Design

Tuesday, 4 December 2018 11:48 AM

Software Design

- Transforming problem to solution
- Product/ external design (meet user requirements)
- Implementation/ internal design (how product will meet required external behaviour)

Design Fundamentals

Abstraction

- Complexity is the enemy -> multiple levels of abstraction
- Classes, methods, architecture/ class diagrams are examples of abstraction
    - Method intention (name) vs mechanism (implementation)
- "leaky": reveal details which should be hidden
- Example: data/ control abstractions

Coupling

- Degree of dependence: tight/ strong?
- Changing B _might_ require changing A
- Dependencies based on time, data, control flow
- Reduce coupling for testability, reusability, maintainability and integration
- Unnecessary dependency: e.g. Parser --> Logic, parser doesn't need to know logic

A has access to the internal structure of B (this results in a very high level of coupling)

A and B depend on the same global variable

A calls B

A receives an object of B as a parameter or a return value

A inherits from B

A and B are required to follow the same data format or communication protocol

- STRONG: A calls method to B --> B might return value
- WEAK: A and B access the same pool of global variables (hidden dependency)

Content coupling: one module modifies or relies on the internal workings of another module e.g., accessing local data of another module

Common/Global coupling: two modules share the same global data

Control coupling: one module controlling the flow of another, by passing it information on what to do e.g., passing a flag

Data coupling: one module sharing data with another module e.g. via passing parameters

External coupling: two modules share an externally imposed convention e.g., data formats, communication protocols, device interfaces.

Subclass coupling: a class inherits from another class. Note that a child class is coupled to the parent class but not the other way around.

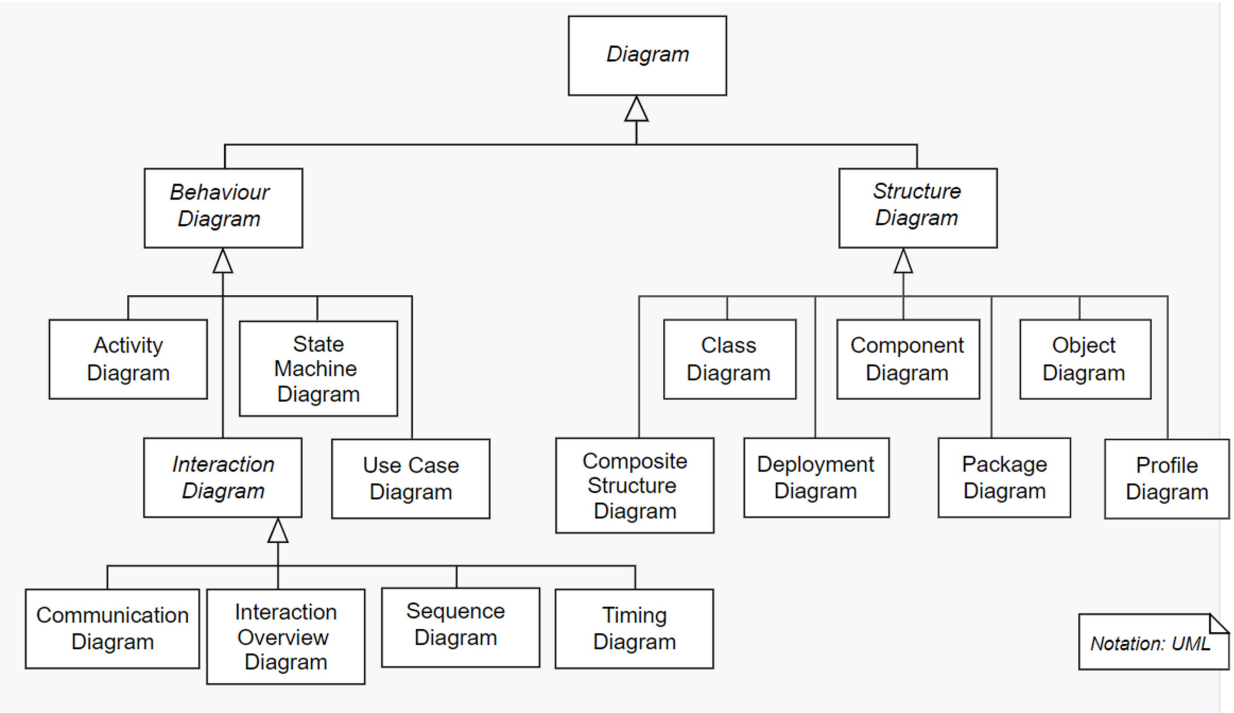
Temporal coupling: two actions are bundled together just because they happen to occur at the same time e.g. extracting a contiguous block of code as a method although the code block contains statements unrelated to each other

Cohesion

- Components do one thing, and do it all
- More cohesion tends to lead to less coupling

