Azrieli College of Engineering Jerusalem

עזריאלי מכללה אקדמית להנדסה ירושלים

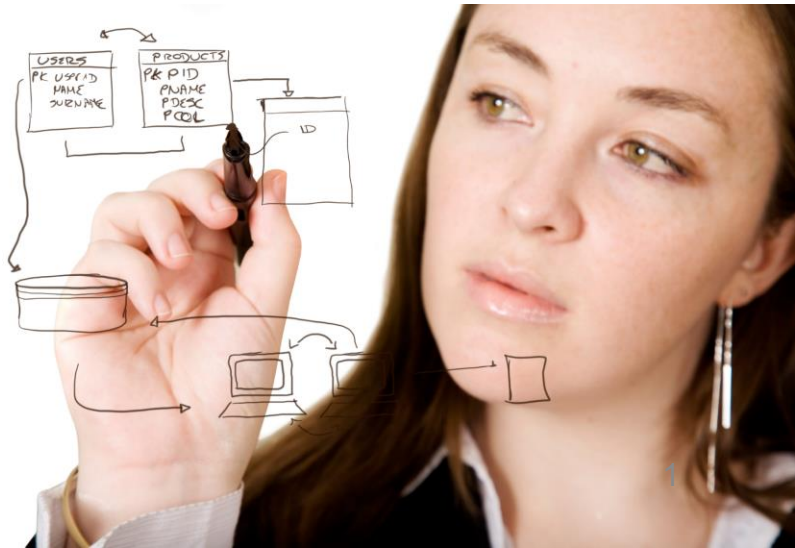
מודלים לפיתוח מערכות תוכנה

Software Systems Modeling

קורס 12003

סמסטר ב' תשע"ה

9-10. Models for Software Engineering Processes: Design



ד"ר ראובן יגל
robi@post.jce.ac.il

הפעם

- מידול תהליכי פיתוח תוכנה
  - עבודת צוות ומחזור חיים
  - תיכון וארכיטקטורה -> High Level Models
  - בדיקות
  - מימוש (git? Code-review?)
- המשך מצגות פרויקט
- שב בנושא מידול ותיכון

נושאים

- תיכון וארכיטקטורה
- האם עוד צריך אותם בסביבת אג'ייל
- מודלים כלליים לארכיטקטורה
  - Adaptable Design –
  - design patterns – הקשר לתבניות תיכון –
  - ~~MDA? –~~
  - Domain Driven Design –
  - =>: CQRS / EventSourcing, –

מקורות

- JCE Soft.Eng. Class (resources over there)
- Haim Macabbe, Adaptable Design Up Front, http://effectivesoftwaredesign.com/adaptable-design-up-front/
- Rotem Hermon, Change Driven Design, http://effectivesoftwaredesign.com/2015/05/12/rotem-hermon-on-change-driven-design/

DDD

- Evans, Domain Driven Design…
- Infoq minibook, DDD quickly
- Domain Driven Design (slideshare)
- Pluralsight course (subscription)
  - Related sample
  - Materials (private)
- Other Repos:
  - http://www.geoffmazeroff.com/2015/05/27/domain-driven-design-concepts-reference/
  - https://github.com/andreazevedo/petstore-ddd-csharp

שאלות

- מה הוא תיכון (design)?
- מה היא ארכיטקטורת תוכנה?
- מה תפקידם בתהליך הפיתוח?
- מה תפקידו של ארכיטקט?
- האם עדיין צריך אותם?
  - "Working Software over Comprehensive Documentation"…
  - Just Enough Software Architecture
  - Simon Brown: "a good software architecture enables agility"
- האם שפות וכלים יכולים לעזור?

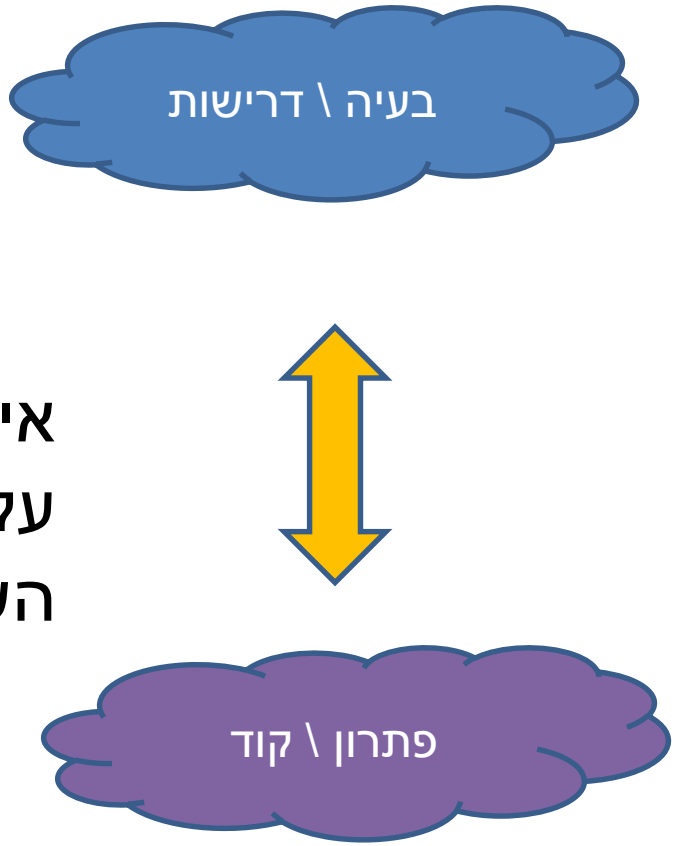
Design תיכון

- Design (wikipedia): A plan (with more or less detail) for the structure and functions of an artifact, building or system.
- “sufficient information for a craftsman of typical skill to be able the build the artifact without excessive creativity” – Guy Stelle
- “Design is the thinking one does before building” - Richard P. Gabriel
- “design is there to enable you to keep changing the software easily in the long term” - Beck
- פלט משלב זה: SDS – מפרט תיכון תוכנה

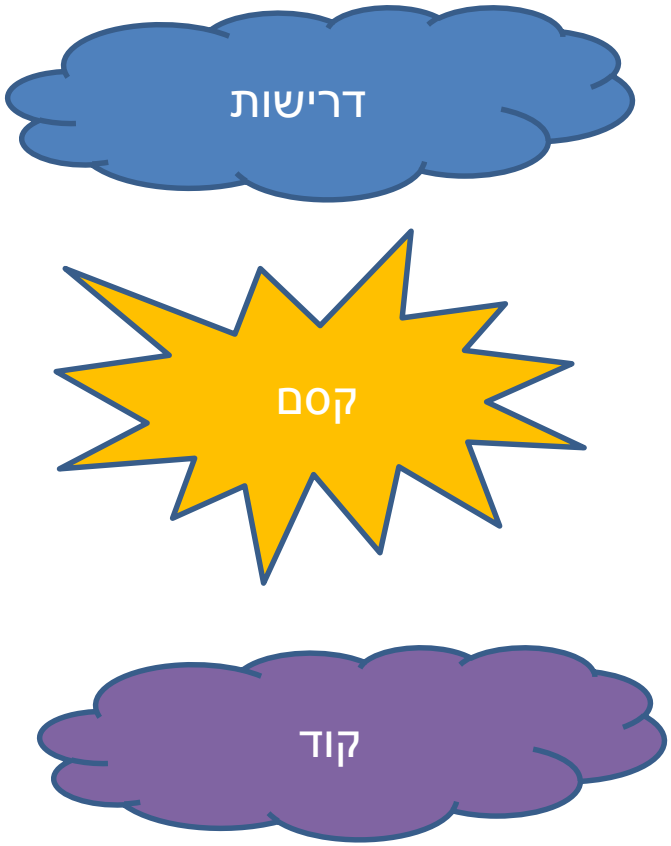
תיכון ואיכות

- From: Jones, "Software Defect Origins-and Removal", 2013:
  - The major defect origins include:
  - 1. Functional requirements
  - 2. Non-functional requirements
  - 3. Architecture
  - 4. Design
  - 5. New source code
  - 6…

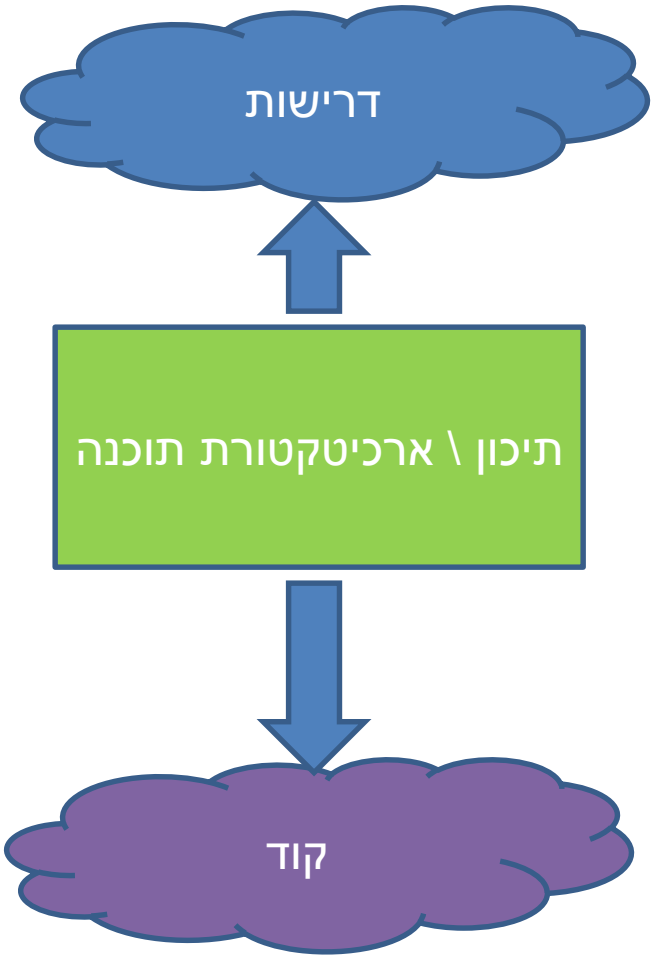
הבעיה העיקרית



תשובה ראשונה



תשובה יותר מעשית (היום)



מספקת מסגרת
כללית לבנות
ולפתח את התוכנה

Booch: Traits of Successful Projects

- A successful software project is one in which the deliverables satisfy and possibly exceed the customer expectations, the development occurred in a timely and economical fashion, and the result is resilient to change and adaption.
- … several traits that are common to virtually all successful oo systems we have encountered and noticeable absent from the ones we count as failures:
  - Existence of a strong architectural vision
  - Application of a well managed iterative and incremental development lifecycle

ארכיטקטורה: הגדרה

- ארכיטקטורה של מערכת מתארת את המבנה העיקרי שלה, כך שהיא תתאים לצורכי הלקוח תוך כדי עמידה באילוצי טכנולוגיה ותקציב
  - המרכיבים העיקריים וההתנהגות שלהם
  - הקשרים בין מרכיבים אלו

[מתוך מכתב של Brooks]

ייצוג היבטים שונים של התוכנה באופן מופשט

Yegge About Bezos

- His Big Mandate went something along these lines:

  1) All teams will henceforth expose their data and functionality through service interfaces.

  2) Teams must communicate with each other through these interfaces.

  3) There will be no other form of interprocess communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. The only communication allowed is via service interface calls over the network.

  4) It doesn't matter what technology they use. HTTP, Corba, Pubsub, custom protocols -- doesn't matter. Bezos doesn't care.

  5) All service interfaces, without exception, must be designed from the ground up to be externalizable. That is to say, the team must plan and design to be able to expose the interface to developers in the outside world. No exceptions.

  6) Anyone who doesn't do this will be fired.

  7) Thank you; have a nice day!

  Ha, ha! You 150-odd ex-Amazon folks here will of course realize immediately that #7 was a little joke I threw in, because Bezos most definitely does not give a shit about your day.

- “One begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a design decision from the others.”
  - David L. Parnas “On the criteria To Be Used in Decomposing Systems into Modules”, 1972

- “The dependencies between packages in a design should be in the direction of the stability of the packages. A package should only depend upon packages that are more stable than it is.”
  - Robert C. Martin The Stable Dependencies Principle, 1997

עוד הגדרות

- Architecture represents the significant design decisions that shape a system, where significant is measured by cost of change (Booch 2006)
- The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment and the principles guiding its design and evolution (IEEE1471 2007)
- The form of a system (Coplien, Lean Arch. 2010)
- “things that people perceive as hard to change” (Fowler, Who Needs an Architect, 2003)
- “the set of design decision which, if made incorrectly, may cause your project to be cancelled” - Eoin Woods (SEI 2010)
- “In a sense we get the architecture without really trying. All the decisions in the context of the other decisions simply gel into an architecture” – Cunningham, 2004
- "irreversible decisions in the large“ J. B. Rainsberger
- Many more @ SEI Community Software Architecture Definitions

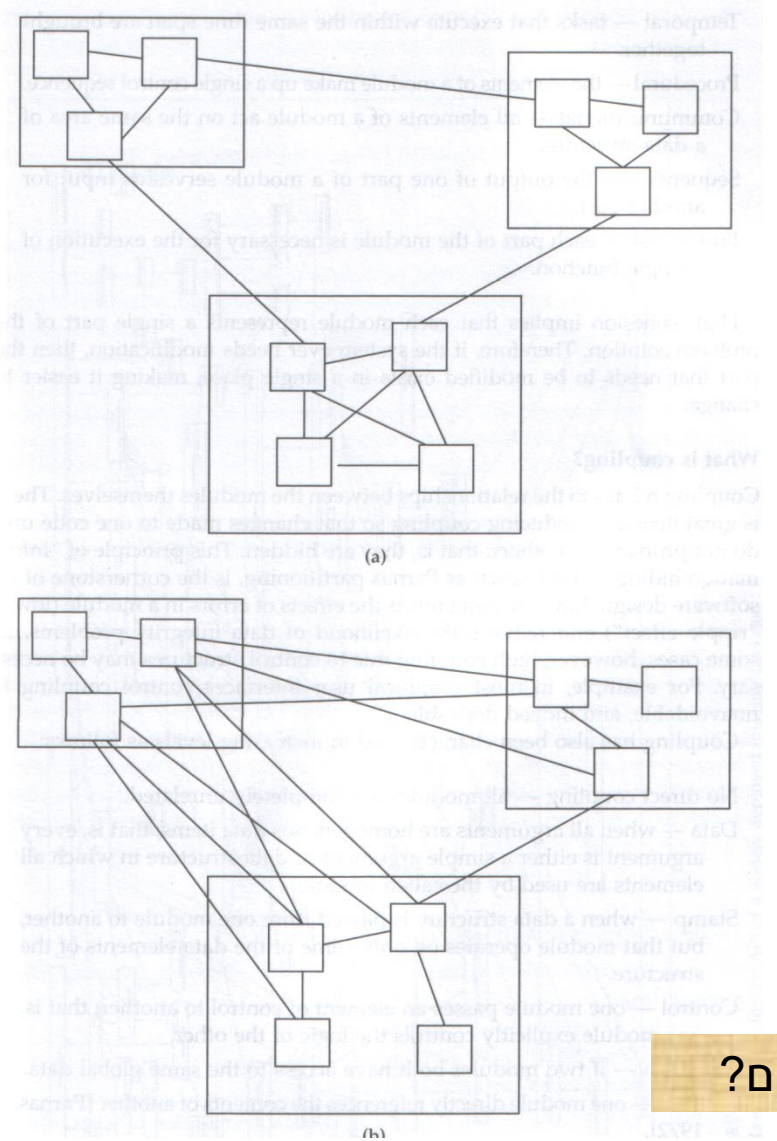
- http://en.wikipedia.org/wiki/Software_architecture
  'To date there is still no agreement on the precise definition of the term “software architecture”. However, this does not mean that individuals do not have their own definition of what software architecture is. This leads to problems because many people are using the same terms to describe differing ideas.'

מאפייני ארכיטקטורה טובה

- חלוקה לשכבות ברורות המייצגות כל אחת הפשטה* של המערכת ומספקת ממשק ברור
- הפרדה בין הממשק והמימוש של כל שכבה
- פשטות: שימוש בהפשטות* ומנגנונים מקובלים

* Uncle Bob Martin: “Abstraction is the elimination of the irrelevant and the amplification of the essential.“

מדדים מרכזיים לתיכון תוכנה



- אנליזה ומודולריות: חלוקה לרכיבים והקשרים בינהם
- Coupling – צימוד
  - מדד תלות באחרים
- Cohesion – לכידות
  - מדד ענייניות
- עקרונות מפורטים יותר בהמשך

האם כדאי שהמדדים יהיו גבוהים או נמוכים?

