Table of Contents

C1: Scripting

Type of Scripting Languages

Shell Scripting	• **Automate and glue together operating system commands**
	• Contains sequence of commands that are executed by the operating system's command interpreter (shell).
	• Commonly used for **system administration, task automation and batch processing**
	• Examples:
	  ○ Bash (Linux / macOS)
	  ○ PowerShell (Windows)
	  ○ Batch Script (.bat, .cmd)
	• Use Cases:
	  ○ Automating backups
	  ○ Installing software packages
	  ○ Managing files and directories
	  ○ Running scheduled system maintenance tasks

Web Scripting	• **Create dynamic web pages** and handle interactions between users and web applications
	• Divided into:
	  ○ **Client-side scripting** - runs in the user's browser
	  ○ **Server-side scripting** - runs on the web server
	• Examples:
	  ○ Client-side languages:
	  ■ JavaScript
	  ○ Server-side languages:
	  ■ PHP
	  ■ Python
	  ■ JavaScript
	• Use Cases:
	  ○ Form validation
	  ○ User authentication
	  ○ Database interaction
	  ○ Generating dynamic content

Extension Language	• **Control, customize or extend the functionality of existing applications** • These scripts do not run independently but are embedded inside software to add automation and customization features.

	• Languages:
	  ○ VBScript (Windows applications)
	  ○ Python (Extensions in Blender, AutoCAD)
	  ○ Lua (game engines)
	  ○ JavaScript (Adobe Acrobat scripting)
	• Use Cases:
	  ○ Automating repetitive tasks in software
	  ○ Creating plugins or macros
	  ○ Customizing application behavior

Scripting VS Application Languages

	Scripting Languages	Application Languages
Execution Method	**Interpreted at runtime**, meaning code is executed line by line.	Usually **compiled into machine code before execution**.
Performance vs Expressiveness	**Trade performance for expressiveness**, making them easier and faster to write.	**Prioritize performance**, producing faster and more efficient executables.
Development Speed and Prototyping	**Encourage rapid prototyping and faster development** due to simpler syntax and dynamic execution.	**Require more development effort**, including compilation and stricter structure.
Type System	**Tend to be weakly or dynamically typed**, with type checking done at runtime.	**Often strongly or statically typed**, with type checking done at compile time.
Typical Usage	Commonly used for **automation, web scripting and gluing components together**.	Used to build **large-scale, performance-critical applications** such as desktop, mobile and system software.

Advantages of Strong Typing

Catch Errors Earlier	• Strongly typed languages perform **type checking at compile time**. • Many errors, such as assigning an incorrect type, are

	detected before the program runs. • This reduces runtime errors and improves software reliability. • e.g. Assigning a string to an integer variable will cause a compile-time error in Java or C++
More Manageable Code	• Type information makes large codebases easier to understand and maintain. • Developers can more easily track how data flows through the system. • Strong typing is beneficial for **team-based and long-term projects**. • e.g. Enterprise systems written in Java and C# benefit from strong typing for maintenance.
Clearer and More Readable Code	• Explicit types clarify **how variables, parameters and return values are used**. • Code becomes more self-documenting. • This improves readability and reduces misunderstanding between developers. • e.g. Method signatures clearly indicate expected input and output types.
More Efficient Code Execution	• The compiler can use type information to perform **optimizations**. • This results in more efficient machine code. • Strong type is suitable for **performance-critical applications**. • e.g. Numerical computations in C++ can be optimized by the compiler using static type information.

Advantages of Weak Typing

Fewer Restrictions and Greater Flexibility	• Weakly typed languages impose fewer constraints on how variables are used. • Values can be easily converted between types. • This allows developers to experiment and prototype quickly. • e.g. A variable in JavaScript can hold a number, then later a string.
Easier to "Hook	• Weak typing allows different components to interact

Things Together"	without strict type requirements. • This is especially useful in **scripting and automation** environments. • Everything being treated as a common type (e.g. strings) simplifies integration. • e.g. Unix shell scripts treat command output as strings, enabling easy piping between commands.
Encourages Code Reuse	• Functions can work with many different types without needing separate interfaces. • This reduces duplication of code. • Generic behavior is easier to implement. • e.g. A JavaScript function can accept numbers, strings or objects without modification.
Simpler and More Succinct 简洁 Code	• Less boilerplate code is needed since type declarations are minimal or absent. 由于类型声明极少或完全省略，因此所需的冗余代码更少。 • Code is shorter and faster to write. • This improves developer productivity for small or quick tasks. • e.g. Python scripts often require fewer lines of code than equivalent Java programs.

2 Types of Web Development Scripting

	Client-Side Scripting	Server-Side Scripting
Definition	Scripts that **run inside the user's web browser**. The code is executed **after the web page has been downloaded**, and **no interaction with the web server is required for the script to run**.	Scripts that **run on the web server**. The server executes the code and then **sends only the generated HTML output** to the client's browser.
How it works	1. The server sends HTML (and script files) to the browser. 2. The browser executes the client-side script locally.	1. The browser sends a request to the server. 2. Server executes server-side code. 3. Server generates HTML.

	3. The script manipulates the web page or responds to user actions.	4. Browser receives and displays the HTML result.
Key Characteristics	• Runs **on the client (browser)** • Executes **after page loads** • Does **not require server processing** for execution • Immediate response to user actions	• Runs **on the server** • Client **never sees the source code** • Used to generate dynamic web pages • Handles business logic and data processing
Common Uses	• Form validation (e.g. checking empty fields) • Interactive UI (buttons, pop-ups, animations) • Dynamic content updates (without reloading the page)	• Database operations (search, insert, update) • User authentication and login • Processing orders and payments • Generating dynamic content
Examples	• Language: JavaScript • Scenario: Checking whether a password field is empty before submitting a form.	• `.php` - PHP (Hypertext Preprocessor) • `.js` - Node.js • `.jsp` - Java Server Pages • `.aspx` - ASP.NET
Highlighted Info	Limitations • Client-side scripts **cannot access databases directly**. • Code is **visible to users**, so it is **not suitable for sensitive logic**.	Advantages • More **secure** than client-side scripting • Suitable for **critical operations** such as payments

C2: Software Security Practices

Recommended Software Security Practices

- Clearly define roles and responsibilities
- Provide development teams with adequate software security training
- Implement a secure software development lifecycle
- Establish secure coding standards
- Build a reusable object library
- Verify the effectiveness of security controls
- Establish secure outsourced development practices

Goal of Software Security

- Maintain confidentiality, integrity and availability of information resources in order to enable successful business operations
- This goal is accomplished through the implementation of security controls

Security Issues*

1. Data Breaches

- **Confidential, sensitive or protected information is accessed, viewed, stolen or exposed by someone who is NOT authorized**.
- Examples:
    - Unauthorized person accessing database
    - Employee leaking customer info
    - Sensitive data sent to the wrong email
    - Publicly exposed database or storage bucket
    - Passwords or API keys accidentally pushed to GitHub
    - Stolen devices containing personal data
    - Malware stealing login credentials

Prevention

- **Strong authentication**: MFA, Complex passwords
- **Encryption**: Encrypt data at rest and in transit
- **Access control**: Role-based permissions, Least privilege principle
- **Regular security patching**: Update servers, apps, libraries
- **Monitoring & logging**: Detect unusual access patterns
- **Employee training**: Avoid phishing, Handle data safely
- **Secure configuration**: No public-facing databases, Hardened servers
- **Secure coding**: Avoid SQL injection, XSS, CSRF

2. Social Engineering

- Attacker **tricks people into giving information, access, or doing an action that helps the attacker**

Common Social Engineering Techniques	
Phishing	Fake emails or websites used to trick victims into giving login credentials or personal information
Spear Phishing	Highly targeted phishing aimed at specific individuals (e.g. CEO, manager)
Smishing	Phishing attacks through SMS, WhatsApp or messaging apps
Vishing (Voice Phishing)	Phone calls pretending to be banks, IT support, police or companies to steal information
Pretexting	Attacker creates a believable story or identity to gain trust and extract information
Baiting	Offering something attractive (free files, gifts, USB drives) to

	lure victims into executing malware or giving access
Tailgating / Piggybacking	Attacker physically follows someone into a restricted building or secure area
Impersonation	Pretending to be someone with authority (IT staff, manager, technician) to gain access or information
Prevention	
Education & Awareness	Train users to recognize and avoid social engineering and cyber threats.
Verification Procedures	Always confirm identity or requests through trusted channels before taking action
Zero Trust Principle	Never trust by default to verify every user, device and request
Strong Authentication	Use strong passwords and multi-factor authentication to secure logins
Access Control	Limit user permissions so only authorized people can access sensitive data

Access to Sensitive Data

Sensitive data:

- Personally identifiable information (PII)
- Financial details
- Medical records
- Proprietary business information

Authentication	Implement strong authentication techniques to only allow authorized users accessing sensitive data:

	• **Multi-factor authentication (MFA)** • **Password policies (e.g. length, complexity)** • **Biometrics** • **Hardware security keys**	
Authorization	**Role-Based Access Control (RBAC)**	Restrict access to sensitive data based on the user's role within the organization
	Least Privilege Principle	Users only have access to those data important for their tasks to reduce the risk of exposing information unnecessarily
	Attribute-Based Access Control (ABAC)	Fine-grained control based on user attributes to allow dynamic access control decisions (e.g. department, location or other contextual information)
Data encryption	• Sensitive data should be encrypted when stored on disks, databases or cloud storage • To prevent unauthorized access even if attackers gain access to the storage system	
Audit logs and monitoring	• Maintain detailed logs of who accessed sensitive data, when and why • Detect unauthorized access and provide insights for investigation after a breach	
Data masking and tokenization	• Mask sensitive data in environments where it is not necessary for the user or system to see the actual data • Showing the last 4 digits of a credit card number or a hashed version of sensitive information	

Modification of Data

- Process where **data is changed, altered, or tampered with malicious intent**
- Unauthorized data modification may cause:
    - Loss of data integrity
    - Financial fraud
    - System misbehavior
    - Breaches of privacy

Server-side validation	• The **server checks whether the incoming data is valid regardless what the client sends**
	• Pros:
	  ○ Highly reliable because client cannot bypass it
	  ○ Protected against tampered request, injection, invalid formats and missing fields
	  ○ Ensures data integrity and security consistently
	• Cons:
	  ○ Adds server workload because every request is validated again
	  ○ Usually needs more detailed coding compared to client-side validation

Whitelisting	• Practice of **adding emails, IP addresses and apps to a list of entities that are safe to receive messages and attachments from or use**
	• Pros:
	  ○ Strict access control
	  ○ Reduced risk of malicious intent
	  ○ Increased safety of internet-connected systems
	  ○ Ability to define what an app or service can and can't do

	within your system ● Cons: ○ Difficult to compile a list ○ Administrative burden of updating the list ○ Limited creativity and access to tools used to complete tasks ○ Risk of blocking wanted traffic
Blacklisting	● Practice of **compiling a list of entities that are unsafe or unwanted to prevent them from reaching your computer or email** ● Pros: ○ Easy to implement quickly ○ Useful for common known bad inputs ● Cons: ○ Not secure because attackers can bypass it, there are many variations of malicious input exist ○ Requires constant updates and still never complete ○ Should never be used as the only protection

3. Denial of Service

Attacker attempts to **make a computer, network or service unavailable by overwhelming it with flood of traffic** or by **exploiting a vulnerability to cause it to crash**	
Traffic filtering and rate limiting	● Use firewalls and intrusion prevention systems (IPS) to filter out malicious traffic ● Controls how many requests a user or IP address can send to the server in a specific timeframe (e.g. 100 requests per minute)

Load balancing and redundancy	• Use load balancers to distribute traffic across multiple servers • Prevents any single server from becoming overwhelmed by too much traffic
Web Application Firewall (WAF)	• Protect against application-layer attacks by filtering out malicious HTTP/HTTPS traffic before it reaches the server

Blackholing and Sinkholing	• Blackholing
	  ◦ Routing attack traffic to a "black hole" where the traffic is dropped and discarded
	• Sinkholing
	  ◦ Similar concept, but redirecting the attack traffic to a monitored server for analysis

4. SQL Injection

Attacker **injects malicious SQL code into a query to modify data in a database**

• **Filter all user input at server-side** as well
  ◦ Check and validate all input on the server, not only on the browser
  ◦ Client-side validation can be bypassed easily using tools
  ◦ Example:
  ▪ Ensure numbers are actually numbers
  ▪ Block special characters like `'`, `"`, `;`, `–` where appropriate
  ▪ Reject overly long input
• **Sanitization of user input** (e.g. trim())
  ◦ Cleaning the input before processing
  ◦ Examples:
  ▪ `trim()` → remove leading / trailing spaces

        - Removing dangerous characters
        - Converting special characters to safe versions
- **Escaping output**
    - Converting special characters into a safe format before sending them to database or back to the browser
    - Prevent database from interpreting the input as part of SQL commands
    - e.g. `'` becomes `\'`
- Use **prepared statement / parameter binding in SQL statements**
    - Strongest and most recommended defense
    - separate SQL code from data
    - Database treats user input only as data, not as executable code
- **Stored procedures**
    - SQL statements stored in the database itself
    - If written safely (without dynamic SQL), they can reduce SQL injection risk
- **Least Privilege Database Accounts**
    - Application database user have minimum permissions (e.g. can't drop tables)
    - Examples:
        - App user cannot DROP table
        - App user cannot create new users
        - App user only as SELECT / INSERT / UPDATE on specific tables
        - Admin privileges only for actual DB admin accounts

5. Cross-Site Scripting (XSS)

Attacker **injects malicious JavaScript code into a web page that is then executed by other users**

Examples:

- Steal session cookies → hijack user accounts
- Redirect users to phishing websites

- Deface the website 篡改网站
- Display fake login forms
- Send malicious requests on behalf of the user
- Modify visible page content

- **Input Sanitization**
    - Sanitize incoming data so dangerous characters are removed or neutralized.
    - Examples:
        - Removing `<script>` tags
        - Removing JavaScript event handlers like `onclick=`
        - Filtering out `<` and `>` where not needed
        - Limiting input length
- **Escaping Output**
    - Ensures that user-supplied content is displayed as text, not interpreted as code
    - Browser shows the tags instead of executing them
    - Example:
        - Before escaping: `<script>alert('xss')</script>`
        - After escaping: `&lt;script&gt;alert('xss')&lt;/script&gt;`
- **Content Security Policy (CSP)**
    - Tells browser which scripts are allowed
    - Prevents most reflected and stored XSS
    - Limits impact even if the attacker injects code
- **Input whitelisting instead of blacklisting**
    - Allow only safe characters (letters, numbers) instead of block bad patterns
    - Blacklisting tries to block bad characters like `<`, `>`, `'`, `"` but attackers can bypass it with encoding tricks
    - Whitelisting allows only safe characters

6. Phishing Attack

- Attackers **deceive individuals into revealing sensitive information** such as:
    - Passwords
    - Credit card numbers
    - Personal identification
    - Login credentials
- Attackers **impersonate legitimate entities**
- Examples:
    - Fake emails pretending to be from banks, universities or government
    - Fake login pages that look identical to the original
    - SMS messages claiming "Your parcel is held" or "Your account is locked"
    - Phone calls pretending to be customer service agents

- **Security Awareness Training**
    - Employees or users are trained to recognize phishing attempts
    - Examples:
        - How to identity suspicious emails
        - Checking sender address
        - Not clicking unknown links
        - Recognizing fake URLs
        - Reporting suspicious messages
- **Simulated Phishing Exercises**
    - Organizations send fake (safe) phishing emails to test whether employees fall for them
    - Purposes:
        - Measure employee vulnerability
        - Identify areas needing more training

        - Create a culture of awareness and caution
- **Email Filtering**
    - Filter suspicious emails before they reach the user
    - May block:
        - Known malicious domains
        - Emails with dangerous attachments
        - Emails with suspicious patterns or spoofed addresses
- **Spam Filters**
    - Automatically detect and isolate emails that look like spam or phishing based on behavior patterns
    - Analyse:
        - Keywords used in scams
        - Sender reputation
        - Attachment types
        - URL patterns
- **Enable MFA**
    - Adds an extra layer of verification (e.g. OTP, authenticator app, fingerprint)
    - Even if attacker successfully steals user's password, they can't log in without the second authentication factor

7. Cross-Site Request Forgery - CSRF

- Web attack where an attacker tricks a logged-in user's browser into performing unwanted actions on a website **without their consent**.
- Attacker uses users' own login session to do something they didn't intend

How?

