Designing Classes in Real Projects

Introduction

Learning classes is easy when the examples look like this:

```
class User:
    pass
```

Real projects are different.

A real application may contain:

- users
- resources
- payments
- notifications
- files
- databases
- API clients
- services
- reports
- authentication
- configuration

The difficult part is no longer:

> “How do I create a class?”

The difficult part becomes:

> **"What classes should I create, what should each class be responsible for, and how should those classes work together?"**

This is called **class design**.

Good class design helps a project remain:

- understandable
- maintainable
- testable
- reusable
- extendable

The goal is not to create as many classes as possible.

The goal is to create **useful boundaries around responsibilities**.

---

1. Classes in Real Projects

In a small program, you might write everything in one file:

`app.py`

As the project grows, this becomes difficult to manage.

A real project might look like:

```
project/
□
□□□ models/
□   □□□ user.py
□   □□□ resource.py
□
□□□ services/
□   □□□ resource_service.py
□   □□□ notification_service.py
□
□□□ repositories/
□   □□□ resource_repository.py
□
□□□ clients/
□   □□□ email_client.py
□
□□□ main.py
```

Each class has a particular role.

The important question is:

> **Why does this class exist?**

---

2. Start With the Problem, Not the Classes

A common beginner mistake is starting with:

“I need 5 classes.”

Instead, start with the problem.

For example:

> Build a system that allows users to browse, search, and save learning resources.

Before writing code, identify:

- What data exists?
- What actions can users perform?
- What rules exist?
- What external systems are involved?
- What parts may change later?

From that, classes begin to emerge naturally.

---

3. Identify the Main Objects

For a learning platform, you might identify:

```
User
Resource
Category
Quiz
```

These are possible **domain objects**.

For example:

```
class User:
    ...

class Resource:
    ...
```

The important word is **possible**.

Not every noun needs to become a class.

---

4. Not Every Noun Should Become a Class

Suppose the requirements say:

> A user can save a resource.

You might identify:

```
User
Resource
```

But you do not necessarily need:

```
SavedResourceManager
SavedResourceObject
UserResourceConnection
```

for such a simple relationship.

Avoid creating classes simply because you see nouns in the requirements.

Ask:

> Does this concept have meaningful state, behavior, or responsibility?

If not, another structure may be better.

---

5. Identify Responsibilities

After identifying possible objects, ask:

> What should this class be responsible for?

For example:

`User`

might be responsible for:

- identity
- profile information
- user preferences

While:

`Resource`

might be responsible for:

- title
- category
- difficulty
- resource metadata

But should `User` also:

- send emails?
- connect to the database?
- generate PDFs?
- call external APIs?

Usually, no.

Those responsibilities belong elsewhere.

---

6. Single Responsibility

One of the most important principles in class design is:

> **A class should have one clear responsibility.**

This does not mean:

> “A class can only have one method.”

It means:

> The class should have one coherent reason to change.

For example:

```
class Resource:
    ...
```

could manage resource data.

But this is problematic:

```
class Resource:
    def save_to_database(self):
        ...

    def send_email(self):
        ...

    def generate_pdf(self):
        ...

    def upload_to_cloud(self):
        ...
```

Now one class is responsible for many unrelated things.

---

7. Separate Responsibilities

Instead, responsibilities can be separated:

Resource

Stores resource information

ResourceRepository

Handles persistence

EmailService

Sends emails

PDFGenerator

Generates PDFs

CloudStorage

Handles cloud files

Now each component has a clearer purpose.

---

8. Domain Classes

A **domain class** represents something meaningful in the application's problem domain.

For a learning platform:

class Resource:

```
    ...
```

For an e-commerce application:

```
class Product:
    ...
```

For a banking application:

```
class Account:
    ...
```

For a project management application:

```
class Task:
    ...
```

These classes represent the concepts the application works with.

---

9. Data vs Behavior

A useful design question is:

> What data belongs to this object, and what behavior naturally belongs to it?

For example:

```
class BankAccount:
    def __init__(self, balance):
```

```
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount
```

`deposit()` naturally belongs to the account because it changes account state.