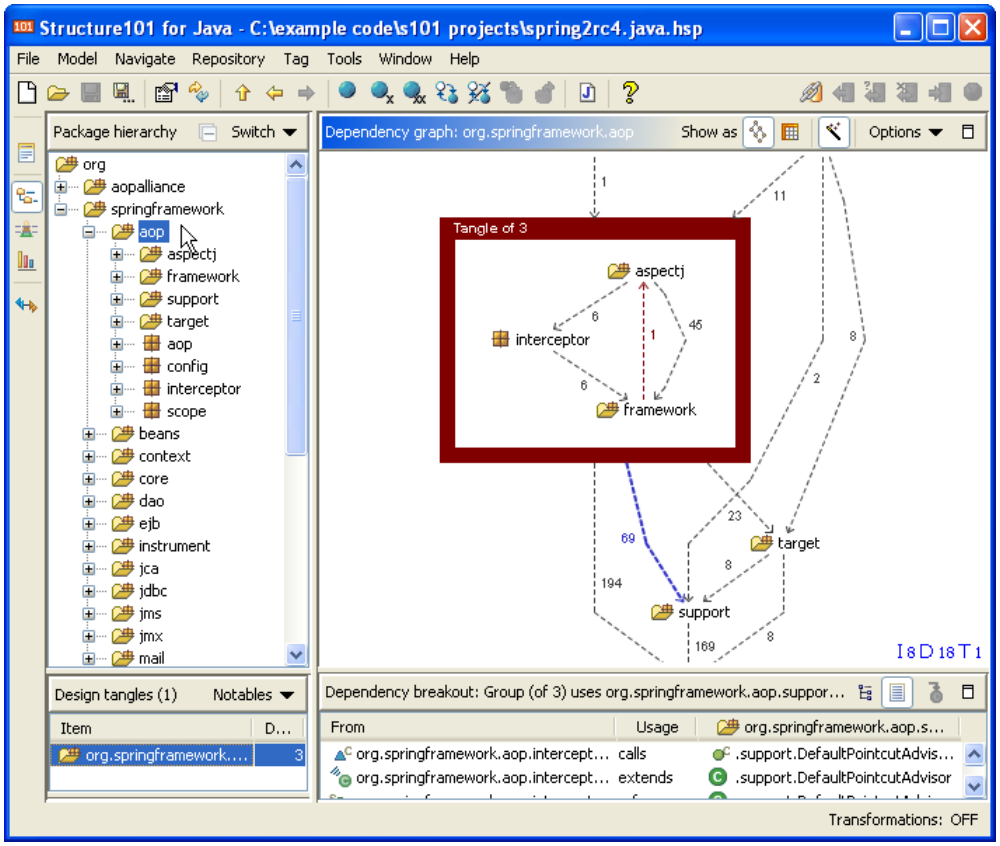
Cohesion Types [Yourdon]

7.	Informational cohesion	(Good)
6.	Functional cohesion	
5.	Communicational cohesion	
4.	Procedural cohesion	
3.	Temporal cohesion	
2.	Logical cohesion	
1.	Coincidental cohesion	(Bad)

http://en.wikipedia.org/wiki/Cohesion_(computer_science)#Types_of_cohesion
http://highered.mcgraw-hill.com/sites/dl/free/0073191264/371536/Ch07.pdf
http://robots.thoughtbot.com/post/23112388518/types-of-coupling

מדדים וכלים

- מדדי סיבוכיות שונים (דוג' Todo MVC)
- כלים, למשל: Structure101, Ndpend, ruby metric_fu, Codeclimate



ארכיטקטורה עוזרת ב:

- הבנת המערכת – תאור הקשרים בין מרכיבים
- שימוש חוזר – לאור החלוקה הכללית לרכיבים, זיהוי הזדמנויות
- טיפול בדרישות לא-פונקציונליות (אילוצים)
- מענה לסיכונים
- מימוש – חלוקה למשימות (במיוחד בצוותים גדולים), וכך נעבור מדרישות למימוש
- ניהול – עוזרת להבין את כמות העבודה ולעקוב אחרי התקדמות
- תקשורת – מייצרת הבנה ואוצר מילים, "תמונה אחת שווה אלף מילים"
  - (אבל ב- Agile: שורת קוד אחת שווה אלף תמונות :-)
- אמורה לאפשר שינוי!

ארכיטקטורה ואילוצי מערכת

- למשל, ביצועים גבוהים:

חלוקה לרכיבים מקביליים, זיהוי צווארי בקבוק, ניהול תקשורת בין רכיבים; קצבים

- אבטחה:

לאלו חלקים מותר לגשת, זיהוי מקומות לשמירה נוספת, הוספת רכיבים שאפשר לסמוך עליהם

- גמישות לשינויים, הרחבתיות:

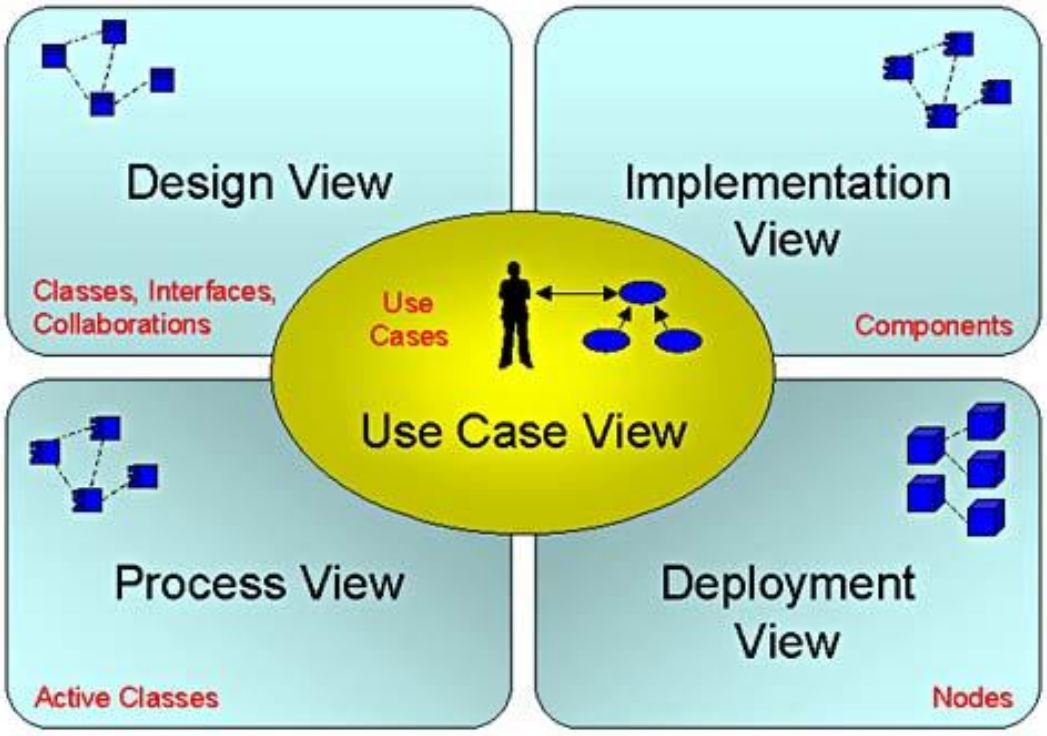
הפרדת רשויות בין החלקים כך ששינויים לא יחלחלו לכלל המערכת

בעצם ישנם היבטים שונים

- "היבט" מאיר ומדגיש אוסף של החלטות ראשיות בתכנונה של מערכת
  - כיצד תורכב המערכת מחלקים שונים
  - היכן הממשקים העיקריים בין החלקים
  - מאפיינים עיקריים של החלקים
  - מידע שמאפשר המשך ניתוח והערכות

החשיבות של היבטים (Views)

- היבטים שונים נדרשים, כדי להבין ממדים שונים של המערכת (Philippe Kruchten)



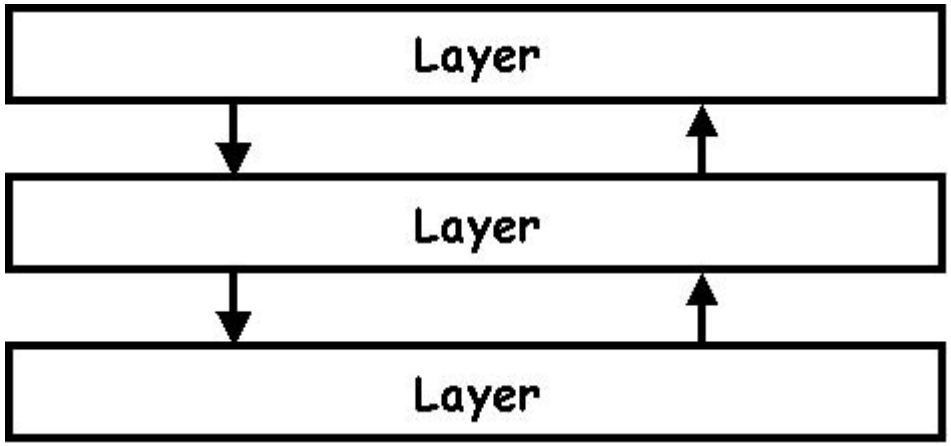
Booch

כמה דוגמאות לארכיטקטורה

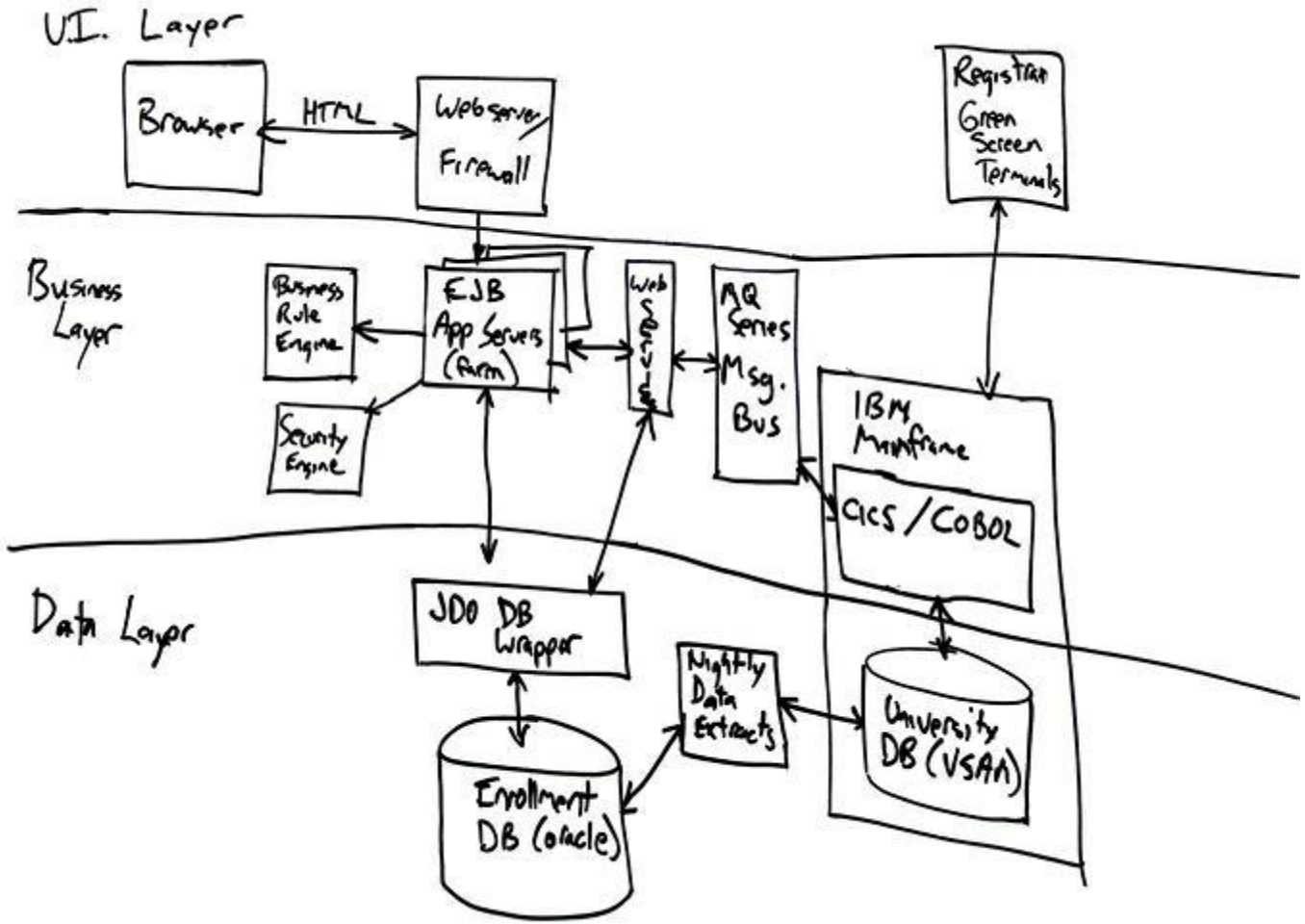
- תבניות שימושיות שהתפתחו מתוך נסיון
- מתאימות בהקשר מסוים
- בסיס להתאמה לבעיה הנוכחית
- בהמשך: תבניות תיכון\עיצוב Design Patterns