1. Hacker creates a request (in the form of URL) for their own benefit from a website
2. Hacker embeds that request into a hyperlink and sends it to a visitor who they hope is logged in to the site

3. The website visitor clicks the link, unwittingly 不知不觉 sending the request to the site
4. Assuming the request is legitimate, the website fulfills the request, sending data, funds or access to the hacker

Prevention

- **CSRF Tokens**
    - Unique hidden token embedded in forms
    - Attacker cannot guess or reuse the token
- **SameSite Cookies**
    - Prevents cookies from being sent in cross-site requests
- **Double-Submit Cookie**
    - Cookie + hidden field must match
- **Re-authentication for sensitive actions**
    - e.g. bank asks for password / OTP before transferring money
- **Avoid GET for dangerous actions**
    - Use POST for data-changing operations
- **CAPTCHA (optional)**
    - For sensitive forms to ensure human interaction

Secure Coding Practices

Input Validation	• Conduct all data validation on a trusted system • Identify all data sources and classify them into trusted and untrusted. Validate all data from untrusted source (e.g. databases, file streams, etc) • Centralized input validation for the application • Specify proper character sets (e.g. UTF-8) for all sources of input • Encode data to a common set before validating (canonicalize 规

	范化) • All validation failures should result in input rejection • Validate all client-provided data before processing, including all parameters, URLs and HTTP header content (e.g. cookie names and values) • Verify that header values in both requests and responses contain only ASCII characters • Validate data from redirects • Validate for expected data types • Validate data range • Validate data length • Validate all input against a white list of allowed characters, whenever possible
	🤔For easier memorise: 1. TRUST - Know where the data comes from 2. CENTRALIZE - Don's scatter validation everywhere 3. FORMAT - Ensure correct encoding & character sets 4. WHITELIST - Allow only safe input 5. RULES - Validate the content itself
Output Encoding	• Conduct all encoding on a trusted system • Use a standard, tested routine for each type outbound encoding • Contextually output encode all data returned to the client that originated outside the application's trust boundary • Contextually sanitize all output of untrusted data to queries for SQL and XML • Sanitize all output of untrusted data to OS commands Before sanitize:

	<script>alert("You have been attacked!")</script> After sanitize: <script>alert("you are attacked")</script>
	🤔For easier memorise: 1. WHERE - Output encoding happens on the server 2. WHEN - Encode whenever data leaves the trust boundary 3. WHERE IT GOES - Encode based on target context ○ sanitize & encode HTML output ○ sanitize output sent into SQL or XML queries ○ sanitize output before sending to OS command line
Authentication	• Require authentication for all pages and resources, except those intended to be public • All authentication controls must be enforced on trusted system • Establish and utilize standard, tested, authentication services • Use a centralized implementation for all authentication controls • Segregate 分离 authentication logic from the resource being requested and use redirection to and from the centralized authentication control • All authentication controls should fail securely • Ensure that only cryptographically strong one-way salted hashes of passwords are stored and that the table/file that stores the passwords and keys is writable by the application (Do not use the MD5 algorithm if it can be avoided) • Password hashing must be implemented on a trusted system • Validate the authentication data only on completion of all data input • Authentication failure responses should not indicate which part of the authentication data was incorrect • Utilize authentication for connections to external systems that

	involve sensitive information or functions • Authentication credentials for accessing services external to the application should be encrypted and stored in a protected location on a trusted system. The source code is NOT a secure location. • Use only HTTP POST requests to transmit authentication credentials • Only send non-temporary passwords over an encrypted connection or as encrypted data (e.g. an encrypted email)
	🤔For easier memorise: 1. CONTROLS - Authentication must be CENTRALIZED 2. TRUSTED - Do NOT trust the browser. Server decides everything 3. STORAGE - Hash storing, store safe 4. RESPONSE - Error must be vague, not helpful to attackers 5. PROTECTION - Credentials = POST + Encrypted Only
Password Management	• Enforce password complexity requirements established by policy or regulation • Enforce password length requirements established by policy or regulation • Password entry should be obscured on the user's screen • Enforce account disabling after an established number of invalid login attempts (normally 5) • Password reset and changing operations require the same level of controls as account creation and authentication • Password reset questions should support sufficiently random answers • If using email-based resets, only send email to a pre-registered address with a temporary link / password • Temporary passwords and links should have a short expiration time • Enforce the changing of temporary passwords on the next use

	• Notify users when a password reset occurs • Prevent password re-use • Passwords should be at least 1 day old before they can be changed • Enforce password changes based on requirements established in policy or regulation • Disable "remember me" functionality for password fields • The last use (successful or unsuccessful) or a user account should be reported to the user at their next successful login • Change all vendor-supplied default passwords and user IDs or disable the associated accounts • Re-authenticate users prior to performing critical operations • Use multi-factor authentication for highly sensitive or high value transaction accounts • If using 3rd party code for authentication, inspect the code carefully to ensure it is not affected by any malicious code
	🤔For easier memorise: 1. COMPLEXITY & LENGTH - Strong + long + not reused 2. LIFECYCLE / LOGIN CONTROLS - Logins must be safe, locked and verified 3. RESET & RECOVERY PROCESS - Reset = as strong as login 4. NOTIFICATIONS - Notify for reset + last login 5. ACCESS PROTECTION - Protect access at all stages
Session Management	• Use the server or framework's session management controls. The application should only recognize these session identifiers (IDs) as valid • Session ID creation must be always be done on a trusted system • Session management controls should use well-vetted 充分验证 algorithms that ensure sufficiently random session IDs • Logout functionality should fully terminate the associated

	session or connection • Logout functionality should be available from all pages protected by authorization • Establish a session inactivity timeout that is as short as possible, based on balancing risk and business functional requirements. In most cases, it should be no more than several hours • Disallow persistent logins and enforce periodic session terminations, even when the session is active • If a session was established before login, close the session and establish a new session after a successful login • Generate a new session ID on any re-authentication • Do not allow concurrent logins with the same user ID • Do not expose session IDs in URLs, error messages or logs. Session ID should only be located in the HTTP cookie header • Protect server-side session data from unauthorized access, by other users of the server, by implementing appropriate access controls on the server • Generate a new session ID and deactivate the old one periodically • Generate a new session ID if the connection security changes from HTTP to HTTPS, as can occur during authentication • Supplement standard session management for sensitive server-side operations, like account management, by utilizing per-session strong random tokens or parameters
	🤔For easier memorise: 1. CREATION - Create safe → Rotate often 2. INVALIDATION - Kill old sessions completely – no leftovers 3. TIMEOUT - Short session = safer session 4. HIDING - Session ID must NEVER be visible 5. PROTECTION - Protect server, sensitive operations
Access Control	• Use only trusted system objects for making access authorization

	decisions • Use a single site-wide component to check access authorization • Access controls should fail securely • Deny all access if the application cannot access its security configuration information • Enforce authorization controls on every request, including those made by server-side scripts, "includes" and requests from rich client-side technologies like AJAX • Segregate privileged logic from other application code • Restrict access to files or other resources, including those outside the application's direct control to only authorized users • Restrict access to protected URLs to only authorized users • Restrict access to protected functions to only authorized users • Restrict direct object references to only authorized users • Restrict access to services to only authorized users • Restrict access to application data to only authorized users • Restrict access to user and data attributes and policy information used by access controls • Restrict access to security-relevant configuration to only authorized users • Server-side implementation and presentation layer representations of access control rules must match • If state data must be stored on the client, use encryption and integrity checking on the server side to catch state tampering • Enforce application logic flows to comply with business rules • Limit the number of transactions a single user or device can perform in a given period of time. The transactions / time should be above the actual business requirement, but low enough to deter automated attacks • If long authenticated sessions are allowed, periodically re-validated a user's authorization to ensure that their privileges have not changed and if they have, log the user out and force

	them to re-authenticate • Implement account auditing and enforce the disabling of unused accounts • The application must support disabling of accounts and terminating sessions when authorization ceases (e.g. changes to role, employment status, business process, etc) • Create an Access Control Policy to document an application's business rules, data types and access authorization criteria and / or processes so that the access can be properly provisioned and controlled. This includes identifying access requirements for the data and system resources
	🤔For easier memorise: 1. CORE PRINCIPLES - If you're not sure → deny 2. CENTRALIZATION - One gatekeeper for everything 3. RESTRICTION - Everything must be restricted unless explicitly allowed 4. STATE SECURITY - If the user's state or role changes, invalidate everything 5. FLOW CONTROL - Follow the flow, limit the speed 6. POLICY & ACCOUNT MANAGEMENT - Write the rules. Audit them. Enforce them.
Cryptographic Practices	• All cryptographic functions used to protect secrets from the application user must be implemented on a trusted system • Cryptographic modules must fail securely • All random numbers, strings, file names, etc. should be generated using the cryptographic module's approved random number generator • Establish and utilize a policy and process for how cryptographic keys will be managed
	🤔For easier memorise: 1. TRUSTED - Crypto belongs on the server

	2. FAIL-SECURE - If encryption breaks, don't leak 3. RANDOMNESS - Tokens → only crypto RNG 4. KEY MANAGEMENT - The crypto is only as strong as the key handling
Error Handling	• Do not disclose sensitive information in error responses (e.g. system details, session IDs, or account information) • Use error handlers that do not display debugging or stack trace information • Implement generic error messages and use custom error pages • The application should handle application errors and not rely on the server configuration • Properly free allocated memory when error conditions occur • Error handling logic associated with security controls should deny access by default
	🤔For easier memorise: 1. HIDE - Show nothing. Hide everything. 2. MANAGE - Your app must own its errors. 3. DENY - On error → always deny
Logging	• All logging controls should be implemented on a trusted system • Logging controls should support both success and failure of specified security events • Ensure logs contain important log event data • Ensure log entries that include untrusted data will not execute as code in the intended log viewing interface or software • Restrict access to logs to only authorized individuals • Utilize a master routine for all logging operations • Do not store sensitive information in logs (e.g. unnecessary system details, session IDs, passwords)

	• Ensure that a mechanism exist to conduct log analysis • Log all input validation failures • Log all authentication attempts, especially failures • Log all access control failures • Log all apparent tampering events, including unexpected changes to state data • Log attempts to connect with invalid or expired session tokens • Log all system exceptions • Log all administrative functions, including changes to the security configuration settings • Log all backend connection failures • Log cryptographic module failures • Use a cryptographic hash function to validate log entry integrity
	🤔For easier memorise: 1. TRUSTED LOGGING - Log securely. Log centrally. Log with integrity. 2. CONTENT RULES - Log what matters, not what can harm you. 3. SAFETY FAILURES - If it fails → log it. 4. RESTRICTION & ACCESS - Logs are sensitive data. Guard them. 5. ANALYSIS & ACTION - Logs must be reviewed, not just collected.
Data protection	• Implement least privilege, restrict users to only the functionality, data and system information that is required to perform their tasks • Protect all cached or temporary copies of sensitive data stored on the server from unauthorized access and purge 清除 those temporary working files as soon as they are no longer required • Encrypt highly sensitive stored information, like authentication verification data, even on the server side. Always use well-vetted 严格验证 algorithms • Protect server-side source code from being downloaded by a

	user • Do not store passwords, connection strings or other sensitive information in clear text or in any non-cryptographically secure manner on the client side • Remove comments in user-accessible production code that may reveal backend system or other sensitive information • Remove unnecessary application and system documentation as this can reveal useful information to attackers • Do not include sensitive information in HTTP GET request parameters • Disable auto complete features on forms expected to contain sensitive information • The application should support the removal of sensitive data when the data is no longer required (e.g. personal information or certain financial information) • Implement appropriate access controls for sensitive data stored on the server. This includes cached data, temporary files and data that should be accessible only be specific system users
	🤔For easier memorise: 1. LEAST PRIVILEGE - If they don't need it → they don't get it 2. SECURE STORAGE - Store encrypted. Never expose. 3. REMOVAL & SANITIZATION - If it's not needed → wipe it. 4. AVOID EXPOSURE - Don't leak. Don't reveal.
Database Security	• Use strongly typed parameterized queries • Utilize input validation and output encoding and be sure to address metacharacters. If these fail, do not run the database command • Ensure that variables are strongly typed • The application should use the lowest possible level of privilege when accessing the database • Use secure credentials for database access

	• Connection strings should not be hard coded within the application. Connection strings should be stored in a separate configuration file on a trusted system and they should be encrypted • Use stored procedures to abstract data access and allow for the removal of permissions to the base tables in the database • Close the connection as soon as possible • Remove or change all default database administrative passwords. Utilize strong passwords / phrases or implement multi-factor authentication • Turn off all unnecessary database functionality (e.g. unnecessary stored procedures or services, utility packages, install only the minimum set of features and options required (surface area reduction)) • Remove unnecessary default vendor content (e.g. sample schemas) • Disable any default accounts that are not required to support business requirements • The application should connect to the database with different credentials for every trust distinction (e.g. user, read-only user, guest, administrators)
	🤔For easier memorise: 1. QUERIES - Safe Queries = Parameters + Stored Procedures 2. VALIDATION & ENCODING - Validated in, safe out 3. ACCESS CONTROL - Each role gets only what ie needs 4. CONFIGURATION - No secrets in code, no defaults allowed 5. HARDENING - Less features = fewer attack paths
File management	• Do not pass user supplied data directly to any function • Require authentication before allowing a file to be uploaded • Limit the type of files that can be uploaded to only those types that are needed for business purposes • Validate uploaded files are the expected type by checking file

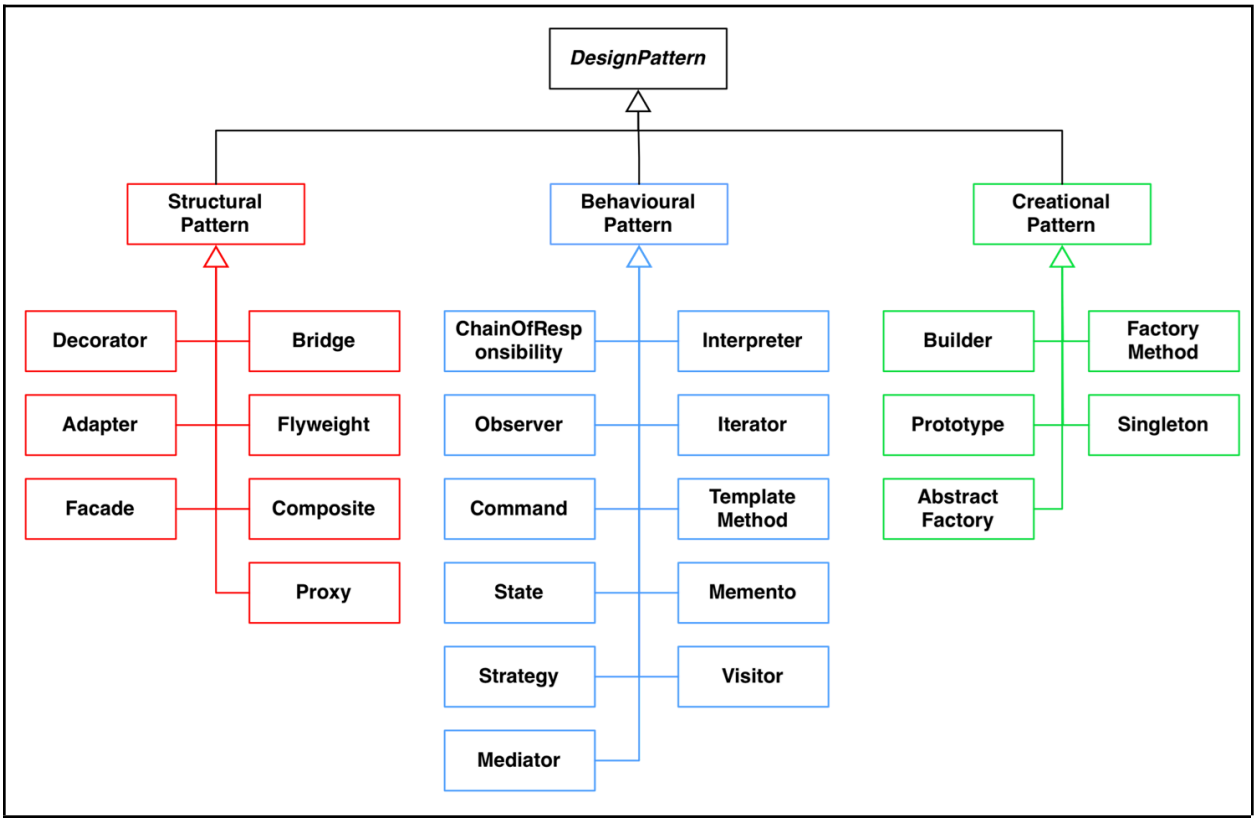
	headers. Checking for file type by extension alone is not sufficient • Do not save files in the same web context as the application. Files should either go to the content server or in the database • Prevent or restrict the uploading of any file that may be interpreted by the web server • Turn off execution privileges on file upload directories • Scan user uploaded files for viruses and malware • When referencing existing files, use a whitelist of allowed file names and types • Do not pass directory or file paths, use index values mapped to pre-defined list of paths • Never send the absolute file path to the client • Ensure application files and resources are read-only
	🤔For easier memorise: 1. UPLOADS CONTROLS - Upload = authenticated + limited + non-executable 2. VALIDATION - Validate by header + whitelist + scan 3. STORAGE LOCATION - Files go somewhere safe - never in the app folder 4. PROTECTION & PERMISSIONS - Read-only + no execute = safe files