Compare this:

```
class BankAccount:
    def send_marketing_email(self):
        ...
```

That behavior does not naturally belong to the account.

It belongs somewhere else.

---

10. Put Behavior Near the Data It Operates On

Consider:

```
class Resource:
    def __init__(self, title):
        self.title = title
```

Suppose a resource needs to generate a URL slug.

This can naturally belong to the resource:

```
class Resource:
    def __init__(self, title):
```

```
        self.title = title

    def slug(self):
        return self.title.lower().replace(" ", "-")
```

The behavior operates directly on resource data.

This often produces more understandable designs.

---

11. Avoid God Classes

A **God class** is a class that tries to do almost everything.

For example:

```
class Application:
    def create_user(self):
        ...

    def login_user(self):
        ...

    def send_email(self):
        ...

    def save_database(self):
        ...

    def generate_pdf(self):
        ...
```

```
    def process_payment(self):
        ...

    def upload_file(self):
        ...

    def create_resource(self):
        ...
```

This class becomes difficult to:

- understand
- test
- modify
- reuse

A growing class is often a signal that responsibilities need to be separated.

---

12. Use Composition to Build Larger Systems

Real applications often work better when smaller classes cooperate.

For example:

```
ResourceService
    |
    +□□ ResourceRepository
    |
    +□□ NotificationService
    |
    +□□ SearchService
```

The service does not need to implement every detail itself.

It coordinates other components.

This is **composition**.

---

13. Composition in a Real Project

Suppose we have:

```
class ResourceRepository:
    def save(self, resource):
        print("Saving resource")


class NotificationService:
    def notify(self, message):
        print(message)
```

Then:

```
class ResourceService:

    def __init__(self, repository, notification_service):
        self.repository = repository
        self.notification_service = notification_service

    def create_resource(self, resource):
        self.repository.save(resource)
        self.notification_service.notify(
            "Resource created"
```

```
    )
```

Now `ResourceService` coordinates the workflow.

It does not need to know how the repository saves data.

---

14. Dependency Injection

Notice:

```
def __init__(self, repository, notification_service):
```

The dependencies are provided from outside.

This is called **dependency injection**.

Instead of:

```
class ResourceService:

    def __init__(self):
        self.repository = ResourceRepository()
        self.notification_service = NotificationService()
```

we can provide them:

```
service = ResourceService(
    repository,
    notification_service
)
```

This makes the class easier to test and change.

---

15. Why Dependency Injection Matters

Suppose production uses:

`DatabaseRepository`

but tests need:

`FakeRepository`

With dependency injection:

```
service = ResourceService(
    FakeRepository(),
    FakeNotificationService()
)
```

The service does not need to change.

Only its dependencies change.

This is one of the most useful patterns in real applications.

---

16. Program to an Abstraction

Sometimes a service should depend on a contract rather than a specific implementation.

For example:

```
from abc import ABC, abstractmethod


class ResourceRepository(ABC):

    @abstractmethod
    def save(self, resource):
        pass
```

Implementations:

```
class DatabaseResourceRepository(ResourceRepository):

    def save(self, resource):
        print("Saving to database")

class FileResourceRepository(ResourceRepository):

    def save(self, resource):
        print("Saving to file")
```

The service can depend on the abstraction.

```
class ResourceService:

    def __init__(self, repository: ResourceRepository):
        self.repository = repository
```

Now the implementation can change without changing the service's core logic.

---

17. Model, Service, Repository

A common organization in applications is:

```
Model
Service
Repository
```

For example:

```
Resource
  
ResourceService
  
ResourceRepository
```

Their responsibilities differ.

Model

Represents the application's data/domain.

```
class Resource:
  ...
```

Service

Contains application or business workflow.

```
class ResourceService:
  ...
```

Repository

Handles persistence or retrieval.

```
class ResourceRepository:
    ...
```

This separation can prevent one class from becoming responsible for everything.

---

18. Example Architecture

Imagine a resource creation operation:

```
User request
    
ResourceService
    
Validate resource
    
ResourceRepository
    
Database
    
NotificationService
    
User notification
```

Each component has a focused responsibility.

---

19. Keep Business Rules Out of Infrastructure Classes

Suppose:

```
class DatabaseRepository:
    def save(self, resource):
        ...
```

It should primarily deal with persistence.

