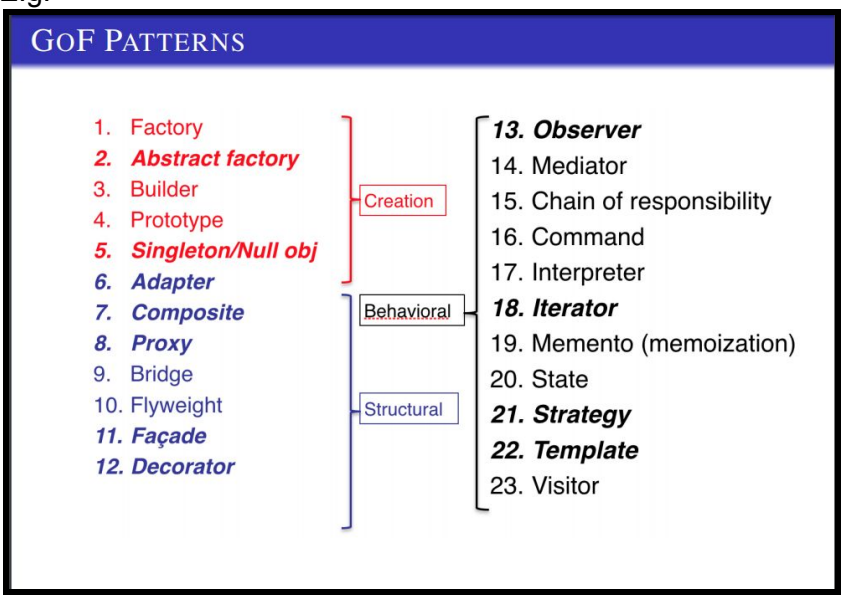
Mock Objects:

- The whole point of unit testing is to test the class in isolation. but sometimes your class depends or collaborates with other classes.
- You can use **mock objects**. The idea of mock objects is to create fake objects for these other classes and instantiate these fake objects so that you fulfill the obligations of your class that is currently under test.
  I.e. A mock object is a dummy implementation for an interface or a class in which you define the output of certain method calls.
- **Fakes** are objects that have working implementations, but not the same as production one.
- **Stubs** are objects that hold predefined data and use it to answer calls during tests.
- Mocks are objects that register calls they receive. In test assertion we can verify on Mocks that all expected actions were performed.

Design Patterns:

- Types of design patterns:
  1. **Architectural Patterns/Macroscale Patterns:**
  - Model-view-controller (MVC)
  - Pipe & Filter (e.g. compiler, Unix pipeline)
  - Event-based (e.g. interactive game)
  - Layering (eg. MEAN stack)
  - Enterprise Level Patterns
  2. **Computation Patterns:**
  - Fast Fourier transform
  - Linear Algebra
  - Symbolic computation
  3. **Gang of Four (GoF):**
  - Structural
  - Creational
  - Behavior
  - E.g.



- **Antipatterns** are code that looks like it should probably follow some design pattern, but doesn't. They are often the result of accumulated technical debt.

- Symptoms of antipatterns are:
    - Viscosity (easier to do hack than Right Thing)
    - Immobility
    - Needless repetition (comes from immobility)
    - Needless complexity from generality
- An example of an antipattern is the **God object**. A God object is an object that knows too much or does too much. A common programming technique is to separate a large problem into several smaller problems (a divide and conquer strategy) and create solutions for each of them. Once the smaller problems are solved, the big problem as a whole has been solved. This follows the Single-responsibility principle.
- SOFA is an acronym that captures symptoms that often indicate code smells.
    - Is it **S**hort?
    - Does it do **O**ne thing?
    - Does it have **F**ew arguments?
    - Is it a consistent level of **A**bstraction?
- Complex tasks need to be divided & conquered.
- Lots of arguments is bad because:
    - Hard to get good testing coverage.
    - Hard to mock/stub while testing.
    - Boolean arguments should be a yellow flag.
    - If a function behaves differently based on a boolean argument value, you should consider breaking the function into 2 functions.
- The **Demeter Principle** states that a module should not have the knowledge on the inner details of the objects it manipulates. In other words, a software component or an object should not have the knowledge of the internal working of other objects or components.
- Chart of refactoring bad designs:

Smell	**Refactoring that may solve it**
Large class	Extract class, subclass or module.
Large method	Decompose conditional. Replace loop with a collection method. Extract method. Replace method with object.
Long parameter list/Data Clump	Replace parameter with method call. Extract method.
Too many comments	Extract method. Replace with internal documentation.
Inconsistent level of abstraction	Extract methods & classes.

Microservices:

- **Monolithic software:**
    - **Introduction:**
    - A **monolithic application** is built as a single unit.
    - Enterprise applications are built in three parts: a database, a client-side user interface, and a server-side application. This server-side application will handle HTTP requests, execute some domain-specific logic, retrieve and update data

from the database, and populate the HTML views to be sent to the browser. It is a monolith – a single logical executable. To make any alterations to the system, a developer must build and deploy an updated version of the server-side application.