C3: Integrative Coding

Pattern Categories

Creational	• Separate the operations of an application from how its objects are created *How is the object created?*
Structural	• Concerned about the composition of objects into larger structures • To provide the possibility of future extension in structure *How are objects connected?*
Behavioral	• Defines how objects interact and how responsibility is distributed among them • Uses inheritance to spread behavior across the subclasses, or aggregation and composition to build complex behavior from simpler components *How do objects talk & act?*

Design Patterns

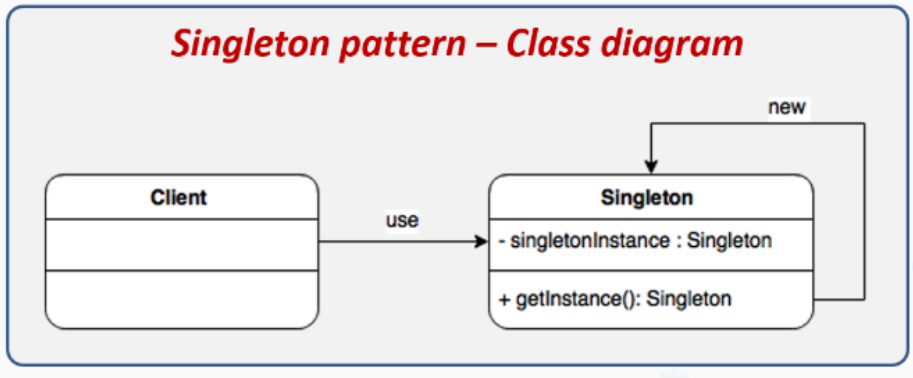


Singleton

Overview	• Singleton Pattern is a **creational design pattern**. • It is one of the **simplest design patterns**. • It ensures that **only one instance of a class is created**. • It provides a **global point of access** to that single instance. • The object can be accessed **without repeatedly instantiating the class**.
Definition	The Singleton Pattern ensures that a class has **only one instance** and provides a **global access point** to that instance.
Design Problem (Why Singleton is needed)	Sometimes, it is necessary to have **exactly one instance** of a class in a system: • Window manager • Print spooler • File system • Configuration manager Creating multiple instances in these cases may lead to

	inconsistency, conflicts or unnecessary resource usage.
Intent of the Singleton Pattern	• To ensure that no more than one instance of a class is created • To provide a single, shared instance that can be accessed globally

Suitable Use Cases	• Configuration Manager
	  ○ Only one configuration file is needed
	  ○ All components share the same settings
	• Logging System
	  ○ A single logger instance writes logs consistently
	• Database Connection Manager
	  ○ Reuses one connection or connection pool
	• Print Spooler
	  ○ Controls access to shared printing resources

Diagram	

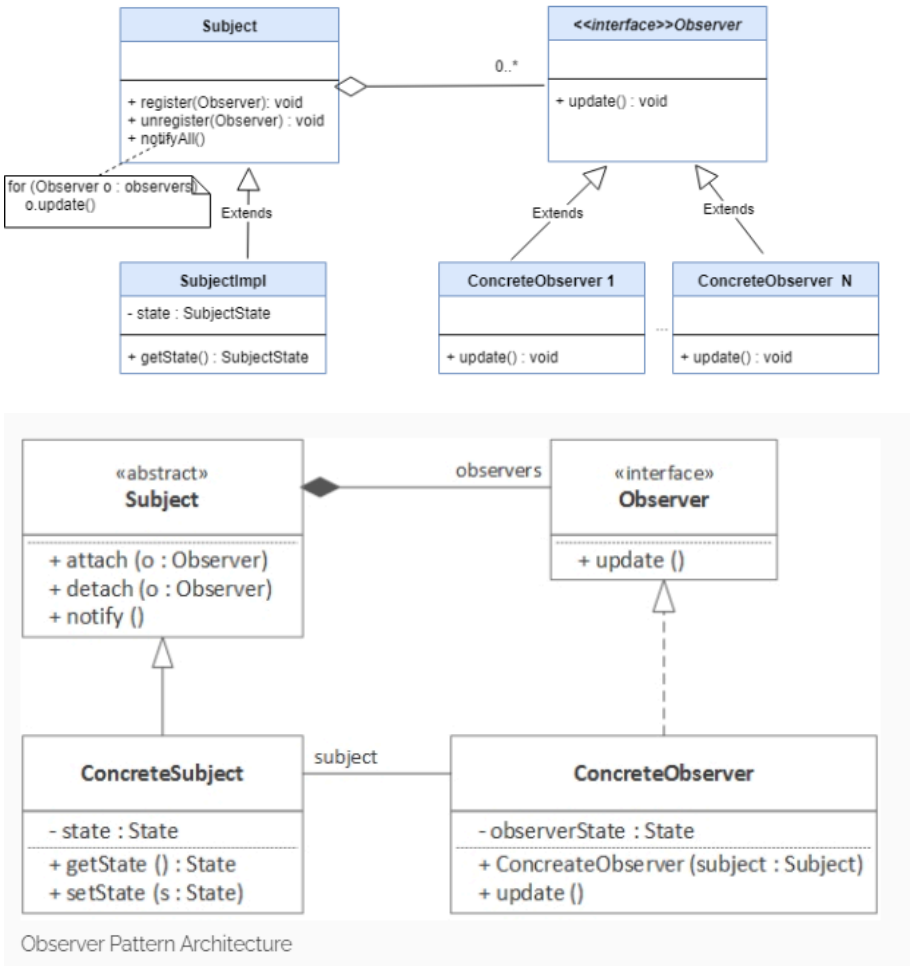
Observer Pattern

Overview	• The Observer Pattern is **behavioral design pattern** • Used to model a **one-to-many dependency** between objects. • When the **subject's state changes**, all **registered observers are notified automatically**.
Definition	The Observer Pattern defines a one-to-many dependency between a subject and its observers such that when the subject changes state, all observers are notified and updated automatically.
Intent	• Keep multiple dependent objects **consistent with a subject** • Allow automatic reaction to state changes

	• **Reduce tight coupling** between objects

Core Roles and Responsibilities	• Subject
	  ◦ Maintains the **state of interest**
	  ◦ Keeps a list of observers
	  ◦ Provides operations to:
	  ▪ `attach()` observers
	  ▪ `detach()` observers
	  ▪ `notify()` observers when state changes
	• Observer
	  ◦ Depends on the subject
	  ◦ Implements an `update()` operation
	  ◦ Updates itself when notified of a change
	• Relationship
	  ◦ **One subject → many observers**
	  ◦ Observers can be **added or removed dynamically**
	  ◦ Subject does **not depend on concrete observer implementations**

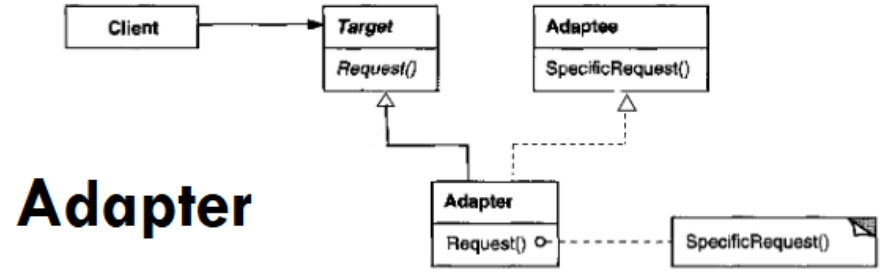
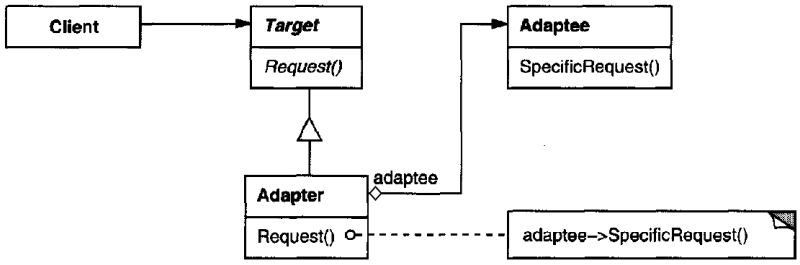
Behavioral Flow	1. Observers **subscribe** to the subject 2. Subject's state changes 3. Subject **notifies all observers** 4. Observers **update themselves**
When to use	• A change in one subject must update **multiple dependent objects** • The number of dependents is **unknown or varies** • Objects should communicate **without tight coupling**
Examples	• Spreadsheet: Data (subject) → charts (observers) • Job recruitment system: Recruiter → job seekers • GUI systems: Event source → event listeners
UML Participants	• **Subject**: declares `attach()`, `detach()`, `notify()` • **ConcreteSubject**: stores state and triggers notification • **Observer**: declares `update()` • **ConcreteObserver**: implements `update()` and syncs state
Advantages	• Loose coupling between objects • Easy to add or remove observers • Supports event-driven design • Follows Open-Closed Principle

Diagram	

Adapter Pattern

Overview	• Adapter Pattern is a **structural design pattern** • Enables **collaboration between classes with incompatible interfaces** • Acts as a **bridge** between a client and an existing class that cannot be used directly • A.k.a. Wrapper Pattern
Definition	The Adapter Pattern converts the interface of a class into another interface that the client expects, allowing classes with incompatible interfaces to work together.
Problem It Solves	• Sometimes, classes need to collaborate but **cannot interact directly** because: ○ They expose **different method names**

	○ They use **different parameter formats** ○ They come from **legacy systems or third-party libraries** ● Modifying the existing class is **not possible or not desirable**
Intent / Purpose	● Enables interaction between **two incompatible interfaces** ● Reuse existing or legacy classes **without modifying their source code** ● Decouple client code from third-party or legacy implementations
Key Participants	● Client ○ The existing system ○ Expects to work with the **Target interface** ● Target ○ Defines the **interface expected by the client** ○ Represents the **intended common functionality** ● Adaptee ○ An existing or external class ○ Provides required functionality ○ Has an **incompatible interface** ● Adapter ○ Implements the **Target interface** ○ Internally uses (wraps) the Adaptee ○ Translates client requests into a format the Adaptee understand
How it works	1. Client **sends a request using the Target interface** **2. Adapter receives the request** 3. Adapter **converts (translates) the request** 4. Adapter **delegates the request to the Adaptee** 5. Result is returned to the client
Types of Adapter Pattern	● Class Adapter ○ Uses **inheritance** ○ Adapter inherits from Adaptee and implements Target ○ Requires multiple inheritance (not supported in many languages) ○ Less flexible ● Object Adapter (Most Common) ○ Uses **composition**

	○ Adapter holds a reference to Adaptee ○ More flexible and safer ○ Avoids unexpected side effects
When to use	• You want to use an **existing class** but its interface does not match • You are integrating **legacy systems** • You are using **third-party libraries** • You want to standardize different interfaces • You want to upgrade or replace components without breaking existing code • You want to mock or stub components during testing • You need cross-platform or cross-API compatibility
Advantages	• Enables reuse of existing code • Avoids modifying legacy or third-party classes • Improves flexibility and maintainability • Supports Open-Closed Principle • Allows building larger systems from incompatible components
Diagram	A class adapter uses multiple inheritance to adapt one interface to another:  Adapter An object adapter relies on object composition: 

Decorator Pattern

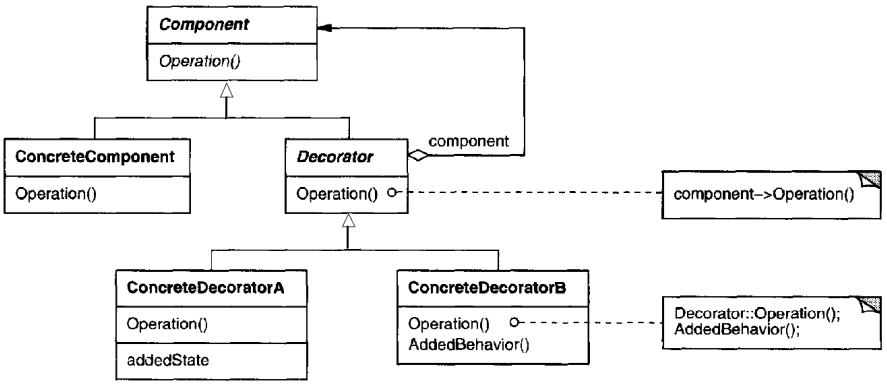
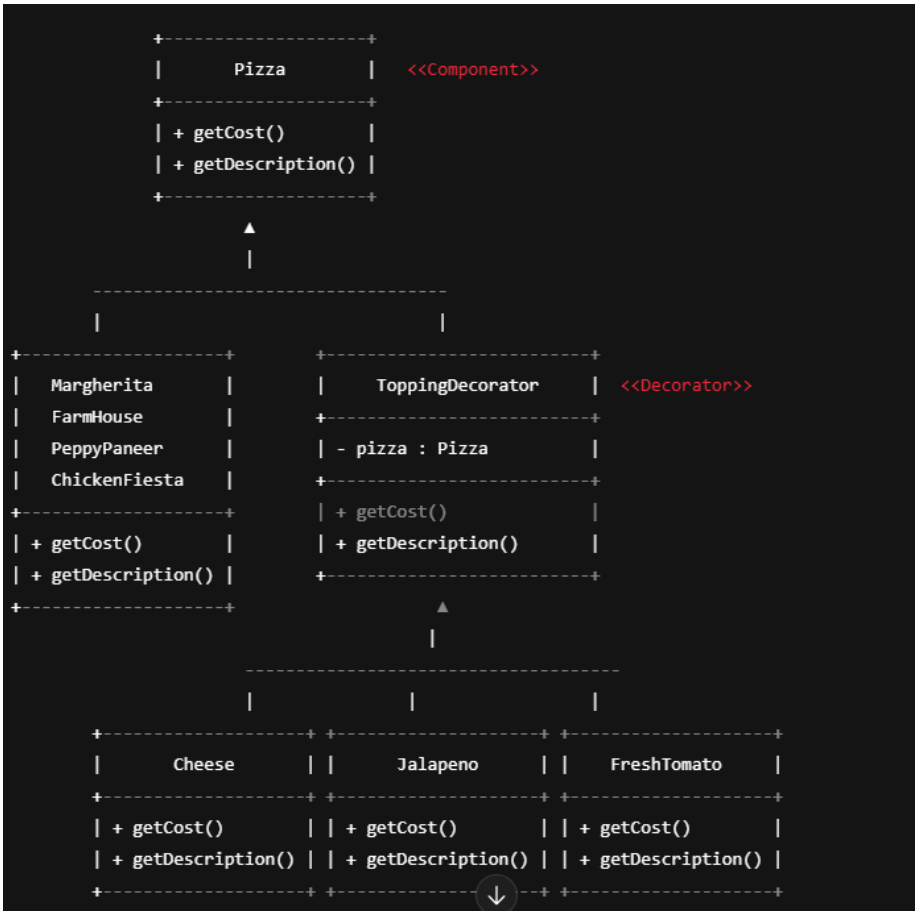
Overview	• Decorator Pattern is a **structural design pattern** • Allows **adding new functionality (responsibilities)** to an object **dynamically at runtime** • The original object's **structure and interface are not modified** • Provides a **flexible alternative to subclassing** for extending behavior • Follows the **Open-Closed Principle**: *open for extension, closed for modification*
Definition	The Decorator Pattern attaches additional responsibilities to an object dynamically by wrapping it without altering the original object.

