Software Design

Dr Sean McGrath

Software Design

- Software design links requirements to coding and debugging.
- It’s crucial in both small and large projects.
- The design process helps manage complexity, ensures structure, and balances trade-offs.

Inherent Challenges

- Wicked Problems: Design problems are often not fully understood until partially solved (e.g., Tacoma Narrows Bridge example).
- Sloppy Process: Design involves trial and error, leading to a tidy result through mistakes.
- Nondeterministic: Multiple valid solutions can exist for a design problem.

Tradeoffs

- Designers must balance multiple objectives like performance, simplicity, and extensibility.
- Different designs will suit different priorities based on system requirements.
- Restricting possibilities helps simplify the design and avoid overcomplication.

Emergent Practice

- Good design doesn’t happen instantly; it evolves.
- Reviews, discussions, and real-world testing help shape the design.
- Flexibility to adapt and improve is a key aspect of emergent design.

Emergent Practice

- Managing **complexity** is the primary technical imperative.
- **Essential** complexity stems from real-world intricacies, while **accidental** complexity arises from poor design.
- Breaking a problem into manageable parts helps reduce complexity.

Design Heuristics

- Minimal Complexity: Simplicity should be prioritized to reduce mental load.
- Ease of Maintenance: Design for the future, thinking about how someone else will maintain your code.
- Loose Coupling: Keep interdependencies between components minimal.

Summary of a Typical Software Application Architecture

- **Presentation** Layer: User interface.
- **Business** Logic Layer: Core application logic.
- **Data** Access Layer: Interacts with databases.
- **Database** Layer: Stores persistent data.

- (optionally)

Types of architecture

- Monolithic Architecture: The entire application is built as a single unit, where the presentation, business logic, and data access layers are all tightly coupled in one codebase. This is simpler to develop initially but can be harder to scale and maintain as it grows.
- Microservices Architecture: The application is

Your experience

- Likely to work either on microservice architecture in professional setting or
- 3 Tier architecture for the applications you build here e.g:
- UI(html/react), Node.JS, Database(SQL)

Question

What does good software look like?

- To the user?
- To the developers?
- To the designers?

Good Software

- **Reusability**: Code should be reusable in other systems.
- **Extensibility**: Changes in one part of the system should not affect other parts.
- **High Fan-In**: Maximise the number of classes that use utility classes.
- **Portability**: Design should be transferable to other platforms.

Architecture

- System Level: Organising the system into subsystems or component parts.
- Class Level: Dividing subsystems into classes.
- Routine Level: Designing individual methods and routines.

Core tennets

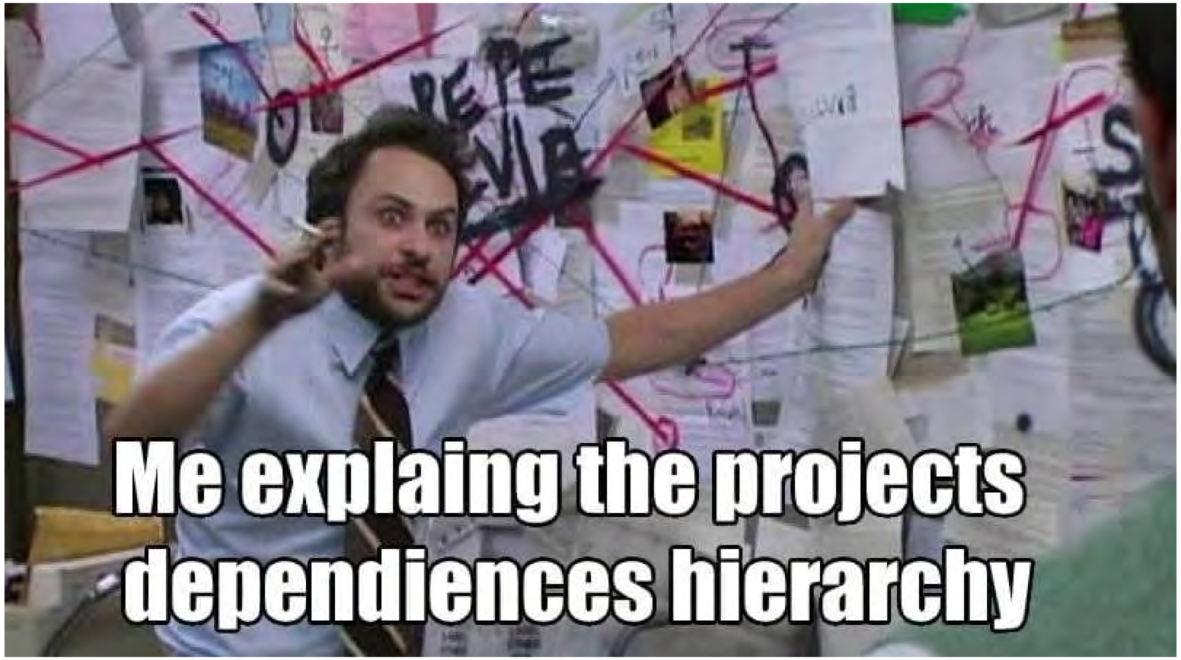
- **Coupling**: Refers to the interdependencies between modules. Low coupling is desirable for flexibility and easier maintenance.
- **Cohesion**: The degree to which elements within a module belong together. High cohesion means a module performs a single task or related tasks.