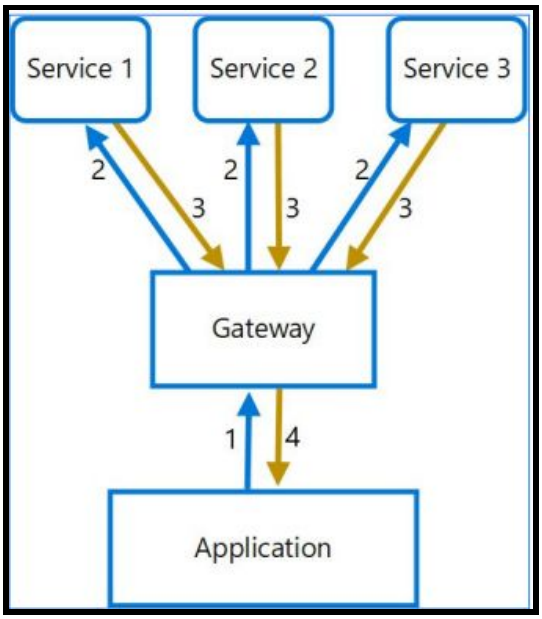
- I.e. A monolithic application is an application where the application components (e.g. UI, Database, Business Logic) are coupled together under a single program.
- A monolithic application is designed without modularity.
- The design philosophy for monolithic applications is that:
  - The application is responsible not just for a particular task, but it can perform every step needed to complete a particular function.
  - It can help the user carry out a complete task end to end.
  - It is a private data silo rather than parts of a larger system of applications that work together.
- **Advantages:**
  - Less complex deployment and shipment.
  - Simpler scaling model.
  - Easier code management.
- **Disadvantages:**
  - Giant codebases
  - Testing is expensive
  - Fault intolerant
  - Resource intensive to scale
  - Requires significant multi-team alignment and co-operation to release
  - Monolithic applications can evolve into a “big ball of mud”; a situation where no single developer or group of developers understands the entirety of the application.

- **Microservice software:**
  - **Introduction:**
  - In microservices, we split the application into multiple services each of which is responsible for some contained functionality.
  - In microservices, application components are decoupled and modularized into services to perform a single piece of functionality. The services connect to form the final program.
  - In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.
  - Each service is responsible for a single application unit/component.
  - Microservices architecture naturally leads itself to decoupling of application components.
  - A service is queried via an API contract shared between services/teams.
  - Each service can be modified/updated without impacting the rest of the application.
  - While a single microservice is more maintainable, the entire application is overall more complex.
  - **Note:** Microservices are not inherently better than monoliths.

  - **Scalability:**

- Deployment strategy is heavily dependent on DevOps culture.
- Scale each service independently.
- Scaling is logistically complex:
    - Single monolith vs many microservice
    - Container management
    - Tracking clients across services
    - Logging across services
- Scaling is less computationally expensive.
- **Resiliency:**
- An individual or set of microservices can be removed without bringing down the entire application.
- Microservice are composed together to create the cohesive application.
- No single point of failure → increased security.
- Allows for greater overall uptime and reliability of application.
- **Decoupling:**
- Each service:
    - Is its own “program”
    - Run as its own application without requiring other services
    - Can be tested independently
    - Can be deployed/scaled/removed independently from the application
- Services are composed together to create the final application.
- Decoupling allows for:
    - Team Autonomy
    - Rapid Iteration
    - Resiliency
    - Scalability
- **Rapid Iteration:**
- Allows for consistent delivery (i.e. Agile methodology).
- Reduced dependency on other teams.
- The more autonomous the team, the faster the delivery.
- Independent release cycles.
- **Team Autonomy:**
- Each team is responsible for one or more services.
- Each service can be independently:
    - Developed
    - Refactored
    - Tested
    - Deployed
- Each service can use any technology stack.
- Less dependency on other teams. However, we need to communicate the API contract to other teams in order to query the microservice.

- **Microservice Design Patterns:**
  1. **Aggregator:**
  - Invokes multiple services and combines all data prior to returning it to the client.



  - The most common design pattern.
  - The aggregator can be scaled.
    I.e. It too can be a microservice.
  - We use it when:
    - The client needs to communicate with multiple services to perform an operation.
    - The client's network requirements utilize significant latency.

  2. **Chain:**
  - Produces a single consolidated response for a single client request.
  - Each service invokes the next service in the chain.
  - The client will be blocked until the chain returns a response.
  - We use it when:
    - There is a direct dependency on another service (e.g. purchasing an item, need purchase and order services).
    - Ordering operations between services is required.

  3. **Branch:**
  - An extension of the Aggregator and Chaining patterns.
  - A microservice can simultaneously request data from 2 or more services.
  - We use it when a service requires data from multiple sources (e.g. product info, seller info, pricing, etc).