Software Development Steps

A General Process for a Team and an Individual

Overview: The Software Development Journey

Like building a house, software development follows a proven process:

- **1. Requirements** → *What do we need to build?*
- **2. Software Design** → *How should we build it?*
- **3. Iterative Development** → *Build it step by step*
- **4. Frequent Tests** → *Make sure it works*

1. Requirements (Define)

The starting point for developing software

"You can't build what you don't understand"

Who makes the requirements?

Clients often start unsure of precisely what they need.

- Our job is to help turn their thoughts and ideas into precise, actionable requirements.
- As we work together, clients may discover new needs or realize they want something different.
- So, we should always stay flexible and ready to adjust our plans to make sure they get the best result.

The Issue

Rumsfeld's Law:

> *"We don't know what we don't know yet."*

Common Problems:

- Clients think they know what they want
- Developers think they understand the client
- Both discover they were wrong... after months of work!

Real-World Example: The Hotel Booking Problem

Client says: *"I want a hotel booking system"*

What they need:

- Booking cancellation with refund rules
- Room availability in real-time
- Integration with payment systems
- Different rates for seasons/events
- Multi-language support
- Mobile-friendly interface

The problem: We only discover these details during development!

The Solution - Prototype

Prototype = Building a Model Home

- **After prototype, we know what we don't know**
- Shows what's possible and what's missing
- Reveals hidden requirements early
- Cheaper to change a prototype than finished software

The Goal of a Prototype

Identify:

- **The Model** → *What data do we need?*
- **The Architecture** → *How should components connect?*

We make a working protytpe to check feasibility; no need to make design/tests yet.

Prototype Example: Hotel Booking

```
# Quick prototype - main data models
class Hotel:
    def __init__(self, name, location):
        self.name = name
        self.location = location
        self.rooms = []

class Room:
    def __init__(self, room_number, room_type, price):
        self.room_number = room_number
        self.room_type = room_type  # Single, Double, Suite
        self.price = price
        self.is_available = True

class Booking:
    def __init__(self, customer, room, check_in, check_out):
        self.customer = customer
        self.room = room
        self.check_in = check_in
        self.check_out = check_out
```

What the prototype reveals:

- Need customer management
- Need availability checking
- Need pricing logic
- Need payment processing
- Need cancellation policies

2. Software Design

"Design before you code"

Two main activities:

- Identify Modules & Interfaces
- Create Test Plan

Identify Modules & Interfaces

Module = A specialized team member

Hotel Booking System Modules:

- **UserModule** → Handle login/registration
- **HotelModule** → Manage hotel data
- **BookingModule** → Handle reservations
- **PaymentModule** → Process payments
- **NotificationModule** → Send emails/SMS

Interface Design Example

```
# Clear interfaces between modules
class BookingService:
    def check_availability(self, hotel_id, dates):
        # Returns available rooms
        pass

    def create_booking(self, customer, room, dates):
        # Creates a new reservation
        pass

    def cancel_booking(self, booking_id):
        # Cancels existing booking
        pass

class PaymentService:
    def process_payment(self, amount, card_info):
        # Handles payment processing
        pass

    def refund_payment(self, booking_id):
        # Processes refunds
        pass
```

The Problems that Design Can Solve

1. Reveals high-level module connections, enabling problem-solving at a broader level.
2. Enables planning for testing individual modules and their interactions.
3. Allows flexibility to modify modules when needed.
4. Separates design from implementation, letting others, including AI, handle coding while we focus on high-level architecture.

Test Plan

Plan your testing before you build

Module	Test Case	Expected Result
BookingService	Book available room	Booking confirmed
BookingService	Book unavailable room	Booking rejected
PaymentService	Valid credit card	Payment successful
PaymentService	Invalid credit card	Payment failed

3. Iterative Development

"Build the software with a Vertical Slice Approach"

- Instead of building perfect modules one by one, modern software engineering uses the vertical slice approach.
- This means creating a small, working feature that includes all layers—from user interface to backend—right from the start.
- Then, the feature is improved continuously based on user feedback.

Vertical Slice vs. Horizontal Development

Wrong Approach: Horizontal Development

- Build all UI layers first
- Then build all business logic
- Finally, build all database layers
- **Problem:** Nothing works until the very end!

