a2 # advanced # design-patterns # optional

Exploring the Road Rage Codebase: Design Patterns in Action

TCSS 305 Programming Practicum

> **This Guide is Optional**
>
> You don't need this guide to complete Assignment 2. Your task is to implement vehicle classes—the GUI and supporting code work "out of the box." But if you're curious about **how** that code is organized, this guide reveals the professional design patterns hidden in plain sight.

---

1 Who This Is For

This guide is for students who:

- Want to understand the "magic" behind the GUI
- Are curious about professional code organization
- Like exploring codebases to see how things work
- Want a preview of design patterns covered later in TCSS 305

If you just want to complete the assignment, skip this guide and focus on your vehicle implementations. Come back later if you're curious!

---

2 What You'll Discover

The Road Rage starter code demonstrates several professional design patterns:

Pattern	Where	What It Does
Strategy	Collision resolution	Swappable algorithms without changing core logic
Factory	Collision modes, vehicle loading	Centralized object creation
Observer	GUI updates	Components react to state changes without tight coupling
Sealed Types	Event system	Type-safe events with exhaustive pattern matching

You'll learn these patterns formally later in TCSS 305. This guide shows you real implementations to build intuition now.

---

3 The Collision System: Strategy + Factory

3.1 What You See in the GUI

The GUI has a **Collision Mode** dropdown with three options:

- Mass (Realistic) - heavier vehicles win
- Random (Chaos) - winner chosen randomly
- Inverted Mass (Underdog) - lighter vehicles win

When you change modes, collision behavior changes instantly. How does this work without rewriting the collision logic?

3.2 The Strategy Pattern

Core idea: Encapsulate different algorithms in separate objects, then swap them at runtime.

Open `logic/CollisionComparators.java` in your project. You'll see this structure:

```
public final class CollisionComparators {
    private static final Random RANDOM = new Random();
```

```
    // Strategy 1: Mass-based comparison
    private static final Comparator<Vehicle> BY_MASS =
        Comparator.comparing(Vehicle::getMass);

    // Strategy 2: Random comparison
    private static final Comparator<Vehicle> RANDOM_OUTCOME =
        (v1, v2) -> RANDOM.nextInt(3) - 1;  // Returns -1, 0, or 1 randomly

    // Strategy 3: Inverted mass (lighter wins)
    private static final Comparator<Vehicle> INVERTED_MASS =
        Comparator.comparing(Vehicle::getMass).reversed();

    // ... factory methods below
}
```

Each collision mode is a **different** `Comparator<Vehicle>` —a different strategy for deciding who wins.

3.3 How the Simulation Uses Strategies

In `logic/RoadRage.java`, the collision resolution code doesn't know which mode is active:

```
private void resolveCollision(final Vehicle theV1, final Vehicle theV2) {
    // Use the current comparator (whatever it is)
    final int result = myCollisionComparator.compare(theV1, theV2);

    if (result > 0) {
        theV2.collide();  // theV1 wins
    } else if (result < 0) {
        theV1.collide();  // theV2 wins
    } else {
        // Equal: randomly pick a loser (50/50)
        if (RANDOM.nextBoolean()) {
            theV1.collide();
        } else {
            theV2.collide();
        }
    }
}
```

Notice: `myCollisionComparator` could be **any** of the three strategies. The code doesn't care—it just calls `compare()` and uses the result.

Polymorphism in Action!

This is **polymorphism**—the same concept from the Polymorphism guide!

The line `myCollisionComparator.compare(theV1, theV2)` is **identical** regardless of which collision mode is active. But the **behavior** changes completely:

- With `BY_MASS`, it compares mass values
- With `RANDOM_OUTCOME`, it returns a random result
- With `INVERTED_MASS`, it compares mass in reverse

Same method call, different behavior. That's the power of polymorphism. The `resolveCollision()` code doesn't know or care which comparator it's using—it just calls `compare()` and trusts the result.