Avoid putting business rules such as:

```
if resource.difficulty == "Beginner":
    ...
```

inside the database repository unless that logic is specifically related to persistence.

Business decisions generally belong in domain/service logic.

---

20. Keep Infrastructure Details Out of Domain Models

Similarly, avoid turning a domain model into a database controller.

Problematic:

```
class Resource:

    def save(self):
        database.connect()
        database.insert(...)
```

Now `Resource` knows about database infrastructure.

A cleaner design:

```
Resource
   
ResourceRepository
   
Database
```

The domain object remains focused on representing the resource.

---

21. Designing Class Relationships

Before implementing classes, draw their relationships.

For example:

```
User
 |
 | saves
 
Resource
```

Another example:

```
Course
  |
  | contains
  
Resource
```

Another:

```
ResourceService
    |
    +□□ ResourceRepository
    |
    +□□ NotificationService
```

This simple diagram can reveal design problems before code is written.

---

22. IS-A vs HAS-A

A useful question is:

> Is this an IS-A relationship or a HAS-A relationship?

IS-A

```
PDF is a Resource
```

This may suggest inheritance.

```
class PDF(Resource):
    ...
```

HAS-A

```
Course has Resources
```

This suggests composition.

```
class Course:
```

```
    def __init__(self, resources):
        self.resources = resources
```

Choosing the correct relationship is important.

---

23. Prefer Composition When Appropriate

Suppose:

```
Car
Engine
```

A car **has an** engine.

So:

```
class Car:

    def __init__(self, engine):
        self.engine = engine
```

is generally more appropriate than:

```
class Car(Engine):
    ...
```

Inheritance should represent a meaningful type relationship.

Composition represents collaboration or ownership.

---

24. Avoid Deep Inheritance Trees

A design such as:

```
A
 □
B
 □
C
 □
D
 □
E
```

can become difficult to understand.

Changes in a parent can affect many children.

Often a simpler structure is:

```
Common abstraction
      |
 □□□□□□□□□□
 A   B   C
```

or composition:

```
Service
  |
 +□□ Component A
 +□□ Component B
```

Use inheritance when the relationship genuinely benefits from it.

---

25. Keep Classes Small Enough to Understand

There is no universal number of lines that makes a class “too large.”

Instead ask:

- Can I describe its responsibility in one clear sentence?
- Does it have unrelated methods?
- Does it know too much about other components?
- Is it difficult to test?
- Does changing one feature require modifying many unrelated methods?

If several answers are yes, reconsider the design.

---

26. Minimize Coupling

Coupling describes how strongly components depend on each other.

High coupling:

```
Class A
  
Class B
  
Class C
  
Class D
  
Database
```

If changing the database requires changing many classes, the system is tightly coupled.

Lower coupling can be achieved through:

- clear interfaces
- dependency injection
- composition
- abstractions
- focused responsibilities

---

27. Aim for High Cohesion

Cohesion describes how closely related the responsibilities inside a class are.

High cohesion:

```
EmailService
  □
email-related operations
```

Low cohesion:

```
UtilityManager
  □
send email
parse CSV
calculate tax
resize image
create user
send notification
```

A well-designed class generally has related responsibilities.

---

28. Class APIs Should Be Simple

A class should expose what other parts of the application actually need.

For example:

```
class Resource:

    def publish(self):
        ...
```

Other components should ideally call:

```
resource.publish()
```

rather than modifying many internal details:

```
resource.status = "published"
resource.published_at = ...
resource.validate()
resource.update_index()
```

A good class can hide internal complexity behind a clear API.

---

29. Encapsulation in Real Projects

Encapsulation is not only about private attributes.

It is about controlling how state changes.

For example:

```
class Resource:

    def publish(self):
        if not self.title:
            raise ValueError("Title required")

        self.status = "published"
```

Instead of allowing every part of the application to directly manipulate:

```
resource.status
```

the class can control the transition.

This helps protect business rules.

---

30. Design Around Use Cases

A practical way to design classes is to start from user actions.

Suppose the requirement is:

> An admin creates a new resource.

