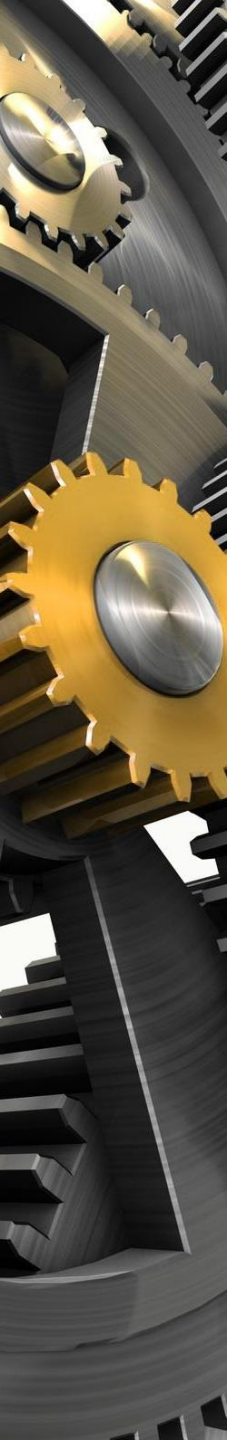
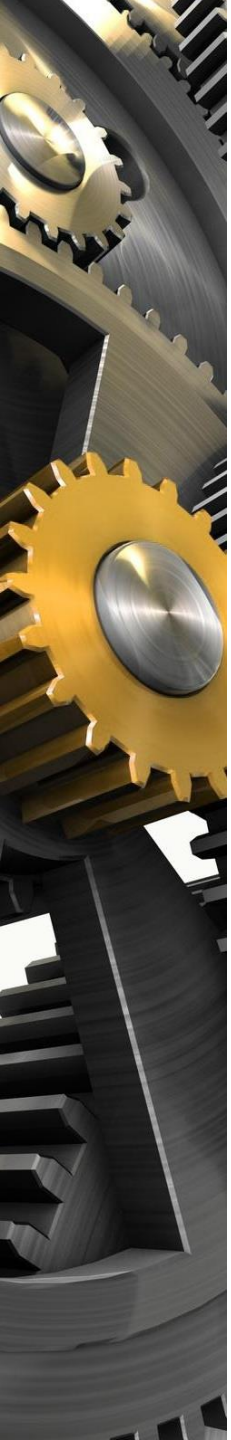


ADUF
Adaptable Design Up Front

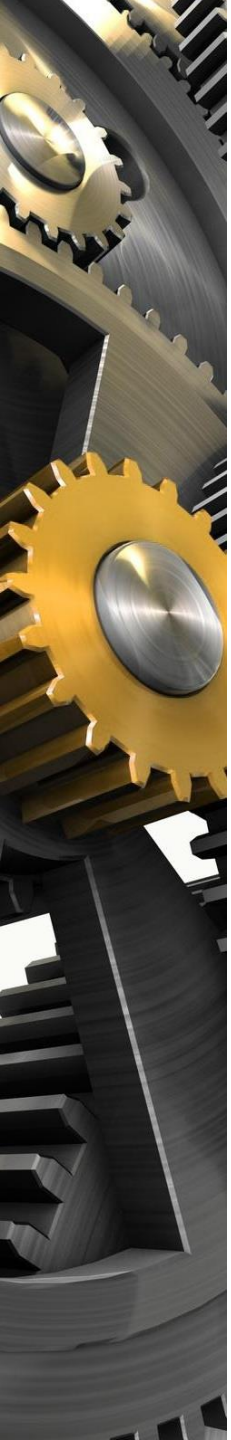
Hayim Makabee

http://EffectiveSoftwareDesign.com



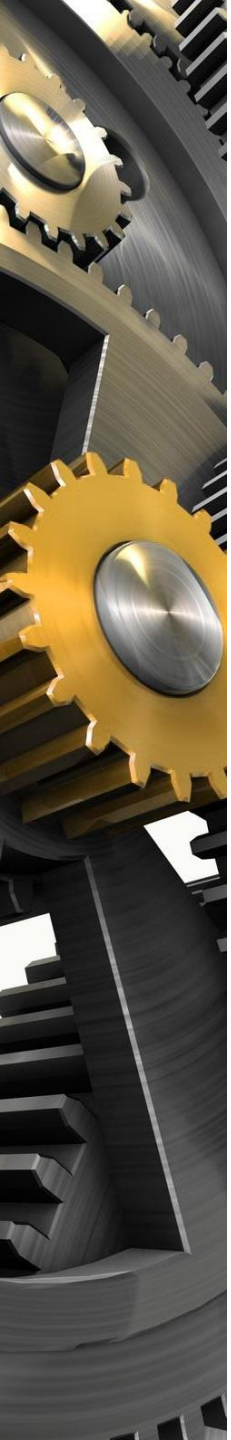
Context

- Iterative Software Development
- Agile, Incremental



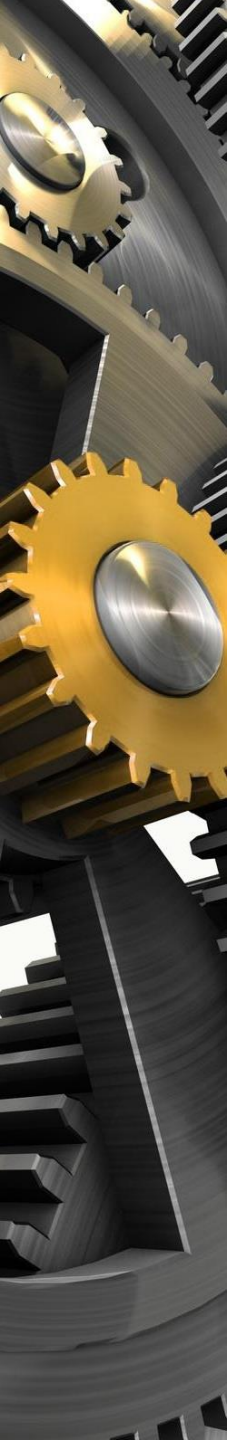
Question

- How much design should be done up front?
- Up front = Before starting the implementation (coding).



Big Design Up Front

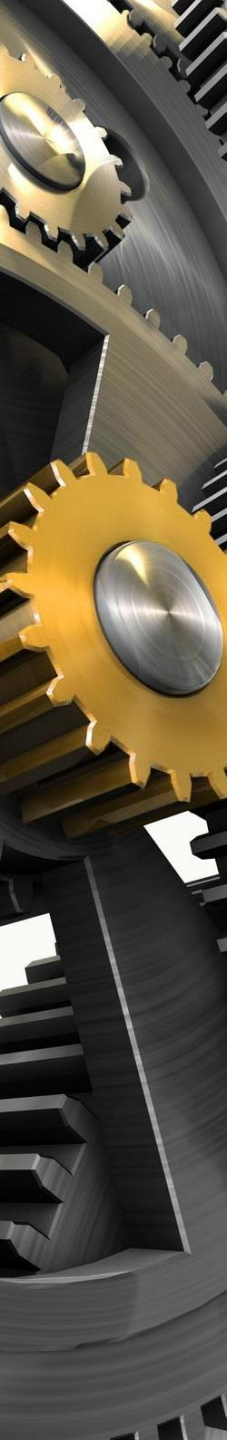
- BDUF = Do detailed design before starting to code.
- Problems:
  - Requirements are incomplete.
  - Requirements may change.
  - *"The Scrum product backlog is allowed to grow and change as more is learned about the product and its customers."*



Emergent Design

- *"With emergent design, a development organization starts delivering functionality and lets the design emerge."*

- First iteration:
  - Implement initial features.
- Next iterations:
  - Implement additional features.
  - Refactor.



Just Enough Design

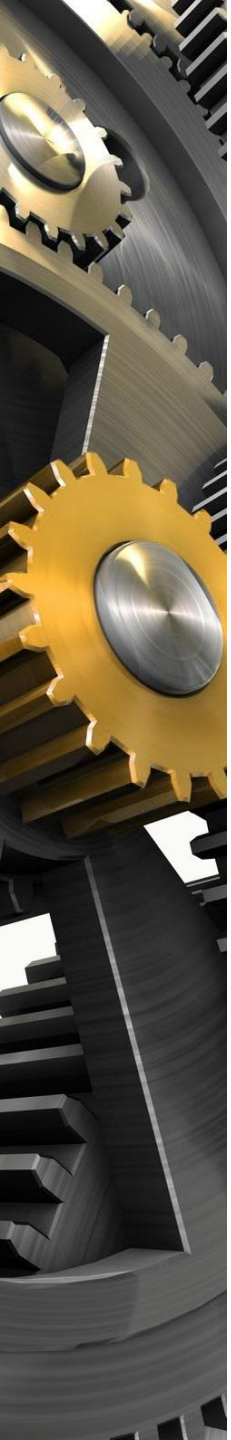
- *"**Just enough** sits somewhere in the chasm between big design up front's analysis paralysis and emergent design's refactor distractor."* – Simon Brown

- Question: How much design is "just enough"?