"Architecture should reflect use cases not a framework" - R. Martin

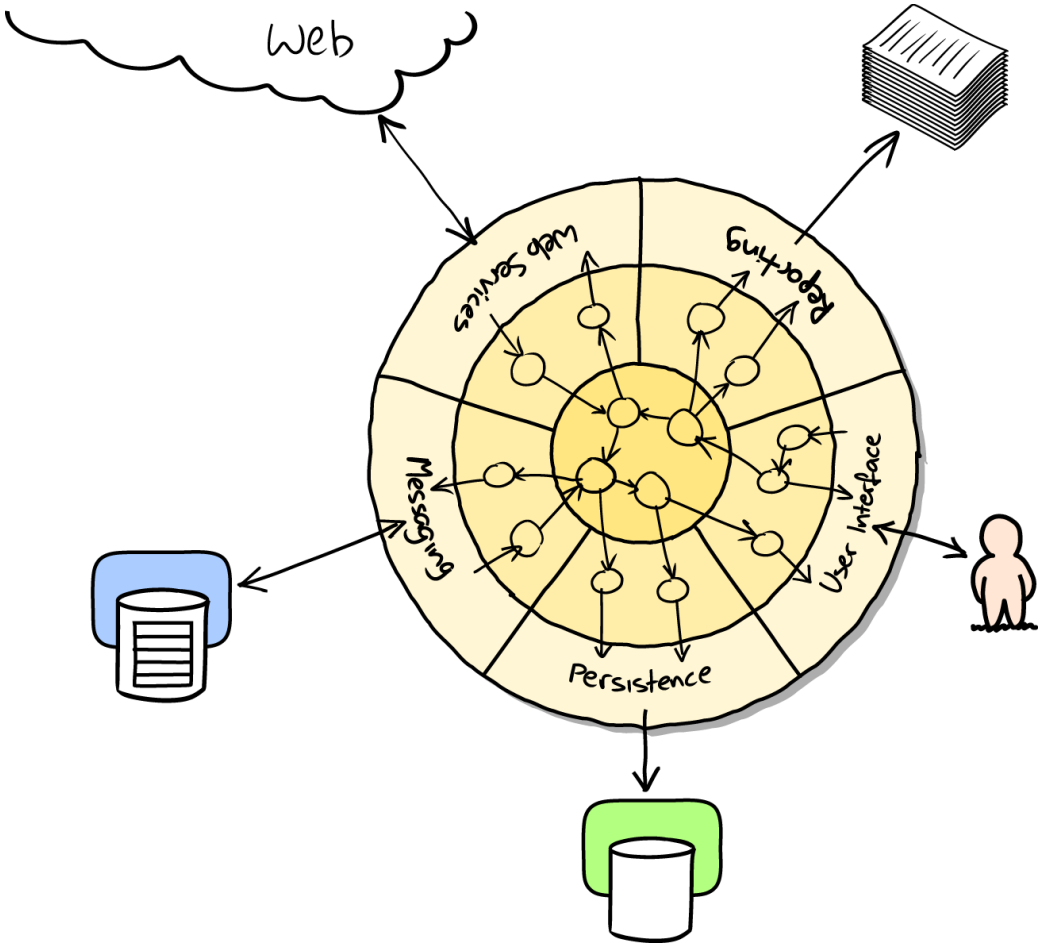
שכבות



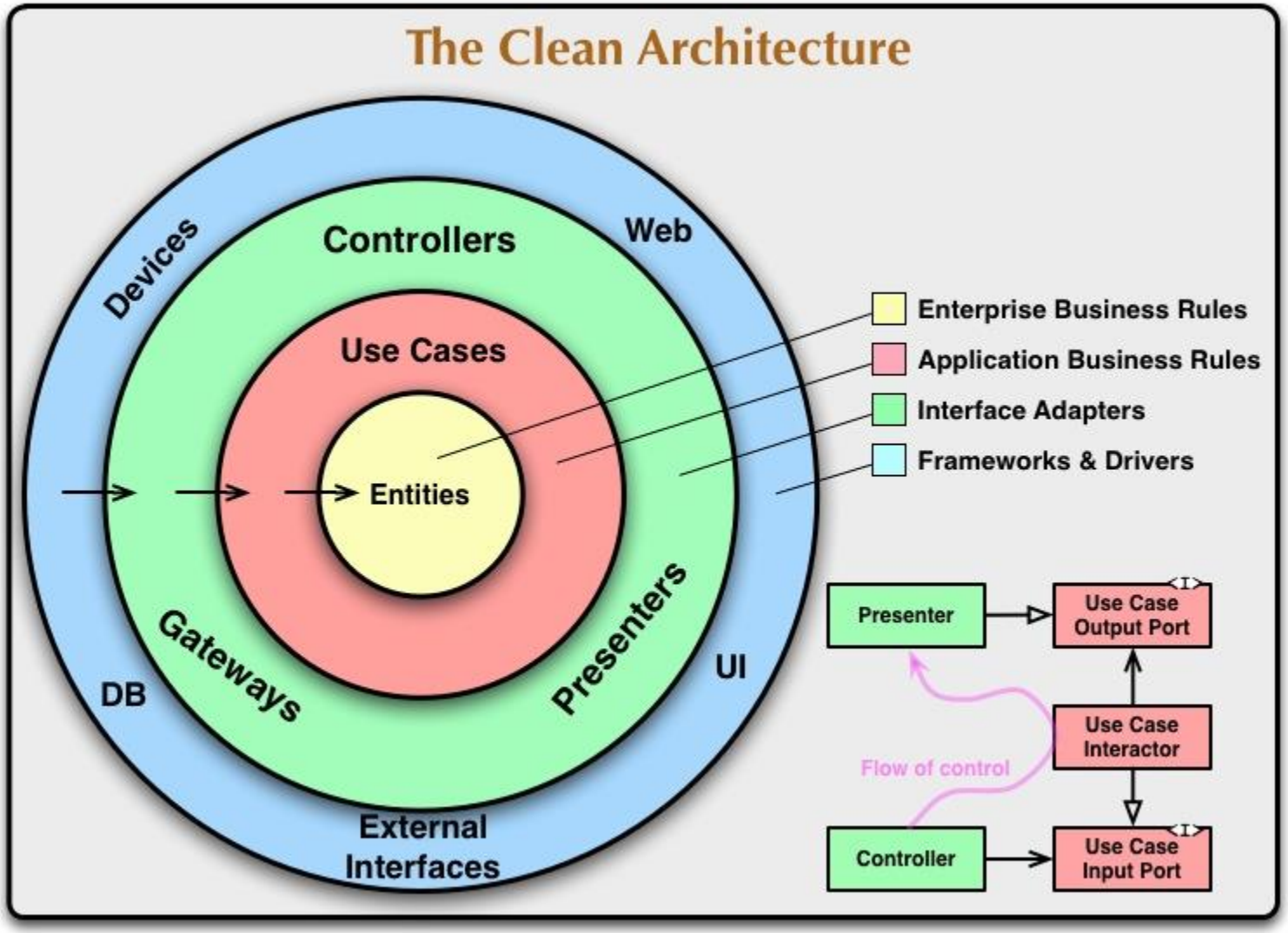
- דוגמאות: מערכות הפעלה, פרוטוקולי תקשורת, N-tier


U.I. Layer
Browser
HTML
Web server
Firewall
Registrar
Green Screen Terminals
Business Layer
Business Rule Engine
EJB App Servers (farm)
Web Services
MQ Series Msg. Bus
Security Engine
IBM Mainframe
CICS / COBOL
Data Layer
JDO DB Wrapper
Nightly Data Extracts
University DB (VSAM)
Enrollment DB (oracle)

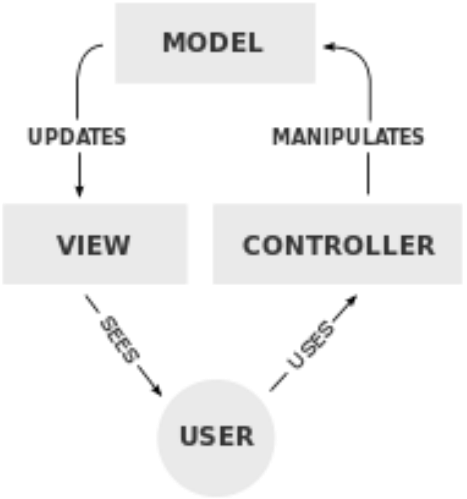
Hexagonal Architecture / Ports & Adapters



Clean Arch.


The Clean Architecture
Devices
Controllers
Web
Use Cases
Entities
Gateways
Presenters
DB
UI
External Interfaces
Enterprise Business Rules
Application Business Rules
Interface Adapters
Frameworks & Drivers
Presenter
Use Case Output Port
Use Case Interactor
Flow of control
Controller
Use Case Input Port

Model View Controller

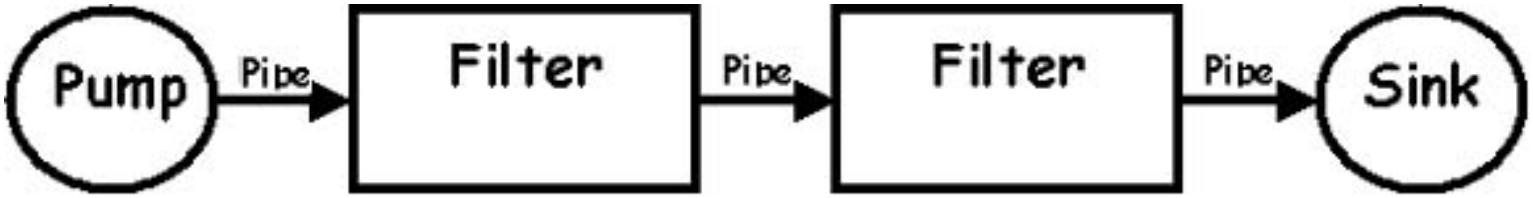


- מימושים עכשוויים, לדוג':
Ruby on Rails, ASP.Net MVC, Angular.js

REST

- ארכיטקטורה לבניית מערכות מבוזרות
- אילוצים
  - הפרדה ללקוח ושרת
  - אין שמירת מצב לקוח
  - שימוש במטמון
  - שכבות
- => מדרגיות, הפצה נוחה ועוד
- דוגמא: שכתוב טוויטר לעמידה בצמיחה

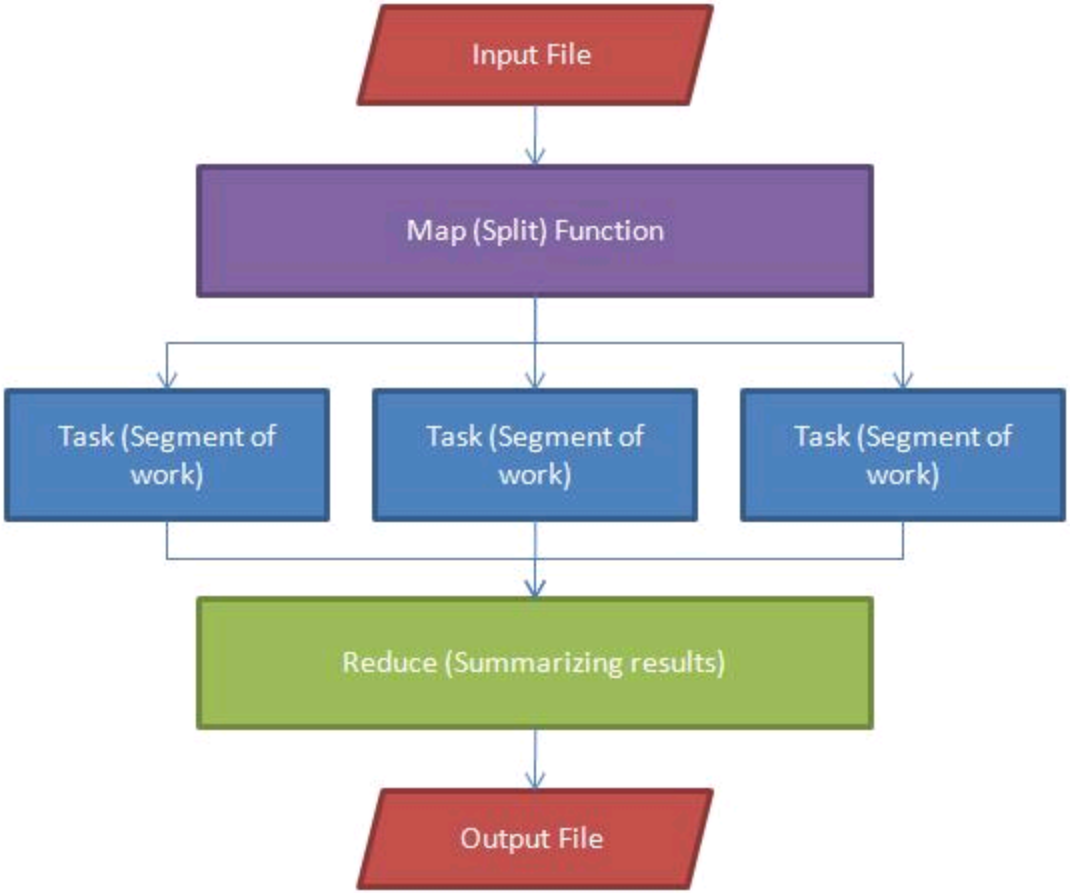
Pipe and filter



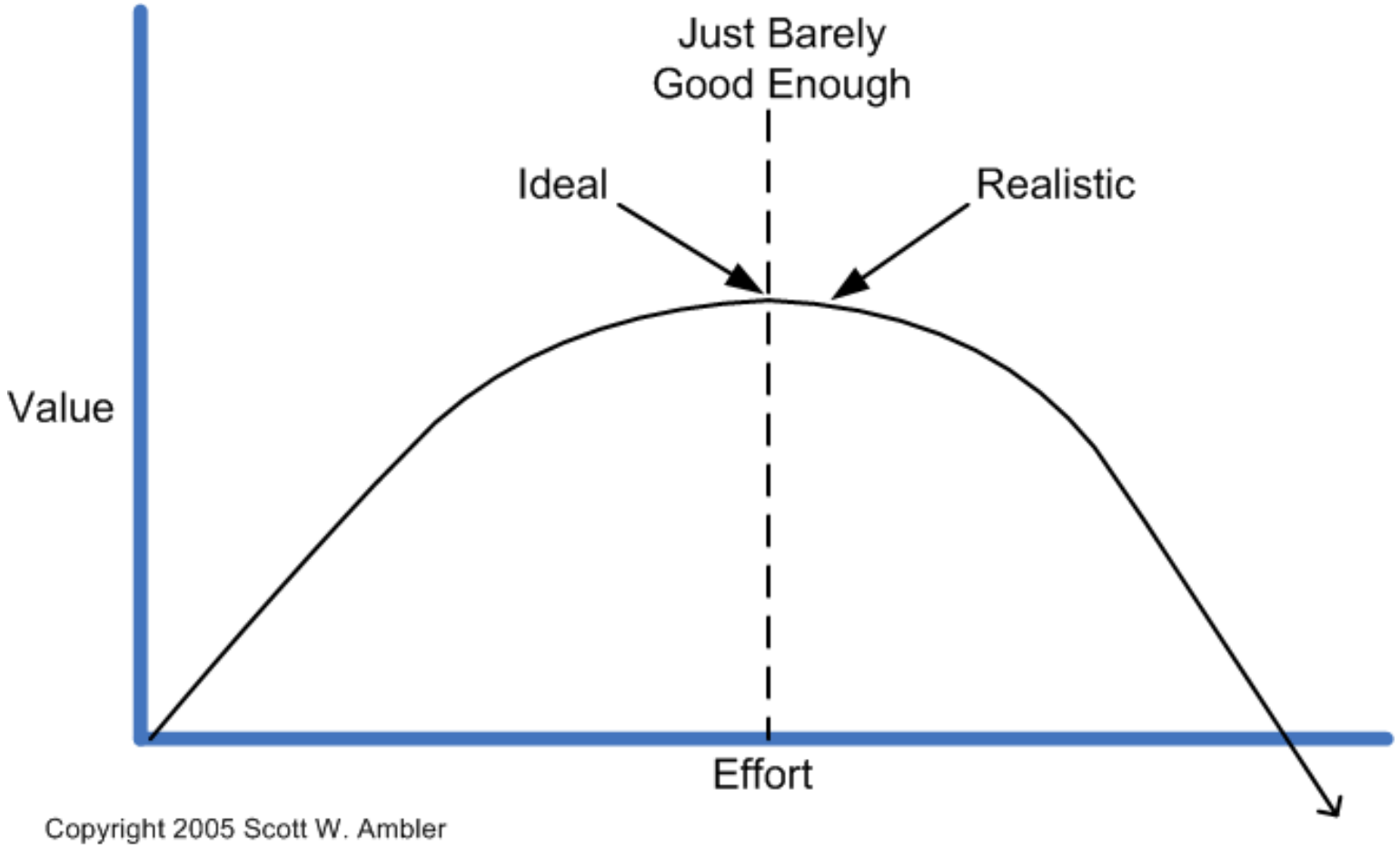
דוגמאות:

- Unix: ps aux | grep init
- מהדרים

Map-Reduce (Google)



Good enough כמה צריך?



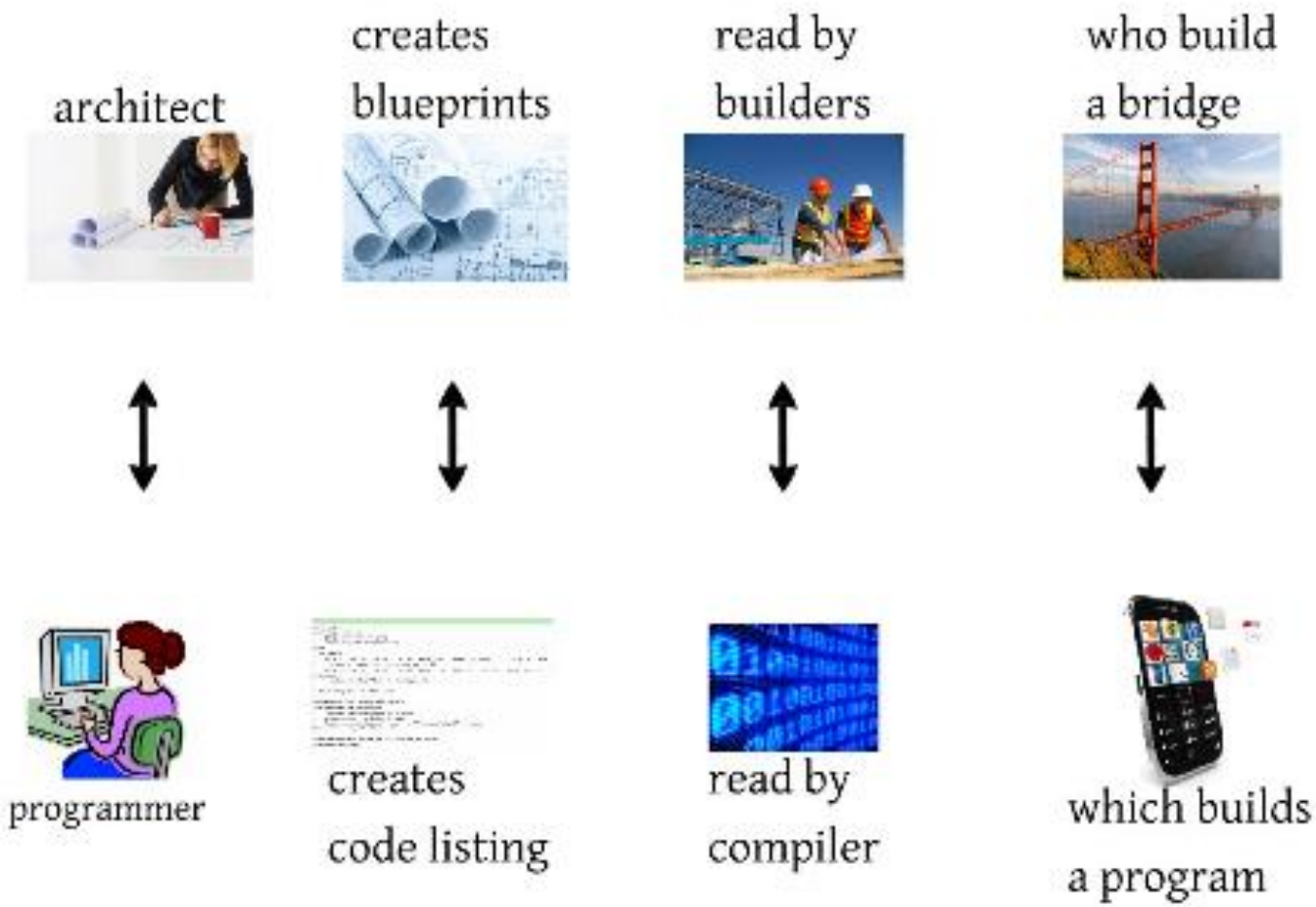
תיכון מעבר לארכיטקטורה

- תיכון מפורט
- תיכון שאינו ארכיטקטוני
- תיכון פונקציונלי
- מונחה עצמים מול מובנה
- כללי אצבע ועקרונות תיכון למשל:

Don’t Repeat Yourself, Tell Don’t Ask, Law of Demeter, Favor Composition Over Inheritance, **S**ingle Responsibility Principle (SOLID), …

- UML, CRC, ...