This is exactly what you saw in the `advance()` loop with `v.chooseDirection(neighbors)` — same call, different behavior for Truck vs Car vs Bicycle.

Why this matters:

- Adding a new collision mode requires ZERO changes to `resolveCollision()`
- Just create a new `Comparator` and make it available
- The algorithm is **pluggable**

3.4 The Factory Pattern

How does the GUI know what collision modes exist? It asks the factory.

Back in `CollisionComparators.java`:

```
// A record pairing display name with comparator
public record CollisionMode(String displayName,
                            Comparator<Vehicle> comparator) { }

// Factory method: returns all available modes
public static List<CollisionMode> getAll() {
    return List.of(
        new CollisionMode("Mass (Realistic)", BY_MASS),
        new CollisionMode("Random (Chaos)", RANDOM_OUTCOME),
        new CollisionMode("Inverted Mass (Underdog)", INVERTED_MASS)
    );
}

// Factory method: returns the default mode
public static Comparator<Vehicle> getDefault() {
    return BY_MASS;
}
```

The GUI calls `CollisionComparators.getAll()` and builds a dropdown from the list. It doesn't hardcode mode names or know how many modes exist—it just asks the factory.

Why this matters:

- Single source of truth: all collision modes defined in one place
- GUI doesn't need to know implementation details
- Adding a new mode: add it to `getAll()`, done

3.5 Try It Yourself

Want to see the Strategy + Factory patterns in action?

1. Open `logic/CollisionComparators.java`
2. Add a new strategy that favors vehicles with **lower disabled duration**:

```
private static final Comparator<Vehicle> BY_DISABLED_DURATION =
    Comparator.comparing(Vehicle::getDisabledDuration);
```

3. Add it to `getAll()`:

```
new CollisionMode("Disabled Duration (Quick Recovery Wins)",
BY_DISABLED_DURATION)
```

4. Run the GUI—your new mode appears in the dropdown automatically!
5. Select it and watch how it changes collision outcomes

No other code changes needed. That's the power of Strategy + Factory.

---

4 The Event System: Observer + Sealed Types

4.1 What You See in the GUI

When vehicles move, the GUI updates. When lights change, the rendering changes.

All of these state changes happen in the **model** and **logic** packages—the core simulation. But how does the GUI in the **view** package know when to redraw? The model and view are in separate packages and shouldn't directly reference each other. How do they communicate?

4.2 The Observer Pattern

Core idea: Objects "subscribe" to state changes and get notified automatically.

The `logic` package (game state) doesn't directly call the `view` package (rendering). Instead, it fires events:

```
// In RoadRage.java (logic package)
private void fireVehicleChange() {
    myPcs.firePropertyChange(
        new RoadRageEvent.VehiclesChanged(
            List.copyOf(myVehicles),
            System.currentTimeMillis()
        )
    );
}
```

The GUI **observes** these events:

```
// In RoadRagePanel.java (view package)
@Override
public void propertyChange(final PropertyChangeEvent theEvent) {
    if (theEvent.getNewValue() instanceof final RoadRageEvent event) {
        switch (event) {
            case final RoadRageEvent.VehiclesChanged e ->
updateVehicles(e.vehicles());
            case final RoadRageEvent.LightChanged e ->
updateLight(e.newLight());
            case final RoadRageEvent.GridChanged e -> updateGrid(e.grid());
            case final RoadRageEvent.TimeChanged e -> updateTime(e.newTime());
        }
    }
}
```

Why this matters:

- Logic and view are **decoupled**—they don't directly reference each other
- Logic says "something changed" without knowing who's listening
- View reacts without the logic knowing it exists
- Easy to add new listeners (debuggers, loggers, etc.) without changing logic

4.3 Sealed Types: Type-Safe Events

Old-school Java uses **string constants** for event names:

```
// Old way (type-unsafe)
pcs.firePropertyChange("vehiclesChanged", oldList, newList);
```

```
// Listener has to cast and hope for the best
if ("vehiclesChanged".equals(evt.getPropertyName())) {
    List<Vehicle> vehicles = (List<Vehicle>) evt.getNewValue();  // Unsafe
cast!
}
```

Problems: typos compile, casts can fail at runtime, no IDE autocomplete.