Planned: Modiin

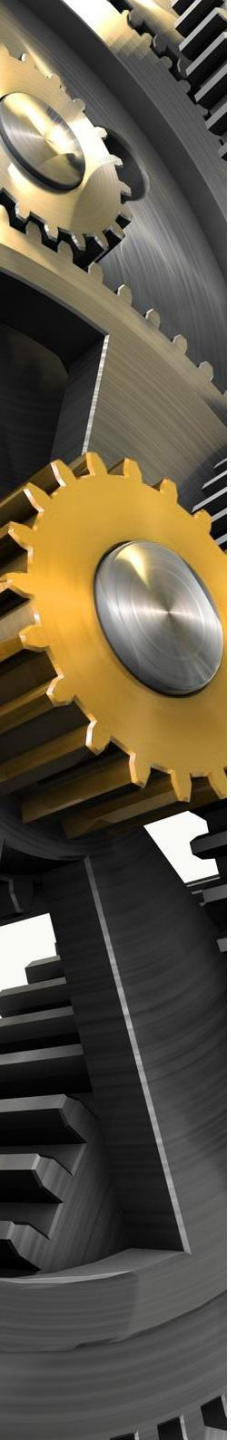
Emergent: Jerusalem Old City

Eroding Design



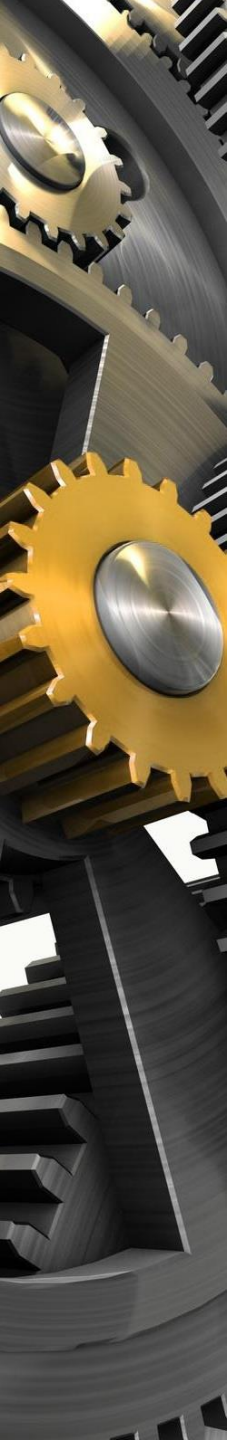
Eroding Design

- *"The biggest risk associated with Piecemeal Growth is that it will* ***gradually erode the overall structure of the system****, and inexorably turn it into a Big Ball of Mud."* - Brian Foote and Joseph Yoder – "Big Ball of Mud"



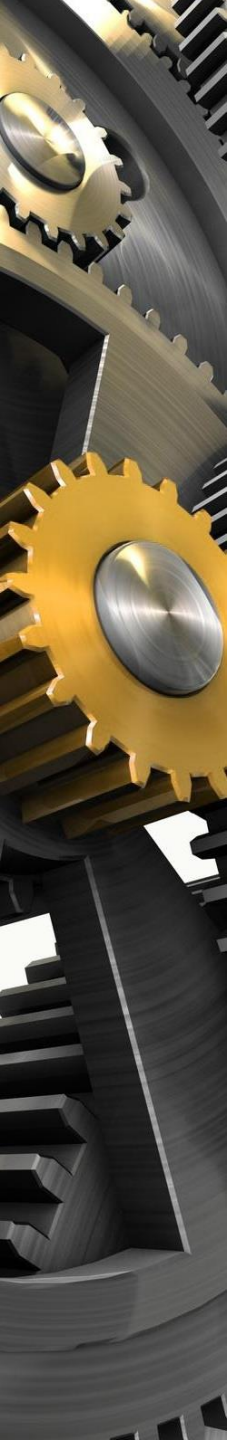
Software Entropy

- Entropy is a measure of the number of specific ways in which a system may be arranged (a measure of disorder).
- The entropy of an isolated system never decreases.



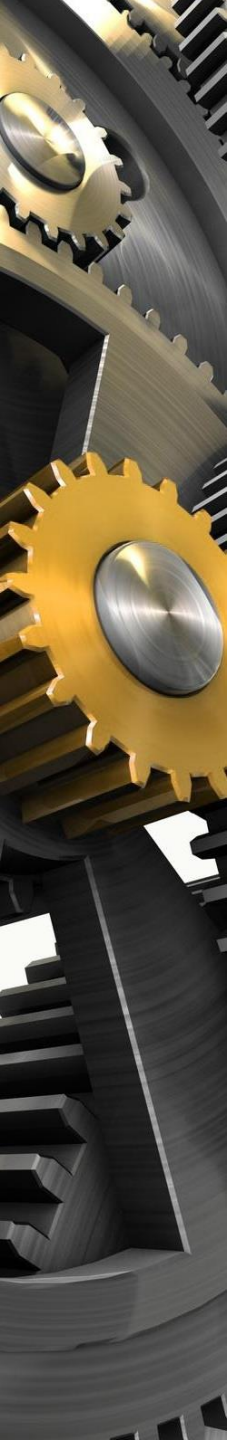
Lehman Laws

- A computer program that is used will be modified.
- When a program is modified, its complexity will increase, provided that one does not actively work against this.



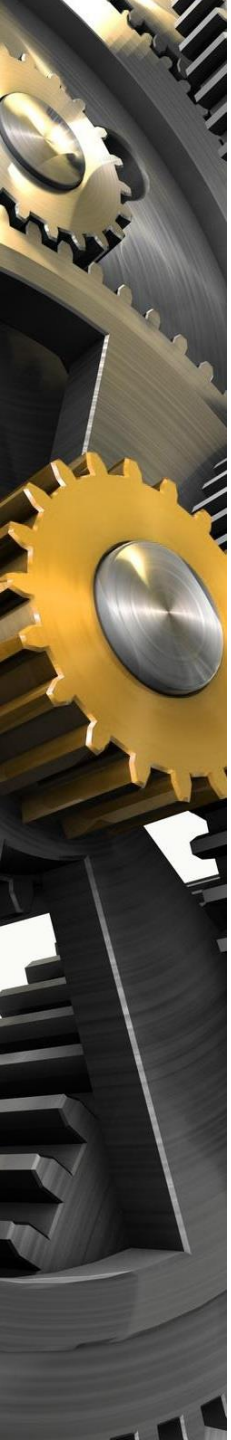
Conclusions so far…

- In summary:
  - We must design up front.
  - BDUF does not support change.
  - Emergent design works at low-level.
  - Just Enough DUF is not clear.
- We need:
  - The design must support change.