Reeves, The Code is the Design!



Beck (XP): Simple Design

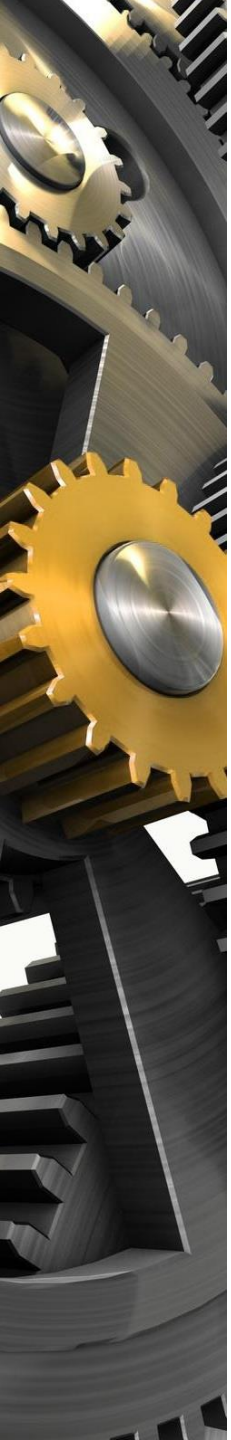
1. כל הבדיקות עוברות
2. ללא כפילויות (DRY)
3. ברור - מבטא את כוונת המתכנת
4. קטן - מינימום של מחלקות ומתודות

- אפשר פחות?
  The Four Elements of Simple Design
- Fowler, Is Design Dead?

זיהיתי קוד כפול אך הוצאתו למחלקה נפרדת עלולה להפוך אותו לפחות קריא - מה לעשות?

1. תמיד נעדיף למנוע כפילויות
2. קריאות הקוד חשובה יותר
3. קריאות חשובה יותר בתנאי שלא נגדיל את מספר המחלקות
4. קודם נכתוב בדיקה למחלקה החדשה ואם היא תעבור נוציא את הקוד

Adaptable Design



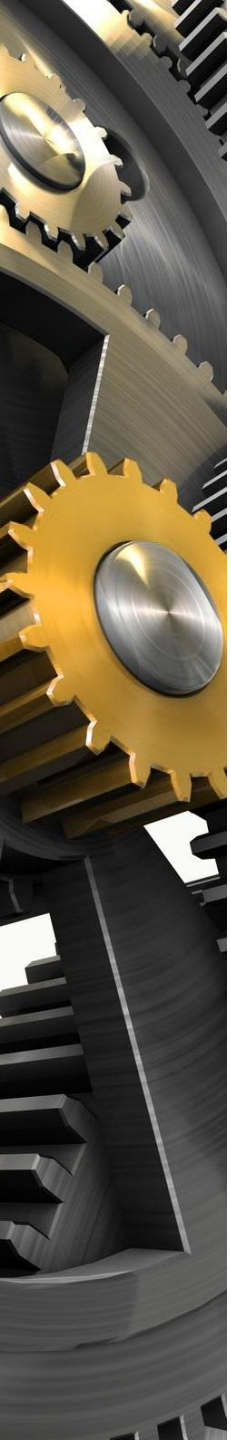
ADUF
Adaptable Design Up Front

Hayim Makabee

http://EffectiveSoftwareDesign.com

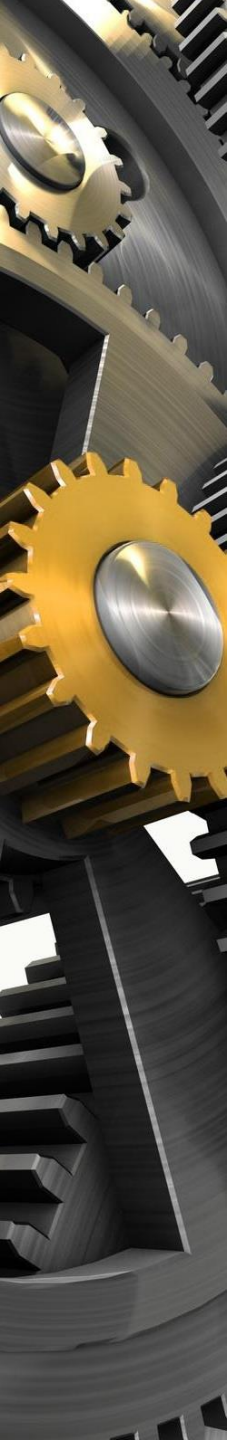
Context

- Iterative Software Development
- Agile, Incremental



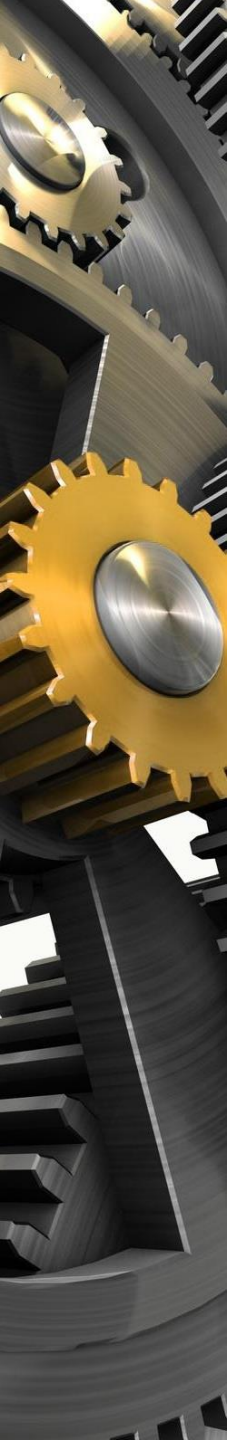
Question

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



Big Design Up Front

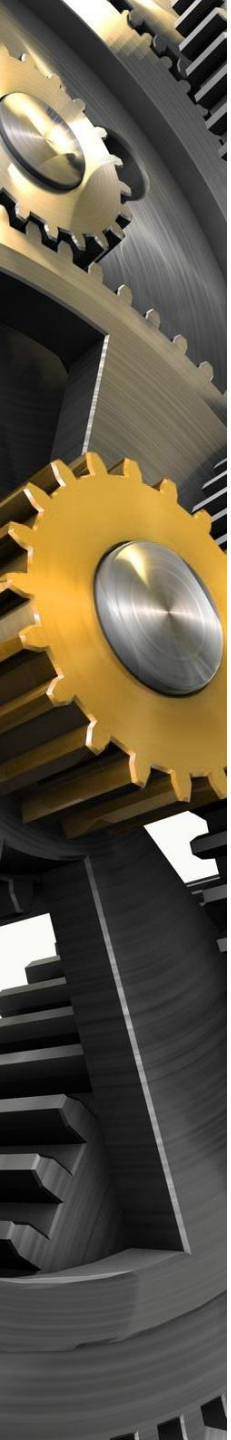
- BDUF = Do detailed design before starting to code.
- Problems:
  - Requirements are incomplete.
  - Requirements may change.
  - *"The Scrum product backlog is allowed to grow and change as more is learned about the product and its customers."*



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

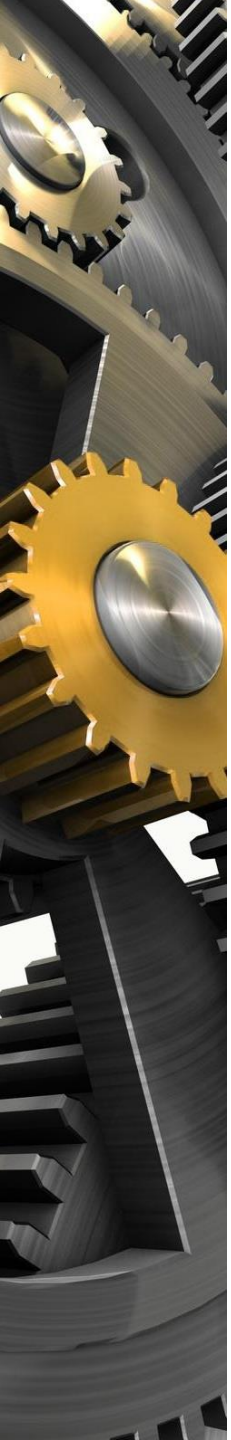
- *"**Just enough** sits somewhere in the chasm between big design up front's analysis paralysis and emergent design's refactor distractor."* – Simon Brown

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

Planned: Modiin

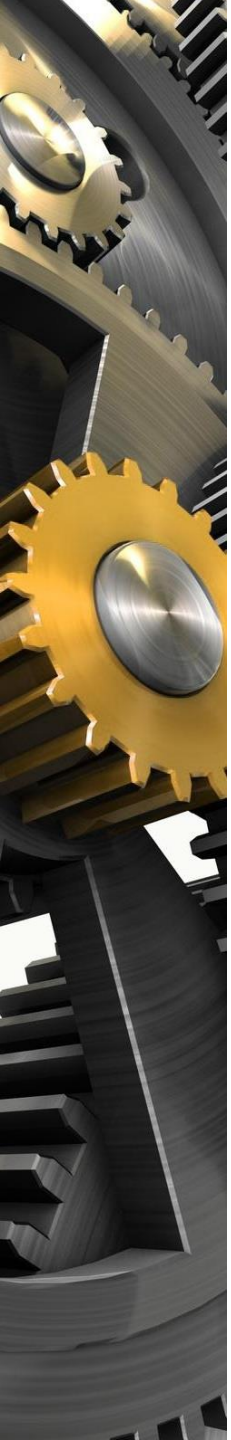
Emergent: Jerusalem Old City

Eroding Design



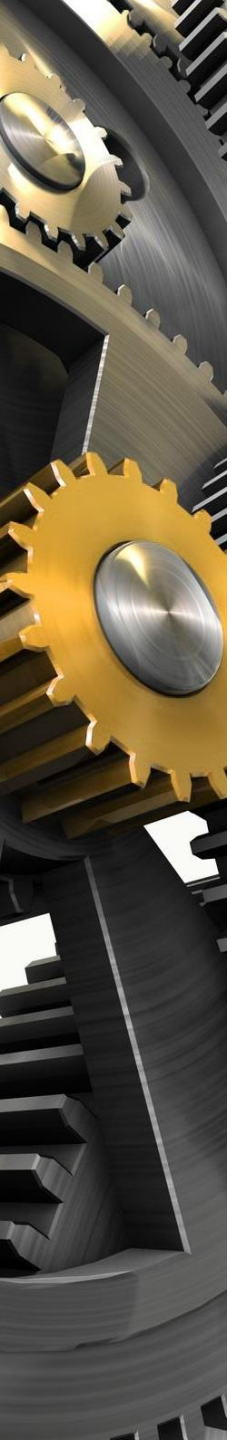
Eroding Design

- *"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"



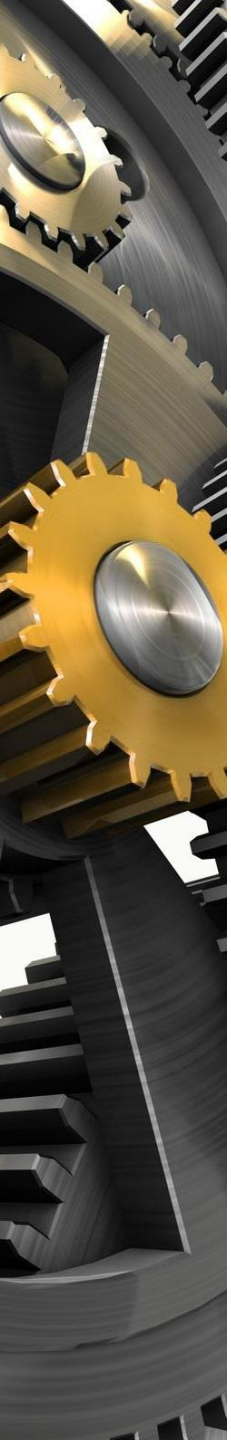
Software Entropy

- 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.



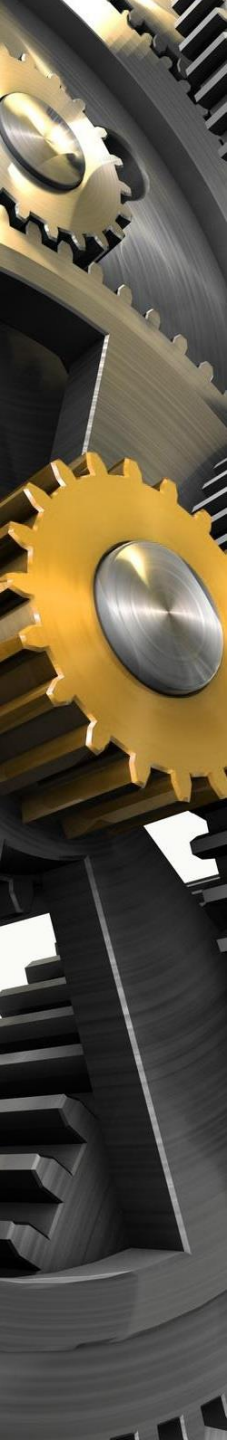
Lehman Laws

- 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.



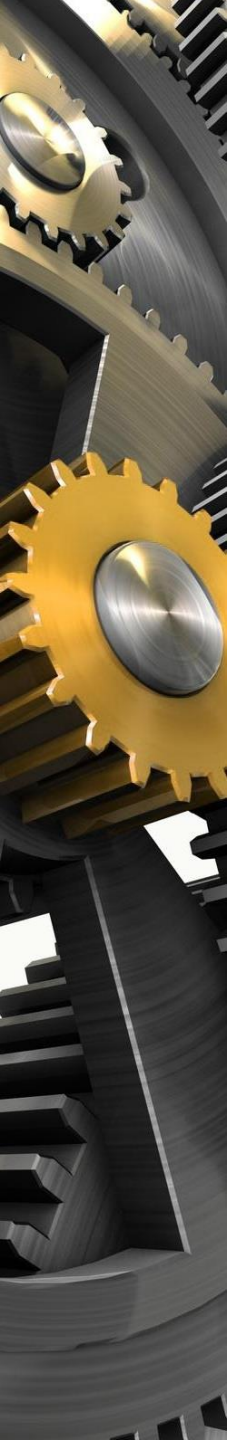
Conclusions so far…

- 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.