Problem it solves	• **Subclass Explosion Problem**
	  ○ Using inheritance for every feature combination leads to:
	  ■ Too many subclasses
	  ■ Difficult maintenance
	  ■ Poor scalability
	• **Boolean Flag / Variable Explosion**
	  ○ Adding variables for every new feature:
	  ■ Forces modification of base class
	  ■ Violates Open-Closed Principle
	  ■ Causes irrelevant features to be inherited

Core Idea	• **Wrap** an object inside another object
	• The wrapper:
	  ○ Implements the same interface
	  ○ Adds extra behavior before or after delegating calls
	• Behavior is **composed**, not inherited

Key Participants	• **Component**
	  ○ Defines the common interface
	  ○ Objects can have responsibilities added dynamically
	• **ConcreteComponent**
	  ○ The original object
	  ○ Base functionality to be decorated
	• **Decorator (Abstract)**
	  ○ Implements the same interface as Component
	  ○ Maintains a reference to a Component object

	● **ConcreteDecorator** ○ Adds new behavior or state ○ Calls the wrapped component and extends its behavior
How it works	1. Client creates a base object (ConcreteComponent) 2. Client wraps it with one or more decorators 3. Each decorator: ○ Adds its own responsibility ○ Delegates remaining work to the wrapped object 4. Final behavior is the **combination of all decorators**
Example	<u>Pizza Example</u> ● Roles Mapping ○ Component → Pizza ○ ConcreteComponents → Magherita, FarmHouse, PeppyPaneer, ChickenFiesta ○ Decorator → ToppingDecorator ○ ConcreteDecorator → Cheese, Jalapeno, Paneer, FreshTomato, Barbeque
When to use	● You need to **add responsibilities dynamically** ● Responsibilities may need to be **added or removed** ● Subclassing would cause **class explosion** ● You want to extend behavior **without modifying existing code** ● Behavior should apply to **individual objects**, not the entire class
Advantages	● Avoids subclass explosion ● Supports Open-Closed Principle ● Flexible and extensible ● Behavior can be added or removed at runtime ● Promotes composition over inheritance

Diagram	 With Pizza Example: 

Facade Pattern

Overview	• Facade Pattern is a **structural design pattern** • **Hides the complexity** of a subsystem

	• Provides a **single, simplified interface** for clients to access the system • Clients interact with the **facade**, not directly with subsystem classes
Definition	The Facade Pattern provides a unified, higher-level interface to a set of interfaces in a subsystem, making the subsystem easier to use by hiding its internal complexity.
Intent	• Provide a **unified and simplified interface** to a complex subsystem • Reduce dependencies between clients and subsystem classes • Improve system **usability and maintainability**

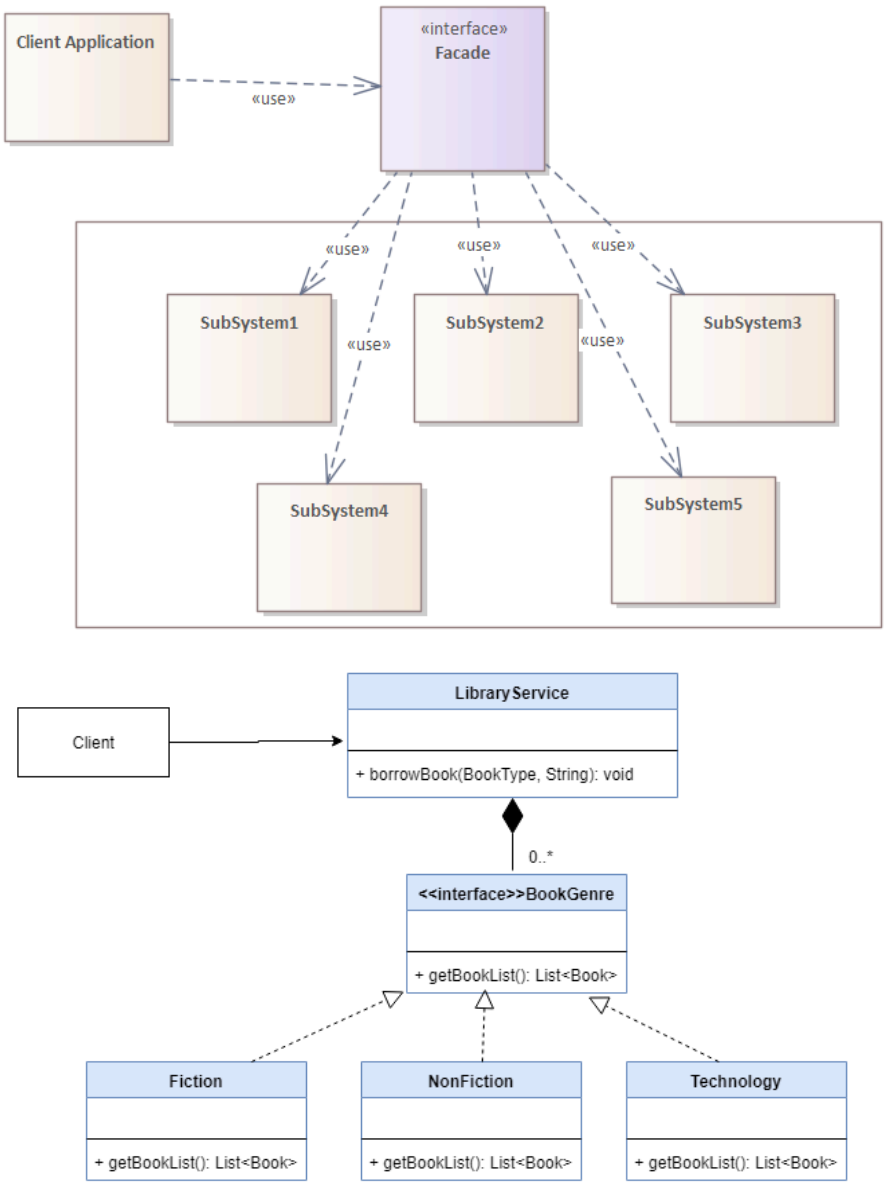
Motivation	• Large systems are often divided into **multiple subsystems**
	• Direct interaction with many subsystem classes:
	  ○ Increases complexity
	  ○ Creates tight coupling
	  ○ Makes the system harder to maintain
	• Facade introduces a **single entry point** to the subsystem, reducing complexity

Core Idea	• A **Facade class**:
	  ○ Knows which subsystem classes handle a request
	  ○ Delegates 委托 client requests to the appropriate subsystem objects
	• Subsystem classes:
	  ○ Perform the actual work
	  ○ Are **unaware of the facade**
	  ○ Do not reference the facade

Key Participants	• **Facade**
	  ○ Provides simplified methods for the client
	  ○ Coordinates calls to multiple subsystem classes
	  ○ Acts as an intermediary between client and subsystems
	• **Subsystem Classes**
	  ○ Implement the core system functionality
	  ○ Handle tasks delegated by the facade
	  ○ Can still be accessed directly if needed

How it works	1. Client calls a method on the Facade

	2. Facade internally calls multiple subsystem methods 3. Subsystems perform their tasks 4. Facade returns the result to the client
When to use	● You want to provide a **simple interface** to a complex subsystem ● There are many dependencies between clients and subsystem classes ● You want to **decouple clients** from subsystem implementations ● You want to **layer subsystems** and define clear entry points ● Subsystems depend on each other and need simplified communication
Advantages	● Reduces complexity for clients ● Promotes **loose coupling** ● Improves readability and maintainability ● Supports **Principle of Least Knowledge** ● Subsystems can change without affecting clients ● Does not prevent direct access to subsystems if required
Example	● Hotel / Restaurant: ○ Client: Customer ○ Facade: Hotel keeper ○ Subsystems: Different restaurants ○ Customer talks only to the hotel keeper, who handles everything internally ● Customer Service: ○ Clients call one customer service number ○ Customer service representative acts as a Facade ○ Internally communicates with: ■ Order fulfillment ■ Billing ■ Shipping

Diagram	

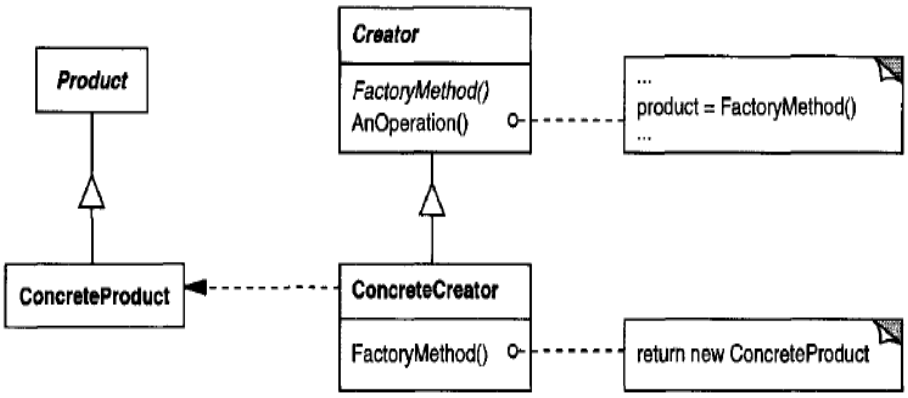
Factory Pattern

Overview	• Factory Method Pattern is a **creational design pattern** • One of the most commonly used patterns (especially in Java) • **Separates object creation logic from the client code** • Objects are created without exposing the instantiation logic • The client interacts with objects through a common interface

Definition	The Factory Method Pattern defines an interface for creating an object, but lets subclasses decide which class to instantiate, thereby deferring instantiation to subclasses.
Intent	• Define an interface for object creation • Defer the decision of which concrete class to create to subclasses • Refer to newly created objects through a common interface • Centralize and encapsulate object creation logic
Problem it solves	Without Factory Method: • Client code uses new directly • Object creation logic is scattered • Code becomes tightly coupled to concrete classes • Adding new product types requires modifying existing code • Violates the Open-Closed Principle
Core Idea	• Encapsulate object creation in a factory method • Client code calls the factory method instead of using new • Subclasses decide which concrete object to instantiate • Promotes loose coupling and flexibility

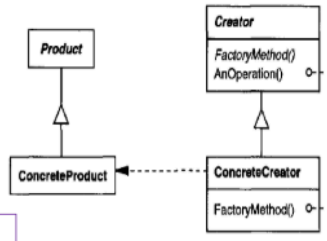
Key Participants	• Product
	  ○ Interface or abstract class
	  ○ Define the type of objects created by the factory
	• ConcreteProduct
	  ○ Implements or extends Product
	  ○ Represents actual objects created
	• Creator
	  ○ Declares the factory method
	  ○ May define default behavior that uses the product
	  ○ Refers to Product through its interface
	• ConcreteCreator
	  ○ Overrides the factory method
	  ○ Returns a specific ConcreteProduct instance

How it works	1. Client calls a method in Creator 2. Creator invokes the factory method 3. ConcreteCreator creates a ConcreteProduct 4. Product is returned via the Product interface
When to use	• A class cannot anticipate the class of objects it must create • A class want its subclasses to specify the objects it creates

	• Object creation responsibility should be delegated • You want to localize knowledge of which subclass is instantiated • You need to support multiple variations • New product types may be added in the future
Advantages	• Encapsulates object creation • Promotes loose coupling • Follows Open-Closed Principle • Improves maintainability and scalability • Centralizes instantiation logic • Simplifies unit testing and dependency injection
Example	<u>Pizza Factory</u> • Client: Customer • Factory: Pizza Factory • Product: Pizza • ConcreteProducts: Margherita, FarmHouse, PeppyPaneer Customer orders a pizza → Factory decides which pizza to create → Customer does not care how it is made.
Diagram	

Factory Method Example

```php
abstract class Connection {

  function __construct() {
  }

  public function description() {
    return "Generic";
  }
}
```




```php
require_once 'OracleConnection.php';
require_once 'SqlServerConnection.php';
require_once 'MySqlConnection.php';

class DBConnectionFactory {
  protected $type;

  function __construct($type) {
    $this->type = $type;
  }

  function createConnection() {
    if ($this->type == "Oracle")
      return new OracleConnection();
    elseif ($this->type == "SQL Server")
      return new SqlServerConnection();
    else
      return new MySqlConnection();
  }
}
```

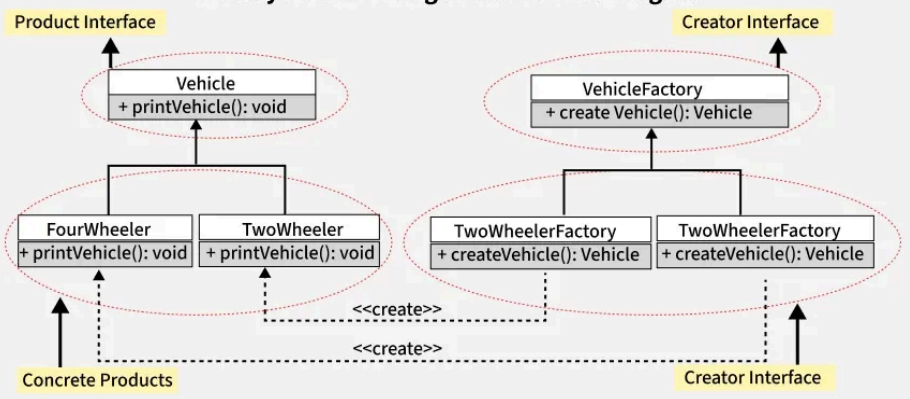
```php
require_once 'Connection.php';

class OracleConnection extends Connection {
  function __construct() {
  }

  public function description() {
    return "Oracle";
  }
}
```

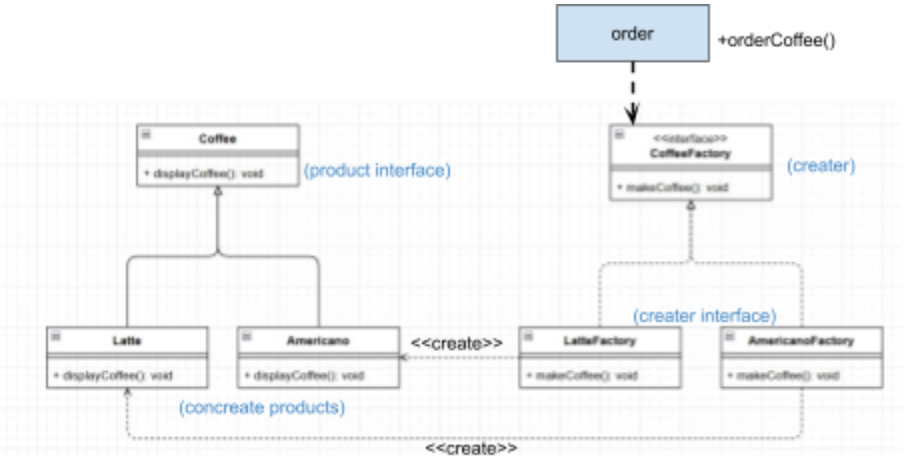
```php
require_once 'Connection.php';

class MySqlConnection extends Connection{
  function __construct() {
  }

  public function description() {
    return "MySQL";
  }
}
```

```php
class SqlServerConnection extends Connection {

  function __construct() {

  }

  public function description() {
    return "SQL Server";
  }

}
```

