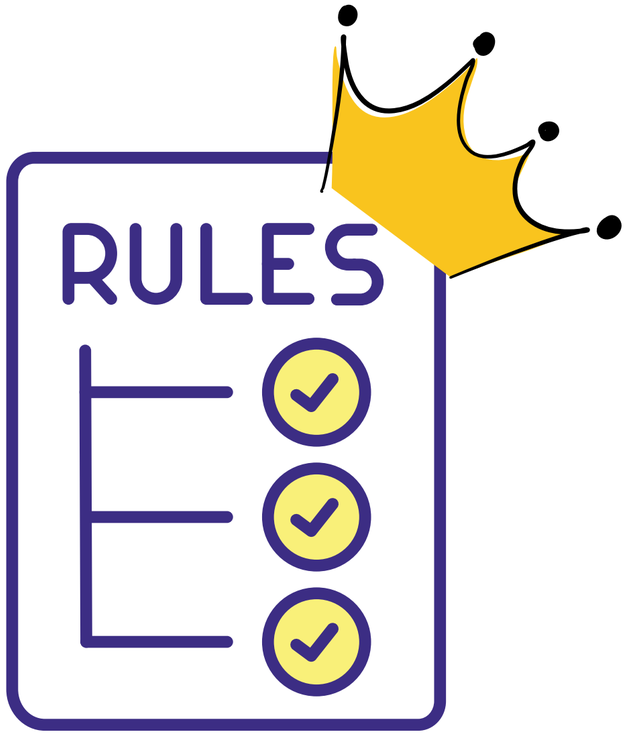
COMP 3550

9.4 — ADDING NEW FEATURES TO LEGACY SYSTEMS SAFELY

Week 9: Legacy Software, Architecture Recovery & Change

GOLDEN RULE — DO NO HARM

- Existing behavior may be relied on in hidden ways
- No/low tests means risk of silent breakage
- Large changes multiply complexity and debugging time



PRACTICAL STEPS

Add Test Coverage First

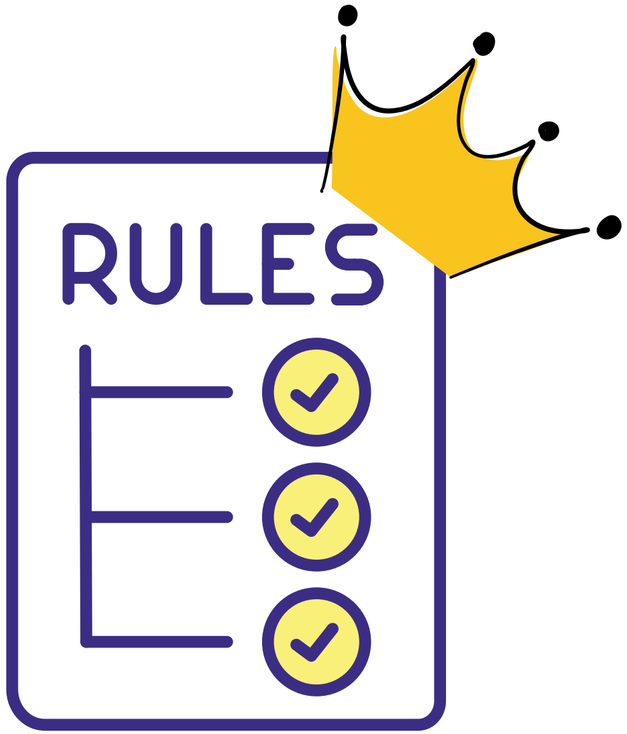
- Write characterization tests to lock in current behavior
- Even minimal coverage builds confidence
- Focus on the parts you must touch

Isolate New Code

- Prefer extension over modification (new methods, classes, modules)
- Keep old code paths intact unless you’re certain it’s safe to change

Commit Small, Review Often

- One small change → one commit → one review
- Makes it easy to spot and revert problems



FEATURE FLAGS & ISOLATION

- Add new features without risking the whole system
- Control rollout and testing in production-like environments
- Allow quick disable if problems appear
- Wrap new behavior in a toggle (config, env var, DB flag)

```
if (featureFlag.isEnabled("newCheckoutFlow")) {
    runNewFlow();
} else {
    runOldFlow();
}
```

Isolation via Adapters

- Keep new logic separate from old code paths
- Use an adapter to connect the old interface to new implementation
- Benefits:
- Test new code in isolation
- Swap implementations without deep rewrites

EXPAND BY INTERFACE OR EXTENSION

Wrap Existing Code (Adapter / Decorator)

- Create a new class that implements the same interface
- Delegate to the old code for unchanged behavior
- Inject new logic where needed
- Benefit: Keeps original tested paths intact

EXPAND BY INTERFACE OR EXTENSION

Wrap Existing Code (Adapter / Decorator)

- Create a new class that implements the same interface
- Delegate to the old code for unchanged behavior
- Inject new logic where needed
- Benefit: Keeps original tested paths intact