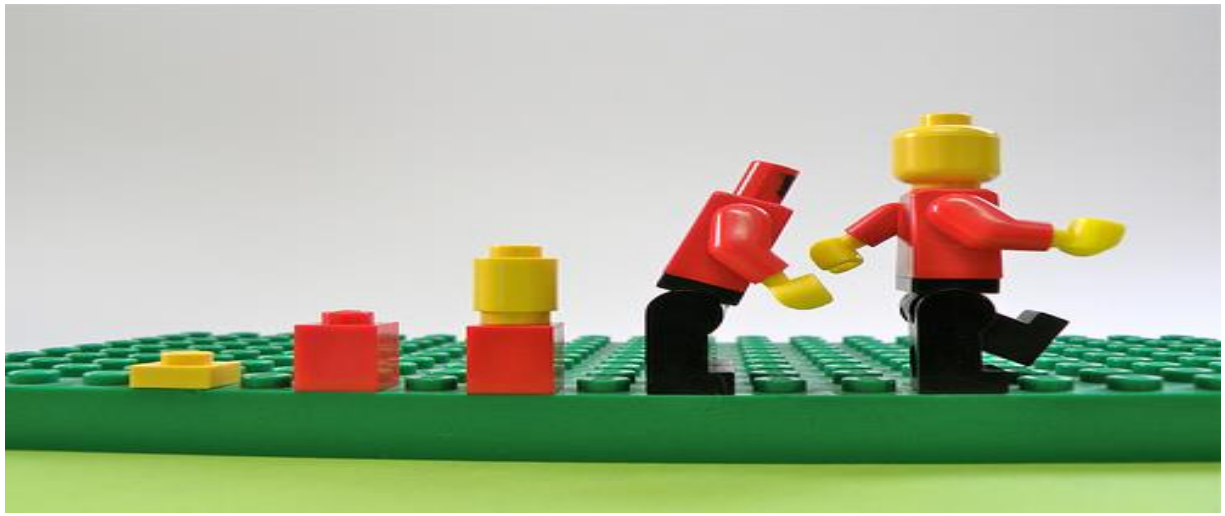
Adaptability

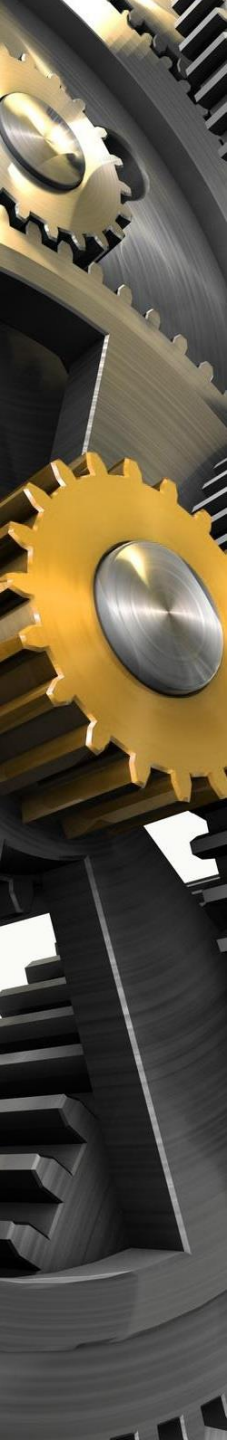
- *"A system that can **cope readily with a wide range of requirements**, will, all other things being equal, have an advantage over one that cannot. Such a system can **allow unexpected requirements to be met with little or no reengineering**, and allow its more skilled customers to rapidly address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



New development mindset

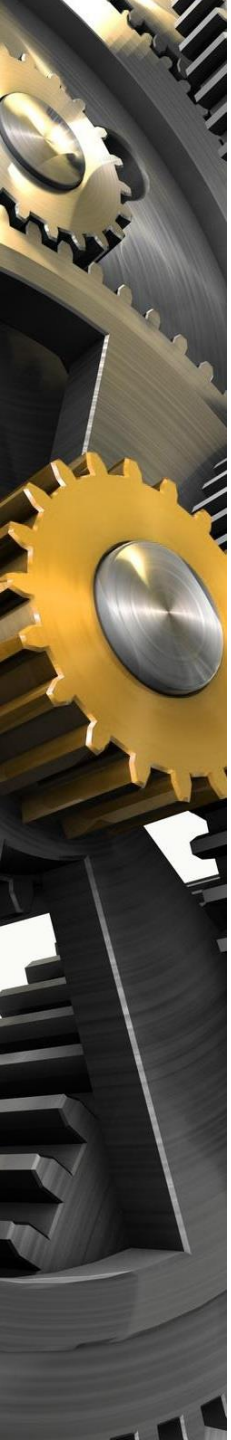
- Instead of planning for software **development**, plan for software **evolution**.
- Instead of designing a **single system**, design a **family of systems**.





Family of Systems

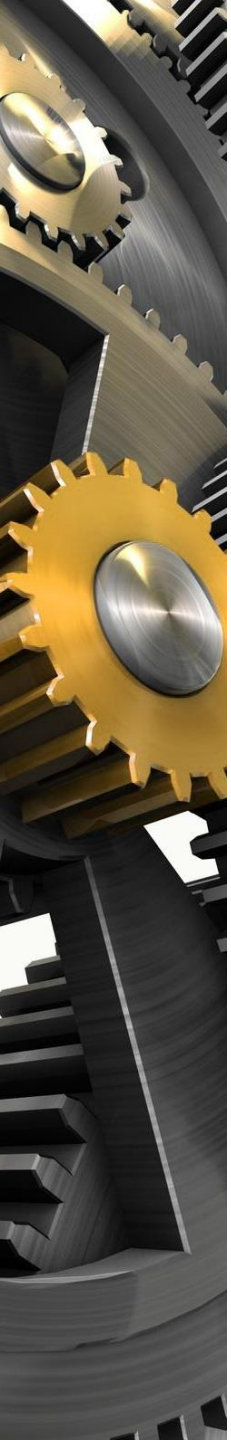
- Software Product Line: "*A set of software-intensive systems that* ***share a common, managed set of features*** *satisfying the specific needs of a particular market segment or mission and that are* ***developed from a common set of core assets*** *in a prescribed way.*" – Software Engineering Institute



Adaptable Design

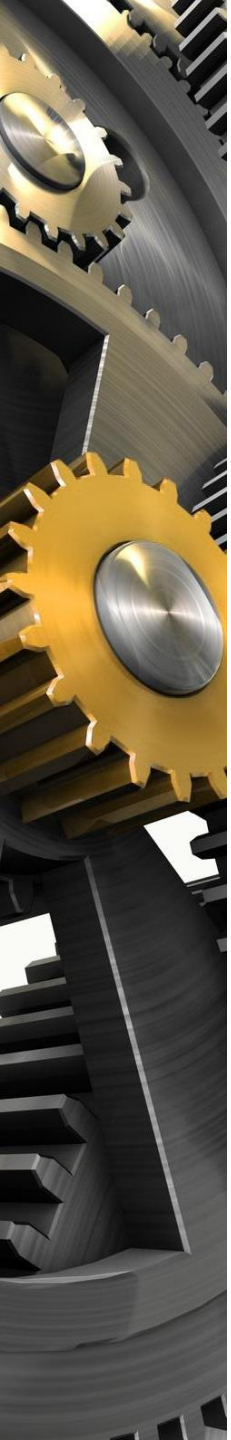
- Adaptable Software Design: A generic software design for a family of systems which does not need to be changed to accommodate new requirements.

- ADUF = Adaptable Design Up Front!

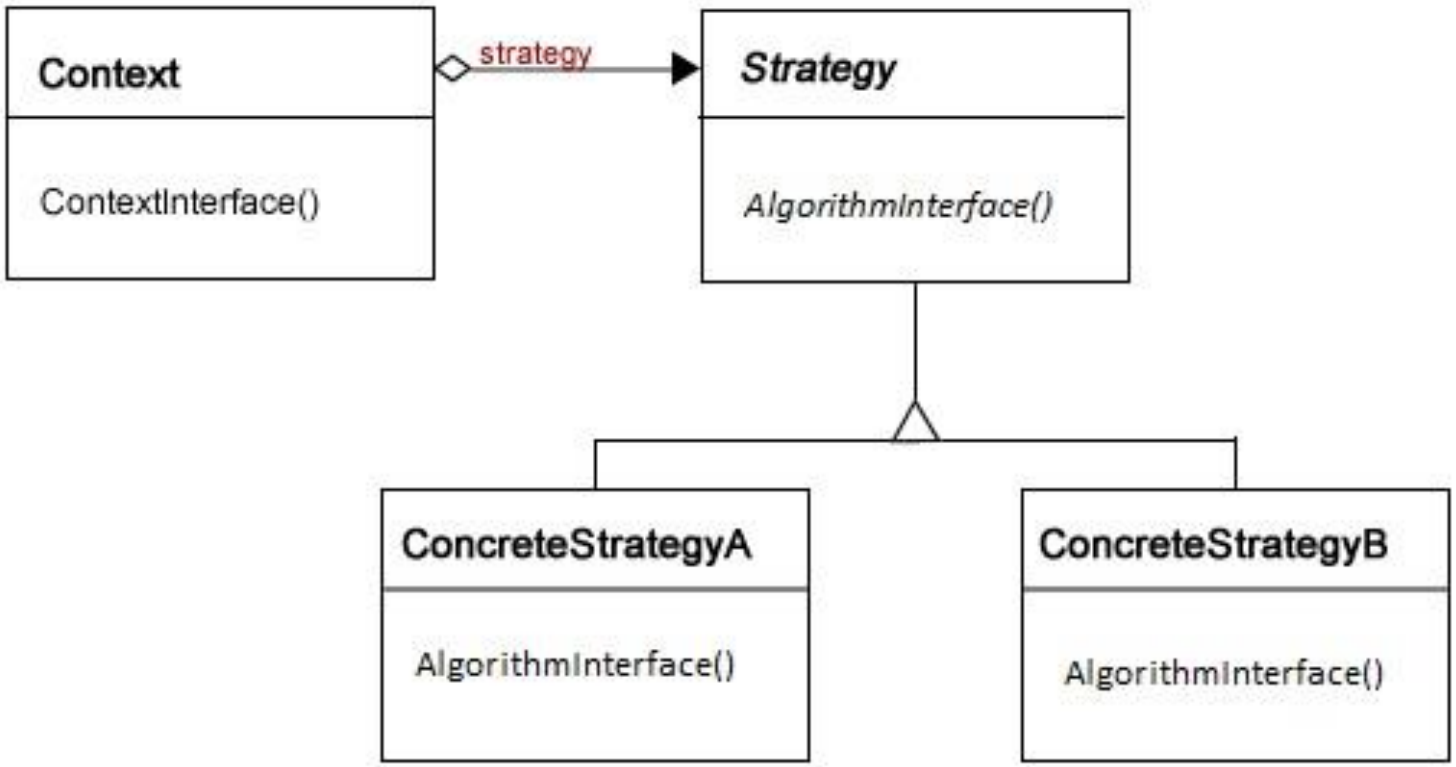


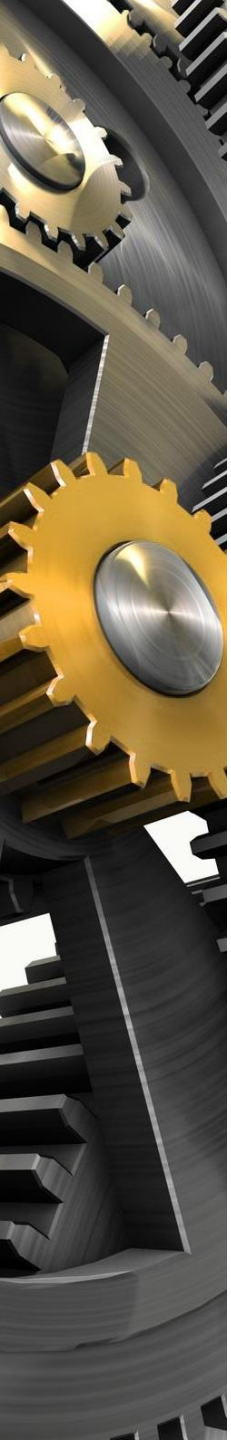
Open/Closed Principle

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



Strategy Design Pattern

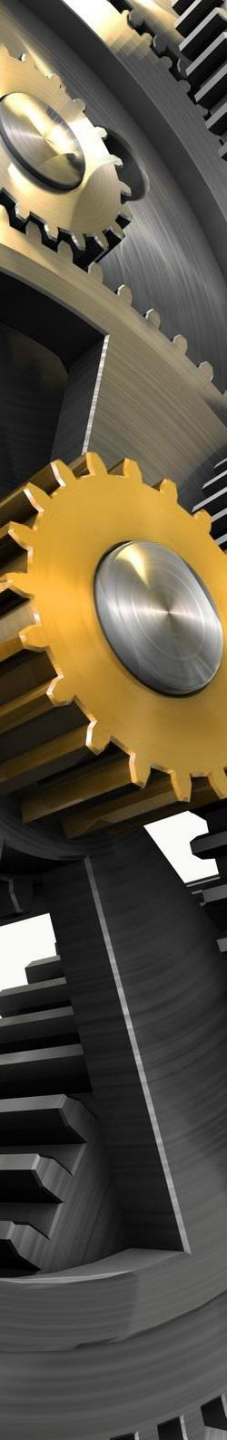


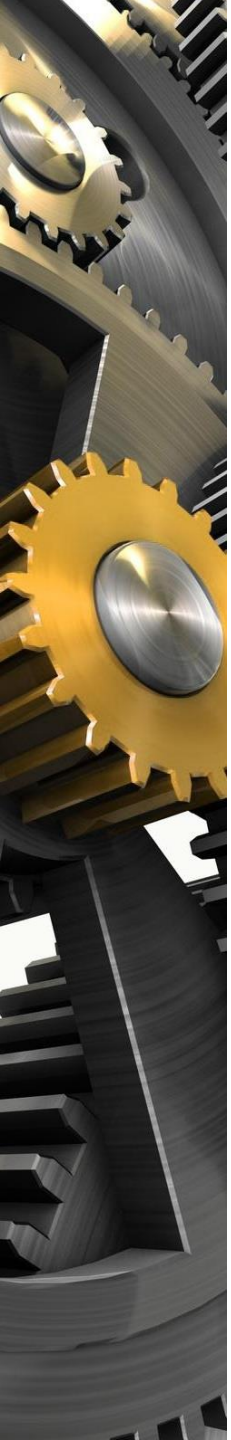


Adaptable Design Up Front and the Open/Closed Principle

- 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.

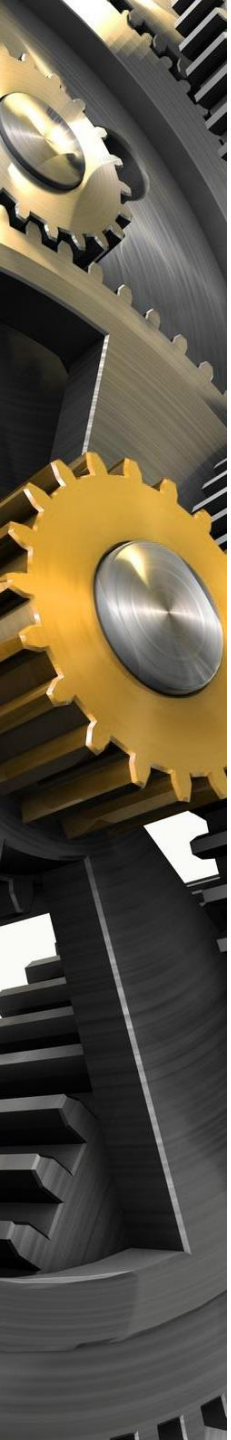
Architecture vs. Interior Design





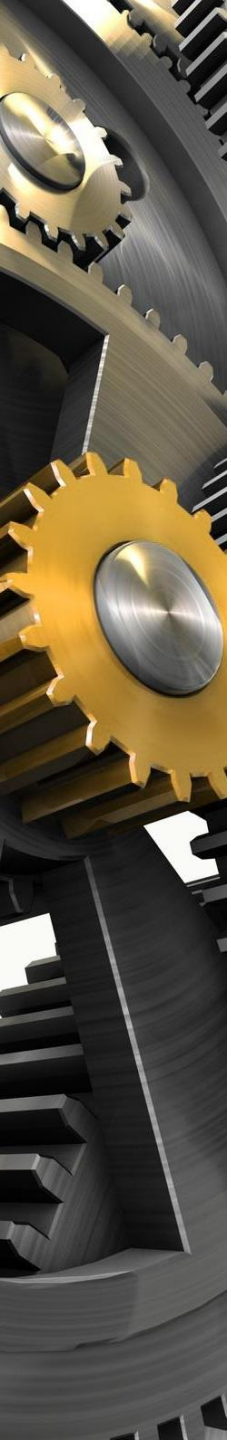
Common Errors

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



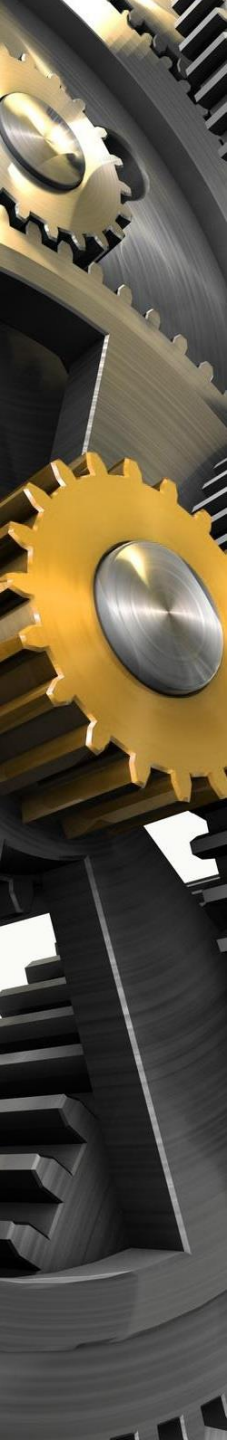
Too much design up front

- 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.