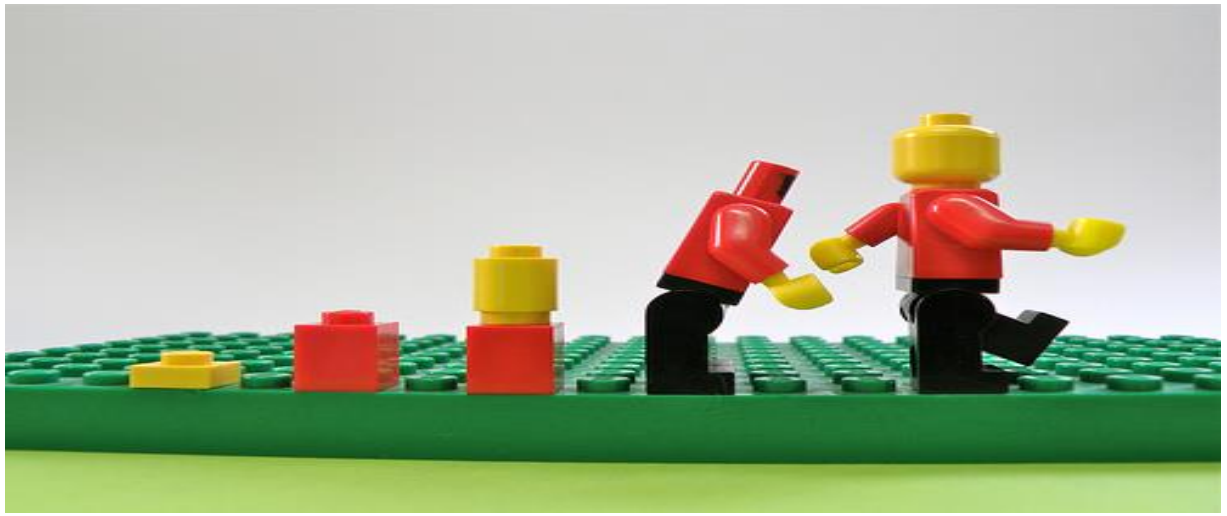
Adaptability

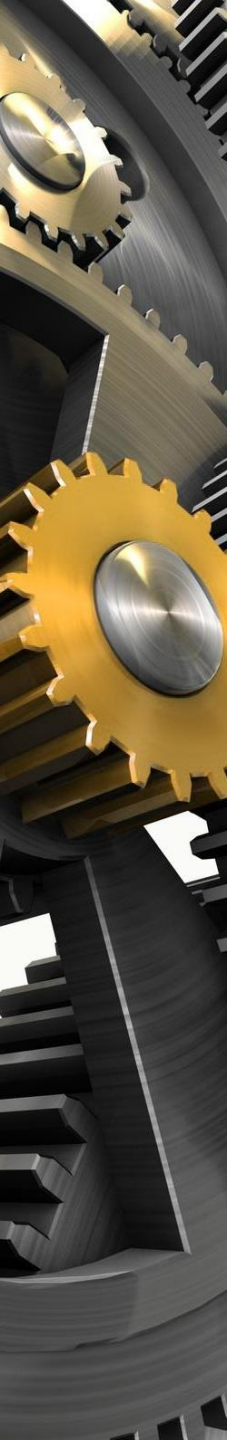
- *"A system that can **cope readily with a wide range of requirements**, will, all other things being equal, have an advantage over one that cannot. Such a system can **allow unexpected requirements to be met with little or no reengineering**, and allow its more skilled customers to rapidly address novel challenges."* - Brian Foote and Joseph Yoder – "Big Ball of Mud"



New development mindset

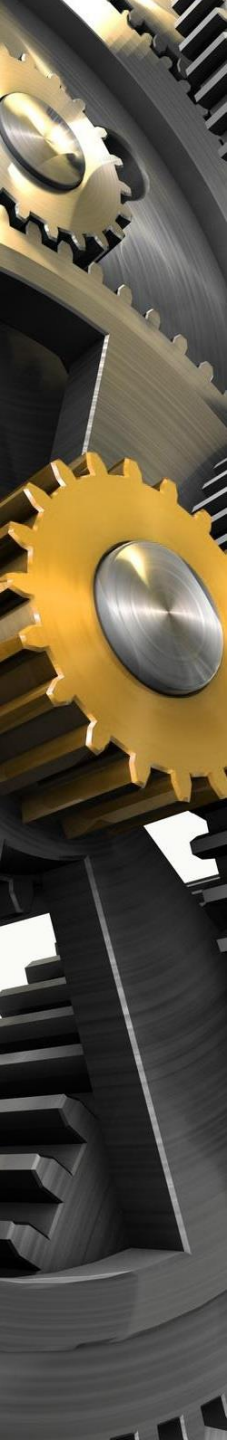
- Instead of planning for software **development**, plan for software **evolution**.
- Instead of designing a **single system**, design a **family of systems**.





Family of Systems

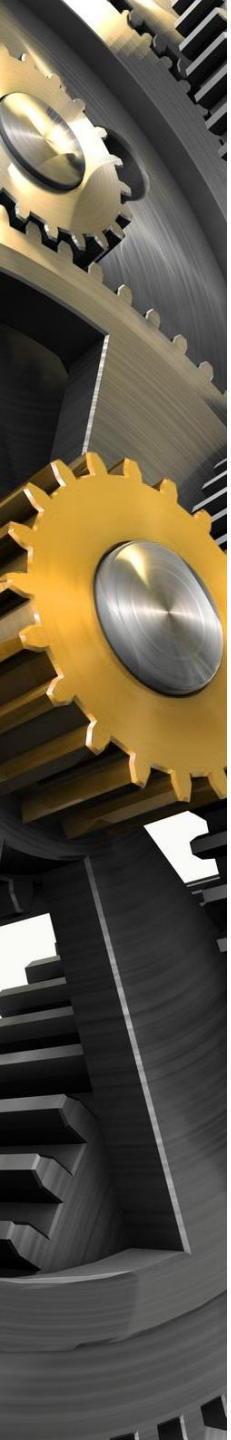
- Software Product Line: "*A set of software-intensive systems that* ***share a common, managed set of features*** *satisfying the specific needs of a particular market segment or mission and that are* ***developed from a common set of core assets*** *in a prescribed way.*" – Software Engineering Institute



Adaptable Design

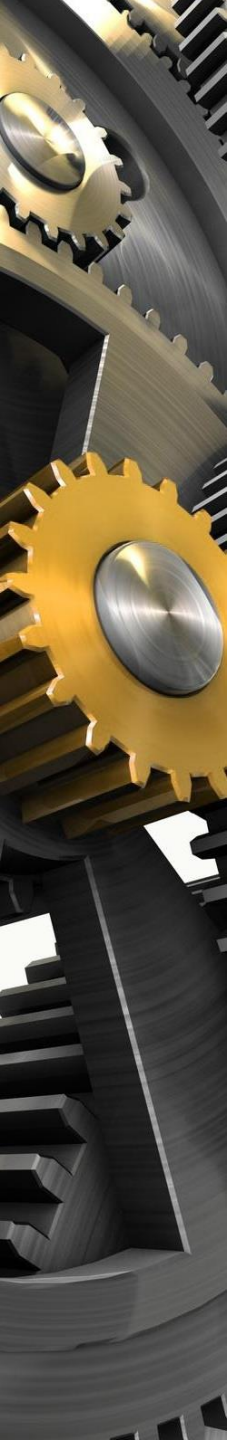
- Adaptable Software Design: A generic software design for a family of systems which does not need to be changed to accommodate new requirements.

- ADUF = Adaptable Design Up Front!