Refer to Chapter3**factorymethod** folder

Factory Method Design Pattern Class Diagram





Proxy Pattern

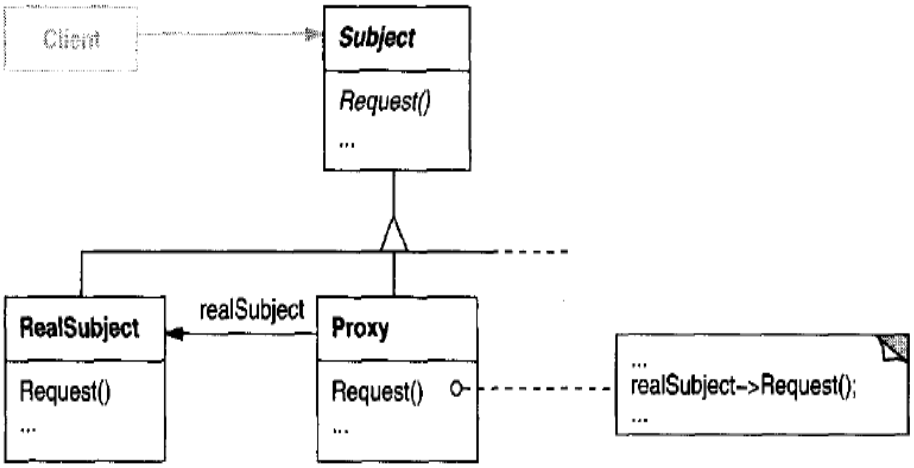
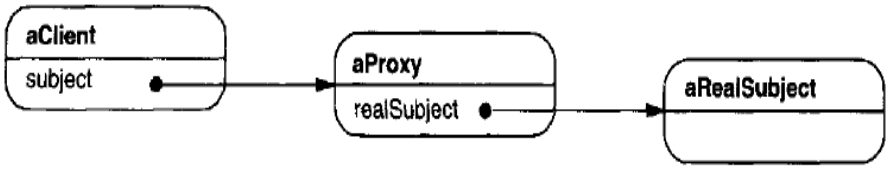
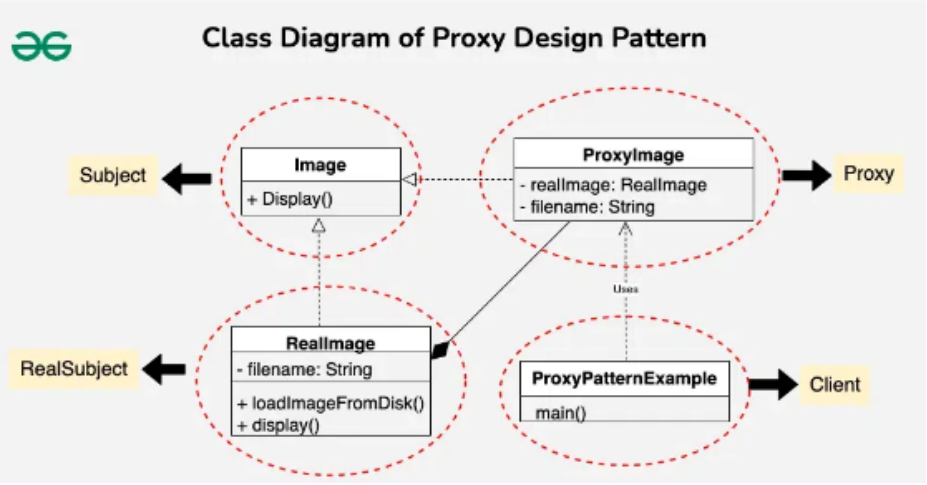
Overview	● Proxy Pattern is a **structural design pattern** ● Provides a **surrogate (placeholder)** 代理（占位符）for another object ● The proxy **controls access** to the real object ● Clients interact with the **proxy**, not directly with the real objects
Definition	The Proxy Pattern provides a substitute or placeholder for another object to control access to it.

Core Idea	● A proxy **represents another object**.
	● The proxy and real object share the **same interface**.
	● The proxy may:
	  ○ Control access
	  ○ Delay object creation
	  ○ Perform extra work before or after forwarding a request

Intent	● Control access to an object ● Add extra behavior **without changing the real object** ● Support **lazy initialization, access control or remote access** ● Improve performance or security

Key Participants	● **Subject**
	  ○ Interface or abstract class
	  ○ Defines the common operations
	  ○ Used by both Proxy and RealSubject
	● **RealSubject**
	  ○ The **actual object**
	  ○ Contains real business logic or resource
	  ○ Performs the real work
	● **Proxy**
	  ○ Maintains a reference to RealSubject
	  ○ Implements the same interface as Subject
	  ○ Controls access to RealSubject
	  ○ May create or destroy RealSubject
	● **Client**
	  ○ Interacts with the Subject interface
	  ○ Does not know whether it is using a proxy or the real object

<table>
<tr><td>How it works</td><td colspan="2">Client calls a method on the proxy
Proxy performs additional checks or actions
Proxy forwards the request to the real object
Real object processes the request
Result is returned to the client</td></tr>
<tr><td rowspan="3">Types of Proxy Pattern</td><td>Remote Proxy</td><td>Acts as a local representative of a remote object
Used in distributed systems
Handles network communication
May cache data to reduce remote calls
Example:
ATM system accessing bank data on a remote server
RPC / CORBA stubs</td></tr>
<tr><td>Virtual Proxy</td><td>Acts as a placeholder for expensive objects
Uses lazy initialization
Creates the real object only when needed
Improves performance and memory usage
Example:
Image loaded only when <code>display()</code> is called</td></tr>
<tr><td>Protection Proxy</td><td>Controls access based on permissions
Acts as an authentication / authorization layer
Ensures only authorized clients can access the real object
Example:
Grade system accessed by administrators, teachers and students</td></tr>
<tr><td>Examples</td><td colspan="2">Credit card / cheque → proxy for bank account
ATM card → proxy for cash
Receptionist → proxy for accessing staff</td></tr>
<tr><td>Advantages</td><td colspan="2">Improves performance (lazy loading, caching)
Enhances security (access control)
Supports distributed systems
Encapsulates housekeeping logic
Transparent to the client</td></tr>
</table>

When to use	• Access to an object must be controlled • Object creation is expensive • Object resides in a remote location • Security or logging is required • Lazy initialization is needed
Diagram	 a possible object diagram of a proxy structure at run-time:  

Strategy Pattern

Overview	• Strategy Pattern is a **behavioral design pattern** • Focuses on **algorithms and behavior selection** • Allows a class's **behavior (algorithm) to be changed at runtime** • Different algorithms are encapsulated as **separate strategy objects**
Definition	The Strategy Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable so that the algorithm can vary independently from the clients that use it.
Intent	• Define multiple algorithms for a task • Allow the algorithm to be **selected at runtime** • Avoid hard-coded conditional logic • Promote **flexibility and reusability**
Core Idea	• Take a class that performs a task in many different ways • Extract each algorithm into a **separate strategy class** • Let a **Context object** delegate the work to a selected strategy • **Behavior changes by changing the strategy object, not the context code**

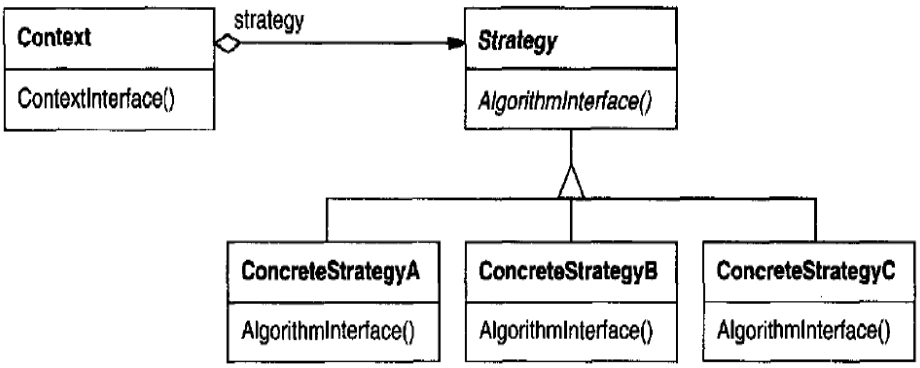
Key Participants	• **Strategy**
	  ◦ Declares a **common interface** for all algorithms
	  ◦ Context uses this interface to execute the algorithm
	• **ConcreteStrategy**
	  ◦ Implements the algorithm defined by Strategy
	  ◦ Each class represents **one specific behavior**
	• **Context**
	  ◦ Maintains a reference to a Strategy object
	  ◦ Delegates algorithm execution to the Strategy
	  ◦ Can change strategy at runtime

How it works	1. Client selects or sets a Strategy 2. Context stores the Strategy reference 3. Context delegates algorithm execution to the Strategy 4. Changing the Strategy changes the behavior
When to use	• Many related classes differ **only in behavior** • You need **different variants of an algorithm** • A class has many `if-else` or `switch` statements

	• You want to hide algorithm details from clients • Algorithms should be **interchangeable**
Advantages	• Eliminates large conditional statements • Supports Open-Closed Principle • Makes algorithms reusable • Improves code readability and maintainability

Examples	• Discount Calculation
	  ◦ Different discount rules:
	  ▪ Flat discount
	  ▪ Percentage discount
	  ▪ Membership discount
	  ◦ Strategy selected during checkout
	• Sorting Algorithms
	  ◦ Different sorting methods:
	  ▪ Quick sort
	  ▪ Merge sort
	  ▪ Bubble sort
	  ◦ Strategy chosen based on data size or performance needs
	• Compression Algorithms
	  ◦ Different compression techniques:
	  ▪ ZIP
	  ▪ RAR
	  ▪ GZIP
	  ◦ Strategy selected based on file type or user preference

Strategy vs Inheritance	• **Inheritance**:
	  ◦ Behavior fixed at compile time
	  ◦ Applies to all instances of a class
	• **Strategy**:
	  ◦ Behavior chosen at runtime
	  ◦ Applies to individual objects
	* *Strategy uses **composition over inheritance***

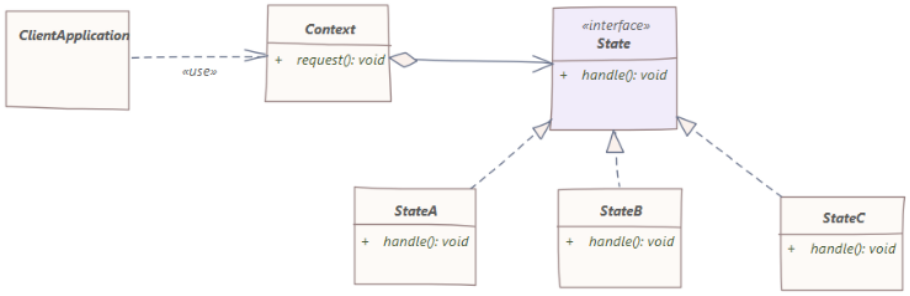
Diagram	

State Pattern

Overview	• The State Pattern is a **behavioral design pattern** • Allows an object to **change its behavior when its internal state changes** • The object appears to change its class at runtime by **delegating behavior to state objects**
Definition	The State Pattern allows an object to alter its behavior when its internal state changes, by encapsulating state-specific behavior in separate state classes
Intent	• Represent **state-dependent behavior explicitly** • Eliminate large conditional statements (`if / else`, `switch`) • Make state transitions **clear, maintainable and extensible**
Core Idea	• An object's behavior depends on its **current state** • Each state is represented by a **separate class** • The **context** delegates requests to the current state object • Changing the state object changes the behavior

Key Participants	• **Context**
	  ◦ Maintains a reference to a **State** object
	  ◦ Delegates behavior to the current state
	  ◦ May allow states to change the current state
	• **State (Interface / Abstract Class)**
	  ◦ Declares state-specific behavior methods (e.g. `handle()`)
	• **ConcreteState**
	  ◦ Implements behavior associated with a specific state
	  ◦ May trigger transitions to other states

How it works	1. Client calls a method on the Context 2. Context forwards the request to the current State 3. State executes behavior specific to that state 4. State may change the Context's state 5. Future behavior changes automatically
When to use	● An object's behavior **changes based on its state** ● There are **many states and transitions** ● You want to avoid large conditional logic ● State transitions should be **explicit and manageable**
Examples	● E-Commerce Order System ○ States: ■ Order Created ■ Delivered ■ Cancelled ■ Returned ○ Each state defines: ■ What actions are allowed ■ How transitions occur ● Smartphone Buttons ○ Locked → shows unlock screen ○ Unlocked → performs actions ○ Low battery → shows charging screen ○ Same button, different behavior, based on state ● User Account Management ○ States: ■ Active ■ Suspended ■ Deactivated ○ Each state controls permissions and actions ● Document Workflow ○ States: ■ Draft ■ Review ■ Approved ■ Published ○ Actions depend on current document state
Advantages	● Removes complex conditional logic ● Improves readability and maintainability ● Makes state transitions explicit ● Follows Open-Closed Principle

Diagram	

C4A: Data Encoding and XML

UTF-8

Definition	• UTF-8 (8-bit Unicode Transformation Format) is a **Unicode character encoding**. • It is a variable-length encoding, using 1 to 4 bytes per character.

How it works	• Characters with code points 0-127 are stored in 1 byte
	• Characters beyond ASCII use 2, 3, or 4 bytes.,
	• **Supports all Unicode characters**, including:
	  ○ Accented letters (é, ñ)
	  ○ Non-Latin scripts (Chinese, Arabic)
	  ○ Symbols and emoji

Why UTF-8 is widely used	• Most widely used encoding on the **web and XML documents** • Efficient for English text • Platform-independent • Fully supports **internationalization**

Extensible Markup Language (XML)*

- A markup language that is extensible
- Can be modified according to the needs of the data being recorded
- Describes the structure and content of any machine-readable information
- Device-independent and system-independent
- Used to create vocabularies of other markup languages
- Structure the data

- Storage purpose
- To share / transfer data from one system to another

XML Syntax Rules

- Every XML element must have a **closing tag** (self-closing tag is permitted)
- XML tags are **case sensitive**
- XML elements must be **properly nested** (All elements can have child (sub)elements)
- Every XML document must have a **root element**
- XML elements can **have attributes in name-value pairs** (Each attribute value must be quoted)

Example 1:

Product Name	Manufacturer	Price
Purrfect Gift Basket	ABC Co	36.00
Stationery Set	Write Well	20.50

```xml
<?xml version="1.0" encoding="UTF-8"?>
<products>
    <product>
        <name>Purrfect Gift Basket</name>
        <manufacturer>ABC Co</manufacturer>
        <price>36.00</price>
    </product>
    <product>
        <name>Stationery Set</name>
        <manufacturer>Write Well</manufacturer>
        <price>20.50</price>
    </product>
</products>
```

Example 2:

Product Name	Items	Price
Purrfect Gift Basket	Pillow	36.00
	Blanket	
Stationery Set	Pen	20.50
	Notebook	
	Ruler	

```xml
<?xml version="1.0" encoding="UTF-8"?>
<products>
	<product>
		<name>Purrfect Gift Basket</name>
		<items>
			<item>Pillow</item>
			<item>Blanket</item>
		</items>
		<price>36.00</price>
	</product>
	<product>
		<name>Stationery Set</name>
		<items>
			<item>Pen</item>
			<item>Notebook</item>
			<item>Ruler</item>
		</items>
		<price>20.50</price>
	</product>
</products>
```

Document Type Definition (DTD)*

- A collection of rules that define the content and structure of an XML document
- Used in conjunction with an XML parser that supports data validation

Application

- Ensure that all required elements are present in the document
- Prevent undefined elements from being used in the document
- Enforce a specific data structure on document content
- Specify the use of element attributes and define their permissible values
- Define default values for attributes

Limitations:

- Lack of data types
- Does not support namespaces
- Uses a different syntax from XML (need to learn additional syntax)

Element Declarations

- Element declaration for each element type: `<!ELEMENT element_name content_specification>`
- `content_specification` can be:
  - `(#PCDATA)` → parsed character data
  - `(child)` → one child element
  - `(c1, .., cn)` → a sequence of child elements c1..cn
  - `(c1|..|cn)` → one of the elements c1..cn
- For each component, possible counts can be specified:
  - `c` → exactly one such element
  - `c+` → one or more

    - `c*` → zero or more
    - `c?` → zero or one
- Variations in element declarations:
    - Arbitrary combinations: `<!ELEMENT f ((a|b), c+, (d|3))>`
    - Elements with mixed content: `<!ELEMENT text (#PCDATA|index|cite))*>`
    - Elements with empty content: `<!ELEMENT image EMPTY>`
    - Elements with arbitrary content: `<!ELEMENT thesis ANY>`