Break it down:

```
Create resource
    ↓
Validate input
    ↓
Create Resource object
    ↓
Save resource
    ↓
Notify relevant users
```

Possible classes:

```
Resource
ResourceService
ResourceRepository
NotificationService
```

The classes emerge from the workflow.

---

31. Example: Creating a Resource

Model

```
class Resource:

    def __init__(self, title, category):
        self.title = title
        self.category = category
```

Repository

```
class ResourceRepository:

    def save(self, resource):
        print(f"Saving {resource.title}")
```

Notification

```
class NotificationService:

    def notify(self, message):
        print(message)
```

Service

```
class ResourceService:

    def __init__(self, repository, notification_service):
        self.repository = repository
        self.notification_service = notification_service

    def create(self, title, category):
        resource = Resource(title, category)

        self.repository.save(resource)

        self.notification_service.notify(
            "Resource created successfully"
        )

        return resource
```

The responsibilities are separated.

---

32. What the Service Should Not Do

Avoid:

```
class ResourceService:

    def create(self, title, category):
        resource = Resource(title, category)

        database.connect()
        database.execute(...)
        database.commit()

        smtp.connect()
        smtp.send(...)

        return resource
```

Now the service knows:

- database implementation
- SQL details
- email implementation
- networking details

This makes the service difficult to maintain.

Instead, delegate those responsibilities.

---

33. Tell, Don't Ask

A useful object-oriented design idea is:

> Tell an object what to do instead of constantly asking for its internal state and changing it externally.

Instead of:

```
if resource.status == "draft":
    resource.status = "published"
```

consider:

```
resource.publish()
```

The object can enforce its own rules.

---

34. Avoid Feature Envy

Suppose:

```
class ResourceService:

    def calculate_resource_score(self, resource):
        return (
            resource.views * 2
            + resource.likes * 5
            - resource.reports * 10
        )
```

If the calculation is fundamentally part of what a resource means, it may belong in `Resource`:

```
class Resource:

    def score(self):
        return (
            self.views * 2
            + self.likes * 5
            - self.reports * 10
        )
```

The service should not become a place where every class's behavior is implemented externally.

---

35. But Don't Put Everything in the Model

There is a balance.

For example, generating a PDF might use an external PDF library.

It may not belong inside:

```
class Resource:
```

Instead:

```
Resource
    
PDFGenerator
```

The key question is:

> Does this behavior naturally belong to the object's domain, or is it an external operation performed using the object?

---

36. Designing Constructors Carefully

Constructors should establish a valid initial state.

Good:

```
class Resource:

    def __init__(self, title, category):
        if not title:
            raise ValueError("Title required")

        self.title = title
        self.category = category
```

Avoid constructors that perform unrelated external work:

```
class Resource:

    def __init__(self, title):
        database.connect()
        send_email()
        upload_file()
```

Object creation should generally be predictable.

---

37. Don't Overuse `__init__()`

A constructor with 15 parameters is difficult to use:

```
User(
    name,
    email,
    age,
    role,
    country,
    timezone,
    language,
    theme,
    ...
)
```

Possible solutions include:

- smaller objects
- configuration objects
- dataclasses
- builders where genuinely useful
- sensible defaults
- separate operations

The goal is not to make every constructor tiny, but to keep object creation understandable.

---

38. Use Dataclasses for Data-Focused Objects

If a class primarily stores structured data, a dataclass may be appropriate.

```
from dataclasses import dataclass


@dataclass
```

```
class Resource:
    title: str
    category: str
    difficulty: str
```

Then a service can operate on it:

```
class ResourceService:

    def publish(self, resource):
        ...
```

This separates:

```
Data model
    
Resource

Application behavior
    
ResourceService
```

when that separation makes sense.

---

39. Classes Should Have Clear Public APIs

Imagine a class:

```
class ResourceService:
    ...
```

Its public methods might be:

`create()`
`update()`
`delete()`
`publish()`
`search()`

Other implementation details should remain internal where appropriate.

This creates a clear boundary between:

`What other code can use`

and:

`How the class performs the work`

---

40. Design for Change

One of the strongest reasons to separate classes is to isolate likely changes.