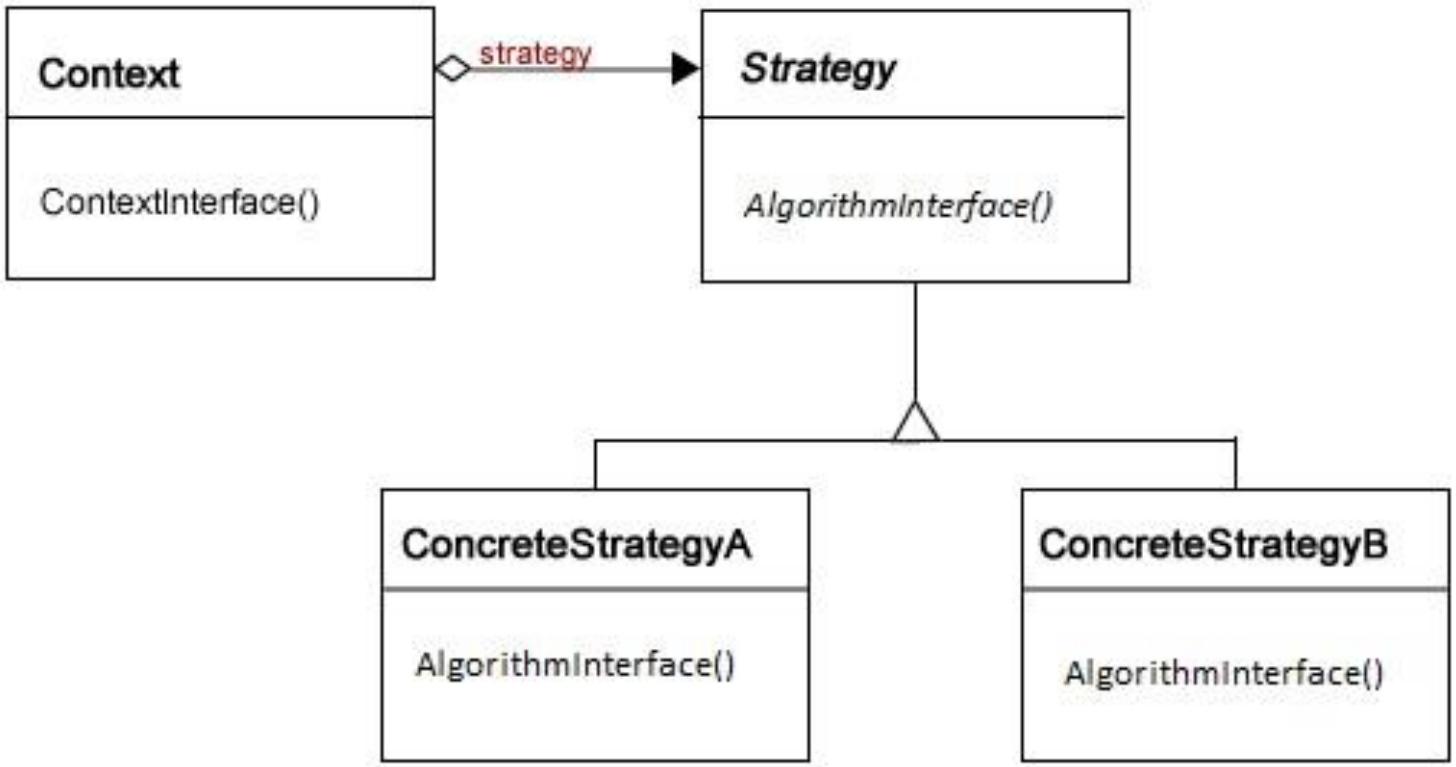


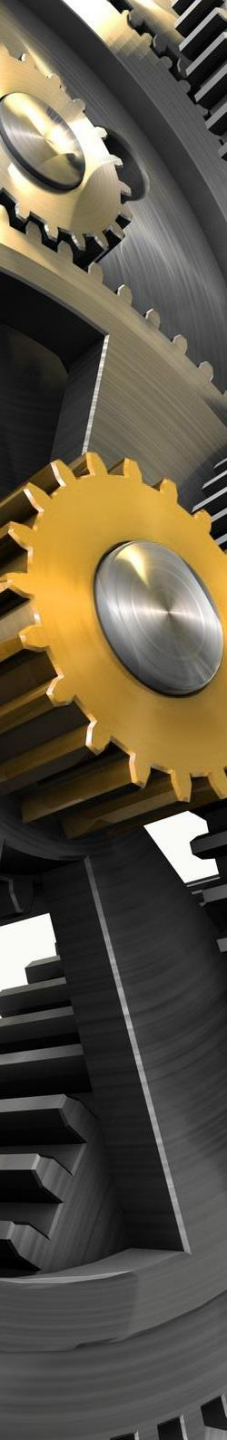
Open/Closed Principle

- *"Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification."* - Bertrand Meyer



Strategy Design Pattern

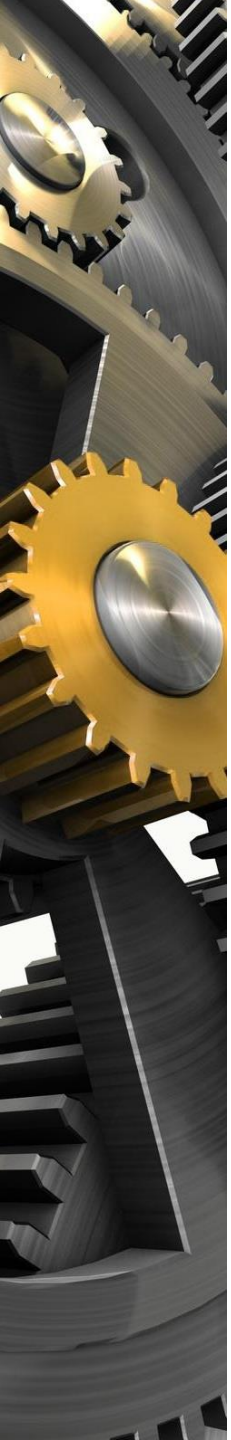


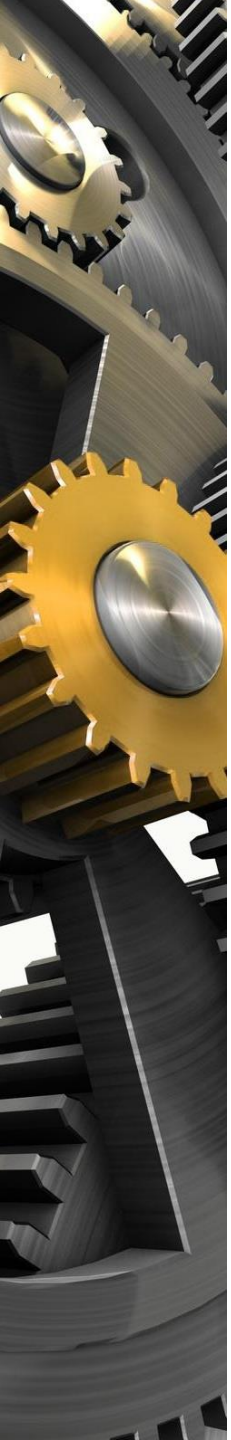


Adaptable Design Up Front and the Open/Closed Principle

- ADUF focuses on the early definition of the "closed" aspects of the design while at the same time allows easy evolution by making this design open to extensions.
- Closed: Captures essential entities and relationships.
- Open: Provides mechanisms for extensibility.

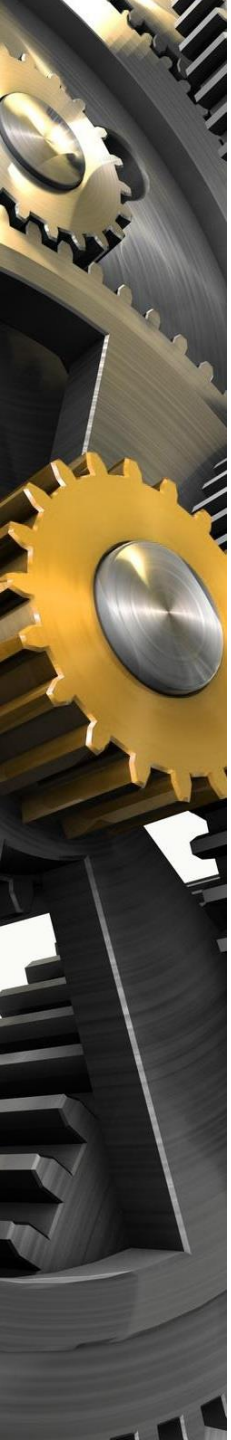
Architecture vs. Interior Design





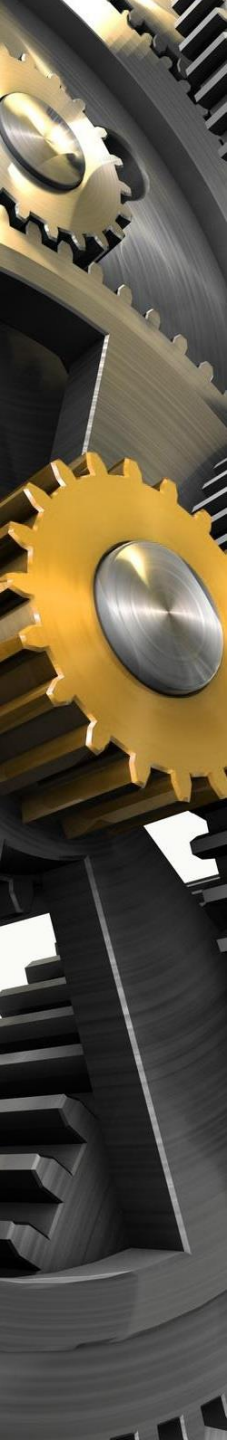
Common Errors

- Too much design up front
- Not enough design up front
- Too much openness
- Not enough openness



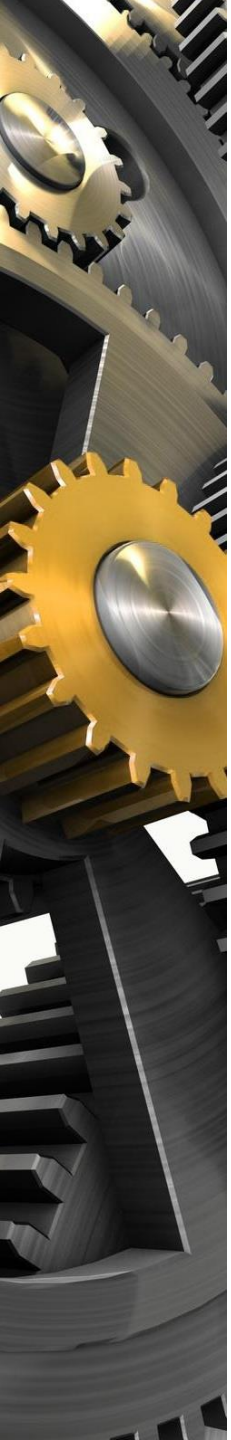
Too much design up front

- Capturing aspects that are not essential as if they were such.
- The design may need to change (thus it is not really "closed").
- Examples:
  - Specific details are not generalized.
  - Designing for a specific system instead of designing for a family of systems.