Attributes

- Attribute list declaration (e.g. A bakery have an attribute *kind*, a character string describing the kind of bakery (e.g. "Gourmet", "Wedding", "Commercial"):

  `<!ELEMENT bakery (name, cake+)>`

  `<!ATTLIST bakery kind CDATA #IMPLIED>`
- `CDATA` → Character string type; no tags
- `#IMPLIED` → Attribute is optional
- `#REQUIRED` → Attribute is compulsory

IDs and IDREFs

- To allow an element to refer to another element with an `ID` attribute, give the element an attribute of type `IDREF`
- To allow an element to refer to any number of other elements, give the element an attribute of type `IDREFS`

Example 1:

bakeries.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE bakeries SYSTEM 'bakeries.dtd'>
<bakeries>
      <bakery>
```

```
            <name>Beautiful Wedding Cakes</name>
            <cake>
                  <name>3-Tier Wedding Cake</name>
                  <price>RM150.00</price>
            </cake>
            <cake>
                  <name>2-Tier Wedding Cake</name>
                  <price>RM100.00</price>
            </cake>
      </bakery>
      <bakery kind="Commercial">
            <name>CakesRUs</name>
            <cake>
                  <name>Butter Cake 1kg</name>
                  <price>RM30.00</price>
            </cake>
      </bakery>
</bakeries>
```

bakeries.dtd

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT bakeries (bakery+)>
<!ELEMENT bakery (name, cake+)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT cake (name, price)>
<!ELEMENT price (#PCDATA)>
<!ATTLIST bakery kind CDATA #IMPLIED>
```

Example 2:

menuF.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE menu SYSTEM 'menu.dtd'>
<menu>
      <item>
            <code>1001</code>
```

```
            <name>Nasi Lemak Ayam</name>
            <price currency="MYR">RM10.90</price>
            <price currency="USD">$2.50</price>
            <price currency="EUR">&#8364;2.20</price>
      </item>
      <item>
            <code>1002</code>
            <name>Seafood Fried Rice</name>
            <price currency="MYR">RM20.00</price>
            <price currency="USD">$4.70</price>
            <price currency="EUR">&#8364;4.10</price>
      </item>
</menu>
```

menu.dtd

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT menu (item+)>
<!ELEMENT item (code, name, price+)>
<!ELEMENT code (#PCDATA)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT price (#PCDATA)>
<!ATTLIST price currency CDATA #REQUIRED>
```

Example 3:

members.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE members SYSTEM 'members.dtd'>
<members>
      <member id="F1234">
            <name>Fred Flinstone</name>
            <age>43</age>
      </member>
      <member id="A5678" spouse_id="F1234">
            <name>Wilma Amber</name>
            <age>38</age>
      </member>
```

```
      <member id="F8888" parent_id="F1234 A5678">
            <name>Pebbles Flinstone</name>
            <age>5</age>
      </member>
</members>
```

members.dtd

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT members (member+)>
<!ELEMENT member (name, age)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT age (#PCDATA)>
<!ATTLIST member
      id ID #REQUIRED
      spouse_id IDREF #IMPLIED
      parent_id IDREFS #IMPLIED
>
```

Example 4:

menuG.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE menu SYSTEM 'menuG.dtd'>
<menu>
      <item code="M1001">
            <name>Nasi Lemak Ayam</name>
            <price currency="MYR">10.90</price>
            <combo paired_item="C1002" price="25.00"></combo>
      </item>
      <item code="C1002">
            <name>Seafood Fried Rice</name>
            <price currency="MYR">20.00</price>
            <combo paired_item="D1003" price="21.00" />
      </item>
      <item code="D1003">
            <name>Ice Lemon Tea</name>
            <price currency="MYR">5.00</price>
```

```
      </item>
</menu>
```

menuG.dtd

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT menu (item+)>

<!ELEMENT item (name, price, combo?)>
<!ATTLIST item code ID #REQUIRED>

<!ELEMENT name (#PCDATA)>

<!ELEMENT price (#PCDATA)>
<!ATTLIST price currency CDATA #REQUIRED>

<!ELEMENT combo EMPTY>
<!ATTLIST combo
      paired_item IDREF #REQUIRED
      price CDATA #REQUIRED
>
```

XML Schemas*

- XML-based alternative to DTD
- Describes the structure of an XML document (validate)
- 3 basic schema designs:
    - Flat Catalog design (Salami Slice design)
    - Russian Doll design
    - Venetian Blind design

xs:element

- Provide definition for an XML element
- Has attributes name and type where

    - `name` - the tag-name of the element being defined
    - `type` - the type of the element which may be
        - an XML-schema type (e.g. `xs:string`) or
        - a custom type defined in the document itself
- Common XML-schema type:
    - `xs:string`
    - `xs:integer`
    - `xs:float` (小数点后多6-7位)
    - `xs:decimal` (unlimited)

xs:complexType & xs:sequence

- Describe elements that consist of subelements
    - Has attribute `name` that gives a name to the type
- Typical subelement is `xs:sequence`
    - Has a sequence of xs:element subelements
    - Use `minOccurs` and `maxOccurs` attributes to indicate the number of occurrences of `xs:element`
    - The default for `minOccurs` and `maxOccurs` is **1**
        - `<xs:element name="item" type="itemType" maxOccurs="unbounded" />`:**unbounded - one to many**

xs:attribute

- Used within complex type to indicate attributes of elements of that type
- Has attributes **`name`**, **`type`** and **`use`** where
    - `name` and `type` - similar as for xs:element
    - **`use`** - either `"required"` or `"optional"`

Example:

```
<xs:complexType name="itemType">
```

```
      <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="code" type="xs:integer"/>
            <xs:element name="price" type="priceType"
maxOccurs="unbounded"/>
      </xs:sequence>
</xs:complexType>

<xs:complexType name="priceType">
      <xs:simpleContent>
            <xs:extension base="xs:float">
                  <xs:attribute name="currency" type="xs:string"
use="required" />
            </xs:extension>
      </xs:simpleContent>
</xs:complexType>
```

Restricted Simple Type

- `xs:simpleType`（里面没有塞type 跟use）can describe **enumerations** and **range-restricted** base types
    - Has attribute `name`
- `xs:enumeration` is a subelement

xs:restriction

- Attribute `base` specifies the simple type to be restricted (e.g. `xs:integer`)
- To specify lower and upper bounds on a numerical range, use the attribute
    - `xs:minInclusive` or `xs:minExclusive`
    - `xs:maxInclusive` or `xs:maxExclusive`
- `xs:enumeration` is a subelement with the attribute value that allows enumerated types

Example 1 - Restriction with enumeration: 几个选一个做老婆

```
<xs:simpleType name="licenseType">
      <xs:restriction base="xs:string">
```

```
            <xs:enumeration value="Learner"/>
            <xs:enumeration value="Probationary"/>
            <xs:enumeration value="Competent"/>
      </xs:restriction>
</xs:simpleType>
```

Example 2 - Restriction with range: 一个老婆的数值

```
<xs:simpleType name="fees">
      <xs:restriction
            base="xs:float"
            minInclusive="30.00"
            maxExclusive="151.00"
      >
      </xs:restriction>
</xs:simpleType>
```

Example 1:

menuH.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<menu
      xmlns:xs='http://www.w3.org/2001/XMLSchema=instance'
      xsi:noNamespaceSchemaLocation='menu.xsd'>
      <item>
            <name>Nasi Lemak Ayam</name>
            <code>1001</code>
            <price>10.90</price>
      </item>
      <item>
            <name>Chicken Lasagne</name>
            <code>1002</code>
            <price>12.90</price>
      </item>
</menu>
```

menu.xsd

```
<?xml version="1.0"?>
<xs:schema version="1.0"
```

```
      xmlns:xs="http://www.w3.org/2001/XMLSchema"
      elementFormDefault="qualified"
>
      <xs:element name="menu">
            <xs:complexType>
                  <xs:sequence>
                        <xs:element name="item" type="itemType"
maxOccurs="unbounded" />
                  </xs:sequence>
            </xs:complexType>
      </xs:element>

      <xs:complexType name="itemType">
            <xs:sequence>
                  <xs:element name="name" type="xs:string" />
                  <xs:element name="code" type="xs:integer" />
                  <xs:element name="price" type="xs:float" />
            </xs:sequence>
      </xs:complexType>
</xs:schema>
```

Example 2:

stationeries.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<Stationeries>
      <Stationery>
            <ItemID>1</ItemID>
            <Name>Pen</Name>
            <Category>Writing</Category>
            <Price>1.50</Price>
      </Stationery>
      <Stationery>
            <ItemID>2</ItemID>
            <Name>Pencil</Name>
            <Category>Writing</Category>
            <Price>0.50</Price>
      </Stationery>
</Stationeries>
```

stationeries.xsd (version 1: Russian Doll Design)

```
<?xml version="1.0"?>
<xs:schema version="1.0"
	xmlns:xs="http://www.w3.org/2001/XMLSchema"
	elementFormDefault="qualified"
>
	<xs:element name="Stationeries">
		<xs:complexType>
			<xs:sequence>
				<xs:element name="Stationery">
					<xs:complexType>
						<xs:sequence>
							<xs:element name="ItemID" type="xs:integer"/>
							<xs:element name="Name" type="xs:string"/>
							<xs:element name="Category" type="xs:string"/>
							<xs:element name="Price" type="xs:float"/>
						<xs:sequence>
					</xs:complexType>
				</xs:element>
			</xs:sequence>
		</xs:complexType>
	</xs:element>
</xs:schema>
```

stationeries.xsd (version 2: Salami Slice Design)

```
<?xml version="1.0"?>
<xs:schema version="1.0"
	xmlns:xs="http://www.w3.org/2001/XMLSchema"
	elementFormDefault="qualified"
>
	<xs:element name="Stationeries">
		<xs:complexType>
			<xs:sequence>
				<xs:element name="Stationery"
type="StationeryType" maxOccurs="unbounded"/>
			</xs:sequence>
		</xs:complexType>
	</xs:element>

	<xs:complexType name="StationeryType">
		<xs:sequence>
			<xs:element name="ItemID" type="xs:integer"/>
			<xs:element name="Name" type="xs:string"/>
			<xs:element name="Category" type="xs:string"/>
			<xs:element name="Price" type="xs:float"/>
		</xs:sequence>
	</xs:complexType>
</xs:schema>
```

C4B: XML Parsing, XPath and XSLT

XPath*

- XML Path Language
- Access and navigate the contents of a DOM tree
- Extract data from XML documents to use it with various applications

`element` nodes	• Contain everything from the start tag to the end tag. This may be data or sub-elements, as well as any attribute nodes • e.g. `<title>Dark Side of the Moon</title>`
`attribute` nodes	• Contain the complete attribute • e.g. `country="UK"`
`text` nodes	• Contains the text of an element or attribute • e.g. `"UK"`

- XPath can translate the XML document's hierarchical structure into an expression called a **location path**
- The location path
  - References a node set (collection of nodes) or a specific node from the source XML document
  - Can be either in **absolute** (starting from the root node) or **relative** terms

Path Expressions Syntax

`/`	2 Meanings: • at the beginning of an XPath expression, / represents the **root node** of the document (The root node contains the root element)

	• / between element names represents a **parent-child** relationship
`//`	• Represents an ancestor-descendant relationship • Can be used as a short cut to *'all titles'* `//title` rather than `/catalog/cd/title`
`@`	• Marks an attribute
`[condition]`	• Specifies a condition • e.g. `[price = 10]`

Expressions

- 2 kinds of expression, returning either
  - a **set of nodes** ("node sets") or
  - a **value** (e.g. number, string, boolean)
- To get the text, use **text()** function, e.g. /catalog/cd/title/text()
  - This return all the title text nodes
  - Not necessary to use the text function in conditions

Aggregation Functions

- count result is always a number
  - e.g. count(//artist)
- sum result is only a number if every node in the argument set can be cast as a number
  - e.g. sum(//cd[price>10]/price)
- Others functions: min, max, and avg

Example 1:

```
<homelist>
      <home id="1">
            <hname>Rose Cottage</hname>
            <location>Inverness</location>
            <url>rosecottage@homes.co.uk</url>
            <contactdetails>
                  <cname>John Smith</cname>
                  <phone>0131 123 1234</phone>
                  <email>jsmith@hotmail.com</email>
            </contactdetails>
            <contactdetails>
                  <cname>Tim Smith</cname>
                  <phone>0131 123 4321</phone>
            </contactdetails>
      </home>
      ...
</homelist>
```

Question:

Find the phone number of homes in Inverness when John Smith is the contact name

Answer:

```
//home[location="Inverness"]/contactdetails[cname="John Smith"]/phone/text()
```

Question:

Find the location of homes with the contact name "John Smith"

Answer:

```
//home[contactdetails/cname="John Smith"]/location/text()
```

Question:

Find the names of homes which have a url

Answer:

```
//home[url]/hname
```

Example 2:

```
<?xml version="1.0" encoding="UTF-8"?>
<catalog>
	<cd country="UK">
		<title>Dark Side of the Moon</title>
		<artist>Pink Floyd</artist>
		<price>10.90</price>
	</cd>
	<cd country="UK">
		<title>Space Oddity</title>
		<artist>David Bowie</artist>
		<price>9.90</price>
	</cd>
	<cd country="USA">
		<title>Aretha: Lady Soul</title>
		<artist>Aretha Franklin</artist>
		<price>9.90</price>
	</cd>
</catalog>
```

Question:

The value of the price of the CD titled "Dark Side of the Moon"

Answer:

```
/catalog/cd[title="Dark Side of the Moon"]/price/text()
//cd[title="Dark Side of the Moon"]/price/text()
//catalog/cd[title="Dark Side of the Moon"]/price/text()
```

Question:

Value of artists of CDs from the USA

Answer:

```
/catalog/cd[@country="USA"]/artist/text()
//cd[@country="USA"]/artist/text()
//catalog/cd[@country="USA"]/artist/text()
```

Question:

Value of the country of the CD sung by David Bowie

Answer:

```
/catalog/cd[artist="David Bowie"]/@country
//cd[artist="David Bowie"]/@country
//catalog/cd[artist="David Bowie"]/@country
```

Question:

Value of the cost of all the CDs

Answer:

```
/catalog/cd/price/text()
//cd/price/text()
//catalog/cd/price/text()
```

Question:

Value of the number of CDs not from the UK (hint: use !=)

Answer:

```
count(/catalog/cd[@category!="UK"])
count(//cd[@category!="UK"])
count(//catalog/cd[@category!="UK"])
```

Question:

Value of the maximum price

Answer:

```
max(/catalog/cd/price/text())
max(//cd/price/text())
max(//catalog/cd/price/text())
```

Question:

Elements of the CDs from the UK

Answer:

```
/catalog/cd[@country="UK"]
//cd[@country="UK"]
//catalog/cd[@country="UK"]
```

Question:

Elements of all title elements

Answer:

```
/catalog/cd/title
//cd/title
//catalog/cd/title
```

Question:

Elements of the title of CDs by Pink Floyd

Answer:

```
/catalog/cd[artist="Pink Floyd"]/title
//cd[artist="Pink Floyd"]/title
```

```
//catalog/cd[artist="Pink Floyd"]/title
```

Question:

Elements of the root element

Answer:

```
//catalog
```

XSL*

- **eXtensible Stylesheet Language**
- XSL is itself an XML vocabulary
- Used to:
    - Present XML data in an easily readable format
    - transform the contents of a source XML document containing data into a result document written in a new format
- XSL is organized into 2 languages:
    - **XSL-FO (Extensible Stylesheet Language - Formatting Objects)**
        - for the layout of paginated documents
        - describes the precise layout of text on a page, indicating the placement of individual pages, text blocks, horizontal rules, headers, footers and other page elements
    - **XSLT (Extensible Stylesheet Language Transformations)**
        - used to transform the contents of an XML documents into another document format