Modern solution: Sealed interfaces with records.

Open `logic/RoadRageEvent.java`:

```
public sealed interface RoadRageEvent permits
        LightChanged, VehiclesChanged, GridChanged, TimeChanged {

    long timestamp();

    default String getPropertyName() {
        return this.getClass().getSimpleName();
    }

    record LightChanged(Light oldLight, Light newLight, long timestamp)
        implements RoadRageEvent { }

    record VehiclesChanged(List<Vehicle> vehicles, long timestamp)
        implements RoadRageEvent { }

    record GridChanged(Terrain[][] grid, long timestamp)
        implements RoadRageEvent { }

    record TimeChanged(long oldTime, long newTime, long timestamp)
        implements RoadRageEvent { }
}
```

Why this matters:

- **Sealed interface:** Only the 4 permitted records can implement `RoadRageEvent`—compiler knows all possible types
- **Pattern matching:** Switch statement is **exhaustive**—compiler ensures you handle all cases
- **Type safety:** `VehiclesChanged` carries `List<Vehicle>`, no casting needed
- **IDE support:** Autocomplete shows all event types

This is modern Java (JDK 17+): type-safe, concise, and compiler-verified.

---

5 Design Patterns: The Big Picture

5.1 What Are Design Patterns?

Design patterns are reusable solutions to common problems in software design. They're not finished code you can copy-paste. Instead, they're templates or blueprints for how to solve recurring challenges.

Think of them like architectural patterns in building construction: - **The problem:** "How do we build a strong roof that spans a large space?" - **The pattern:** "Use an arch—distribute weight through compression" - **The implementation:** Each building uses arches differently, but the core principle is the same

In software: - **The problem:** "How do we change algorithms at runtime without rewriting code?" - **The pattern:** "Strategy—encapsulate each algorithm in its own class" - **The implementation:** Road Rage uses Strategy for collision modes; Netflix uses it for recommendation algorithms

Why patterns matter:

1. **Common vocabulary** - Saying "we'll use the Observer pattern" communicates the entire design in three words
2. **Proven solutions** - These patterns have been tested in thousands of real systems
3. **Easier maintenance** - Other developers recognize the pattern and understand the structure immediately

The patterns in Road Rage aren't just academic exercises. They're the same patterns used in professional systems you use every day.

5.2 What You've Seen

Pattern	Purpose	Where in Road Rage
Strategy	Swap algorithms at runtime	Collision comparators
Factory	Centralize object creation	`CollisionComparators.getAll()`, vehicle loading
Observer	React to state changes without tight coupling	PropertyChangeListener for GUI updates
Sealed Types	Exhaustive type-safe variants	`RoadRageEvent` with records

5.3 Why Professional Code Uses These Patterns

Without patterns:

```
// Collision resolution WITHOUT Strategy pattern
if (collisionMode.equals("mass")) {
    if (v1.getMass() > v2.getMass()) {
        v2.collide();
    } else {
        v1.collide();
    }
} else if (collisionMode.equals("random")) {
    if (Math.random() < 0.5) {
        v1.collide();
    } else {
        v2.collide();
    }
} else if (collisionMode.equals("inverted")) {
    // ... more if-else
}
// Adding new mode = modify this method!
```

Problems: - Violates Open/Closed Principle (modify existing code to add features) - Hard to test (can't test modes in isolation) - Hard to maintain (collision logic mixed with mode selection)

With Strategy pattern:

```
// Collision resolution WITH Strategy pattern
int result = myCollisionComparator.compare(v1, v2);
if (result > 0) {
    v2.collide();
} else if (result < 0) {
    v1.collide();
}
// Adding new mode = add new Comparator, ZERO changes here!
```

Benefits: - Open/Closed: open for extension (new strategies), closed for modification (core logic unchanged) - Testable: test each strategy independently - Maintainable: clear separation of concerns

5.4 When You'll Learn These Formally

Design patterns are woven throughout the rest of TCSS 305. You'll study them systematically, practice implementing them, and learn to recognize when to apply each pattern.