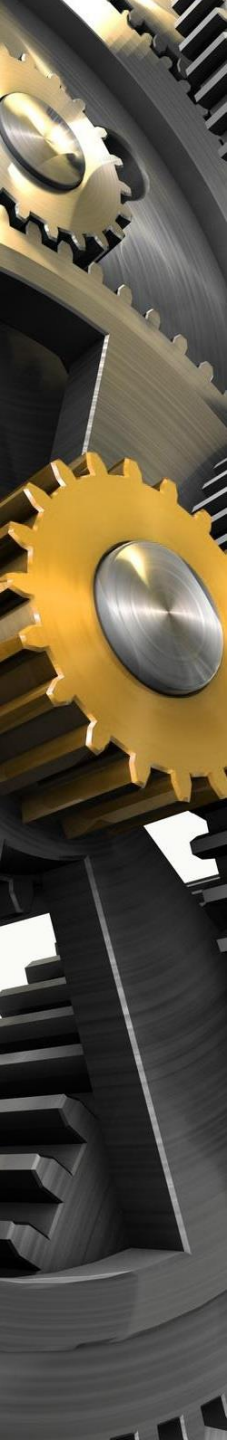
Not enough design up front

- 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.



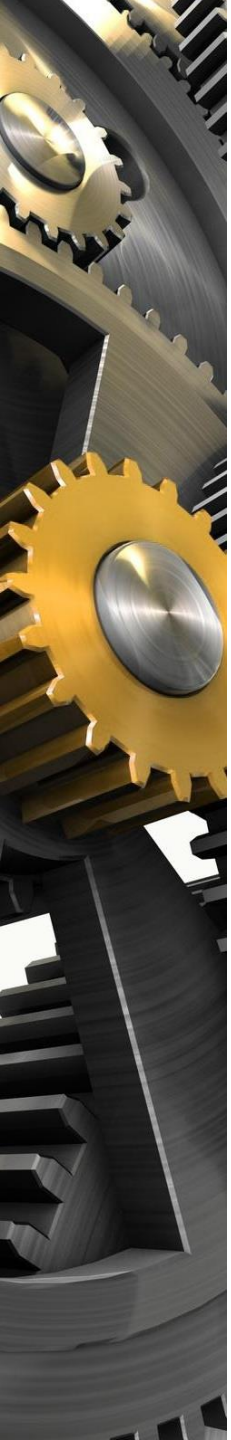
Too much openness

- 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

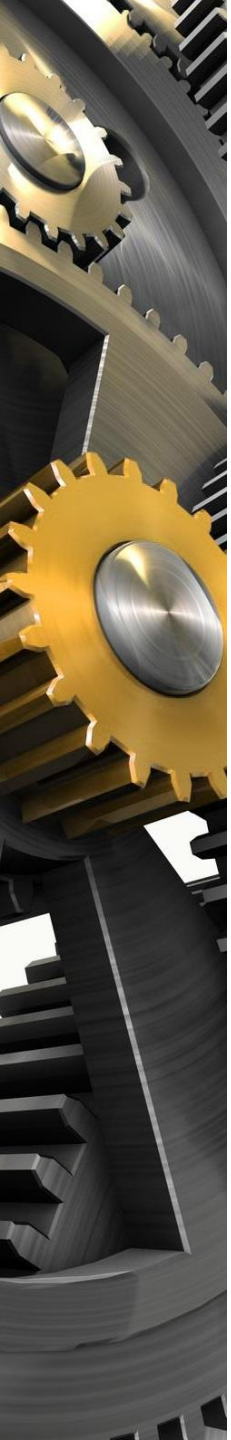
- The design captures all essential aspects, but there are not mechanisms that allow this design to be easily extended.
- Example:
  - If there are Interfaces for the main entities, there should also be Factories to instantiate concrete subclasses.



Extensible Designs

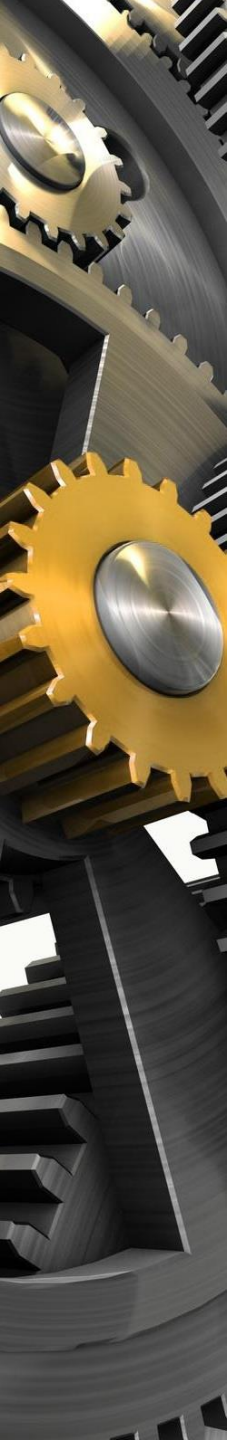
Main alternatives for extensible designs:

- Frameworks
  - Plug-ins
- Platforms



Frameworks

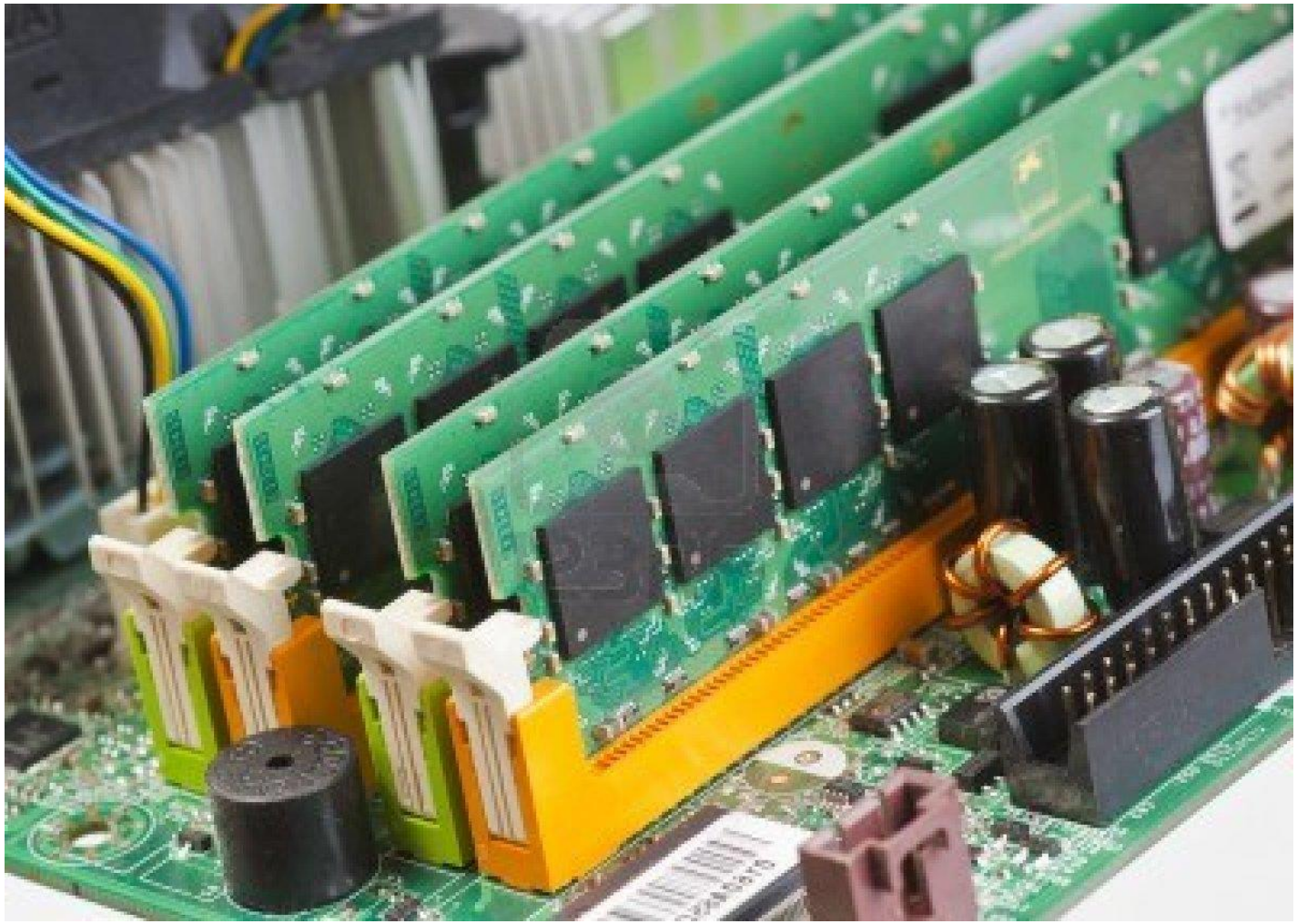
- Framework: "*A software framework is an abstraction in which software providing generic functionality can be selectively changed by additional user-written code, thus providing application-specific software.*"

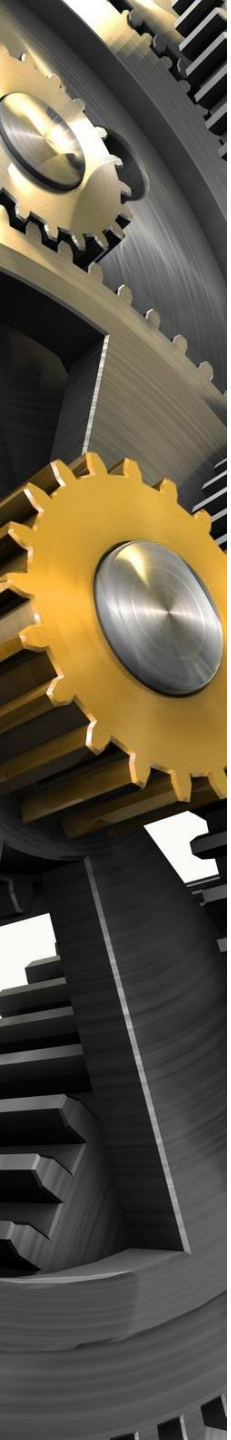


Plug-ins

- 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."*

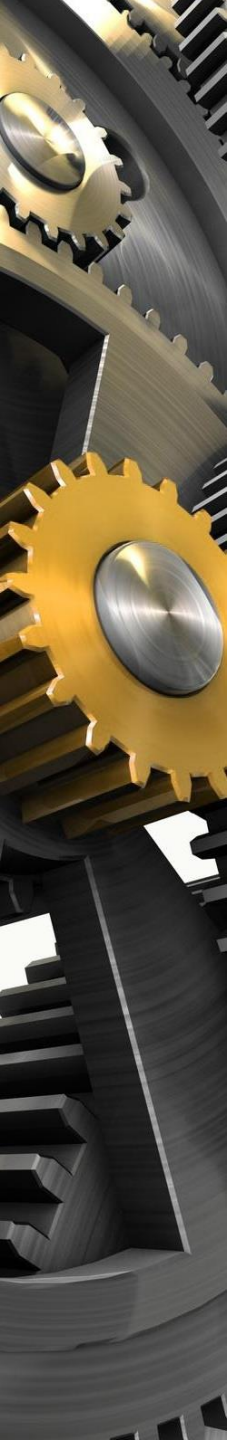
Pluggable Components





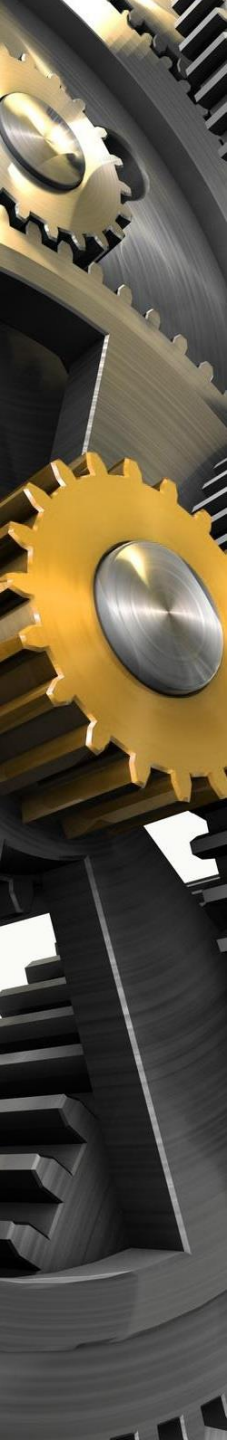
Versioning

- 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.



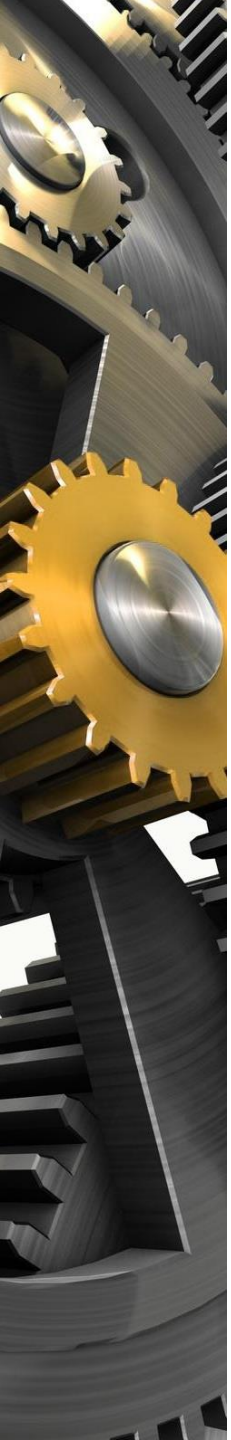
Platforms

- 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.



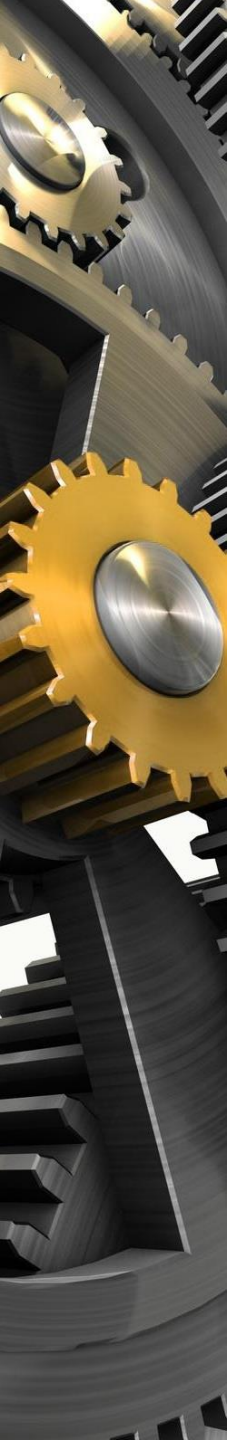
Question 1

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



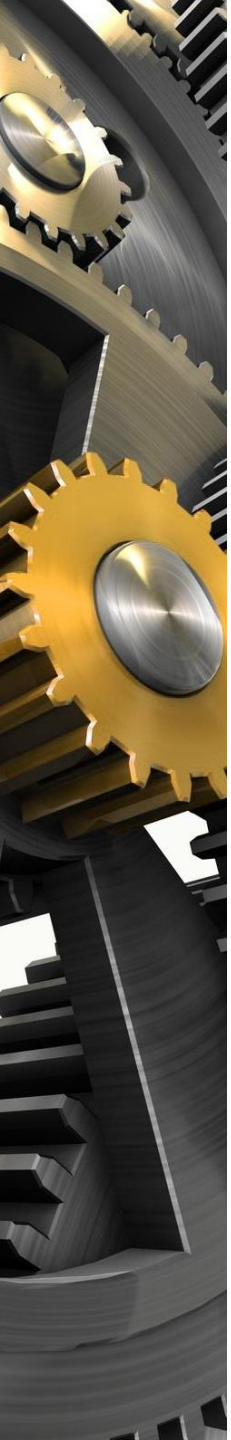
Domain Modeling

- *"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?

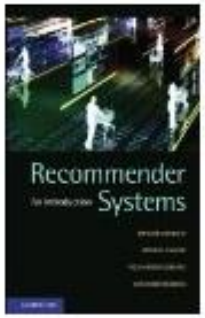


Design Patterns

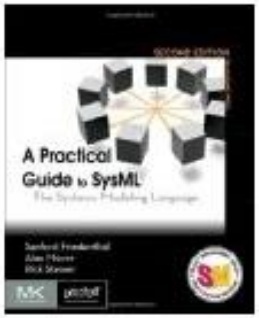
- Reduce coupling using Design Patterns.
- Most patterns avoid direct links between concrete classes, using interfaces and abstract classes.
- Example: Use Observer when several types of objects must react to the same event.