Right Approach: Vertical Slice

- Build complete features from top to bottom
- Each slice is a working piece of software
- Users can test and provide feedback immediately

The Four Principles of Vertical Slice Development

1. "Make it work, and make it better"

- Similar to "Red, Green, Refactor" in TDD
- Emphasizes progression: functional → optimal
- Prevents over-engineering in early iterations

2. "The software should be usable from the beginning."

- Forces developers to think about user value from day one
- Prevents the "nothing works until everything works" trap

3. "Test thoroughly to guarantee the software is working with any changes."

- Emphasizes continuous testing as you add features
- Prevents breaking existing functionality when adding new features
- Builds confidence for rapid iteration

4. "Get feedback early and often"

- Captures the collaborative aspect
- Enables course correction before major investment
- Validates assumptions with real users

Build minimum viable product (MVP)

"Ship the simplest working solution as fast as possible"

Core MVP Principles:

- **Ship early** → Get real user feedback immediately
- **Functional, not fancy** → Works well, looks basic
- **Product, not prototype** → Users actually use and pay for it
- **Be ready to pivot** → Willing to throw it away if unsuccessful

MVP Examples: Simple but Working

Company	MVP	What They Skipped
Airbnb	Simple website + manual email booking	Automated payments, reviews, messaging
Uber	Basic ride request via SMS	GPS tracking, ratings, surge pricing
Facebook	College directory with basic profiles	News feed, photos, messaging

Key: Each MVP effectively addressed a core problem, attracting users from the start.

Visual Example: Building a Car

Horizontal (Wrong):

- Week 1: Build all wheels
- Week 2: Build all body parts
- Week 3: Build all engines
- Week 4: Assemble everything
- **Result:** No working car until week 4!

Vertical (Correct):

- Week 1: Build basic scooter (works!)
- Week 2: Upgrade to bicycle (better!)
- Week 3: Add motor → motorcycle (even better!)
- Week 4: Add roof and doors → car (complete!)

Vertical Slice in Software

Vertical Slice = Complete working feature from UI to database

Example: "Hotel Booking" feature

- **UI Layer** → Booking form and confirmation page
- **Business Logic** → Booking validation and processing
- **Database Layer** → Store booking and room data
- **API Layer** → REST endpoints for booking

Key: Complete the entire stack for basic functionality, then enhance!

Correct Iteration: Hotel Booking System

Iteration	Working System	What Users Can Do	Technical Stack
Week 1	Basic Hotel Booking MVP	View 1 hotel, book 1 room type, simple confirmation	UI + API + Database (minimal)
Week 2	Enhanced Booking	Multiple hotels, room search, email confirmation	+ Search logic + Email service
Week 3	Full Booking Experience	User accounts, booking history, cancellations	+ User management + Advanced business logic
Week 4	Production Ready	Payment processing, advanced search, mobile responsive	+ Payment API + Enhanced UI

Key Point: Every week, we deliver a working hotel booking system that customers can actually use!

Why Vertical Slices Work Better

Benefits of Vertical Approach:

- **Early Feedback:** Users can test and give feedback immediately
- **Risk Reduction:** Discover problems early when they're cheap to fix
- **Motivation:** Team sees working results every iteration
- **Flexibility:** Can pivot or change direction based on real user feedback
- **Value Delivery:** Customers get value even if the project is cancelled early

Real Example: After Week 1, the hotel owner could start taking bookings manually while you build the advanced features!

Code Smell & Refactoring

Code Smell = Warning signs your code needs improvement

During each iteration:

- Look for code smells
- Refactor to improve quality
- Keep tests passing

Common Code Smells in Development

1. Long Method (doing too much)

```
def process_booking():  # 150 lines!
    # Validate customer
    # Check room availability
    # Calculate pricing
    # Process payment
    # Send confirmation
    # Update database
    # Log transaction
```

Better approach:

```
def process_booking(customer, room, dates):
    self.validate_customer(customer)
    self.check_availability(room, dates)
    price = self.calculate_price(room, dates)
    self.process_payment(customer, price)
    booking = self.create_booking(customer, room, dates)
    self.send_confirmation(booking)
```

