COMP 3550

9.3 — RECONSTRUCTING ARCHITECTURE FROM CODE

Week 9: Legacy Software, Architecture Recovery & Change

ARCHITECTURE IS STILL THERE

Even if there’s no documentation, code has structure! It’s just buried in the implementation.

- Architecture influences how code changes ripple through the system
- Recognizing patterns helps you navigate faster
- You can recover missing design intent from the code itself

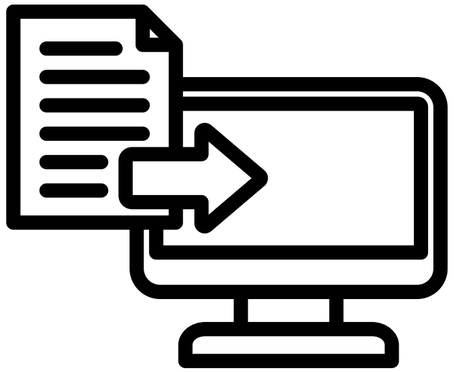
Typical Layered Pattern (as we have seen before)
UI → Controller → Service → Repository

- UI — handles user input/output
- Controller — coordinates requests
- Service — contains business logic
- Repository — reads/writes from data source

START FROM ENTRY POINTS

Why Entry Points?

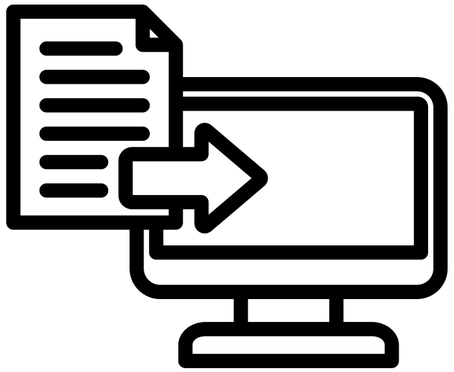
- They show how the system is first invoked
- Great starting points for tracing architecture and flow
- Let you quickly map “where things begin” in the code



START FROM ENTRY POINTS

Why Entry Points?

- They show how the system is first invoked
- Great starting points for tracing architecture and flow
- Let you quickly map “where things begin” in the code



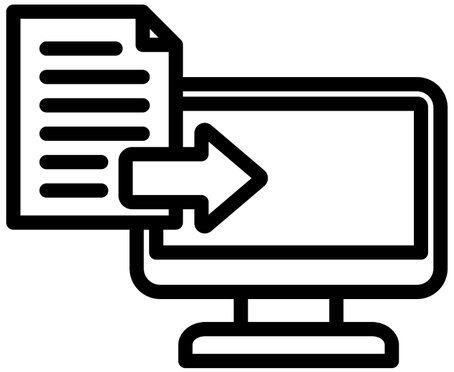
Common Entry Points

- Main class / main() method — console or desktop apps
- Routes / Controllers — web apps (e.g., /login, /dashboard)
- UI Action Listeners — buttons, menus, form submissions in GUI apps

START FROM ENTRY POINTS

Why Entry Points?

- They show how the system is first invoked
- Great starting points for tracing architecture and flow
- Let you quickly map “where things begin” in the code



Common Entry Points

- Main class / main() method — console or desktop apps
- Routes / Controllers — web apps (e.g., /login, /dashboard)
- UI Action Listeners — buttons, menus, form submissions in GUI apps

Look for Layers

- Follow method calls to Controller → Service → Repository
- Use package structure to guess layer boundaries
  - e.g., ui/, controllers/, services/, dao/

SKETCH KEY COMPONENTS

The “Buckets” Approach

Group code into 3 broad areas:

1. UI
    a. anything handling user interaction or presentation
2. Logic
    a. business rules, calculations, workflows
3. Data
    a. persistence, APIs, file I/O

Draw the Connections

- Use directional arrows to show how data and commands flow
- Typical pattern: UI → Logic → Data
- Note any weird reverse flows (e.g., DB triggers calling back into logic)

VALIDATE WITH CODE TRACING

What to Do

1. Pick a User Task
    - e.g., Login → Dashboard load → Recent items query
2. Follow It Through the System
    - Set breakpoints at UI → Controller → Service → Repository boundaries
    - Or add temporary logs at method entry/exit with key params/returns
    - Note external calls (DB, API, filesystem) and side effects (caches, events)
3. Compare Against Your Sketch
    - Does data flow UI → Logic → Data as drawn?
    - Are there unexpected hops (helpers, legacy utilities, cross-calls)?
    - Any missing components (queues, schedulers, interceptors, middleware)?

VALIDATE WITH CODE TRACING

What are we trying to capture?

- Entry/exit points for each layer
- Data shapes (IDs, DTOs, payload sizes)
- Timing (slow steps) and errors/warnings surfaced in logs

VALIDATE WITH CODE TRACING

What are we trying to capture?

