
    //   |       `-- Alice
    //   `-- (more courses could be added here)
    auto sectionA   = std::make_shared<StudentGroup>("Section A");
    auto sectionB   = std::make_shared<StudentGroup>("Section B");
    auto cs101      = std::make_shared<StudentGroup>("CS101");
    auto department = std::make_shared<StudentGroup>("CS Department");
    sectionA->add(std::make_shared<StudentLeaf>(student1));
    sectionA->add(std::make_shared<StudentLeaf>(student2));
    sectionB->add(std::make_shared<StudentLeaf>(student3));
    cs101->add(sectionA);       // a group nested inside a group
    cs101->add(sectionB);
    department->add(cs101);     // three levels deep

    department->display(0);
    // Initial render: each View queries its own Model directly
    view1.draw();
    view2.draw();
    // Simulate user input. The event flows View -> Controller -> Model;
    // the Model then notifies view1 AND Section A, and the notification
    // climbs the tree: Section A -> CS101 -> CS Department.
    view1.userTypedScore(95.0);   // Robert: B -> A
    // Strategy Pattern: swap the controller -> same View, new behaviour
    view1.setController(std::make_shared<ReadOnlyController>());
    view1.userTypedScore(10.0);   // ignored; Robert keeps his A
    // Display the refreshed tree from the root
    department->display(0);
    return 0;
}
```

Expected output

```
CS Department
|-- CS101
    |-- Section A
        |-- Robert (Roll No: 10, Grade: B)
        |-- John (Roll No: 20, Grade: A)
    |-- Section B
        |-- Alice (Roll No: 30, Grade: C)
Student:
Name: Robert
Roll No: 10
Grade: B
Student:
Name: John
Roll No: 20
Grade: A
Student:
Name: Robert
Roll No: 10
Grade: A
Group Section A notified
Group CS101 notified
Group CS Department notified
[read-only mode: change ignored]
CS Department
|-- CS101
    |-- Section A
        |-- Robert (Roll No: 10, Grade: A)
        |-- John (Roll No: 20, Grade: A)
    |-- Section B
        |-- Alice (Roll No: 30, Grade: C)
```

Trace the output to watch the tree work in both directions. The first display(0) call on the department recurses downward through all three levels, printing every group and student with the same code at every level. When view1.userTypedScore(95.0) executes, the Controller calls setScore(95.0), and the Model notifies view1 (which redraws) and the Robert leaf; the leaf's update() re-notifies Section A, which re-notifies CS101, which re-notifies CS Department — the three "Group … notified" lines are that climb made visible. The final display(0) shows Robert's grade has changed from B to A everywhere he appears in the tree, while Alice and Section B are untouched.

Notes:

- **Smart pointers (shared_ptr).** Models are shared between a View, a Controller, and the composite tree; controllers are held by shared_ptr so a View can swap them at runtime. Choosing the right ownership semantics prevents double-deletion and memory leaks.
- **Move semantics.** String parameters in setters use std::move() to transfer ownership rather than copying. Constructor initializer lists use std::move() for the same reason.
- **std::string_view for getters.** Getters return string_view, a non-owning reference to the underlying string. The caller gets read access without any allocation.

- **noexcept.** Methods that cannot throw are marked noexcept. This allows the compiler to generate leaner code at call sites and allows standard containers to use move operations with these types.
- **final.** Concrete leaf classes (SimpleGradeStrategy, CurvedGradeStrategy, StudentView, StudentController, ReadOnlyController, StudentLeaf, StudentGroup) are marked final to prevent unintended subclassing and to enable devirtualisation optimizations.
- **explicit constructors.** Single-argument constructors are marked explicit to prevent implicit conversions that could cause hard-to-diagnose bugs.
- **Rule of Five on StudentView and StudentGroup.** Both manage an observer registration as a resource, a View observes its Model, a Composite observes its children, so both must define or delete all five special member functions. Copy is deleted in both; each destructor deregisters: StudentView from its Model, StudentGroup from every one of its children.

Summary

Recognition: what to spot in code

- *A Model class* that inherits from Subject and calls notifyObservers() in every setter.
- *A View class* that inherits from Observer, holds a shared_ptr to its Model and a swappable ControllerInterface; its update() queries the Model and redraws.
- *A Controller class* that implements ControllerInterface and only translates user events into Model mutations; it never touches the View.
- *A Composite tree* in which leaves (single students) and groups implement one Component interface; groups may contain leaves and other groups, and a change to a leaf climbs the tree, notifying every ancestor group up to the root.

Comprehension: how the pieces fit

- MVC is a compound pattern: Observer + Strategy + Composite, each solving a distinct coupling problem within the same design.
- The Model uses Observer so that it can notify Views and Controllers without knowing they exist. Multiple Views can watch the same Model simultaneously.
- The Controller is the Strategy for the View: swapping the Controller changes the behaviour the user experiences without touching the View or the Model.
- The View is structured as a Composite tree: the Controller updates the root, and the tree handles propagation; a single call refreshes the entire display.
- The Model is completely ignorant of the View. It fires notifications; Views subscribe. This decoupling means the Model can be tested without any display infrastructure.

Application: what to actually do

- **Put all data and business logic in the Model.** Nothing that is not display logic belongs in the View. Nothing that is not user-event handling belongs in the Controller.
- **Every Model setter must call notifyObservers().** Forgetting this call is the most common MVC implementation bug. The View silently shows stale data.
- **The View must register and deregister correctly.** Register in the constructor; deregister in the destructor. Forgetting deregistration leaves a dangling pointer in the observer list.
- **Delete copy on the View.** Two views watching the same model, both trying to deregister the same pointer on destruction, is undefined behaviour.
- **Use shared_ptr for shared ownership.** When a Model is shared between a View, a Controller, and a Composite Group, shared_ptr is the right ownership model.

Practice Problems

1.

Using the student information system from this chapter, trace exactly what happens, method by method, object by object, when this line executes:

```
view1.userTypedScore(95.0);
```

For each method call in the chain, state: (a) the name of the method, (b) which object it is called on, (c) what it does, and (d) what, if anything, it triggers next. How many update() calls fire in total? On which objects?

2.

Without using the word "MVC", identify the three design patterns present in the student information system and explain what coupling problem each one solves in this specific context. For each pattern, name: (a) the abstract role, (b) the concrete class playing that role, and (c) what would go wrong if you removed that pattern.

3.

Add a CSVStudentView class that formats output as comma-separated values:

```
10,Robert,A
```

Attach a CSVStudentView as a second View observing the same student1 model; when view1.userTypedScore(95.0) is called, both views' update() methods fire. Does anything in the Student class or the Composite tree classes (StudentComponent, StudentLeaf, StudentGroup) need to change? Why or why not?

4.

Demonstrate that the Strategy Pattern makes grading algorithms swappable at runtime:

- Create student1 with SimpleGradeStrategy and score 72.0. Print the grade (should be C).
- Call student1->setGradeStrategy(make_shared<CurvedGradeStrategy>(10.0)). Print the grade again (should be B, score 82 after curving).
- Verify that every group on student1's path to the root receives an update notification when the strategy changes (because setGradeStrategy calls notifyObservers()).

What would need to change if grading were hard-coded into the Student class instead of delegated to a strategy?

5.

The following code has a bug that causes undefined behaviour. Identify it, explain why it is undefined behaviour, and fix it:

```
auto student = std::make_shared<Student>("10", "Alice", 88.0,
                   std::make_shared<SimpleGradeStrategy>());
{
    StudentView view(student, std::make_shared<StudentController>(student));
}  // view destroyed here, deregisters from student? Does it?
student->setScore(90.0);  // What happens?
```

6.

Write a test proving that StudentGroup::remove correctly deregisters:

- Add leaves for student1 and student2 to a group.
- Remove student1's leaf from the group.
- Call view1.userTypedScore(99.0). Verify the group's update() does NOT fire.
- Call view2.userTypedScore(50.0). Verify the group's update() DOES still fire.

7.

Apply the MVC pattern to a library management system:

- Model: a Book class with title, author, ISBN, and availableCopies. Every setter calls notifyObservers().
- View: a BookView class with printBookDetails(title, author, isbn, copies).
- Controller: a BookController that exposes checkOut() and returnBook(). Both call notifyObservers() indirectly through the Model.
- Composite tree: a BookshelfGroup (with BookLeaf and a BookComponent interface) that groups books by genre, allowing shelves nested inside shelves, and prints the full shelf listing whenever any book changes.

Write complete C++ class declarations and a main() that demonstrates a checkout, a view update, and a group notification.

8.

The lecture notes state that the Observer Pattern "allows us to use different views with the same model, or even use multiple views at once." Demonstrate this property:

- Create one Student model.
- Attach three different observers: a StudentView, a CSVStudentView, and a StudentGroup.
- Call setScore() once and show that all three observers receive the notification and each renders the student in its own format.

How many lines of code in the Student class need to change to support this? (The answer should be zero.)

9.

Extend the student information system so that each Student has an AcademicStatus governed by the State Pattern:

- States: PROBATION (score < 60), PASSING (60 <= score < 90), HONOURS (score >= 90).
- The AcademicStatus transitions automatically inside setScore(), the State Pattern handles the transition logic.
- The Student's notifyObservers() call fires after the status transition, so the View and Controller receive the update.
- Add getStatusName() to Student and display it in StudentView::draw().

This exercise combines MVC (outer architecture) with State (inner lifecycle management), a combination common in real systems wherever an object's behaviour changes based on accumulated data.

10.

The student information system is not thread-safe. Analyse the following scenario: Thread A calls view1.userTypedScore(95.0) while Thread B calls sectionA->add(std::make_shared<StudentLeaf>(student3)) simultaneously.

- Identify every shared data structure that could be simultaneously read and written.
- Describe the specific data race, what could go wrong, concretely.
- Propose a minimal fix using std::mutex. Where should the lock be held? How long?
- What are the risks of holding the lock inside notifyObservers() while calling observer->update(), and what alternative design would avoid those risks?