Pointers

- Good design is about balancing trade-offs and managing complexity.
- Design must evolve over time, adapting to new information.
- Keeping complexity under control is key to developing maintainable, efficient systems.

Challenges

- Let’s look at some problematic code and identify optimisation strategies.



Challenges

```
// High coupling, Low Cohesion: Fetching and processing data in the same function
function fetchDataAndProcess(url) {
  fetch(url)
    .then(response => response.json())
    .then(data => {
      // Process data inside the same function
      data.forEach(item => console.log(item.value * 2));
    })
    .catch(error => console.error('Error:', error));
```

Challenges

- Problematic to maintain and reuse properly.
- It does two jobs:
  - Fetching data from a URL.
  - Processing the fetched data by logging it to the console.

Why it is bad

- The function does *more than one* thing, making it harder to reuse in other contexts.
- For instance, if you need to fetch data without processing it or want to process data fetched by a different method, you would have to rewrite the function or extract its logic.
- It is difficult to test each responsibility separately. If you want to test data fetching, you’re forced to

Why it is bad

- Difficult to test: Since **fetchDataAndProcess** performs both fetching and processing, it’s harder to isolate and test each part independently.
- Unit testing requires testing both tasks together, potentially increasing the complexity of the test cases.
- Harder to maintain: if the fetch or process logic needs to change, it would be more cumbersome

Why it is bad

- The function does *more than one* thing, making it harder to reuse in other contexts.
- For instance, if you need to fetch data without processing it or want to process data fetched by a different method, you would have to rewrite the function or extract its logic.
- It is difficult to test each responsibility separately. If you want to test data fetching, you’re forced to

```
High Cohesion: Function does one thing -
tching data

nction fetchData(url) {

eturn fetch(url)

 .then(response => response.json())

 .catch(error => console.error('Error:',
ror));
```

```
// Low Coupling: Independent module for
processing data

function processData(data) {

  return data.map(item => item.value *
2); // Process data without external
dependencies

}
```

```
// one function to rule them all...

function main(url) {

  fetchData(url).then(data => {

    const result = processData(data);

    console.log(result);

  });

}
```

NOICE
imgflip.com

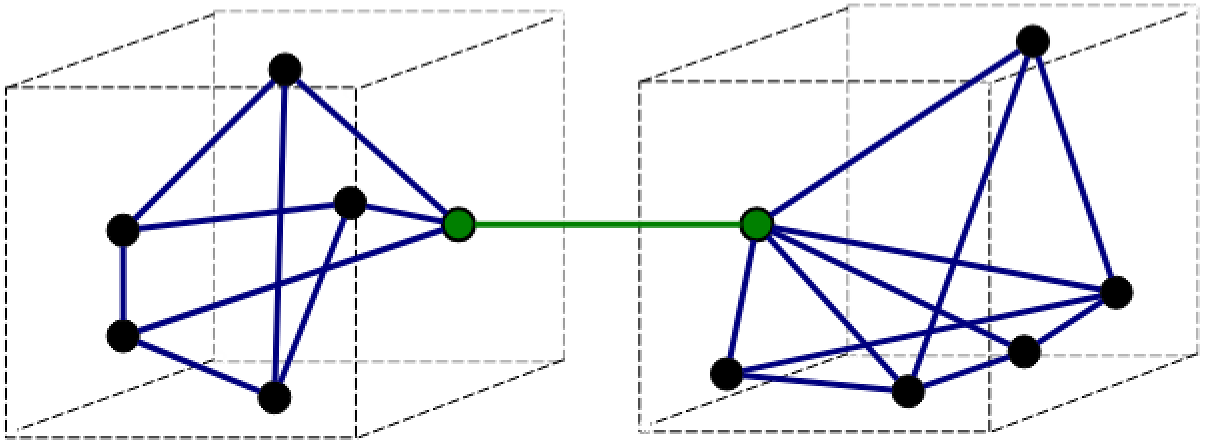
Example 2

```
# Low Cohesion: Fetching and processing data in the same function

def fetch_and_process_data(url):

    import requests

    response = requests.get(url)

    data = response.json()
```