Suppose today you use:

`PostgreSQL`

but later you might use:

`MongoDB`

If database logic is spread throughout the application, the change becomes expensive.

If persistence is isolated:

```
Application
    
Repository
    
Database
```

the database implementation can potentially change without rewriting the entire application.

---

41. Don't Design for Imaginary Requirements

Another common mistake is overengineering.

You might think:

> “Maybe someday we will support 20 database types.”

Then you create:

```
DatabaseFactory
DatabaseManager
DatabaseProvider
DatabaseStrategy
DatabaseAdapter
DatabaseRegistry
```

before the application actually needs them.

Design for:

- current requirements
- clearly foreseeable changes

Do not build a huge architecture for hypothetical problems.

---

42. YAGNI

A useful software design principle is:

> **You Aren't Gonna Need It.**

If the application only needs one simple implementation, do not automatically build a complex abstraction system.

For example, if you only need:

```
class FileRepository:
    ...
```

you may not need five abstraction layers immediately.

Add complexity when there is a real reason.

---

43. DRY and Class Design

DRY means:

>

> Don't Repeat Yourself.

Suppose three classes contain exactly the same logic:

```
def validate_title(self):
    ...
```

This may indicate shared behavior that should be extracted.

But do not create inheritance merely to eliminate a few repeated lines.

Sometimes a helper function or composition is clearer.

DRY should reduce duplication **without creating unnecessary coupling**.

---

44. SOLID and Class Design

Several principles are commonly used when designing classes.

S — Single Responsibility Principle

Keep responsibilities focused.

O — Open/Closed Principle

Design components so new behavior can often be added without modifying stable existing code.

L — Liskov Substitution Principle

Subclasses should behave consistently with the expectations of their base abstraction.

I — Interface Segregation Principle

Prefer focused contracts over large interfaces that force classes to implement unrelated behavior.

D — Dependency Inversion Principle

High-level logic should depend on abstractions rather than tightly coupling itself to low-level implementation details.

You do not need to apply every principle mechanically.

They are design tools for thinking about boundaries.

---

45. A Practical SOLID Example

Suppose we need to send notifications.

Instead of:

```
class UserService:

    def send_notification(self):
        send_email()
```

we can define an abstraction:

```
from abc import ABC, abstractmethod


class NotificationService(ABC):
```

```
    @abstractmethod
    def send(self, message):
        pass
```

Implementations:

```
class EmailNotification(NotificationService):

    def send(self, message):
        print("Email:", message)

class SMSNotification(NotificationService):

    def send(self, message):
        print("SMS:", message)
```

Then:

```
class UserService:

    def __init__(self, notification_service):
        self.notification_service = notification_service

    def welcome_user(self):
        self.notification_service.send(
            "Welcome!"
        )
```

The user service does not need to know how the notification is delivered.

---

46. Testability as a Design Goal

A well-designed class is often easier to test.

Consider:

```
class ResourceService:

    def __init__(self, repository):
        self.repository = repository
```

For testing:

```
class FakeRepository:

    def save(self, resource):
        self.saved = resource
```

Then:

```
repository = FakeRepository()

service = ResourceService(repository)
```

You can test the service without connecting to a real database.

This is one of the practical benefits of good class boundaries.

---

47. Naming Classes

Class names should describe what the object represents or does.

Good:

```
Resource
User
Payment
ResourceRepository
EmailService
PDFGenerator
```

Less useful:

```
Manager
Helper
Utility
Handler
Processor
Thing
```

Generic names can sometimes be valid, but they often hide responsibility.

Prefer names that communicate purpose.

---

48. Avoid Giant Utility Classes

This is common:

```
class Utils:
    ...
```

Then everything gets added to it:

```
format_date()
send_email()
```

```
calculate_tax()
resize_image()
parse_json()
validate_user()
create_slug()
```

This creates a miscellaneous collection rather than a meaningful abstraction.

Prefer focused functions or classes:

```
DateFormatter
EmailService
ImageProcessor
UserValidator
```

when those concepts actually need separate behavior.

---

49. Module Boundaries Matter Too

Good class design is not enough if every class is placed in one huge file.