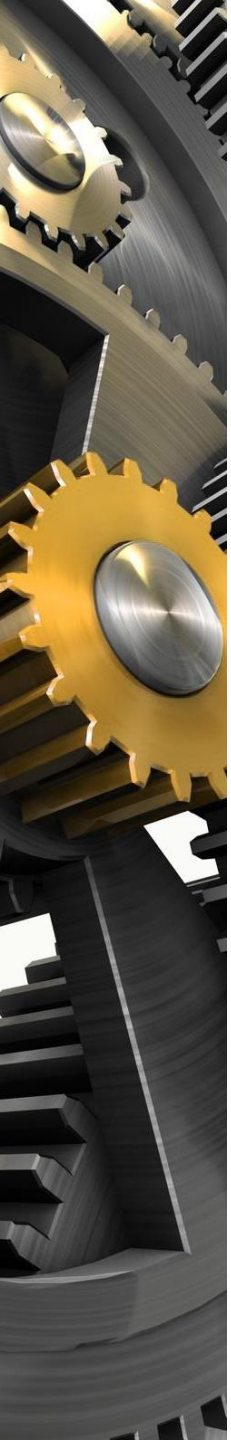
Not enough design up front

- Leaving away some of the essential aspects of the system.
- These aspects will need to be added to the design when other parts of the system are already implemented.
- Example:
  - Adding relationships to new entities may cause changes to existing interfaces.



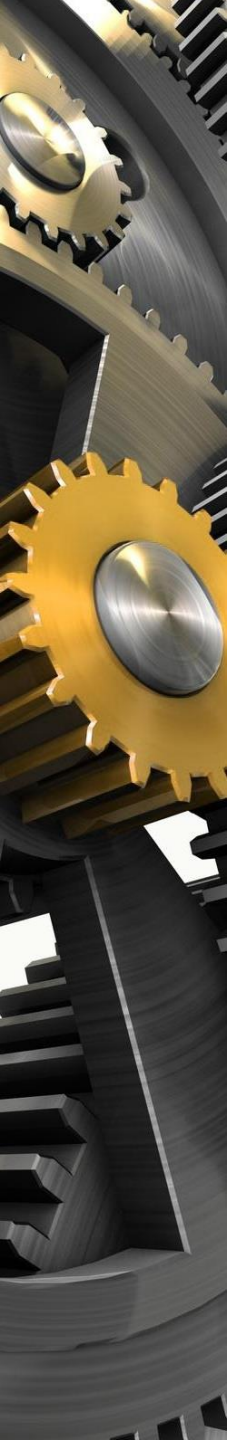
Too much openness

- The design is over engineered, resulting in complexity not really necessary.
- Example:
  - It is always possible to reduce coupling through additional layers of indirection.
  - With many layers, it is difficult to locate where the business logic is implemented.



Not enough openness

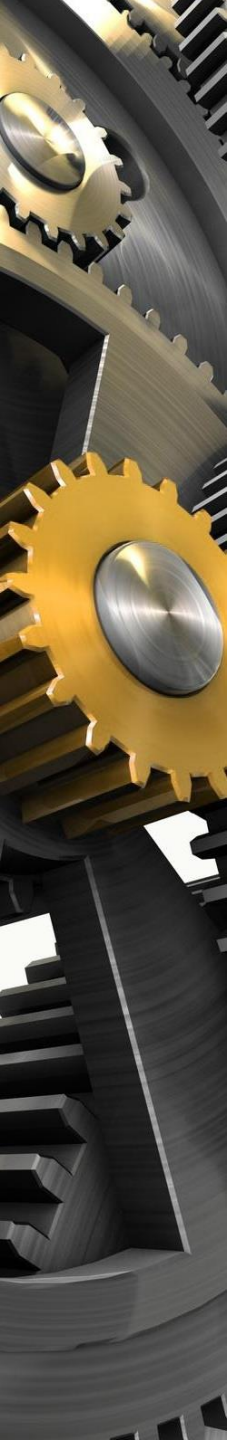
- The design captures all essential aspects, but there are not mechanisms that allow this design to be easily extended.
- Example:
  - If there are Interfaces for the main entities, there should also be Factories to instantiate concrete subclasses.



Extensible Designs

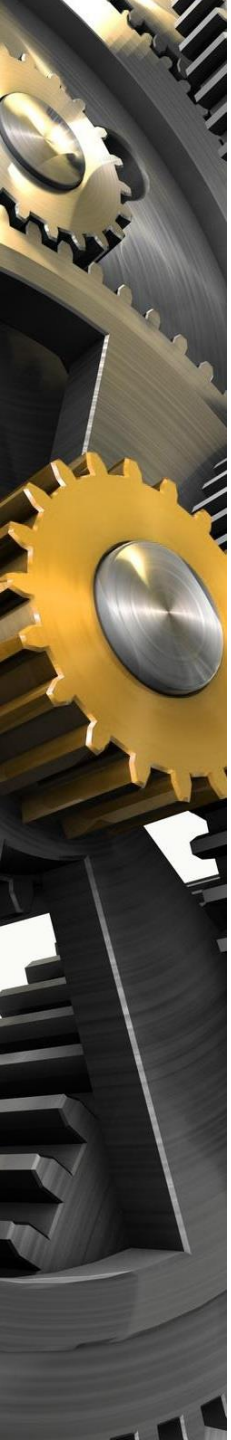
Main alternatives for extensible designs:

- Frameworks
  - Plug-ins
- Platforms



Frameworks

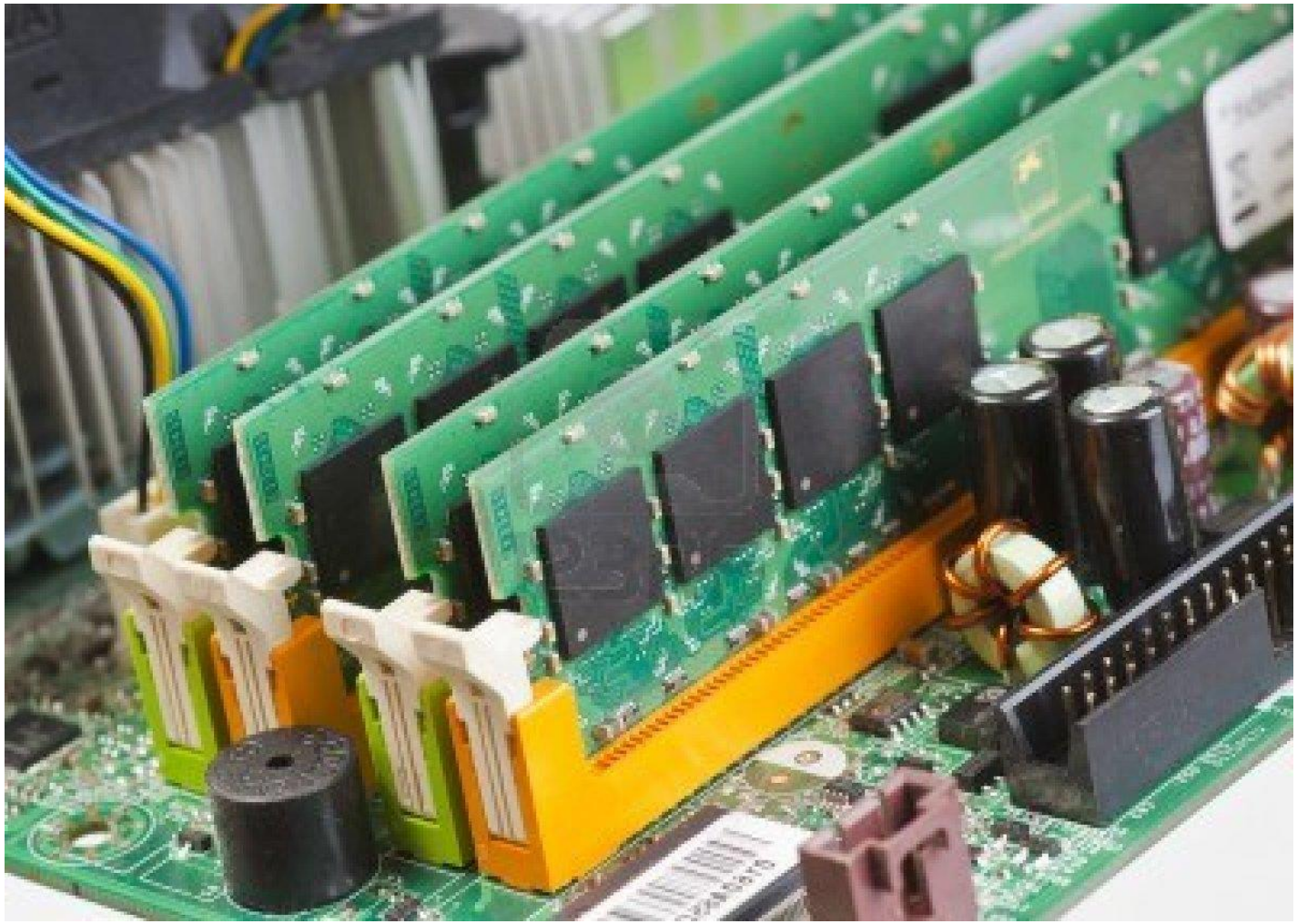
- Framework: "*A software framework is an abstraction in which software providing generic functionality can be selectively changed by additional user-written code, thus providing application-specific software.*"

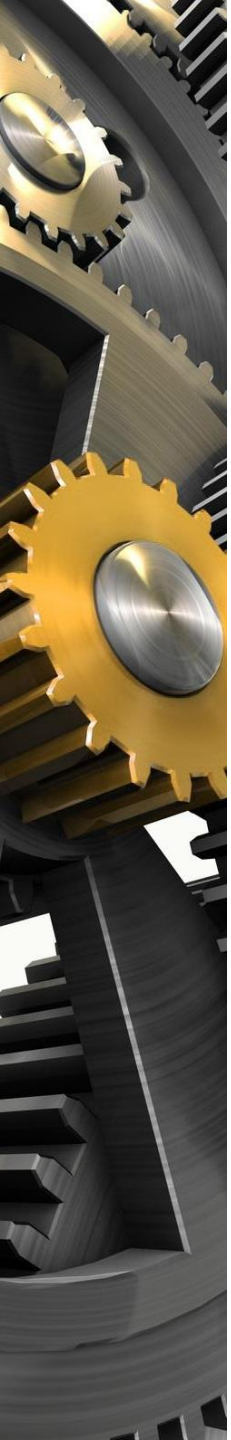


Plug-ins

- Plug-in: *"A plug-in is a software component that adds a specific feature to an existing software application. When an application supports plug-ins, it enables customization."*

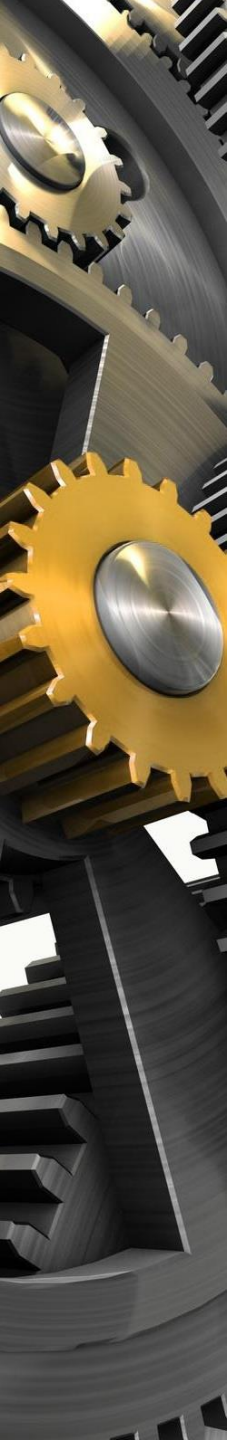
Pluggable Components





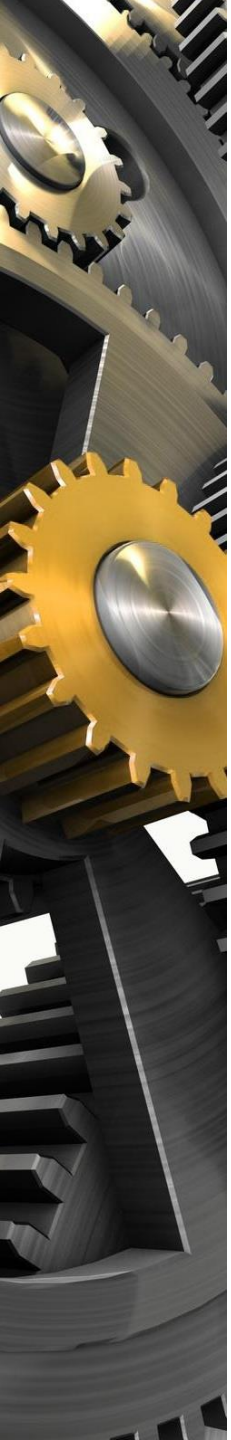
Versioning

- Traditional:
  - Version = modifications + additions.
  - After n iterations: n versions.
- Component-based:
  - Version = combination of new implementations of components.
  - System with m components, after n iterations: n^m versions.



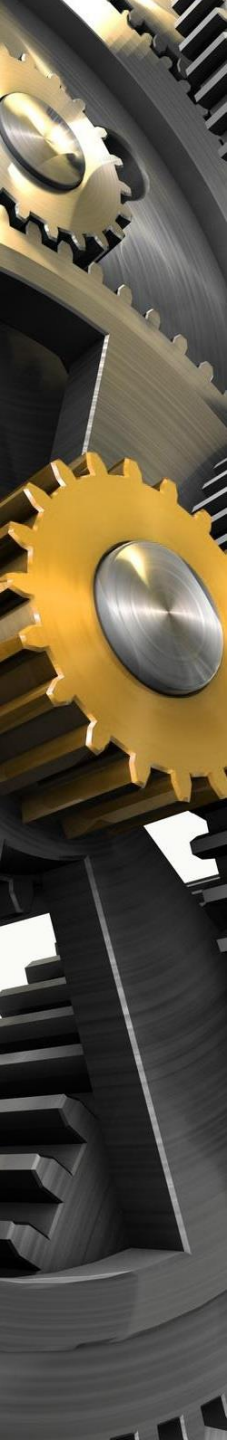
Platforms

- Platform: "*Technology that enables the creation of products and processes that support present or future development.*"
- Software platforms provide services that are used by several applications.
- Platforms and applications may evolve independently of each other.



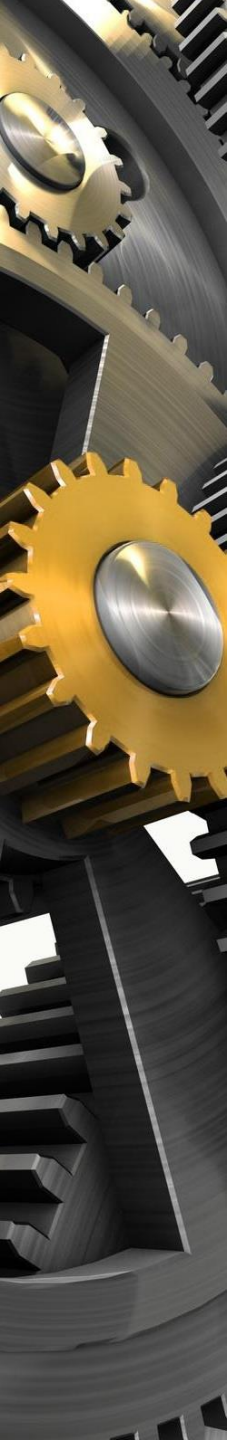
Question 1

- How do you identify the components in your framework or the services in your platform?