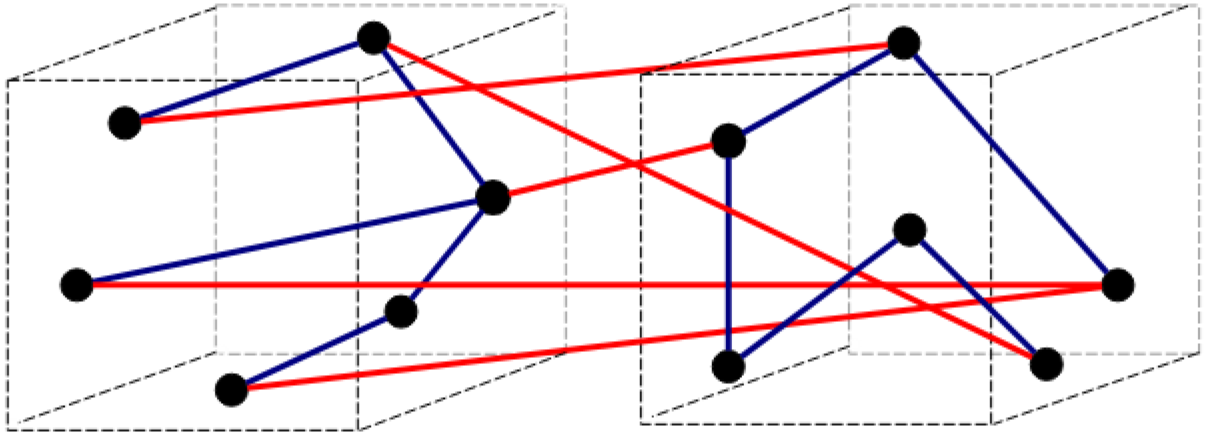
(fixed) Example 2

```
# High Cohesion: Function does one thing - fetching data

def fetch_data(url):

    import requests

    response = requests.get(url)

    return response.json()
```

```
# Main function orchestrating both

def main(url):

    data = fetch_data(url)

    result = process_data(data)

    print(result)
```



a) Good (loose coupling, high cohesion)



b) Bad (high coupling, low cohesion)

THE OFFSPRING ALREADY KNEW IN '94
YOU GOTTA KEEP 'EM SEPARATED

Coupling

```
function fetchUserData() {
  return { id: 1, name: "John" };
}

function getUserDetails() {
  const user = fetchUserData();
/* Directly dependent on fetchUserData
 Changes have to be performed to both
in order to refactor
*/
  console.log(user.name);
}

getUserDetails();
```

```
function fetchUserData() {
  return { id: 1, name: "John" };
}

function getUserDetails(fetchFn) {
  const user = fetchFn();
// Inject dependency
  console.log(user.name);
}

getUserDetails(fetchUserData);
```

Low cohesion vs high cohesion

```
function fetchAndProcessUserData(userId) {
  // Fetch user data
  const user = { id: userId, name: "John" };

  // Process user data
  console.log(user.name + " (" + user.id + ")");

  // Unrelated task: Logging extra info
  console.log("Logging extra info: Operation
complete.");
}

fetchAndProcessUserData(1);
```

```
function fetchUserData(userId) {
  return { id: userId, name: "John" };
// Only fetch user data
}

function processUserData(user) {
  console.log(user.name + " (" + user.id + ")");
// Only process user data
}

function logCompletion() {
  console.log("Logging extra info: Operation
complete.");  // Only log info
}

const user = fetchUserData(1);
processUserData(user);
```

Lessons

- Hardcoding dependencies is bad. Parameters being passed between functions creates better flow of information. Swap out implementations does not require new code!