XSL File: The Outer Section

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<xsl:stylesheet version="1.0"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
      <xsl:template match="/">
            <html>
                  <head>
                        <title>transform</title>
                  </head>
                  <body>
                        ...
                  </body>
            </html>
      </xsl:template>
</xsl:stylesheet>
```

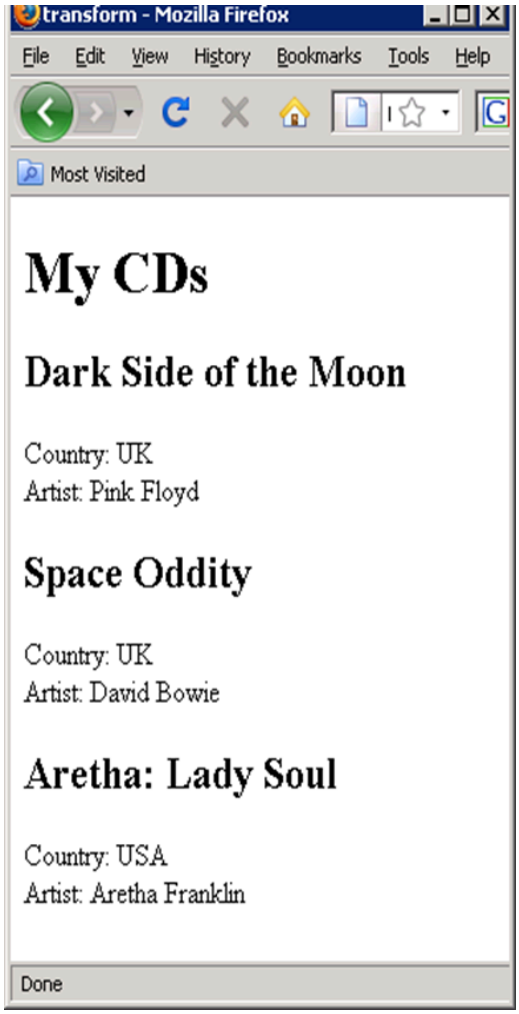
Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<catalog>
      <cd country="UK">
            <title>Dark Side of the Moon</title>
            <artist>Pink Floyd</artist>
            <price>10.90</price>
      </cd>
      <cd country="UK">
            <title>Space Oddity</title>
            <artist>David Bowie</artist>
            <price>9.90</price>
      </cd>
      <cd country="USA">
            <title>Aretha: Lady Soul</title>
            <artist>Aretha Franklin</artist>
            <price>9.90</price>
      </cd>
</catalog>
```

XSL File: the Inner Section

```
<body>
      <h1>My CDs</h1>
```

```
      <xsl:for-each select="//cd">
            <h2><xsl:value-of select="title"/></h2>
            <p>
                  Country: <xsl:value-of select="@country"/><br/>
                  Artist: <xsl:value-of select="artist"/>
            </p>
      </xsl:for-each>
</body>
```



- `xsl:for-each` → xsl loop element
- `select="<path/subelement>"`

    - If it is attribute of `xsl:for-each` element → XPath used to find CD elements
    - If it is attribute of `xsl:value-of` element → find matched subelements
- `xsl:value-of` → extracts data

Example: Displaying in a table

```
<xsl:template match="/">
      <html>
            <head>
                  <title>My CD Collection</title>
            </head>
            <body>
                  <table border="1">
                        <tr>
                              <th>Title</th>
                              <th>Artist</th>
                        </tr>
                        <xsl:for-each select="catalog/cd">
                              <tr>
                                    <td><xsl:value-of select="title"
/></td>
                                    <td><xsl:value-of select="artist"
/></td>
                              </tr>
                        </xsl:for-each>
                  </table>
            </body>
      </html>
</xsl:template>
```


My CD Collection
Title
Artist
Dark Side of the Moon
Pink Floyd
Space Oddity
David Bowie
Aretha: Lady Soul
Aretha Franklin
Don
Local intranet

Code Summary

XML Version

```xml
<xml? version="1.0" encoding="UTF-8"?>
<menu>
	<item code="M1001">
		<name>Nasi Lemak</name>
		<price currency="MYR">10.90</price>
		<combo paired_item="C1002" price="25.00"></combo>
	</item>

	<item code="C1002">
		<name>Seafood Fried Rice</name>
		<price currency="MYR">20.00</price>
		<combo paired_item="D1003" price="21.00" />
	</item>

	<item code="D1003">
		<name>Ice Lemon Tea</name>
		<price currency="MYR">5.00</price>
	</item>
</menu>
```

DTD Version

```dtd
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT menu (item+)>
<!ELEMENT item (name, price, combo?)>
<!ATTLIST item code ID #REQUIRED>

<!ELEMENT name (#PCDATA)>

<!ELEMENT price (#PCDATA)>
<!ATTLIST price currency CDATA #REQUIRED>

<!ELEMENT combo EMPTY>
<!ATTLIST combo
		paired_item IDREF #REQUIRED
		price CDATA #REQUIRED
```

```
>
```

XML Schema (XSD) Salami Slice Version

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified">

    <!-- Root element -->
    <xs:element name="menu" type="menuType"/>

    <!-- menuType -->
    <xs:complexType name="menuType">
        <xs:sequence>
            <xs:element name="item" type="itemType" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>

    <!-- itemType -->
    <xs:complexType name="itemType">
        <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="price" type="priceType"/>
            <xs:element name="combo" type="comboType" minOccurs="0"/>
        </xs:sequence>
        <xs:attribute name="code" type="xs:string" use="required"/>
    </xs:complexType>

    <!-- priceType -->
    <xs:complexType name="priceType">
        <xs:simpleContent>
            <xs:extension base="xs:float">
                <xs:attribute name="currency" type="xs:string"
use="required"/>
            </xs:extension>
        </xs:simpleContent>
    </xs:complexType>

    <!-- comboType -->
    <xs:complexType name="comboType">
```

```
        <xs:attribute name="paired_item" type="xs:string" use="required"/>
        <xs:attribute name="price" type="xs:float" use="required"/>
    </xs:complexType>

</xs:schema>
```

XSL Version

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
    version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

    <xsl:template match="/">
        <html>
            <body>
                <h2>Menu List</h2>
                <table border="1">
                    <tr>
                        <th>Name</th>
                        <th>Price</th>
                        <th>Combo</th>
                    </tr>

                    <xsl:for-each select="menu/item">
                        <tr>
                            <td><xsl:value-of select="name"/></td>

                            <!-- price with currency -->
                            <td>
                                <xsl:value-of select="price"/>
                                <xsl:text> </xsl:text>
                                <xsl:value-of select="price/@currency"/>
                            </td>

                            <!-- combo attributes -->
                            <td>
                                <xsl:if test="combo">
                                    <xsl:value-of
select="concat(combo/@paired_item, ' - ', combo/@price)"/>
                                </xsl:if>
```

```
                                <xsl:if test="not(combo)">
                                    <xsl:text>--</xsl:text>
                                </xsl:if>
                            </td>
                        </tr>
                    </xsl:for-each>

                </table>
            </body>
        </html>
    </xsl:template>

</xsl:stylesheet>
```

C5A: Web Services

JSON vs XML

Aspect	**JSON**	**XML**
Full Name	JavaScript Object Notation	eXtensible Markup Language
Data Format	Key-value pairs	Tag-based markup
Readability	More human-readable and concise	More verbose 冗长 and harder to read
Syntax Complexity	Simple syntax	Complex syntax with opening / closing tags
Data Size	Smaller, lightweight	Larger due to tags
Parsing Speed	Faster	Slower
Data Structure Support	Objects, arrays, numbers, strings, booleans, null	Elements, attributes, text
Schema Support	Optional (JSON Schema)	Strong support (DTD, XSD)
Validation	Less strict	Very strict and formal
Typical Usage	RESTful APIs, web & mobile apps	SOAP web services, enterprise systems
Learning Curve	Easy	Steep
HTTP Friendly	Very friendly	Less friendly

REST & RESTful Web Services

REST (Concept)

- **REST (Representational State Transfer)** is an **architectural style**, not a protocol.
- It defines a **set of principles / constraints** for designing web services
- It describes **how data (resources) should be transferred** between systems

- Systems that follow these principles are called **RESTful**

RESTful Web Services (Core Concept)

- A RESTful service is a web service that follows REST principles
- Key characteristics:
  - Operations are applied to **resources**
  - Resources are accessed using **URIs (URLs)**
  - Communication happens over **HTTP**
  - Data is transferred as **representations** (usually JSON, sometimes XML)
- Examples:
  - `/users` → collection of users
  - `/users/123` → specific user with ID 123

REST Architecture (Key Elements)

Resources

- Everything is a **resource** (user, product, order, article, etc)
- Each resource has a **unique URI**
- A collection contains multiple resources of the same type

HTTP Methods (Verbs)

HTTP Verb	Operation	Description
GET	Read	Retrieve a resource or collection
POST	Create	Create a new resource
PUT	Update	Update an existing resource
DELETE	Delete	Remove a resource

Request and Response

- Client sends an **HTTP request**
- Server returns an **HTTP response**
- Data is usually in **JSON format**
- Responses include **HTTP status codes**
- Examples:
  - 200 → OK
  - 201 → Created

- 400 → Unauthorized
- 404 → Not Found
- 500 → Server Error

Statelessness

- Each request is **independent**
- Server **does not remember previous requests**
- All required information must be included in the request

RESTful API

A REST API allows clients (web app, mobile app, etc) to:

- Fetch data from a server
- Send data to a server

This is done using:

- HTTP methods (GET, POST, PUT, DELETE)
- URIs to identify resources

RESTful Pros, Cons, Application

Advantages of RESTful Services	• **Simple** and **easy** to understand • **Lightweight** (especially with JSON) • **High performance and scalability** • Works well with **web** and **mobile applications** • Widely supported by browsers and tools • **Stateless** design improves reliability
Limitations of RESTful Services	• **No strict standard enforcement** (can be poorly designed) • **Not ideal for complex enterprise-level transactions** • **Lacks built-in security**, transactions and reliability features • **Depends heavily on correct API design practices**
When to use RESTful Services	• Building web or mobile applications • Performance and scalability are important • Simple CRUD operations are required • Lightweight communication is preferred • Public APIs are needed
Not ideal when	• Strong security and transaction control are required • Complex message structures are needed

	• Enterprise-level standards must be enforced (SOAP may be better)

SOAP, WSDL and UDDI

SOAP (Concept)

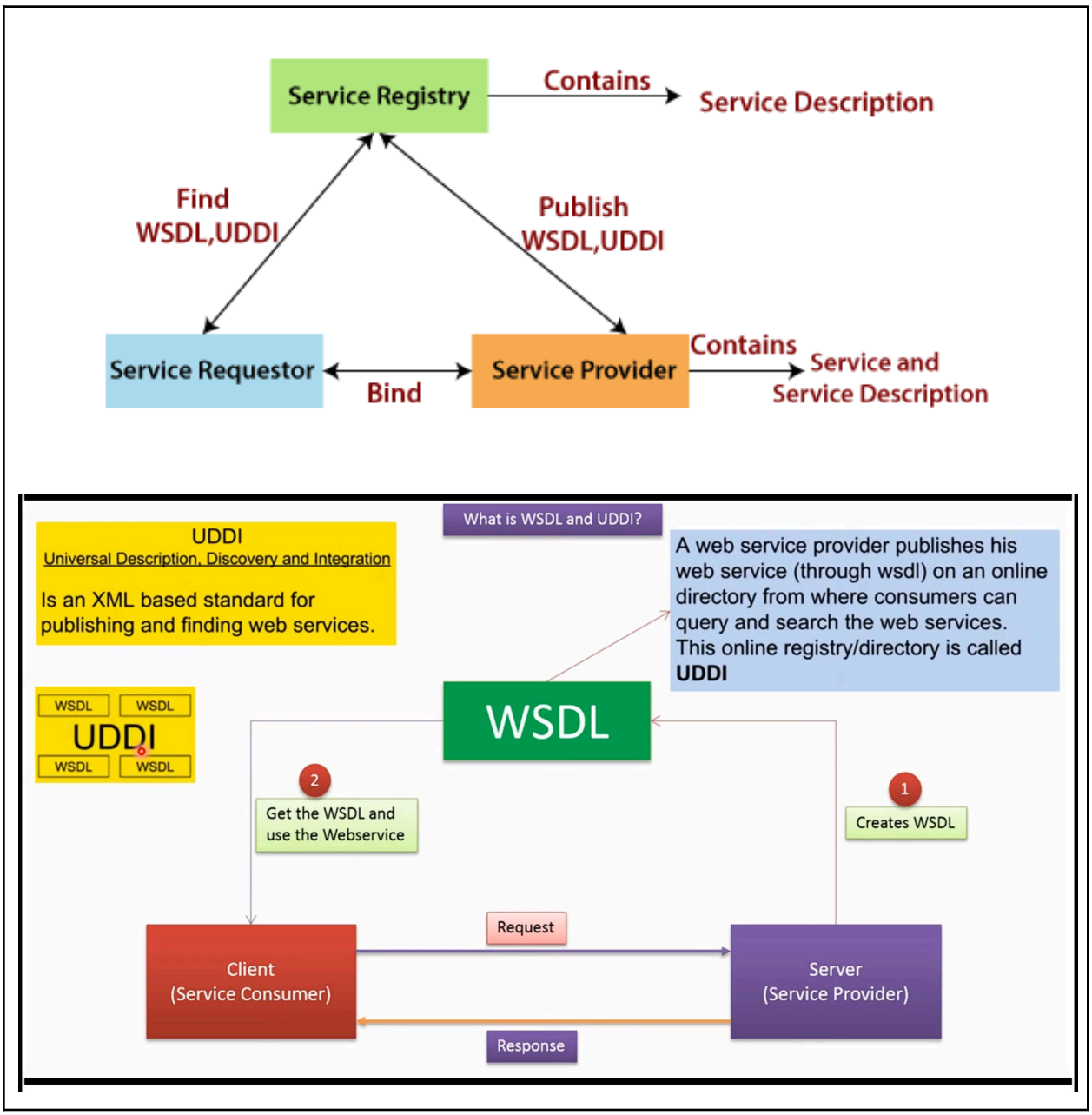
Concept	SOAP (Simple Object Access Protocol) is an XML-based communication protocol used for exchanging structured messages between systems.

Key Characteristics	• XML-based message format
	• Platform-independent
	• Language-independent
	• A W3C recommendation
	• Well-defined and structured
	• SOAP messages are sent inside a SOAP envelope, which contains:
	  ○ Header (optional)
	  ○ Body (mandatory message content)

SOAP Request and Response Model	• Client uses a proxy object generated from WSDL • Client sends a SOAP request • Server processes the request and sends a SOAP response • Communication is strictly structured using XML
Advantages of SOAP	SOAP is suitable for enterprise and B2B environments because: • Strongly standardized and well-defined • Easy integration when WSDL is available • Platform and language independent • Supports complex operations • Suitable for B2B integration across marketplaces • Reliable for structured communication
Limitations of SOAP	• Verbose and heavyweight (XML-only) • Slower performance compared to REST • Complex to implement and maintain • Not ideal for lightweight or mobile applications • Less popular in modern web development
When to use SOAP	• Strong standards and contracts are required • Enterprise-level systems are involved

	• B2B integrations are needed • Complex transactions and structured messaging and required • Strict validation is important
Not ideal when	• Lightweight, fast APIs are required • Mobile or web applications are the primary consumers

Relationship between SOAP, WSDL and UDDI



What is WSDL and UDDI in SOAP Web Service? | Web Service Tutorial	

Overview	• SOAP
	  ◦ Protocol for exchanging XML messages
	• WSDL
	  ◦ Describes the web service
	• UDDI
	  ◦ Registry for discovering web services

WSDL (Web Services Description Language)	• Used to describe a SOAP web service
	• What WSDL describes:
	  ◦ **What** a service does → Operations (methods)
	  ◦ **How** a service is accessed → Data format and protocol
	  ◦ **Where** a service is located → Service endpoint (URL)
	• Key WSDL components:
	  ◦ `types` → XML Schema data types
	  ◦ `message` → input / output messages
	  ◦ `portType` → abstract operations
	  ◦ `binding` → protocol binding
	  ◦ `service` / `port` → actual endpoint URL

UDDI (Universal Description, Discovery and Integration)	• A **directory (registry) of web services**.
	• Purpose:
	  ◦ Describe services
	  ◦ Discover services
	  ◦ Integrate services
	• Often described as the "yellow pages of web services"
	
	<u>Problems Solved by UDDI</u>
	• **Broader B2B integration**
	  ◦ Easier integration across marketplaces
	• **Smarter search**
	  ◦ Programmatic discovery of services
	• **Easier aggregation**
	  ◦ Combine services from multiple organizations
	
	<u>UDDI Components</u>
	• White Pages (Who?)
	  ◦ Company name
	  ◦ Contact information
	  ◦ Key services offered

	• Yellow Pages (What?)
	  ○ Categorization by:
	  ■ Industry codes (NAICS)
	  ■ Product codes (UN / SPSC)
	  ■ Geographic location
	• Green Pages (How?)
	  ○ Technical service details
	  ○ Supported XML formats
	  ○ References to WSDL

SOAP-WSDL-UDDI	• **SOAP** → the envelope that carries the message • **WSDL** → describes how to use the service • **UDDI** → lists and helps find services described by WSDL
Example	Airline reservation system 1. Airlines **publish their services to UDDI** 2. Travel agencies **search UDDI** to find reservation services **3. UDDI provides the WSDL** 4. Travel agencies **use WSDL to generate a client** 5. Communication happens using **SOAP messages**
Summary SOAP is a protocol for exchanging XML messages, WSDL describes how to access SOAP services, and UDDI acts as a registry that allows services described by WSDL to be discovered.	