The course culminates in a **group project** where you'll design and implement a complete application using several common design patterns—applying the same architectural principles you're seeing in Road Rage.

For now, seeing these patterns "in the wild" builds intuition. You'll return to this guide later in the quarter with deeper understanding of why these design choices matter.

---

6 Try It Yourself: Challenges

Want to explore further? Try these challenges:

6.1 Challenge 1: Add a New Collision Mode

Goal: Create a "Disabled Duration" collision mode where vehicles that recover faster win.

Steps:

1. Open `logic/CollisionComparators.java`
2. Add a new comparator:

```
private static final Comparator<Vehicle> BY_DISABLED_DURATION =
    Comparator.comparing(Vehicle::getDisabledDuration);
```

3. Add to `getAll()`:

```
new CollisionMode("Disabled Duration (Quick Recovery)",
BY_DISABLED_DURATION)
```

4. Run the GUI and test your new mode

Bonus: What happens in ties? How would you handle equal disabled durations?

6.2 Challenge 2: Add Logging Observer

Goal: Create a new observer that logs events to console without modifying logic code.

Steps:

1. Create a new class that implements `PropertyChangeListener`
2. In the `propertyChange()` method, log events using pattern matching:

```
if (theEvent.getNewValue() instanceof final RoadRageEvent event) {
    switch (event) {
        case RoadRageEvent.VehiclesChanged e ->
            System.out.println("Vehicles updated at " + e.timestamp());
        // ... handle other events
```

```
        }
    }
```

3. Register your listener in `RoadRageMain` (you'll need to access the model)

Question: Did you have to change any logic code? Why not?

Summary

The Road Rage codebase demonstrates professional software engineering practices:

- **Strategy Pattern:** Swap algorithms without changing core logic
- **Factory Pattern:** Centralize creation, single source of truth
- **Observer Pattern:** Decouple components, react to changes
- **Sealed Types:** Type-safe events, exhaustive pattern matching

Key insight: These patterns make code **extensible** (easy to add features) and **maintainable** (changes stay localized).

As you work on Assignment 2, you're not just implementing vehicles—you're participating in a professionally-structured codebase. Your vehicle classes plug into this architecture seamlessly because the design patterns create clear contracts and boundaries.

Later in TCSS 305, you'll design systems like this yourself. For now, enjoy exploring how the pieces fit together!

Further Reading

External Resources

- Refactoring Guru: Design Patterns - Visual, example-rich pattern catalog
- Baeldung: Java Sealed Classes - Modern Java sealed types
- Observer Pattern Explained - Deep dive with examples

References

Primary Texts:

- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
- Strategy Pattern (pp. 315-323)
- Factory Method Pattern (pp. 107-116)
- Observer Pattern (pp. 293-303)
- Bloch, J. (2018). *Effective Java* (3rd ed.). Addison-Wesley.
- Item 1: Consider static factory methods instead of constructors

Language Documentation:

- Oracle JDK 25: Sealed Classes - Sealed classes and interfaces
- Oracle JDK 25: Pattern Matching - Pattern matching for switch

Design Patterns:

- Freeman, E., & Freeman, E. (2020). *Head First Design Patterns* (2nd ed.). O'Reilly Media.
- Strategy Pattern (Chapter 1)
- Observer Pattern (Chapter 2)

This guide is part of TCSS 305 Programming Practicum, School of Engineering and Technology, University of Washington Tacoma.