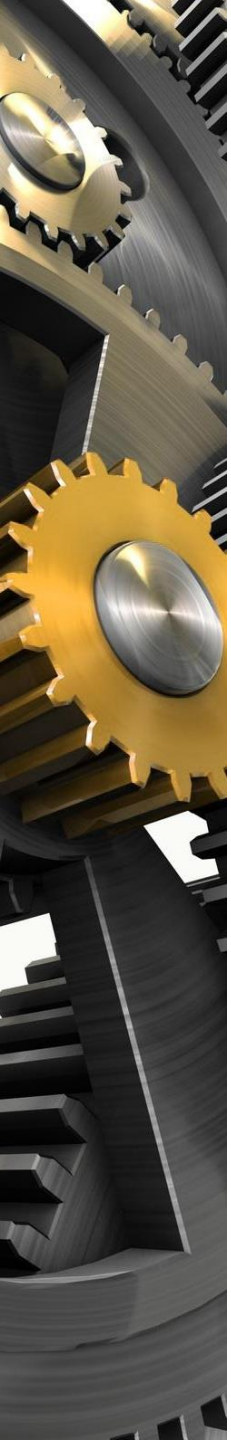
Domain Modeling

- *"A domain model is a conceptual model of all the topics related to a specific problem. It describes the various entities, their attributes, roles, and relationships, plus the constraints that govern the problem domain."*



Question 2

- How do you decouple the components in your framework?

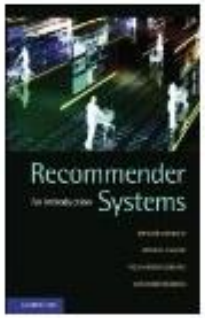


Design Patterns

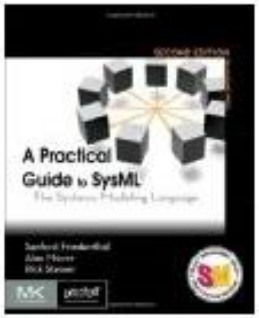
- Reduce coupling using Design Patterns.
- Most patterns avoid direct links between concrete classes, using interfaces and abstract classes.
- Example: Use Observer when several types of objects must react to the same event.

Example: Recommender System

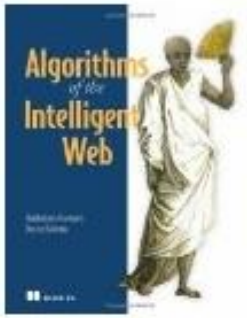
Recommendations for You in Books



Recommender Systems: An Introduction
Dietmar Jannach, Markus Zanker, ...
Hardcover
★★★★½ (3)
~~$69.00~~ $55.96
Why recommended?

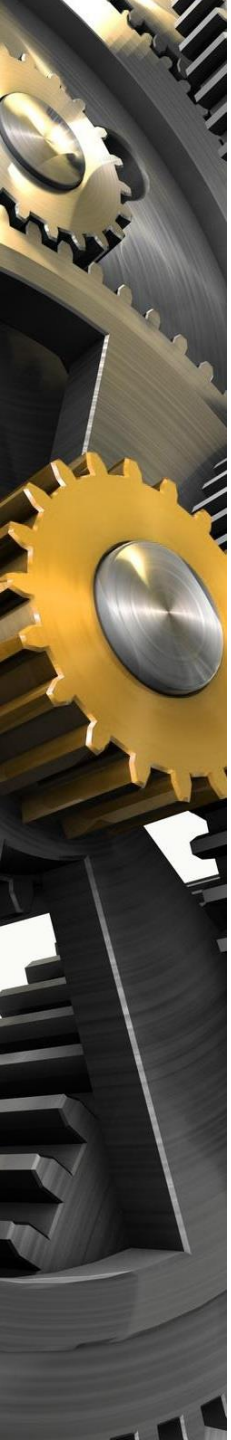


A Practical Guide to SysML, Second...
Sanford Friedenthal, Alan Moore, Rick...
Paperback
★★★★☆ (3)
~~$59.95~~ $51.99
Why recommended?

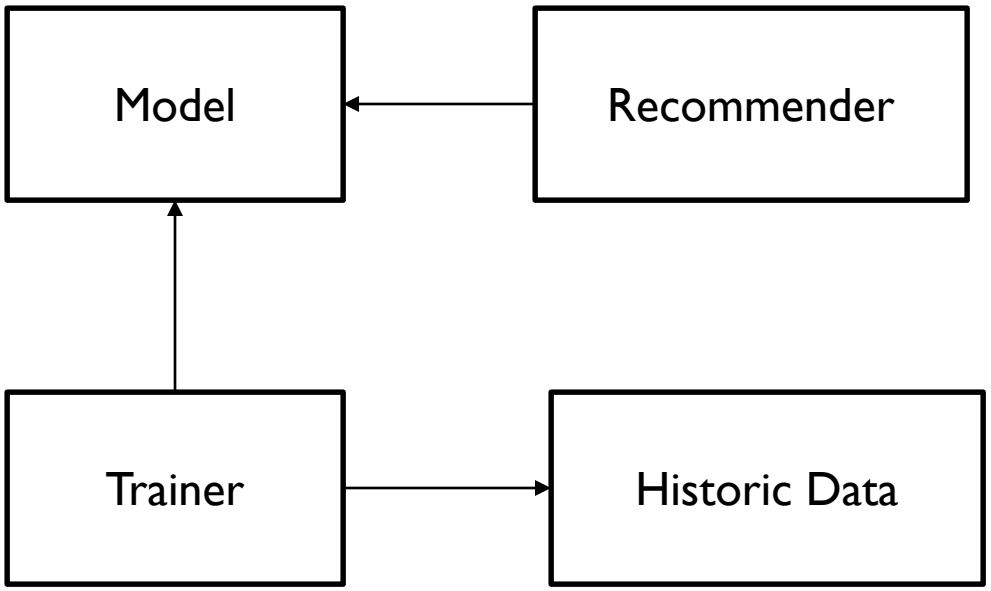


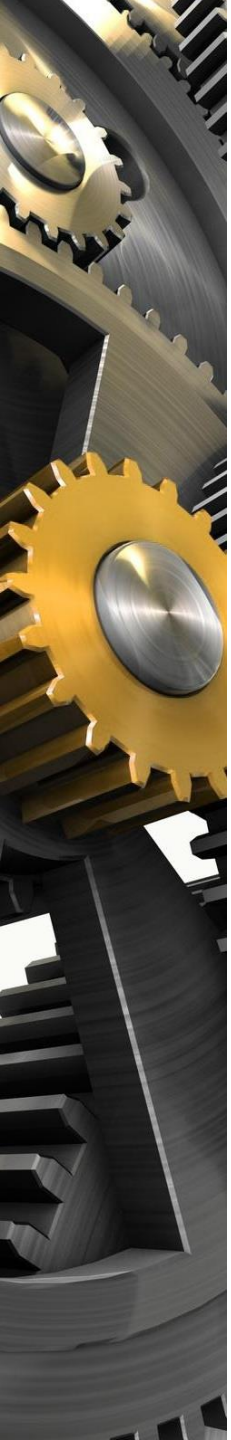
Algorithms of the Intelligent Web
Haralambos Marmanis, Dmitry Babenko
Paperback
★★★★½ (14)
~~$44.99~~ $26.76
Why recommended?