RESTful vs SOAP

Aspect	RESTful	SOAP
Type	RESTful is an **architectural style** that defines guidelines for building web services.	SOAP is a **protocol** that strictly defines how messages are structured and exchanged.
Strictness	RESTful **provides design principles that are recommended** but **not strictly enforced**.	SOAP **enforces strict standards** that must be followed for interoperability.
HTTP Usage	RESTful **fully utilizes HTTP methods and status codes** as part of its design.	SOAP typically **uses HTTP only as a transport mechanism**.

HTTP Methods	RESTful uses HTTP verbs such as **GET, POST, PUT and DELETE** to perform CRUD operations.	SOAP mainly uses **HTTP POST** to **send XML-based requests**.
Data Format	RESTful commonly uses **JSON** for **lightweight data exchange** but can also support XML.	SOAP **uses XML exclusively for all message formats**.
Message Size	RESTful messages are **lightweight and efficient**.	SOAP messages are **verbose** 冗长 due to **XML and additional envelope structure**.
Performance	RESTful services generally provide **faster performance** and **better scalability**.	SOAP services are typically **slower** because of **heavier message processing**.
Contract	RESTful services **do not require a strict contract**, though tools like OpenAPI may be used.	SOAP services **require a WSDL contract** that formally defines the service interface.
Security	RESTful **security depends on external mechanisms** such as HTTPS and tokens.	SOAP provides **built-in security standards** such as WS-Security.
Complexity	RESTful APIs are **simpler to design, implement and maintain**.	SOAP services are more **complex to develop and manage**.
Typical Usage	RESTful is widely used for **web**, **mobile applications** and **public APIs**.	SOAP is commonly used in **enterprise** and **B2B systems**.

REST vs SOAP APIs: The key differences explained for beginners

C5C: Middleware

RMI (Remote Method Invocation)

Concept	**Java RMI (Remote Method Invocation)** is Java's mechanism for **object-to-object communication** in a distributed system. • Allows a Java program to **invoke methods on objects located on another machine** • Supports **distributed computing** using Java objects • Built **on top of sockets**, but hides low-level networking details • Designed specifically for **Java-to-Java communication**
Remote Objects	A remote object: • Lives on a **server** • Is accessed by clients on **different hosts** • Implements a **remote interface** ○ Defines which methods can be called remotely Key idea: • Clients invoke remote methods **almost the same way as local methods** • RMI handles all networking details transparently
Why use RMI?	• Avoids manual socket programming • Allows **direct method invocation** instead of messaging parsing • Supports object-oriented distributed computing • Reduces data transfer by **executing logic on the server**
Example	• Client sends a database query (as a `string`) to a remote database object • Server executes the query and computes the result • Server returns the result (e.g. a `double`) to the client
RMI Object Function Calls	RMI allows: • Method calls **between JVMs** • JVMs running on **different machines** • Passing objects as parameters • Dynamic class loading if required
General RMI Architecture	1. **Server registers** its remote object with the RMI Registry 2. **Client looks up** the remote object in the registry 3. Client calls a method on the remote object

	4. The call is transmitted via **stub and skeleton** 5. Server executes the method 6. Result is returned to the client
Stub and Skeleton	● **Stub (Client Side)** ○ Acts as a **proxy** ○ Sends method calls to the server ○ Marshals parameters ○ Opens a socket connection ● **Skeleton (Server Side)** ○ Receives remote calls ○ Unmarshals parameters ○ Invokes the actual method ○ Sends results back to the stub
RMI Registry	● The **RMI Registry** is a naming service that allows: ○ Servers to **register remote objects** ○ Clients to **locate remote objects**
When to use RMI	● Both client and server are written in Java ● Object-oriented communication is required ● You want simplicity over low-level control
Do not use RMI when	● Systems are written in different languages ● Lightweight or web-based APIs are required ● Internet-scale services are involved

CORBA (Common Object Request Broker Architecture)

Concept	CORBA is a **standard for object-based distributed computing** that allows: ● Objects written in **different programming languages** ● Running on **different platforms** ● To communicate **across a network** Example: A C++ client on Windows can communicate with a Java object on UNIX.
Purpose and Usage of CORBA	CORBA is mainly used as **middleware** for large, enterprise-level systems Typical use cases: ● Systems with **many clients**

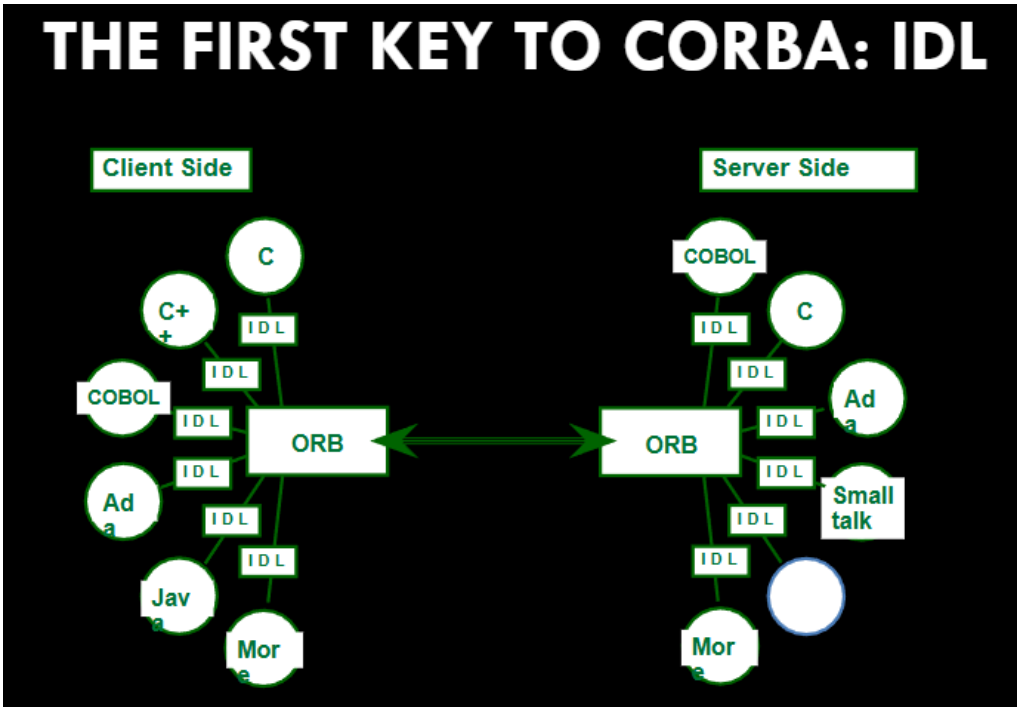
	• High request (hit) rates • High reliability requirements • Heterogeneous environments 异构环境 (different languages & platforms)
CORBA Communication Model	• **Object-based communication** • Clients invoke **methods** on remote objects • Similar goals to RMI, but **language-independent** • Differs from Web Services, which are **message-based**
Core Components	• **IDL (Interface Definition Language)** • **Client / Server CORBA Objects** • **ORB (Object Request Broker)** • **GIOP / IIOP protocols**
Interface Definition Language (IDL)	**IDL** is used to define **interfaces of CORBA objects** in a **language-independent way**. Key points: • Describes what methods are available • Defines public interfaces of CORBA servers • Not a programming language • Client and server are generated from the **same IDL** • OMG provides IDL mappings for many languages: ○ C, C++, Java, COBOL, Smalltalk, Ada, Lisp, Python, etc

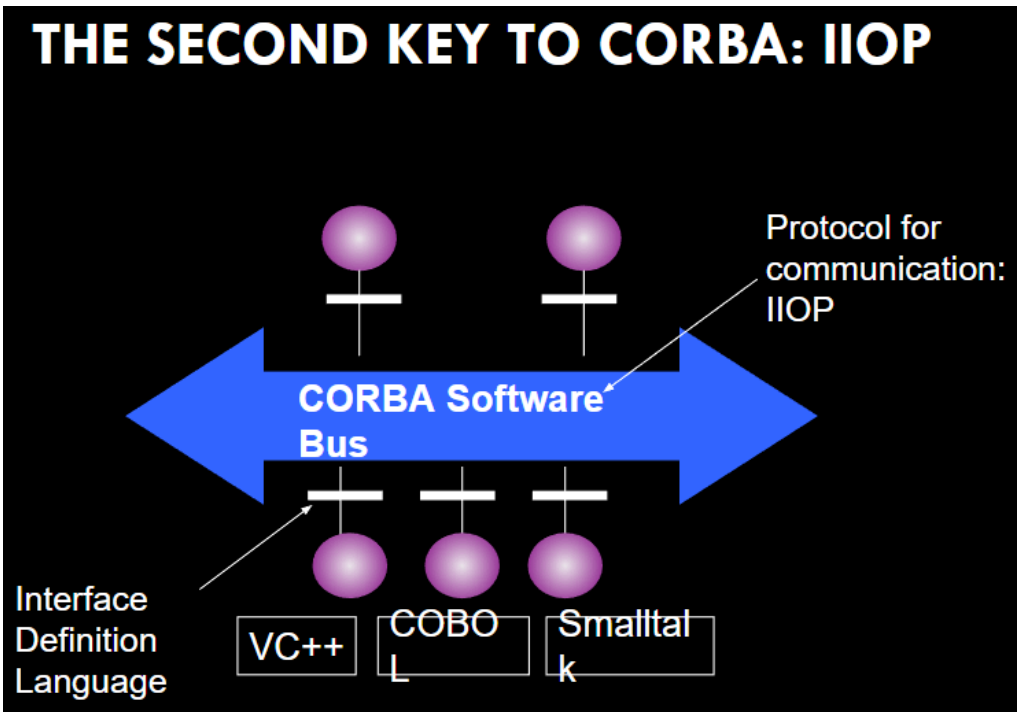
Object Request Broker (ORB)	• **ORB** is the heart of CORBA
	• An ORB is middleware that enables method calls between objects across a network.
	• Role of ORB:
	  ○ **Locates remote objects**
	  ○ **Sends method invocations from client to server**
	  ○ **Returns results back to the client**
	  ○ **Hides all networking details**

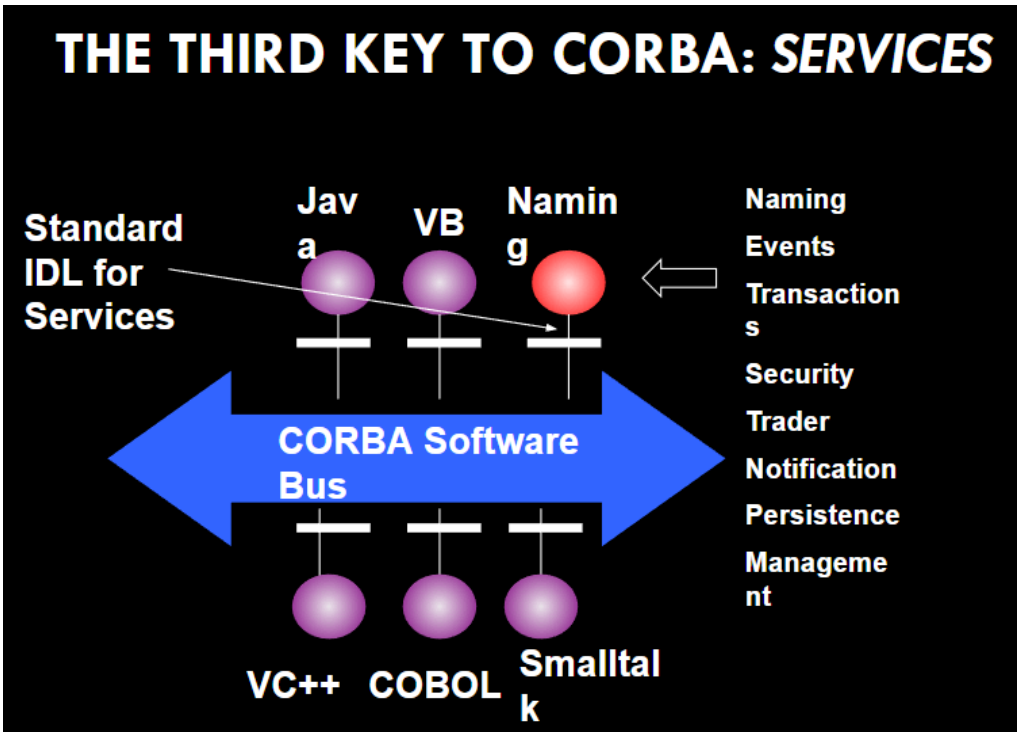
Stub and Skeleton (ORB Internals)	• **Client Side (Stub)**
	  ○ Acts as a proxy for the remote object
	  ○ Connected to the ORB
	  ○ Converts method calls into network requests
	• **Server Side (Skeleton)**
	  ○ Receives requests from the ORB
	  ○ Converts requests into local method calls
	  ○ Returns results or errors back to the client

GIOP / IIOP Protocols	• **GIOP**: General Inter-ORB Protocol
	• **IIOP**: Internet Inter-ORB Protocol
	• Purpose:
	  ○ Define **how ORBs communicates**
	  ○ Ensure interoperability across vendors and platforms
	• IIOP allows CORBA objects to communicate over the Internet

CORBA Services (Third Key of CORBA)	• Provides **standard services**, defined using IDL, such as:
	  ○ Naming service
	  ○ Event service
	  ○ Transaction service
	  ○ Security service
	  ○ Trader service
	  ○ Notification service
	  ○ Persistence & management services
	• These services run on top of the **CORBA software bus**

Diagram	





RMI vs CORBA

Aspect	**RMI**	**CORBA**

Basic Idea	RMI is a Java-specific technology that allows Java objects in different JVMs to invoke methods on each other remotely.	CORBA is a language-independent standard that allows objects written in different programming languages to communicate across platforms.
Language Support	RMI supports only Java and is designed for communication in a homogeneous Java environment.	CORBA supports multiple programming languages such as C++, Java and COBOL, enabling heterogeneous environments.
Interface Definition	In RMI, remote interfaces are defined directly using Java interfaces.	In CORBA, interfaces are defined using IDL (Interface Definition Language), which is independent of any programming language.
Middleware Component	RMI uses the Java RMI runtime and registry to locate and invoke remote objects.	CORBA uses an Object Request Broker (ORB) to manage communication between distributed objects.
Communication Protocol	RMI typically uses Java-specific protocols over TCP/IP for communication.	CORBA uses standard protocols such as IIOP to enable inter-ORB communication over networks.
Object Management	RMI objects are automatically garbage-collected by the Java Virtual Machine. ** Garbage collection = automatic memory cleanup of unused objects*	CORBA objects are not garbage-collected because they must remain language-independent.
Complexity	RMI is relatively simple to implement and maintain due to its tight integration with Java.	CORBA is more complex to design and maintain because of its broad language and platform support.
Typical Usage	RMI is commonly used in Java-based distributed applications.	CORBA is typically used in large enterprise systems requiring cross-language interoperability.