For example:

```
models/
  resource.py
  user.py

services/
  resource_service.py
  user_service.py
```

```
repositories/
  resource_repository.py
```

The exact structure depends on the project.

The goal is to make related code easy to find.

---

50. Avoid Circular Dependencies

Suppose:

```
user.py
  
resource.py
  
user.py
```

This can create circular import problems.

If two classes depend heavily on each other, reconsider the design.

Possible solutions include:

- moving shared concepts into another module
- using composition
- introducing a service
- reducing dependencies
- using abstractions

Circular imports are often a signal that boundaries need examination.

---

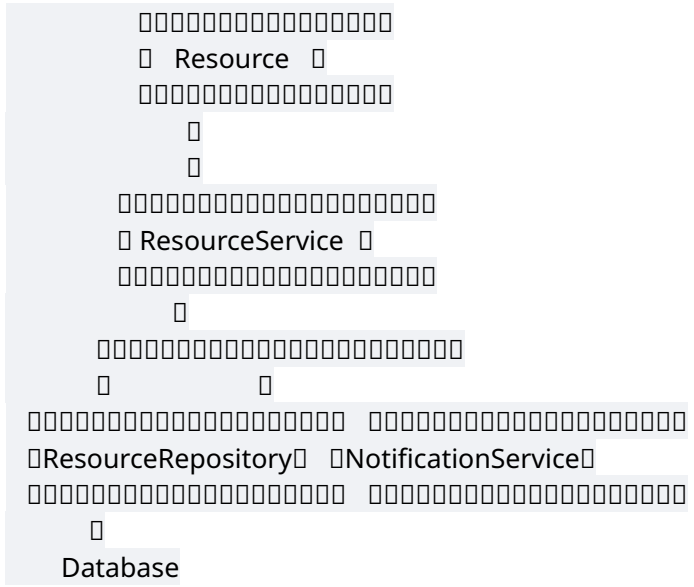
51. Real Project Design Example

Imagine building a haas.dev resource backend.

Requirements:

1. Create resources.
2. Search resources.
3. Save resources.
4. Notify users when new resources are published.

Possible design:



Each part has a clear purpose.

---

52. Example Implementation

Resource

```
from dataclasses import dataclass


@dataclass
class Resource:
    title: str
    category: str
    difficulty: str
```

Repository

```
class ResourceRepository:

    def save(self, resource):
        print(f"Saving: {resource.title}")

    def find(self, title):
        print(f"Searching for: {title}")
```

Notification

```
class NotificationService:

    def notify(self, message):
        print(f"Notification: {message}")
```

Service

```
class ResourceService:

    def __init__(self, repository, notification_service):
```

```
        self.repository = repository
        self.notification_service = notification_service

    def publish(self, resource):
        self.repository.save(resource)

        self.notification_service.notify(
            f"New resource: {resource.title}"
        )
```

Usage:

```
resource = Resource(
    "Python Classes",
    "Python",
    "Intermediate"
)

repository = ResourceRepository()
notifications = NotificationService()

service = ResourceService(
    repository,
    notifications
)

service.publish(resource)
```

The important lesson is not the amount of code.

It is the **separation of responsibilities**.

---

53. A Better Way to Think About Classes

Instead of asking:

> “What classes can I create?”

Ask:

Question 1

What are the important concepts?

Question 2

What data belongs to each concept?

Question 3

What behavior naturally belongs to each concept?

Question 4

Which operations belong to application services?

Question 5

Which operations belong to infrastructure?

Question 6

Which components need to collaborate?

Question 7

Which dependencies are likely to change?

Question 8

How can the design remain easy to test?

---

54. Class Design Workflow

Use this process when starting a project.

1. Understand the requirements

2. Identify domain concepts

3. Identify data

4. Identify behavior

5. Assign responsibilities

6. Identify relationships

7. Choose composition/inheritance

8. Separate infrastructure

9. Define public APIs

10. Inject dependencies

11. Test the design

12. Refactor when necessary

Do not start with folder names or design patterns.

Start with the problem.

---

55. Refactoring a Bad Design

Suppose you start with:

```
class ResourceManager:

    def create_resource(self):
        ...

    def save_database(self):
        ...

    def send_email(self):
        ...

    def generate_pdf(self):
        ...

    def search_resources(self):
        ...
```

This class is doing too much.