- Entry/exit points for each layer
- Data shapes (IDs, DTOs, payload sizes)
- Timing (slow steps) and errors/warnings surfaced in logs

Outcomes

- **Confirm**: Mark paths that match your diagram
- **Mismatch**: Update arrows/components to reflect reality
- **Unknowns**: Create a short list to investigate next

COMMON WILD ARCHITECTURES

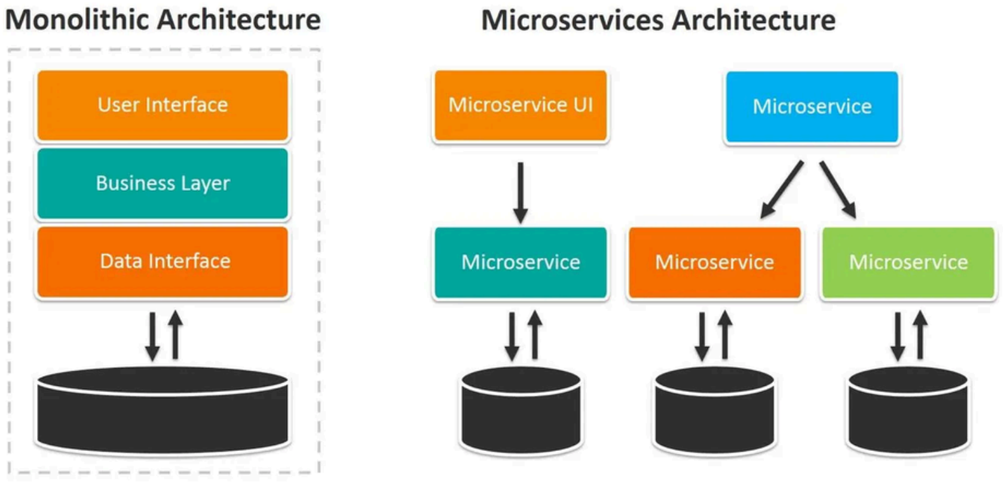
Monoliths

- **What it is:** One deployable with everything inside
- **Symptoms**: Huge codebase, shared DB, cross-cutting “utility” modules
- **Why it happens**: Grew fast, never split
- **Risk**: Change ripple effects; slow builds/deploys
- **How to work with it**: Identify seams; add tests at module boundaries; extract one feature at a time

COMMON WILD ARCHITECTURES

Monoliths

- **What it is:** One deployable with everything inside
- **Symptoms**: Huge codebase, shared DB, cross-cutting “utility” modules
- **Why it happens**: Grew fast, never split
- **Risk**: Change ripple effects; slow builds/deploys
- **How to work with it**: Identify seams; add tests at module boundaries; extract one feature at a time

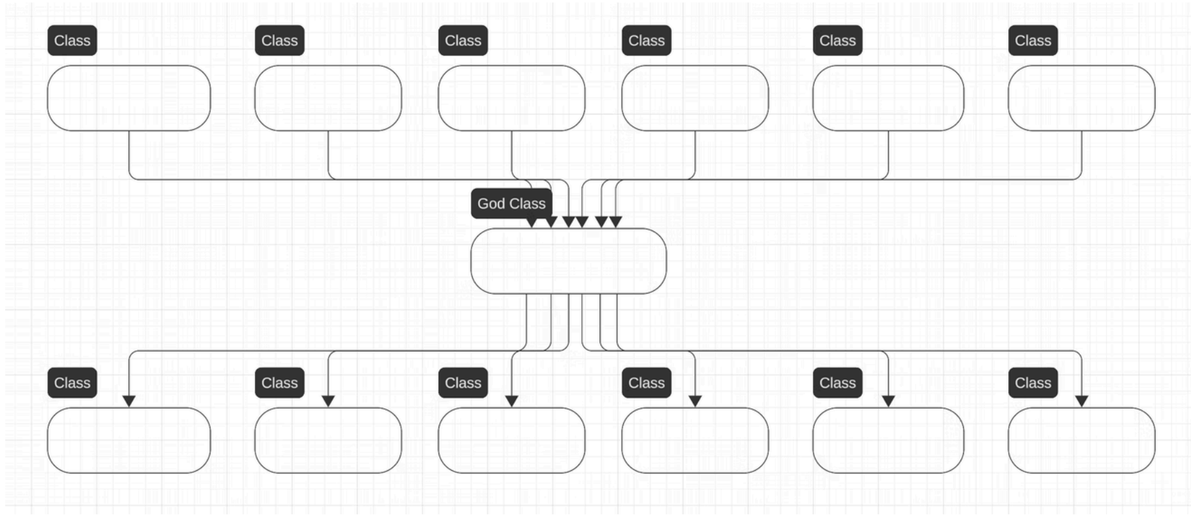


Yes, we have been working with a **monolithic n-tier architecture**

COMMON WILD ARCHITECTURES

God Classes

- **What it is**: One class knows/does too much
- **Symptoms**: Thousands of lines, many responsibilities, long parameter lists
- **Why it happens**: Convenience, “just add it here” drift
- **Risk**: Fragile; any change risks everything
- **How to work with it**: Strangle method-by-method; extract cohesive helpers; introduce interfaces around hotspots



COMMON WILD ARCHITECTURES

“Accidental” MVC

- **What it is:** Looks like MVC, but responsibilities bleed across layers
- **Symptoms:** Controllers with business logic; views hitting DB; services as pass-throughs
- **Why it happens:** Deadline pressure; unclear boundaries
- **Risk:** Hard to test; unpredictable side effects
- **How to work with it:** Push logic into services; keep controllers thin; enforce DTOs between layers

Different versions of MVC exist these days, the point is to plan and design with intention.

PROJECT PAUSE & REFLECT

Draw your app’s current architecture without looking at your old diagram.
How close is it to what your diagram looks like? Should you be updating your diagram?

https://github.com/topics/python

Head over to the python repos (even if you don’t know python) and see if you can draw one of the project’s architectures based on what you’ve learned in the last few videos.