› See more recommendations



Recommender System Model

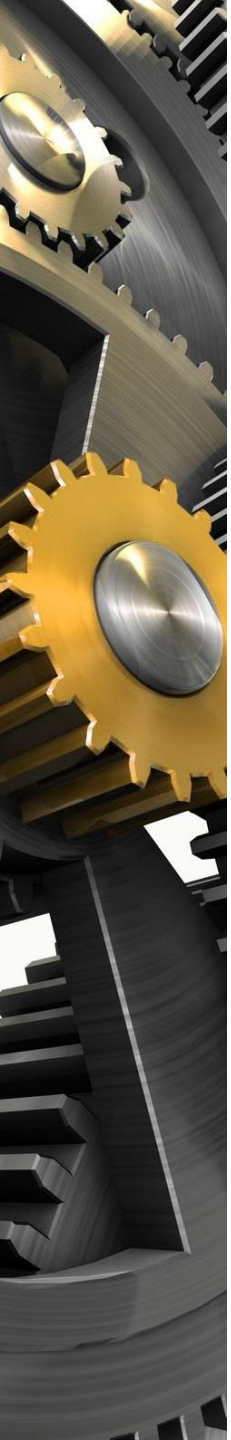




Recommender System Framework

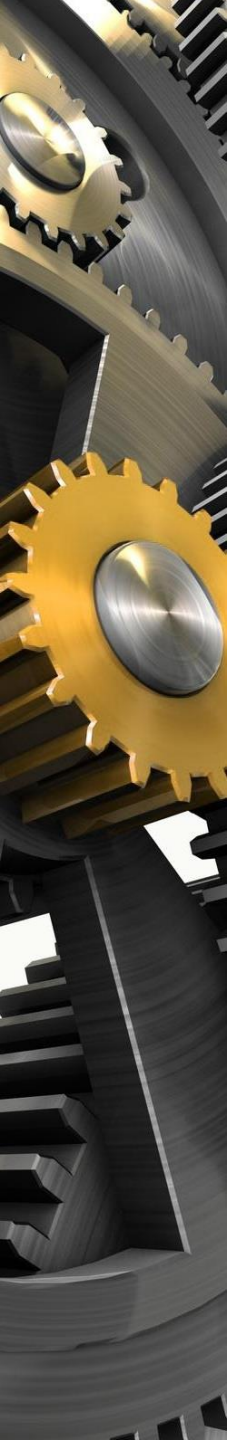
After 1 year of development:

- Historic Data: 13 subclasses.
- Model: 9 subclasses.
- Trainer: 9 subclasses.
- Recommender: 19 subclasses.



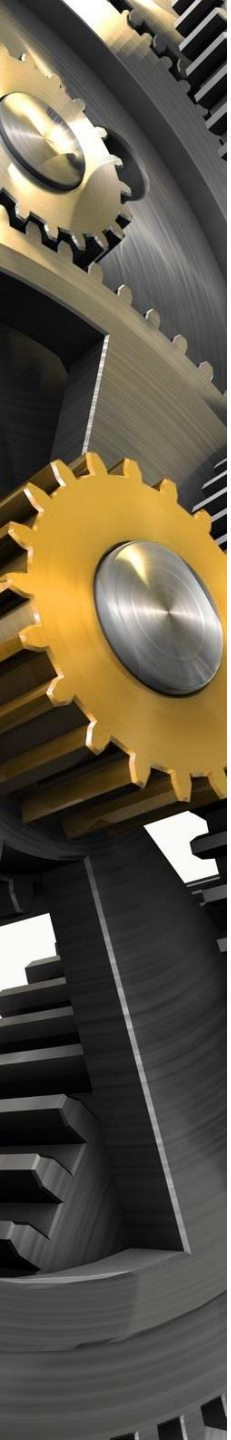
What about YAGNI?

- YAGNI = You aren't gonna need it.
- A principle of extreme programming (XP) that states that a programmer should not add functionality until deemed necessary.
- *"Always implement things when you actually need them, never when you just foresee that you need them."* - Ron Jeffries



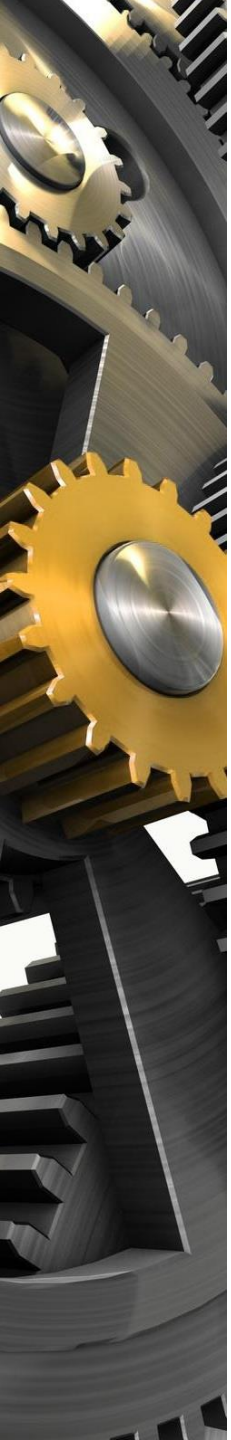
YAGNI vs. ADUF

- YAGNI tells us to avoid doing what we are not sure we will need.
- ADUF tells us:
  - Define the things you are sure you will need in any case.
  - Prepare for the things that you may need in the future (adaptability).
  - NIAGNI: "No, I am gonna need it!"



Conclusions

- Software systems must evolve over time.
- This evolution should be planned and supported trough Adaptable Software Designs.
- Domain Modeling: Identify the entities and relationships that form the "closed" part of the design.
- Framework Design: Design a framework that represents the model and add mechanisms for extensibility, making it "open".



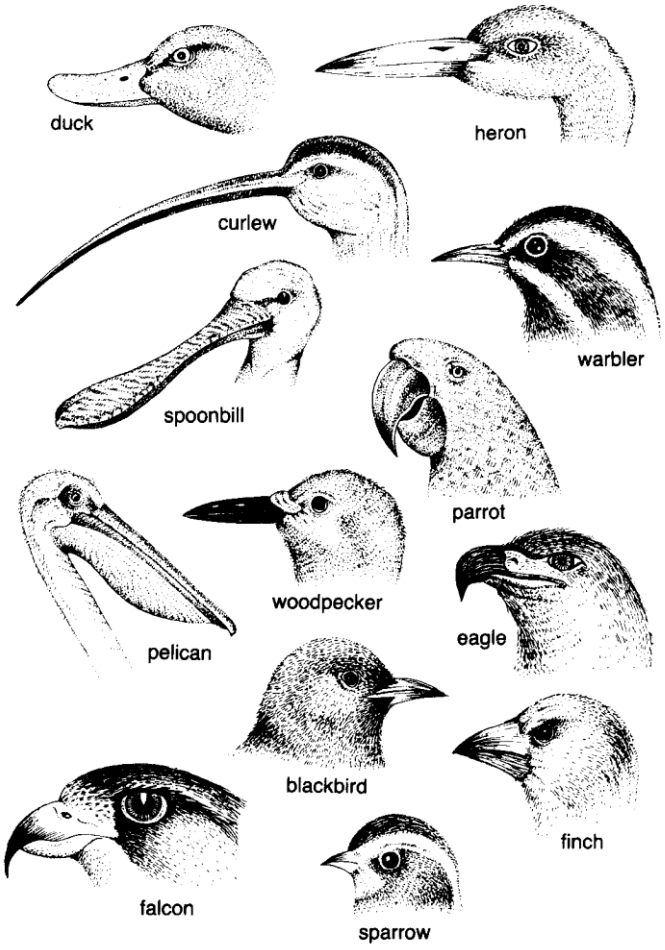
Related Work

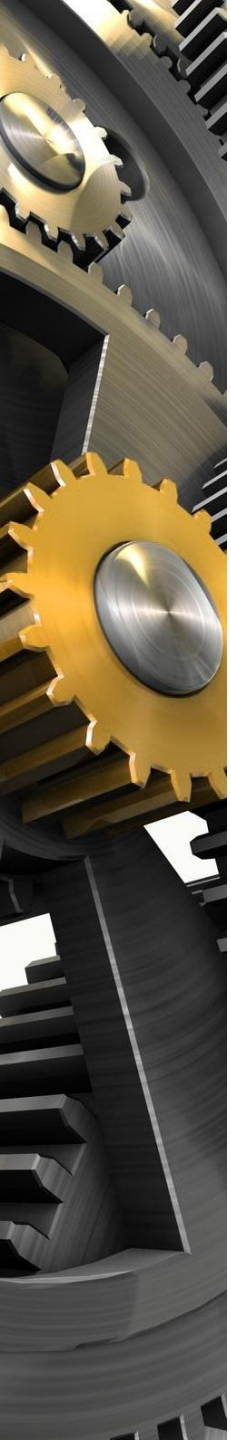
- Software Engineering Institute:
  - Software Product Lines
- Alistair Cockburn:
  - Walking Skeleton
  - Incremental Rearchitecture
- Neal Ford:
  - Emergent Design
  - Evolutionary Architecture
- Simon Brown:
  - Just Enough Design Up Front

Adaptability & Evolution

"It is not the strongest of the species that survives, nor the most intelligent that survives. It is the one that is the most adaptable to change."

Charles Darwin





Thanks!

Q&A

http://EffectiveSoftwareDesign.com