Refactor:

ResourceManager

ResourceService

Database logic

ResourceRepository

Email logic

NotificationService

PDF logic

PDFGenerator

Search logic

ResourceSearch

The goal is not to split every method into a new class.

The goal is to create **meaningful boundaries**.

---

56. Refactor Based on Change

A powerful question is:

> “What would cause this class to change?”

Suppose a class changes because:

- database technology changes
- email provider changes
- resource validation changes
- PDF format changes

These are different reasons.

That suggests the class may have multiple responsibilities.

Separate components around meaningful change boundaries.

---

57. Keep Abstractions Honest

An abstraction should represent something real.

Bad:

```
class AbstractManager:
    ...
```

Better:

```
class ResourceRepository:
    ...
```

The second name tells developers what the component actually represents.

Good abstractions make code easier to understand.

---

58. Don't Over-Abstract Simple Code

This:

```
class StringFormatterInterface(ABC):
    ...
```

may be unnecessary if all you need is:

```
def format_title(title):
    return title.strip().title()
```

Not every piece of logic requires a class.

Python supports:

- functions
- classes
- modules
- dataclasses
- dictionaries
- lists
- composition

Choose the simplest structure that clearly solves the problem.

---

59. Classes Should Communicate Intent

Compare:

`processor.process(data)`

with:

`resource_service.publish(resource)`

The second communicates more about what the program is doing.

Good class names and method names make code readable even before you inspect the implementation.

---

60. Mini Project: Design a Learning Platform

Design the classes for a small learning platform.

Requirements:

- users can register
- users can browse resources
- resources belong to categories
- resources can be published
- published resources can trigger notifications
- resources are stored in a database
- the application should be easy to test

Step 1: Identify domain concepts

User
Resource
Category

Step 2: Identify application services

UserService
ResourceService

Step 3: Identify infrastructure

UserRepository
ResourceRepository
NotificationService

Step 4: Identify relationships

User

UserService

UserRepository

Resource

ResourceService

ResourceRepository

NotificationService

Step 5: Think about dependencies

Instead of:

```
self.repository = ResourceRepository()
```

inject the dependency:

`ResourceService(repository)`

Step 6: Think about testing

Can you replace:

`DatabaseRepository`

with:

`FakeRepository`

without changing `ResourceService`?

If yes, the design is easier to test.

---

61. Five Minute Challenge

Take this class:

```
class AppManager:

    def create_user(self):
        ...

    def save_user(self):
        ...

    def send_email(self):
```

```
        ...

    def create_resource(self):
        ...

    def save_resource(self):
        ...

    def search_resource(self):
        ...

    def generate_pdf(self):
        ...
```

Identify at least four responsibilities.

Then design a better structure.

Possible categories:

```
UserService
UserRepository
ResourceService
ResourceRepository
EmailService
PDFGenerator
```

Do not blindly use these names.

Think about which responsibilities actually belong together.

---

62. Exercises

Exercise 1: E-commerce

Design classes for:

- Product
- Cart
- Order
- PaymentService
- ProductRepository

Decide:

- which are domain objects
- which are services
- which handle persistence

Exercise 2: Banking

Design:

`Account`
`Transaction`
`AccountService`
`AccountRepository`
`NotificationService`

Decide where these operations belong:

- deposit

- withdraw
- save account
- send notification
- calculate transaction total

Exercise 3: Blog

Design a system containing:

`User`
`Post`
`Comment`
`PostService`
`PostRepository`
`NotificationService`

Determine:

- what data each class owns
- what behavior belongs to each
- which classes should collaborate

Exercise 4: haas.dev Resources

Design classes for:

`Resource`
`Category`
`ResourceService`
`ResourceRepository`

`SearchService`
`NotificationService`

Draw their relationships before writing code.

---

63. Mini Quiz

Question 1

What should you identify before creating classes?

A. Design patterns
B. Requirements and responsibilities
C. Number of files
D. Number of methods

Answer: B

Question 2

What does high cohesion mean?

