Part VI - Details

30. The Database Is a Detail

- 👎 The database is NOT an architectural element!
- 👍 The data model IS!
- Relational Databases
  - Many data access frameworks allow database rows and tables to be passed around the system as objects
  - 👎 Allowing this is an architectural error
    - It couples the use cases, business rules, and in some cases even the UI to the relational structure of the data
- Why Are Database Systems So Prevalent?
  - Because of disks - they are slow
    - File systems are document-based
      - Hard to find file by content
    - Database systems are content-based
      - Poor at storing/retrieving opaque documents
- What If There Were No Disk?
  - Disks are dying breed
  - Data will soon be stored in RAM only
    - 👍 In a form of linked lists, trees, hash tables, stacks, queues, etc.
      - This is what we already do after we loaded data from dabadase!
- Details
  - ⚠️ We should not care about the form that the data takes while it is on the surface of a rotating magnetic disk
- But What about Performance?
  - ⚠️ it's a concern that can be entirely encapsulated and separated from the business rules
    - that's a low-level concern
- Anecdote
  - 😐 Sometimes there is no architectural rationale for using a database, but there is a marketing/political rationale

31. The Web Is a Detail

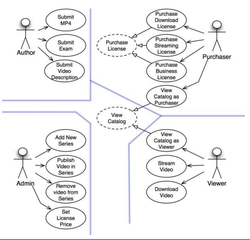
- 👍 The web is just an a series of oscillations that move back and forth between
  - putting all the computer power in central servers
  - and putting all computer power out at the terminals
- The Endless Pendulum
  - As architects, though, we have to look at the long term
    - 👍 Those oscillations are just short-term issues that we want to push away from the central core of our business rules
- The Upshot
  - The GUI is a detail
    - The web is a GUI. So the web is a detail
  - 👍 The WEB is an IO device

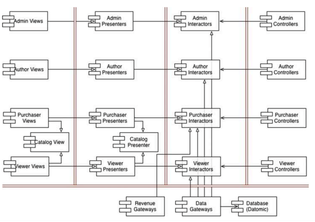
32. Frameworks are Details

- Framework Authors
  - Frameworks solve the author's problems, not yours
    - Your problems do overlap, but to a limited extent
- 👎 Asymmetric Marriage
  - You must make a huge commitment to the framework
  - Yhe framework author makes no commitment to you whatsoever
- The Risks
  - 👎 Frameworks tend to violate he Dependency Rule
    - They ask you to inherit their code into your business objects — your Entities (the innermost circle)!
- The Solution
  - ⚠️ Don't marry the framework!
  - You can use the framework, still
    - If the framework wants you to derive your business objects from its base classes, say no!
      - 👍 Derive proxies instead, and keep those proxies in components that are plugins to your business rules.
- I Now Pronounce You ...
  - There are some frameworks that you simply must marry
    - C++
      - STL
    - Java
      - the standard library
  - ⚠️ it should still be a decision

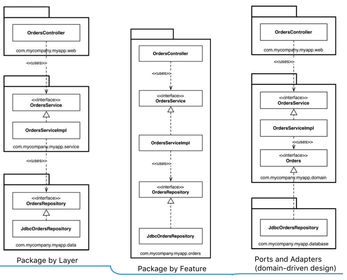
33. Case Study: Video Sales

- The Product
  - the software for a website that sells videos
- Use Case Analysis

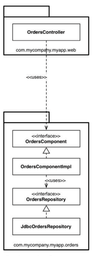




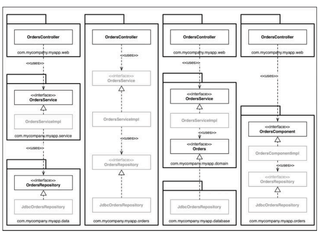
- Component Architecture
  - Yet we still can deploy them as five or just two dll/jar files
    - one for views, presenters, interactors, controllers, and utilities, respectively
    - or views and presenters in one file, and everything else in the other
- Dependency Management
  - the Dependency Rule
    - 👍 All dependencies cross the boundary lines in one direction, and they always point toward the components containing the higher-level policy



- Package by Layer
- Package by Feature
- Ports and Adapters (domain-driven design)



- Package by Component
- A better architecture
- Definition of a component
  - A grouping of related functionality behind a nice clean interface, which resides inside an execution environment like an application
    - https://c4model.com/
- The Devil Is in the Implementation Details
  - 👎 The first three approaches allow you to implement details in a wrong way
- All four architectural approaches are the same
  - But they have different encapsulation options (what should be made PUBLIC)



  - Grayed-out types are where the access modifier can be made more restrictive
    - 👍 The fewer PUBLIC types you have, the smaller the number of potential dependencies
- Organization versus Encapsulation
- Other Decoupling Modes
  - In Java 9 you can make a distinction between types that are public and types that are published
  - You can decouple your dependencies at the source code level
    - by splitting code across different source code trees
- 👍 Conclusion: The Missing Advice
  - Think about
    - how to map your desired design on to code structures
    - how to organize that code
    - which decoupling modes to apply during runtime and compile-time
  - Leave options open where applicable
    - but be pragmatic
      - take into consideration
        - the size of your team,
        - their skill level
        - the complexity of the solution
      - in conjunction with
        - your time
        - and budgetary constraints
  - Also
    - think about using your compiler
      - to help you enforce your chosen architectural style
      - and watch out for coupling in other areas,
        - such as data models

34. The Missing Chapter