SOLID principles

- Single Responsibility Principle (SRP)
- Definition: A class or function should have only one reason to change, meaning it should have only one responsibility.
- Why: It promotes high cohesion, making the code easier to maintain, test, and understand.
- Example: A function should either fetch data or process data, but not do both.

SOLID principles

- Open/Closed Principle (OCP)
- Definition: Software entities (classes, modules, functions) should be open for extension, but closed for modification.
- Why: It allows you to add new features without modifying existing code, reducing the risk of introducing bugs in previously working code.
- Example: Instead of modifying a function, you can

SOLID principles

- Liskov Substitution Principle (LSP)
- Definition: Subclasses should be able to replace their parent classes without affecting the behaviour of the system.
- Why: Ensures that derived classes extend base classes properly, without altering expected functionality.
- Example: If you have a Bird class and a Penguin

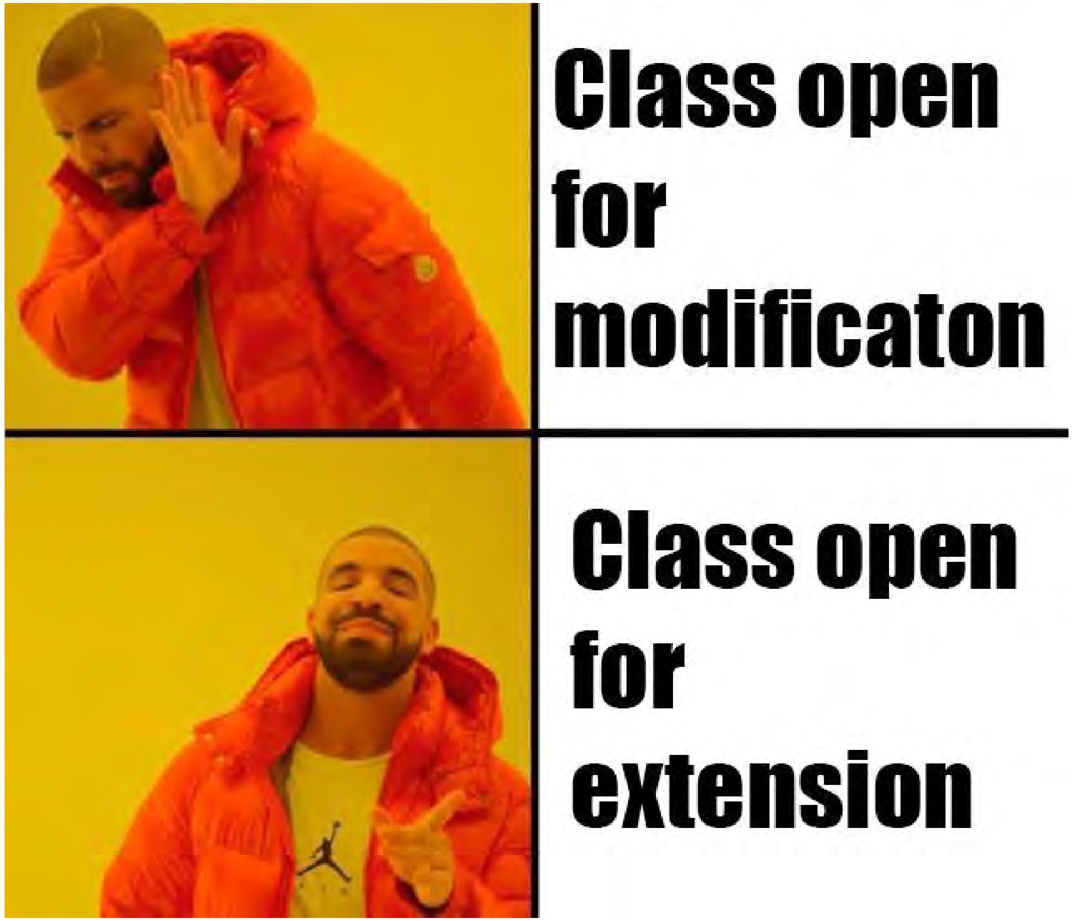
SOLID principles

- Interface Segregation Principle (ISP)
- Definition: Clients should not be forced to depend on interfaces they don't use. Instead of one large interface, break it into smaller, more specific interfaces.
- Why: Promotes high cohesion by ensuring that clients only need to know about the methods they use, and reduces the burden of implementing