Modelling

- Model is a representation (abstraction to reduce complexity), multiple models needed to capture different aspects
- Analyse complex entity, communicate information, blueprint for creating software
    - Blueprint = MDD, rare
- Modelling notation (UML) is not used often, but gives an edge (better than nothing)
    - Three Amigos - each pushing their own
    - Usually used in OOP contexts
- Model structure (class/ object diagram) vs model behaviour (use case diagram)



Modelling Structures

- OO Solution: network of interacting objects, modelled with Class/ Object Diagrams
    - Object structures change over time, based on rules in class structure
- Object Oriented Domain Model - model problem domain
    - Translate nouns to classes
    - Conceptual class diagram (subset of class diagram)
        - Class diagram describes solution domain
        - No solution-specific classes, only classes that exist in problem domain
    - No methods and navigability
    - Object structure in problem domain = object diagram without navigability
- Deployment Diagram: shows physical layout (which software on which hardware)
- Component Diagram: shows components and how they are connected via interfaces
- Package Diagram: shows packages and dependencies (grouping construct for UML elements like classes and use cases)
- Composite Structure Diagram: hierarchically decomposes class into internal structure

Modelling Behaviours

- Activity Diagrams, Sequence Diagrams, Use Case Diagrams
- Timing Diagram: shows timing constraints
- Interaction Overview Diagram: activity + sequence diagram combined
- Communication Diagram: sequence diagram with emphasis on data links
- State Machine Diagram: state-dependent behaviour in state chart
    - Captures object's behaviour over full life cycle (vs sequence diagram: one scenario)

Software Architecture

- Software elements + external properties + relationships among them

- Overall organisation, high-level design by software architect

Architecture Diagrams

- Free-form, no standard notation
- Basic guidelines: minimise symbols, go for widely-understood meanings
    - E.g. Drum = database

Architectural Styles

- Common architectural patterns
- N-tier: layered, abstraction levels, cannot talk to higher layers
    - Higher layers make use of lower layers
- Client-Server: at least one client + server
    - Common in distributed applications
- Transaction Processing: transactions given to dispatcher
    - E.g. ModelManager in AB4, API-based
- Service-Oriented (SOA): combining functionalities as programmatically accessible services (e.g. XML web services)
- Event-driven: detecting events from event emitters and communicating those events to event consumers
    - Common in GUI-driven applications
- Others: pipes-and-filters, broker, p2p, message-oriented

Software Design Patterns

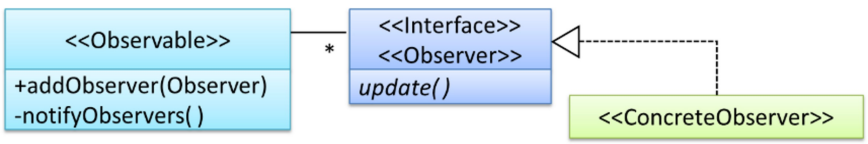
- Elegant, reusable solutions to common problems (in certain contexts)
- Context, Problem, Solution, Anti-Patterns (incorrect/ inferior solutions), Consequences (pros and cons), Others (code examples, known uses, related patterns) - only first three are compulsory
- Singleton: only one instance
    - Private constructor, public class-level method to access single instance
    - Global variable, increases coupling across code base (e.g. In testing)
    - Difficult to replace with stubs (static methods cannot be overriden)
- Abstraction Occurrence
    - Instead of abstracting common occurrence in a superclass, use a separate relationship class (abstraction class and occurrence class)
    - More of an analysis pattern

- Facade: class that sits between internals and users, do not reveal internals
    - E.g. ModelManager in AB4
- Command: use abstract class or interface
- MVC: reduce coupling from storage, UI and user changes

  View: Displays data, interacts with the user, and pulls data from the model if necessary.

  Controller: Detects UI events such as mouse clicks, button pushes and takes follow up action. Updates/changes the model/view when necessary.

  Model: Stores and maintains data. Updates views if necessary.
    - Many variations of MVC
    - More of an architectural pattern
- Observer (e.g. UI observes model)
    - Observer, obseverable
    - Used for A to initiate activity in B without direct dependency



- Design patterns can be combined
- Gang of Four (GoF) book of 23 patterns
    - Creational: separate object application from creation (e.g. Singleton)
    - Structural: composing objects into larger structures (e.g. Facade)
    - Behavioural: defining interactions and distribution of responsibility (e.g. Command)
- Patterns come with overhead (more classes and abstraction)

Design Principles

- Varying degrees of formality: rules, opinions, rules of thumb, observations, axioms
- More applicable and also more overlap among them

Design Approaches

- Multi-level (e.g. Architectural, class diagram...)
    - Top-down (big, novel systems) vs. Bottom-up (components re-use)
    - Start from deliverables (UI) to define APIs systematically
    - Bottom-Up is more common and familiar
- Agile: emergent designs not defined up front
    - Diagrams used for communication
    - Design changes over time, too much precision is a waste of effort
        - Future-proof: omit less important details