Extend via Interfaces

- Define an interface if one doesn’t exist
- Code to the interface, not the legacy class directly
- Allows swapping in new implementations with minimal disruption

EXPAND BY INTERFACE OR EXTENSION

Wrap Existing Code (Adapter / Decorator)

- Create a new class that implements the same interface
- Delegate to the old code for unchanged behavior
- Inject new logic where needed
- Benefit: Keeps original tested paths intact

Extend via Interfaces

- Define an interface if one doesn’t exist
- Code to the interface, not the legacy class directly
- Allows swapping in new implementations with minimal disruption

Composition > Inheritance

- Compose behavior by holding an instance of legacy code, not subclassing it
- Inheritance can force you to carry over unwanted behavior
- Composition gives you finer control over what’s reused

REFACTORING IN LEGACY

Refactor as you go, but only as much as needed to support your change.
Avoid “grand rewrites” in fragile systems.

Sprouting (Michael Feathers)

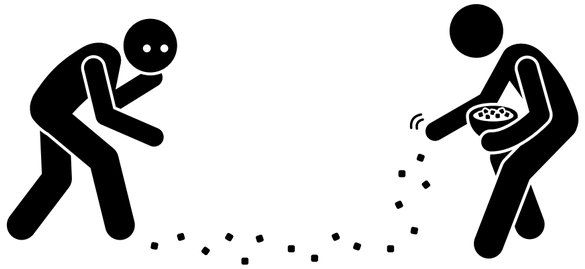
- Create new classes or methods rather than changing old ones directly
- Route calls from old code into the new structure
- Benefits:
  - Reduces risk to existing behavior
  - Lets you write tests for the new code independently

REFACTORING IN LEGACY

Refactor as you go, but only as much as needed to support your change.
Avoid “grand rewrites” in fragile systems.

Sprouting (Michael Feathers)

- Create new classes or methods rather than changing old ones directly
- Route calls from old code into the new structure
- Benefits:
  - Reduces risk to existing behavior
  - Lets you write tests for the new code independently



Leave Breadcrumbs

- Comments — explain oddities you discover
- TODOs — mark places for future cleanup
- Test Scaffolds — minimal tests that capture current behavior, even if imperfect
- Helps future maintainers (including you) pick up where you left off

CASE STUDY: REPLACING LEGACY CHECKOUT WITH THE STRANGLER FIG PATTERN

Scenario

- **Legacy**: LegacyCheckoutService—big, tangled class that calculates totals, discounts, taxes, and talks to the DB directly.
- **Goal**: Replace discount + tax logic first, then gradually the rest, without breaking production.

CASE STUDY: REPLACING LEGACY CHECKOUT WITH THE STRANGLER FIG PATTERN

Scenario

- **Legacy**: LegacyCheckoutService—big, tangled class that calculates totals, discounts, taxes, and talks to the DB directly.
- **Goal**: Replace discount + tax logic first, then gradually the rest, without breaking production.

Phase 0 — Characterize the Current Behavior (Golden Master)

- Lock in what “correct” means before touching code.

```
class LegacyCheckoutCharacterizationTest {
    @Test
    void goldenMaster_cart123() {
        LegacyCheckoutService svc = new LegacyCheckoutService(new LegacyDb());
        Money total = svc.calculateTotal("cart-123"); // uses real-ish fixtures
        // Persist or compare against a known-good snapshot
        assertEquals(new Money("47.35"), total);
    }
}
```

CASE STUDY: REPLACING LEGACY CHECKOUT WITH THE STRANGLER FIG PATTERN

Phase 1 — Introduce a Facade (the “Strangler” front door)

- Create a small interface in front of legacy. Everyone else calls the facade.

```
public interface Checkout {
    Money calculateTotal(String cartId);
}

public final class CheckoutStrangler implements Checkout {
    private final Checkout legacy; // adapter to legacy
    private final Checkout modern; // new implementation slice
    private final FeatureFlag flags;

    public CheckoutStrangler(Checkout legacy, Checkout modern, FeatureFlag flags) {
        this.legacy = legacy;
        this.modern = modern;
        this.flags = flags;
    }

    @Override
    public Money calculateTotal(String cartId) {
        if (flags.isEnabled("checkout_modern_total")) {
            return modern.calculateTotal(cartId);
        }
        return legacy.calculateTotal(cartId);
    }
}
```

CASE STUDY: REPLACING LEGACY CHECKOUT WITH THE STRANGLER FIG PATTERN

Phase 1 — Introduce a Facade (the “Strangler” front door)

- Adapter around the legacy class:

```
public final class LegacyCheckoutAdapter implements Checkout {
    private final LegacyCheckoutService legacy;

    public LegacyCheckoutAdapter(LegacyCheckoutService legacy) {
        this.legacy = legacy;
    }

    @Override
    public Money calculateTotal(String cartId) {
        return legacy.calculateTotal(cartId); // unchanged path
    }
}
```

CASE STUDY: REPLACING LEGACY CHECKOUT WITH THE STRANGLER FIG PATTERN

Phase 1 — Introduce a Facade (the “Strangler” front door)