Example: Recommender System

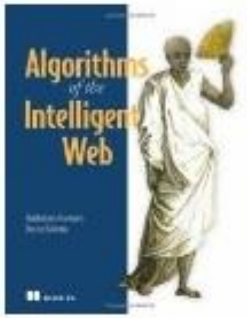
Recommendations for You in Books



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

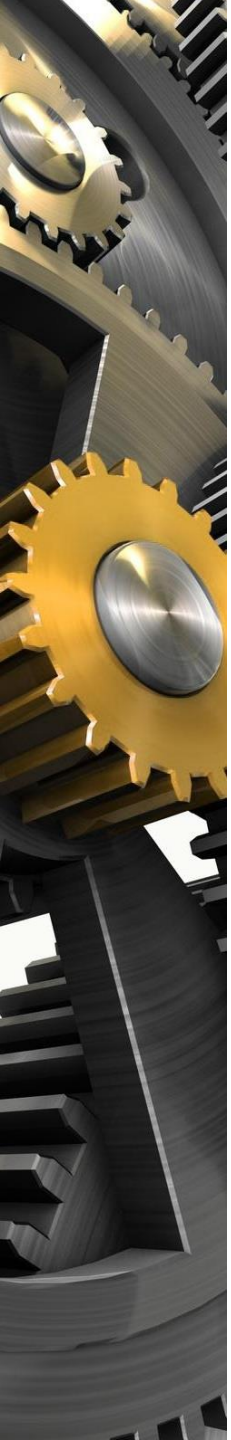


A Practical Guide to SysML, Second...
Sanford Friedenthal, Alan Moore, Rick...
Paperback
(3)
~~$59.95~~ $51.99
Why recommended?

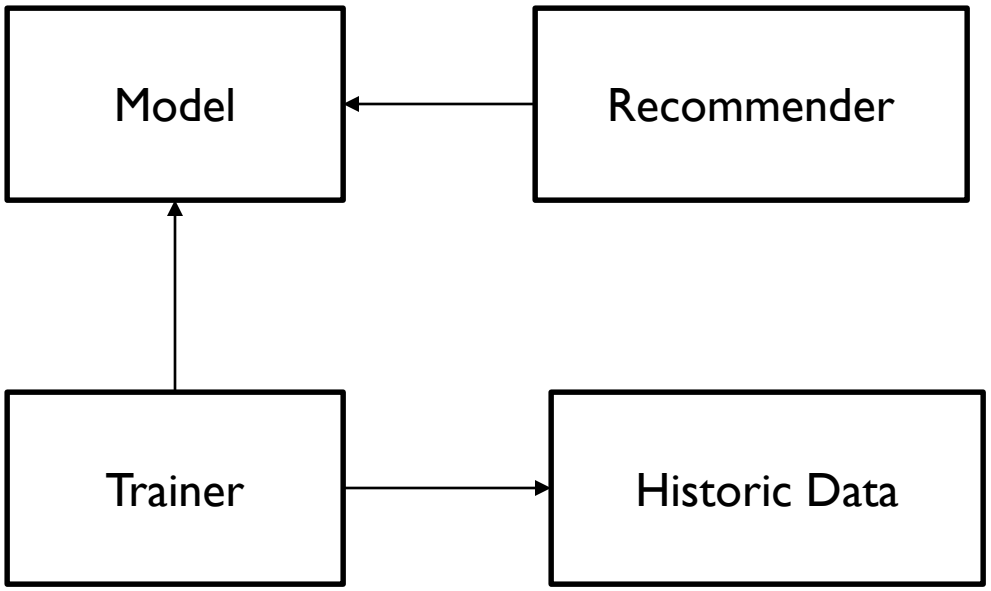


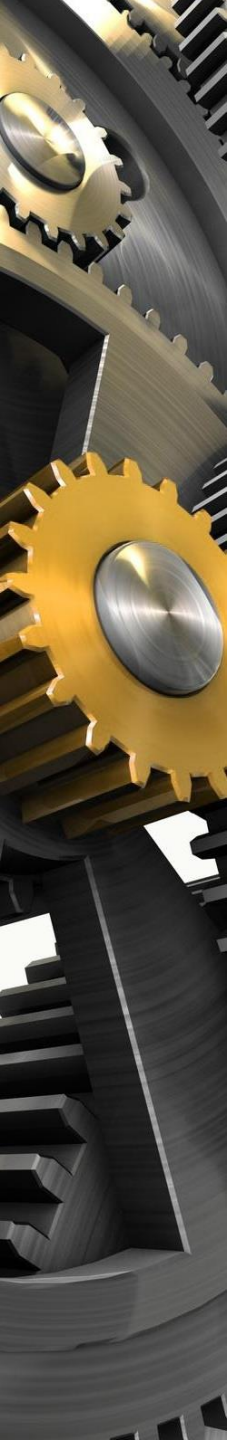
Algorithms of the Intelligent Web
Haralambos Marmanis, Dmitry Babenko
Paperback
(14)
~~$44.99~~ $26.76
Why recommended?

› See more recommendations



Recommender System Model

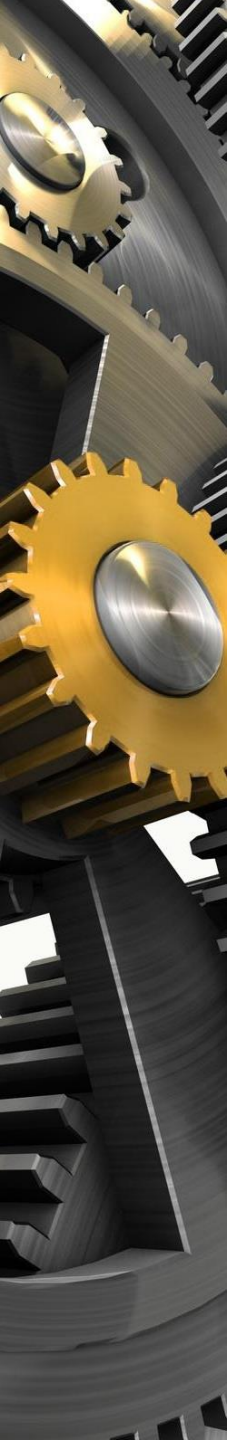




Recommender System Framework

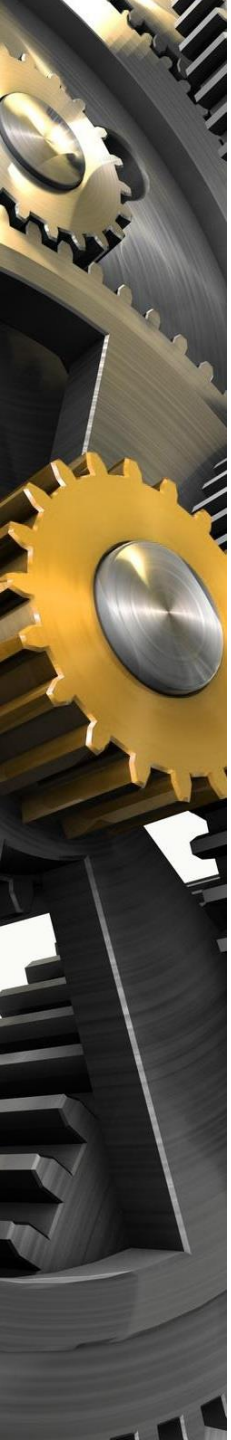
After 1 year of development:

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



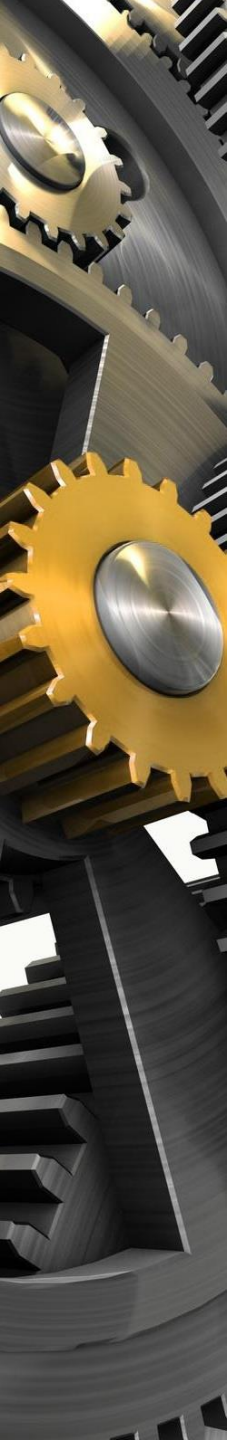
What about YAGNI?

- 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



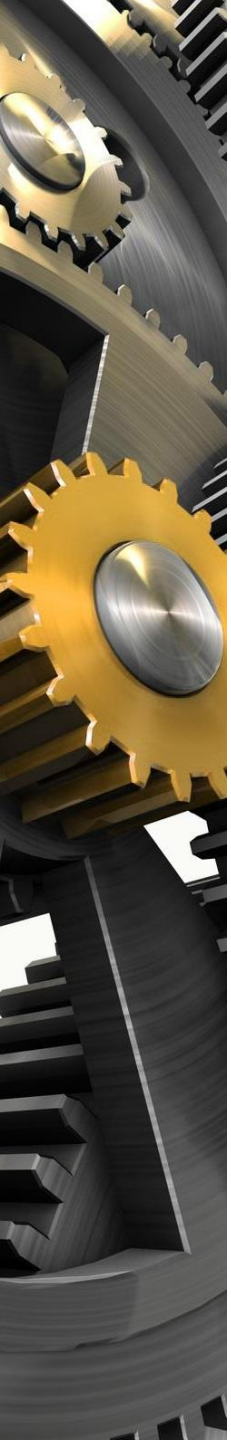
YAGNI vs. ADUF

- 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".



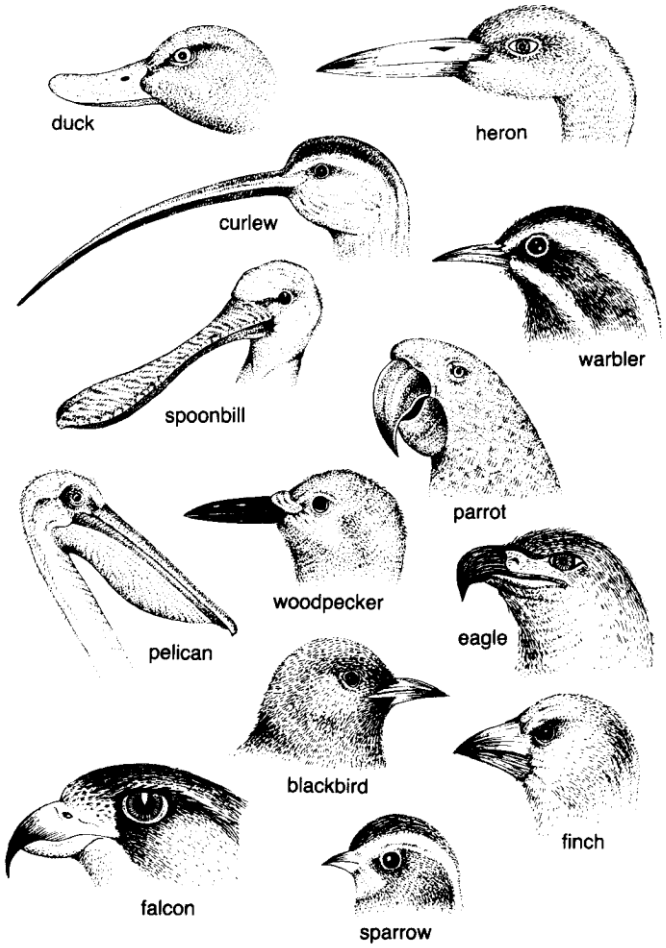
Related Work

- 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

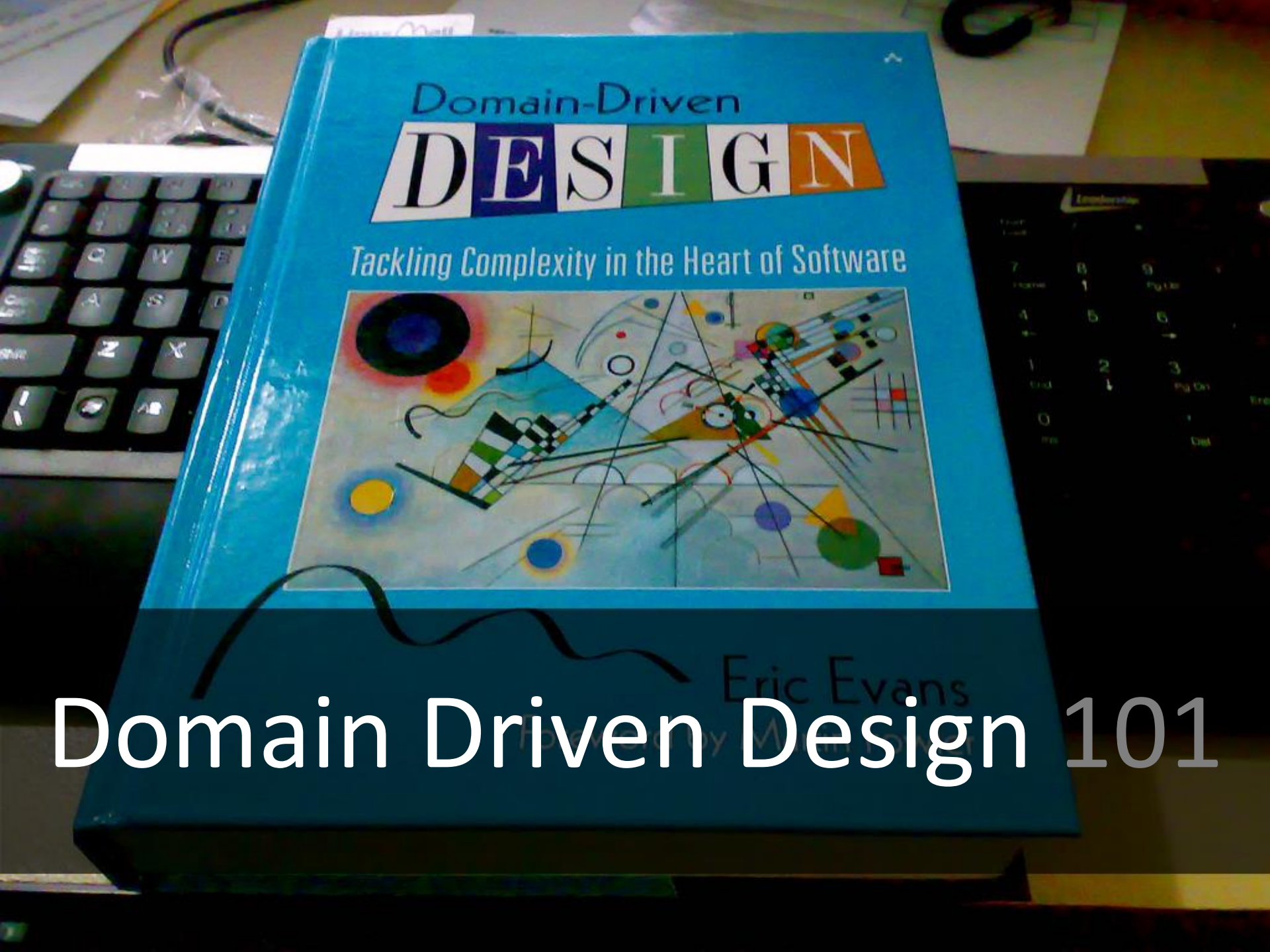
Adaptability & Evolution

"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



Domain Driven Design

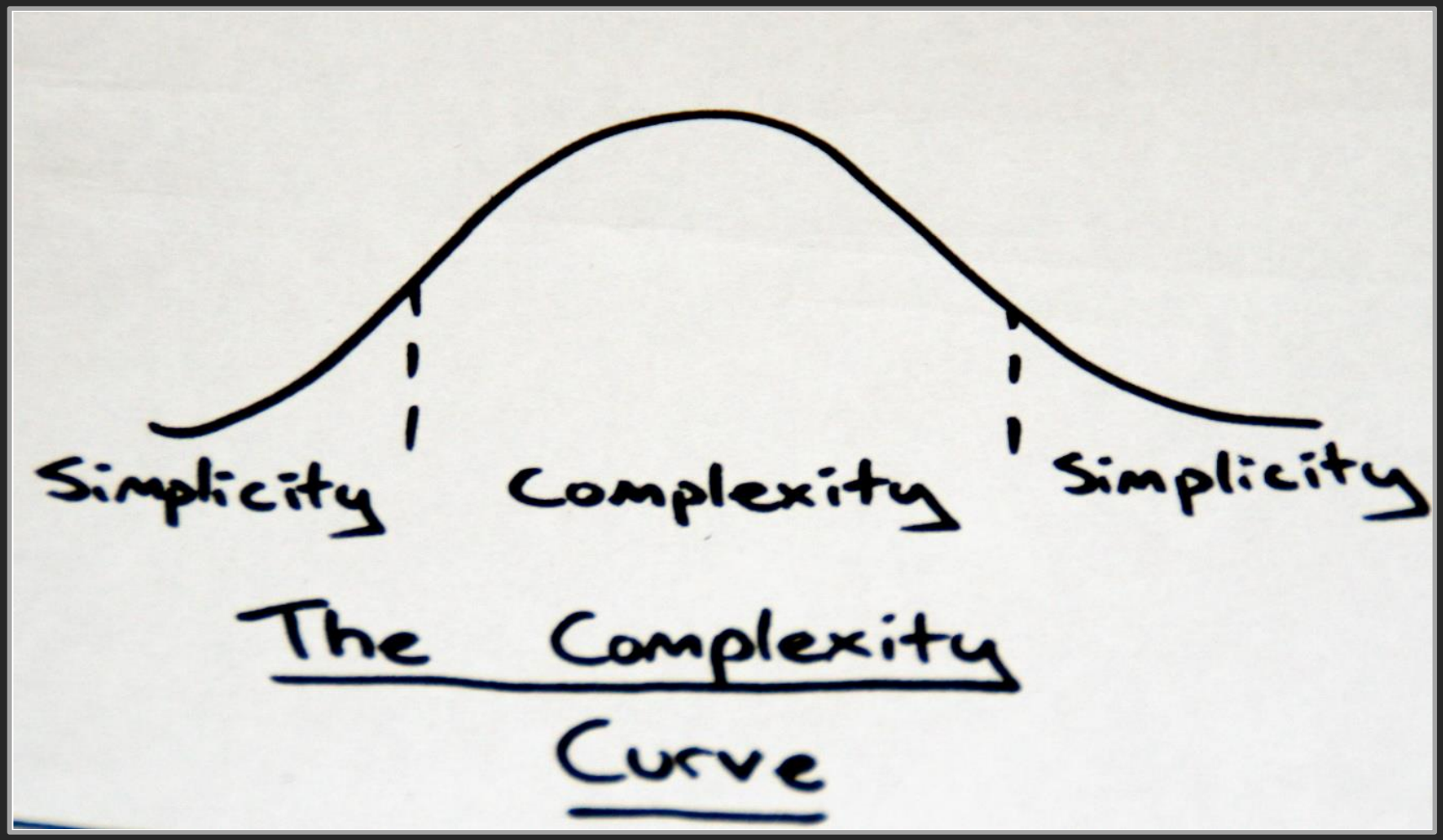


Domain Driven Design 101

Agenda

- Why
- Building blocks
  - Repositories, entities, specifications etc
- Putting it to practice
  - Dependency injection
  - Persistence
  - Validation
  - Architecture
- Challenges
- When not to use DDD
- Resources

