Part IV - Component Principles

Intro

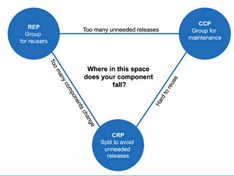
- Principles for high-level software
  - How to build software systems out of components

12. Components

- ⚠️ Components are the units of deployment
- A Brief History of Components
  - App and function library were loaded at specific predefined memory address
- Relocatability
  - Linking loader
    - Modifies references from the app to libraries
    - Loads only functions that are needed
  - External references
  - External definitions
- Linkers
  - Slow linked builds binary that could be loaded with fast linker
  - Murphy's law of program size
    - Programs will grow to fill all available compile and link time

13. Component Cohesion

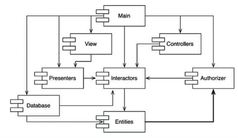
- REP: The Reuse/Release Equivalence Principle
  - 👍 The granule of reuse is the granule of release.
  - Reusable element is a release (version) of set of classes/modules that make sense to keep together
- CCP: The Common Closure Principle
  - 👍 Gather into components those classes that change for the same reasons and at the same times.
  - Separate into different components those classes that change at different times and for different reasons.
  - ⚠️ This is the Single Responsibility Principle (SRP) restated for components
  - ⚠️ It is closely associated with the Open Closed Principle (OCP)
- CRP: The Common Reuse Principle
  - 👍 Don't force users of a component to depend on things they don't need.
  - Classes that are not tightly bound to each other should not be in the same component
  - The CRP is the generic version of the ISP



- The Tension Diagram for Component Cohesion
  - 👍 Early in the development of a project, the CCP is much more important than the REP, because developability is more important than reuse.
    - 🏝️ i.e. always start with monorepo!
  - 👍 Later the project will slide from developability to the left to reusability

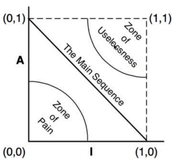
14. Component Coupling

- ADP: The Acyclic Dependencies Principle
  - 👍 Allow no cycles in the component dependency graph
  - 👎 Developer have working code. But in the morning the is not working because another developer has changed something that the code was dependent on
    - It is common problem especially on large teams
  - ⚠️ The Weekly Build
    - Work four days using local copy of all the dependencies ignoring other team members
    - On the Friday integrate all the changes together
    - Sometimes one day is not enough and integration moves to Thursday and before. Or team switches to biweekly builds
      - This scenario will eventually lead to crisis
  - 👍 Eliminating Dependency Cycles
    - Responsible developers publish released components and continue to work on the next tasks
    - Other developers decide which release to use
    - But you have to manage dependencies
      - There can be no cycles (directed acyclic graph)
  - The Effect of a Cycle in the Component Dependency Graph

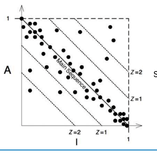


Entities now wants to use Authorizer

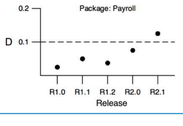
  - Breaking the Cycle
    - 👍 Apply the Dependency Inversion Principle (DIP)
      - Use interfaces
    - 👍 Create a new component that both Entities and Authorizer depend on.
  - The "Jitters"
    - ⚠️ Components structure should be monitored for cycles. And it will change over time
- Top-Down Design
  - Components structure does NOT reflect functionality of the app
    - Instead it reflects
      - Buildability
      - Maintainability
  - ⚠️ The component dependency structure grows and evolves with the logical design of the system
    - They are not designed all at the beginning!
      - Because initially we have no software to build and maintain
- SDP: The Stable Dependencies Principle
  - 👍 Depend in the direction of stability
  - Stability
    - Stability is related to the amount of work required to make a change
      - 😊 One sure way to make a software component difficult to change, is to make lots of other software components depend on it
  - Stability Metrics
    - Count number of dependencies
    - Fan-in: Incoming dependencies
    - Fan-out: Outgoing depenencies.
    - I: Instability: I = Fan-out / (Fan-in + Fan-out)
    - 👍 I metrics should decrease in the direction of dependency
  - Not All Components Should Be Stable
    - When Stable need to depend on very Flexible (potentially unstable) component
      - We use DIP 👈
      - 👍 Abstract Components
- SAP: The Stable Abstractions Principle
  - 👍 A component should be as abstract as it is stable. 👈
  - Where Do We Put the High-Level Policy?
  - Introducing the Stable Abstractions Principle
    - The SAP and the SDP combined amount to the DIP for components
      - 👍 dependencies run in the direction of abstraction
  - Measuring Abstraction
    - Nc: The number of classes in the component
    - Na: The number of abstract classes and interfaces in the component
    - A: Abstractness. A = Na / Nc.
      - ranges from 0 to 1



  - The Main Sequence
    - The Zone of Pain
      - Examples
        - database schemas
        - utility library
    - The Zone of Uselessness
      - Examples
        - unused abstract classes
    - Avoiding the Zones of Exclusion
      - Need to position most of the compontnts to the Main Sequence
    - Distance from the Main Sequence
      - D: Distance. D = |A+I-1|
      - design can be analyzed for its overall conformance to the Main Sequence



Scatterplot of the components



Plot of D for a single component over time