Applying Design Patterns

Use proven solutions during development

Example: Observer Pattern for Notifications

- When booking is confirmed → notify the customer
- When payment fails → notify admin
- When the room is cancelled → notify the hotel manager

```python
class BookingSystem:
    def __init__(self):
        self.observers = []  # List of notification services

    def add_observer(self, observer):
        self.observers.append(observer)

    def notify_booking_confirmed(self, booking):
        for observer in self.observers:
            observer.handle_booking_confirmed(booking)

# Usage
booking_system = BookingSystem()
booking_system.add_observer(EmailService())
booking_system.add_observer(SMSService())
booking_system.add_observer(AdminNotifier())
```

4. Frequent Tests

"Test early, test often, test everything"

Testing pyramid approach:

- **Unit Tests** → Test individual methods
- **Integration Tests** → Test module interactions
- **System Tests** → Test complete features
- **User Acceptance Tests** → Test with real users

Testing Example: Hotel Booking

```
import unittest

class TestBookingService(unittest.TestCase):

    def test_book_available_room(self):
        # Arrange
        booking_service = BookingService()
        customer = Customer("john@email.com")
        room = Room("101", "Single", 100)
        dates = ("2024-03-01", "2024-03-03")

        # Act
        result = booking_service.create_booking(customer, room, dates)

        # Assert
        self.assertTrue(result.is_confirmed)
        self.assertEqual(result.total_price, 200)

    def test_book_unavailable_room(self):
        # Test booking when the room is not available
        pass
```

Continuous Integration (CI)

Automate testing with every code change

Every time someone commits (deploys) code:

1. **Run all tests automatically**
2. **Build the application**
3. **Deploy to testing environment**
4. **Notify team of results**

Benefits: Catch problems immediately, not weeks later!

Tools for Software Development

The right tools make development faster and easier

Essential Development Tools (Examples)

Software Engineers master their tools—using the best available, and creating new ones when needed, just like any true expert in their craft.

Category	Tool Examples	Purpose
Code Editor	VS Code, PyCharm	Write and edit code
Version Control	Git, GitHub	Track code changes
Testing	pytest, JUnit	Automated testing
Database	PostgreSQL, MongoDB	Store application data
Deployment	Docker, AWS	Deploy applications

Version Control Example: Git Workflow

We should be able to revert to any point in time to recover from mistakes.

```
# Start new feature
git checkout -b feature/hotel-search
git add .
git commit -m "Add hotel search functionality"

# Push to remote repository
git push origin feature/hotel-search

# Create pull request for team review
# After review, merge to the main branch
git checkout main
git pull origin main
git merge feature/hotel-search
```

Development Environment Setup

We should isolate development environments to avoid conflict in any form.

Always use the tool for isolating the programming environment.

```
# Create an isolated environment
python -m venv hotel_booking_env

# Activate environment (Windows)
hotel_booking_env\Scripts\activate

# Activate environment (Mac/Linux)
source hotel_booking_env/bin/activate

# Install project dependencies
pip install -r requirements.txt

# Deactivate when done
deactivate
```

Use the same libraries/packages that others use.

```
# requirements.txt - Define project dependencies
Flask==2.3.0
SQLAlchemy==2.0.0
pytest==7.4.0
requests==2.31.0

# Install dependencies
pip install -r requirements.txt

# Project structure
hotel_booking/
├── app/
│   ├── models/
│   ├── services/
│   └── controllers/
├── tests/
├── requirements.txt
└── README.md
```

Summary: The Complete Development Cycle

1. Requirements (Define) & Prototype

- Understand what to build
- Build a prototype to discover unknowns

2. Design

- Plan modules and interfaces
- Create a comprehensive test plan

3. Iterative Development

- Build an application using a vertical slice approach.
- Make it work and make it better.
- Apply design patterns to make the design better.

4. Frequent Testing (before Deployment)

- Test every change
- Automate the testing process
- Catch problems early

Key Takeaways

- **Start with prototype** → Reduce unknown risks
- **Design before coding** → Save time and effort
- **Make MVP** → Vertical slice
- **Build incrementally** → Deliver value early
- **Test continuously** → Maintain quality
- **Use the right tools** → Work more efficiently

Remember: Good software is built step by step, not all at once!