SOLID principles

- Dependency Inversion Principle (DIP)
- Definition: High-level modules should not depend on low-level modules. Both should depend on abstractions (e.g., interfaces or abstract classes). Also, abstractions should not depend on details; details should depend on abstractions.
- Why: This reduces the coupling between high-level and low-level modules, making the system

SOLID principles



DRY (Don’t repeat yourself)

- What is the DRY Principle?
- Definition: The DRY principle states that every piece of knowledge or logic must have a single, unambiguous representation in the system.
- Goal: Avoid repetition in code, data, logic, and documentation to reduce redundancy and make maintenance easier.
- Why It Matters:

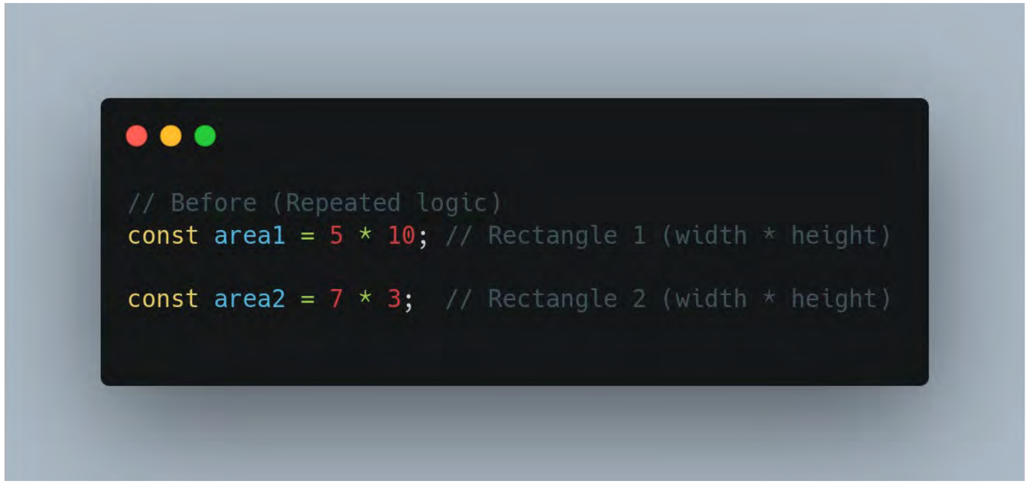
Why DRY?

- Easier Maintenance:
- Changing one instance of a repeated logic block affects all occurrences, so less chance for bugs.
- Code Readability:
  - More concise and understandable code because there’s no clutter of repeated logic.
  - Reduced Complexity:
  - Code remains simpler as you centralise

DRY in Practice

- Refactor Repeated Logic:
  - Identify patterns of repeated code and extract them into functions, classes, or modules.
    - Example: Instead of having validation logic in multiple places, create a single validateUser() function.
- Use Variables and Constants:
  - Avoid using hard-coded values in multiple

```
// Before (Repeated logic)

const area1 = 5 * 10; // Rectangle 1 (width * height)


const area2 = 7 * 3;  // Rectangle 2 (width * height)
```



```
// Before (Repeated logic)
const area1 = 5 * 10; // Rectangle 1 (width * height)

const area2 = 7 * 3;  // Rectangle 2 (width * height)
```

```
// After (Refactored)

function calculateArea(width, height) {

  return width * height;

}



const area1 = calculateArea(5, 10);

const area2 = calculateArea(7, 3);
```

```
// Before (Hardcoded in multiple places)

const priceWithTax1 = 100 * 1.2;

const priceWithTax2 = 200 * 1.2;
```

```
// After (Using a constant)

const TAX_RATE = 0.20;

const priceWithTax1 = 100 * (1 + TAX_RATE);

const priceWithTax2 = 200 * (1 + TAX_RATE);
```

```
// Before (Repeated formatting logic)

const price1 = '$' + 100;

const price2 = '$' + 200;
```

```
// After (Refactored)

function formatPrice(amount) {

  return '$' + amount;

}


const price1 = formatPrice(100);

const price2 = formatPrice(200);
```

Good coding practice

- Often requires you to think about solutions to problems.
- Whiteboarding, flow charts and UML can help with this process.
- Do not just jump in and start writing code. Even simple problems have elegant craft opportunities for solutions that are easily manageable and less error prone.

Important lesson

- I might have to work on your codebase one day.
- You might be responsible for writing the code that powers the self-driving car my children are in.
- I have a vested interest in you getting this right!