A. A class has unrelated responsibilities
B. A class contains closely related responsibilities
C. Every class inherits from another class
D. Every class has one method

Answer: B

Question 3

What does high coupling mean?

A. Components have strong dependencies on each other
B. Components have no relationships
C. A class has many attributes
D. A class uses a dataclass

Answer: A

Question 4

Which relationship is usually represented by composition?

A. IS-A
B. HAS-A
C. IS-NOT-A
D. SAME-AS

Answer: B

Question 5

Why is dependency injection useful?

A. It removes all classes
B. It allows dependencies to be supplied from outside
C. It automatically creates databases
D. It prevents inheritance

Answer: B

---

64. Interview Questions

Beginner

1. What is class design?

Class design is the process of deciding what classes an application needs, what responsibilities they have, what data they contain, and how they collaborate.

2. What is a responsibility?

A responsibility is a specific job or concern that a class should handle.

3. What is cohesion?

Cohesion describes how closely related the responsibilities inside a component are.

4. What is coupling?

Coupling describes how strongly one component depends on other components.

Intermediate

5. What is the Single Responsibility Principle?

It suggests that a class should have one clear responsibility or one primary reason to change.

6. What is dependency injection?

Dependency injection means providing a class's dependencies from outside instead of having the class construct them internally.

7. When should you use composition instead of inheritance?

Composition is appropriate when one object uses or contains another object rather than being a specialized type of it.

8. What is a God class?

A God class is an overly large class that handles many unrelated responsibilities.

Advanced

9. How do you decide whether behavior belongs inside a domain class or a service?

Behavior that naturally operates on and represents the object's own state often belongs to the domain class. Workflow that coordinates multiple objects or external systems often belongs in a service.

10. Why should infrastructure logic be separated from domain logic?

It reduces coupling and makes business logic easier to understand, test, and change independently of databases, APIs, files, or external services.

11. How can good class design improve testing?

Focused classes with injected dependencies can be tested independently and can use fake or mock implementations instead of real external systems.

12. Why should developers avoid premature abstraction?

Because unnecessary abstractions increase complexity and coupling. Abstractions should solve a real design problem rather than hypothetical future requirements.

13. How can you identify a class that has too many responsibilities?

Look for unrelated methods, multiple independent reasons for change, excessive dependencies, difficult tests, and methods that operate on unrelated pieces of data.

---

65. Cheat Sheet

Start With

Requirements

Domain concepts

Data

Behavior

Then Decide

What belongs together?
What should be separate?

Common Components

Model

Represents domain data

Service

Coordinates application behavior

Repository
Handles persistence

Client
Communicates with external systems

Generator
Produces files/output

Relationships

IS-A
Inheritance

HAS-A
Composition

Design Goals

High cohesion
Low coupling
Clear responsibilities
Simple APIs
Testable dependencies
Meaningful abstractions

Avoid

God classes
Deep inheritance
Circular dependencies
Unnecessary abstractions
Generic utility classes
Premature optimization
Premature architecture

---

66. Final Mental Model

Think of a real application as a group of specialists.

Instead of one person doing everything:

```
        ONE GIANT CLASS
             |
    ┌────────┬────────┬────────┐
    ▼        ▼        ▼
 Database   Email    PDF
    ▼        ▼        ▼
 Search    Users   Resources
```

give each responsibility a clear owner:

```
        APPLICATION
             |
    ┌────────┬────────┬────────┐
    ▼        ▼        ▼
Resource  Services  Infrastructure
   |        |         |
   ▼        ▼         ▼
 Model   Workflow  Database/API/File
```

Then connect those components through clear interfaces and dependencies.

The central idea is:

> **Good class design is not about creating more classes. It is about giving each class a clear responsibility and designing simple relationships between those responsibilities.**

When designing a real project, think in this order:

Understand the problem
  ↓
Identify responsibilities
  ↓
Create meaningful boundaries
  ↓
Choose composition or inheritance
  ↓
Separate domain and infrastructure
  ↓
Inject dependencies
  ↓
Keep APIs simple
  ↓
Test
  ↓
Refactor when the design reveals a problem

Visit haas.dev for more resources and guides.

https://dev-roast-app.vercel.app