- A trivial feature-flag:

```
public final class FeatureFlag {
    private final Set<String> enabled = ConcurrentHashMap.newKeySet();
    public boolean isEnabled(String name) { return enabled.contains(name); }
    public void enable(String name) { enabled.add(name); }
    public void disable(String name) { enabled.remove(name); }
}
```

CASE STUDY: REPLACING LEGACY CHECKOUT WITH THE STRANGLER FIG PATTERN

Phase 1 — Introduce a Facade (the “Strangler” front door)

- Wiring (Dependency Injection or manual):

```
FeatureFlag flags = new FeatureFlag();
Checkout checkout = new CheckoutStrangler(
    new LegacyCheckoutAdapter(new LegacyCheckoutService(new LegacyDb())),
    new ModernCheckout(new PricingService(), new TaxService(), new CartRepo()),
    flags
);
```

CASE STUDY: REPLACING LEGACY CHECKOUT WITH THE STRANGLER FIG PATTERN

Phase 2 — Implement the First “New” Slice (Composition, not Rewrite)

- Start with discounts + tax but reuse legacy data access via a thin repository.

```
public final class ModernCheckout implements Checkout {
    private final PricingService pricing;
    private final TaxService tax;
    private final CartRepo carts;

    public ModernCheckout(PricingService pricing, TaxService tax, CartRepo carts) {
        this.pricing = pricing; this.tax = tax; this.carts = carts;
    }

    @Override
    public Money calculateTotal(String cartId) {
        Cart cart = carts.load(cartId);                // isolates legacy DB shape
        Money subtotal = pricing.subtotal(cart.items());
        Money discounts = pricing.discounts(cart);
        Money taxable = subtotal.minus(discounts);
        Money taxAmount = tax.compute(cart.customerLocation(), taxable);
        return taxable.plus(taxAmount);
    }
}
```

CASE STUDY: REPLACING LEGACY CHECKOUT WITH THE STRANGLER FIG PATTERN

Phase 2 — Implement the First “New” Slice (Composition, not Rewrite)

- Repository shields you from legacy DB quirks:

```
public interface CartRepo { Cart load(String id); }

public final class LegacyCartRepo implements CartRepo {
    private final LegacyDb db;
    public LegacyCartRepo(LegacyDb db) { this.db = db; }

    @Override public Cart load(String id) {
        LegacyCartRow row = db.findCart(id); // weird joins, nullable fields, etc.
        return CartMapper.fromLegacy(row);   // normalize into clean domain model
    }
}
```

CASE STUDY: REPLACING LEGACY CHECKOUT WITH THE STRANGLER FIG PATTERN

Phase 3 — Compare Outputs; Ramp Up the Flag

- Keep both paths alive; prove equivalence (within tolerance) before switching traffic.

```
class StranglerShadowTest {
    @Test
    void modern_matches_legacy_for_sample_set() {
        FeatureFlag flags = new FeatureFlag(); // keep modern OFF for prod traffic
        Checkout strangler = wiring(flags);

        // Shadow compare without exposing to users
        List<String> sampleCarts = List.of("cart-123", "cart-456", "cart-789");
        for (String id : sampleCarts) {
            Money legacy = ((CheckoutStrangler) strangler).calculateWithOverride(id,
/*forceLegacy=*/true);
            Money modern = ((CheckoutStrangler) strangler).calculateWithOverride(id,
/*forceLegacy=*/false);
            assertTrue(legacy.closeTo(modern, new Money("0.01")),
                () -> "Mismatch for " + id + ": legacy=" + legacy + " modern=" + modern);
        }
    }
}
```

CASE STUDY: REPLACING LEGACY CHECKOUT WITH THE STRANGLER FIG PATTERN

Phase 3 — Compare Outputs; Ramp Up the Flag

Rollout plan:

- Enable flag in dev → staging.
- In prod, start with 1% of traffic (or internal users).
- Monitor metrics/logs; increase to 10%, 50%, 100%.
- When stable, leave the flag ON permanently for a while; then delete legacy path.

CASE STUDY: REPLACING LEGACY CHECKOUT WITH THE STRANGLER FIG PATTERN

Phase 4 — Extend the Strangler to More Endpoints

- Add more methods to Checkout or create additional strangler facades (payments, shipping quotes, etc.).
- Each time:
  - Add an interface or expand it carefully.
  - Route via the strangler.
  - Build new in isolation (new classes/services).
  - Compare behaviors, then flip.

CASE STUDY: REPLACING LEGACY CHECKOUT WITH THE STRANGLER FIG PATTERN

Phase 5 — Remove the Old Vines

- When a slice is fully migrated:
  - Delete the legacy method(s) and their adapters.
  - Collapse the feature flag (set default ON, then remove code).
  - Remove dead tables/columns last (after a safe window).

PROJECT PAUSE & REFLECT

Looking for a small challenge?

- Go to GitHub or even just a fellow group’s project
- Pick a small feature and try implementing it *without modifying any existing method bodies*