Software is complicated

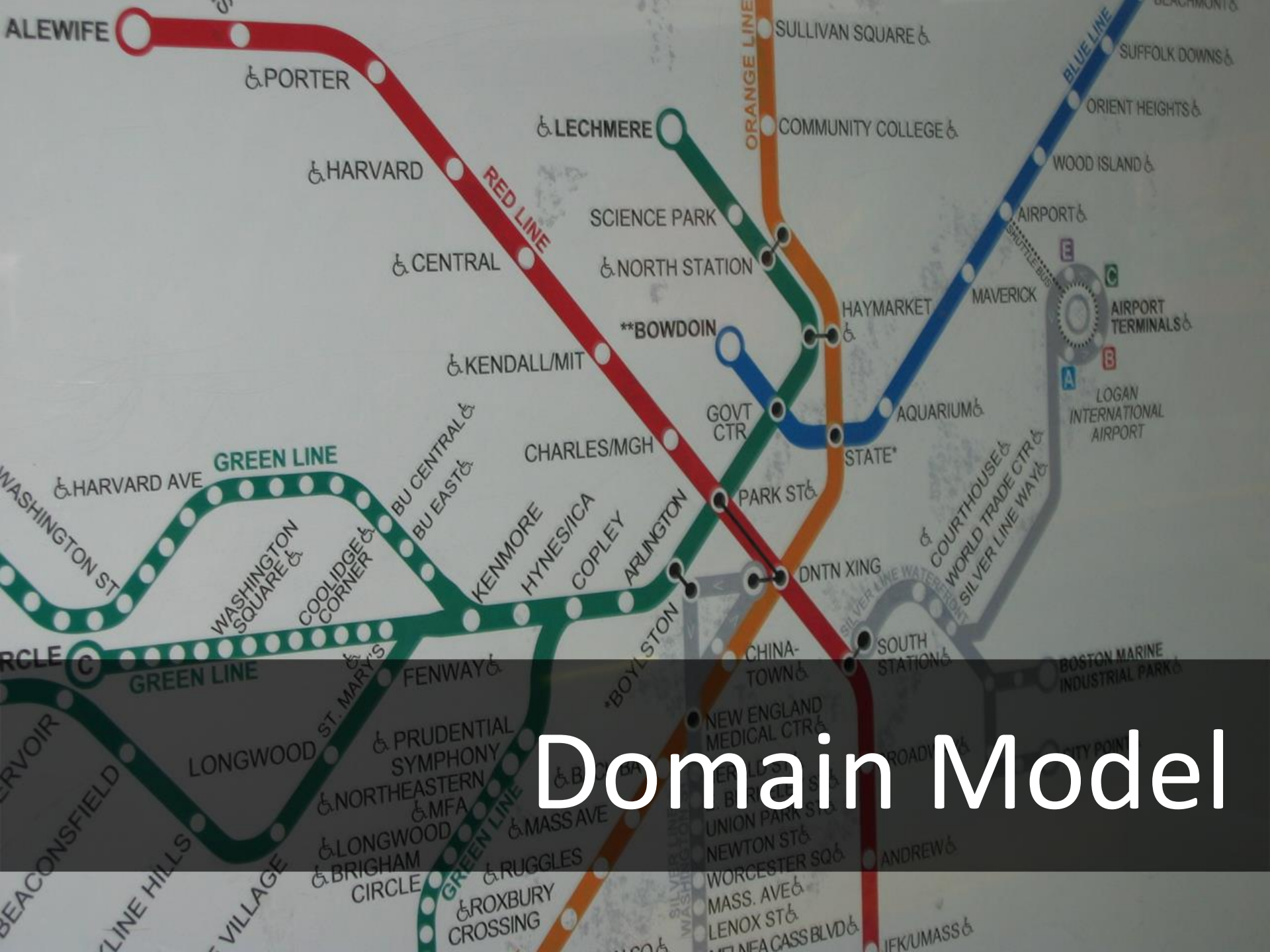


We solve complexity in software by distilling our problems

```
public bool CanBook(Cargo cargo, Voyage voyage)
{
    double maxBooking = voyage.Capacity * 1.1;
    if (voyage.BookedCargoSize + cargo.Size > maxBooking)
        return false;

    ...
}
```

```
public bool CanBook(Cargo cargo, Voyage voyage)
{
    if (!overbookingPolicy.IsAllowed(cargo, voyage))
        return false;

    ...
}
```

DDD is about making concepts explicit



Domain Model

Ubiquitous language

```
public interface ISapService
{
    double GetHourlyRate(int sapId);
}
```

A poor abstraction

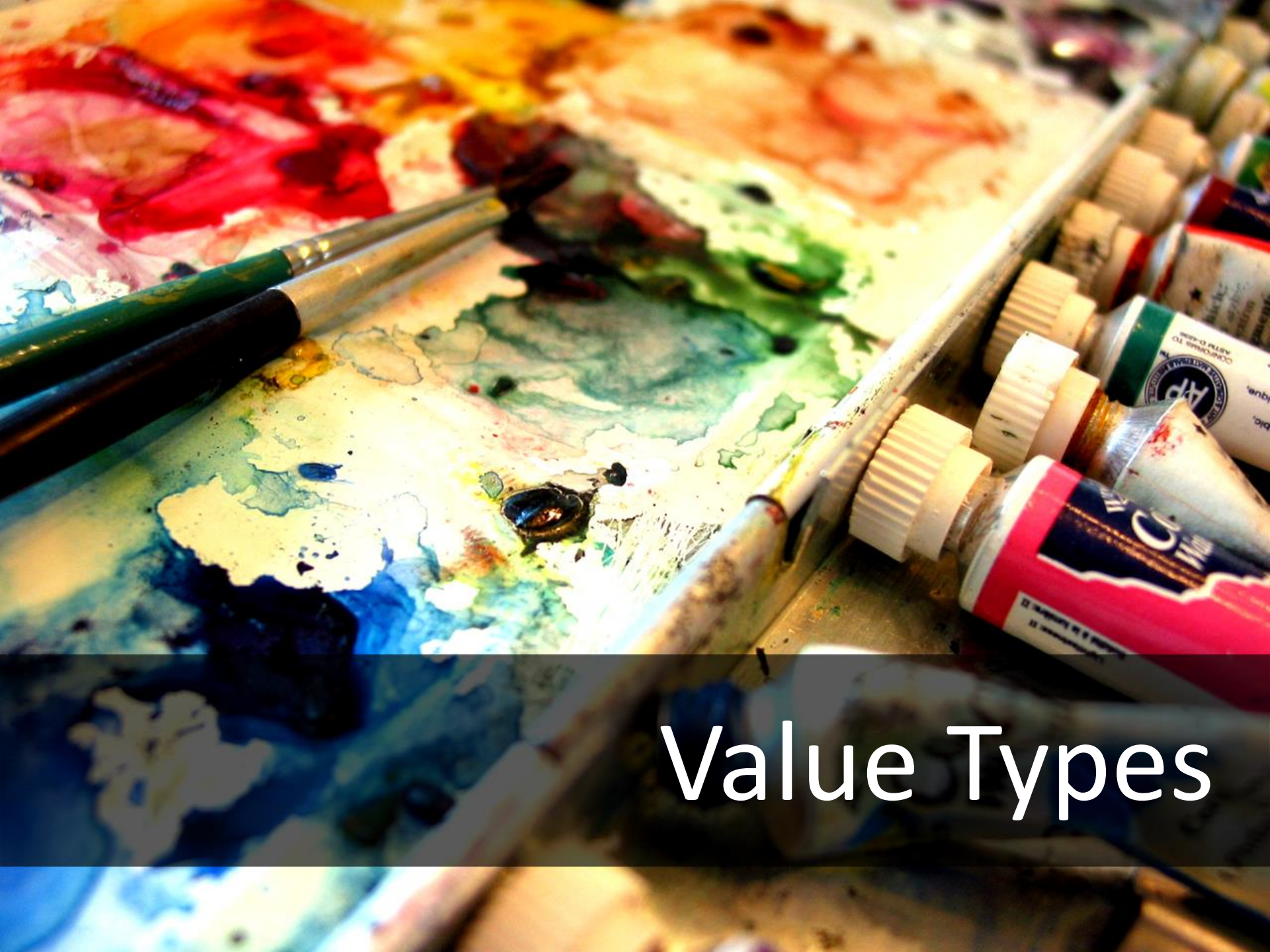
```
public interface IPayrollService
{
    double GetHourlyRate(Employee employee);
}
```

Intention-revealing interfaces

```
public class Employee
{
    void ApplyForLeave(DateTime start,
                       DateTime end,
                       ILeaveService leaves)
    {
        ...
    }
}
```


Domain Expert

Entities



Value Types

```
public class Employee : IEquatable<Employee>
{
    public bool Equals(Employee other)
    {
        return this.Id.Equals(other.Id);
    }
}
```

*Entities are the same if they have the same **identity***

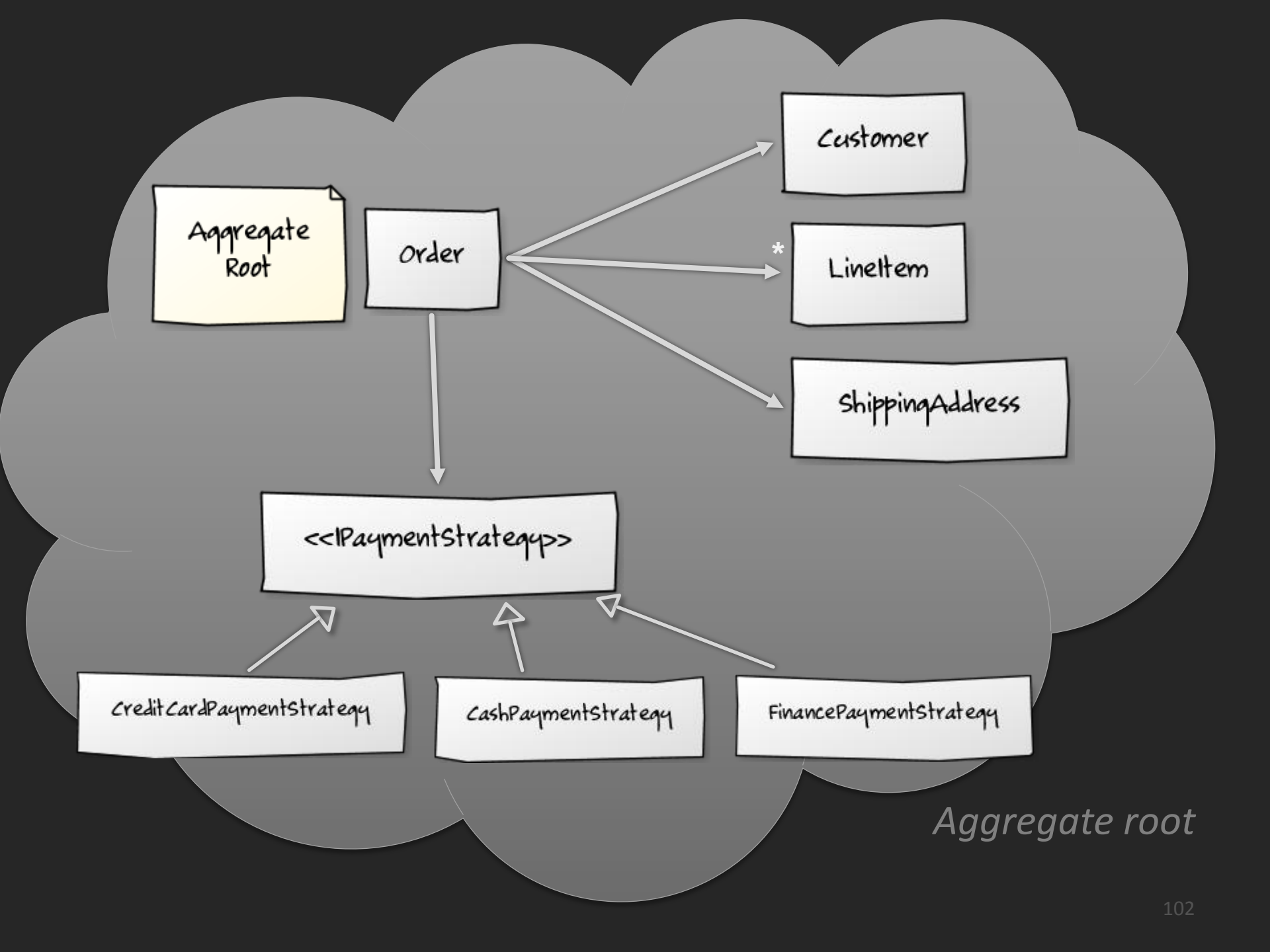
```
public class PostalAddress : IEquatable<PostalAddress>
{
    public bool Equals(PostalAddress other)
    {
        return this.Number.Equals(other.Number)
            && this.Street.Equals(other.Street)
            && this.PostCode.Equals(other.PostCode)
            && this.Country.Equals(other.Country);
    }
}
```

*Value Types are the same if they have the same **value***

```
public class Colour
{
    public int Red { get; private set; }
    public int Green { get; private set; }
    public int Blue { get; private set; }

    public Colour(int red, int green, int blue)
    {
        this.Red = red;
        this.Green = green;
        this.Blue = blue;
    }

    public Colour MixInTo(Colour other)
    {
        return new Colour(
            Math.Avg(this.Red, other.Red),
            Math.Avg(this.Green, other.Green),
            Math.Avg(this.Blue, other.Blue));
    }
}
```

*Value Types are **immutable***

Aggregates



Aggregate root

Repositories

```
public interface IEmployeeRepository
{
    Employee GetById(int id);
    void Add(Employee employee);
    void Remove(Employee employee);

    IEnumerable<Employee> GetStaffWorkingInRegion(Region region);
}
```

Repositories provide collection semantics and domain queries

Domain Services

```
public interface ITripService
{
    float GetDrivingDistanceBetween(Location a, Location b);
}
```

Specifications

```
class GoldCustomerSpecification : ISpecification<Customer>
{
    public bool IsSatisfiedBy(Customer candidate)
    {
        return candidate.TotalPurchases > 1000.0m;
    }
}
```

```
if (new GoldCustomerSpecification().IsSatisfiedBy(employee))
    // apply special discount
```

Specifications encapsulate a single rule

Specifications can be used…

to construct objects

```
var spec = new PizzaSpecification()
               .BasedOn(new MargaritaPizzaSpecification())
               .WithThickCrust()
               .WithSwirl(Sauces.Bbq)
               .WithExtraCheese();

var pizza = new PizzaFactory().CreatePizzaFrom(spec);
```

Constructing objects according to a specification

Specifications can be used…

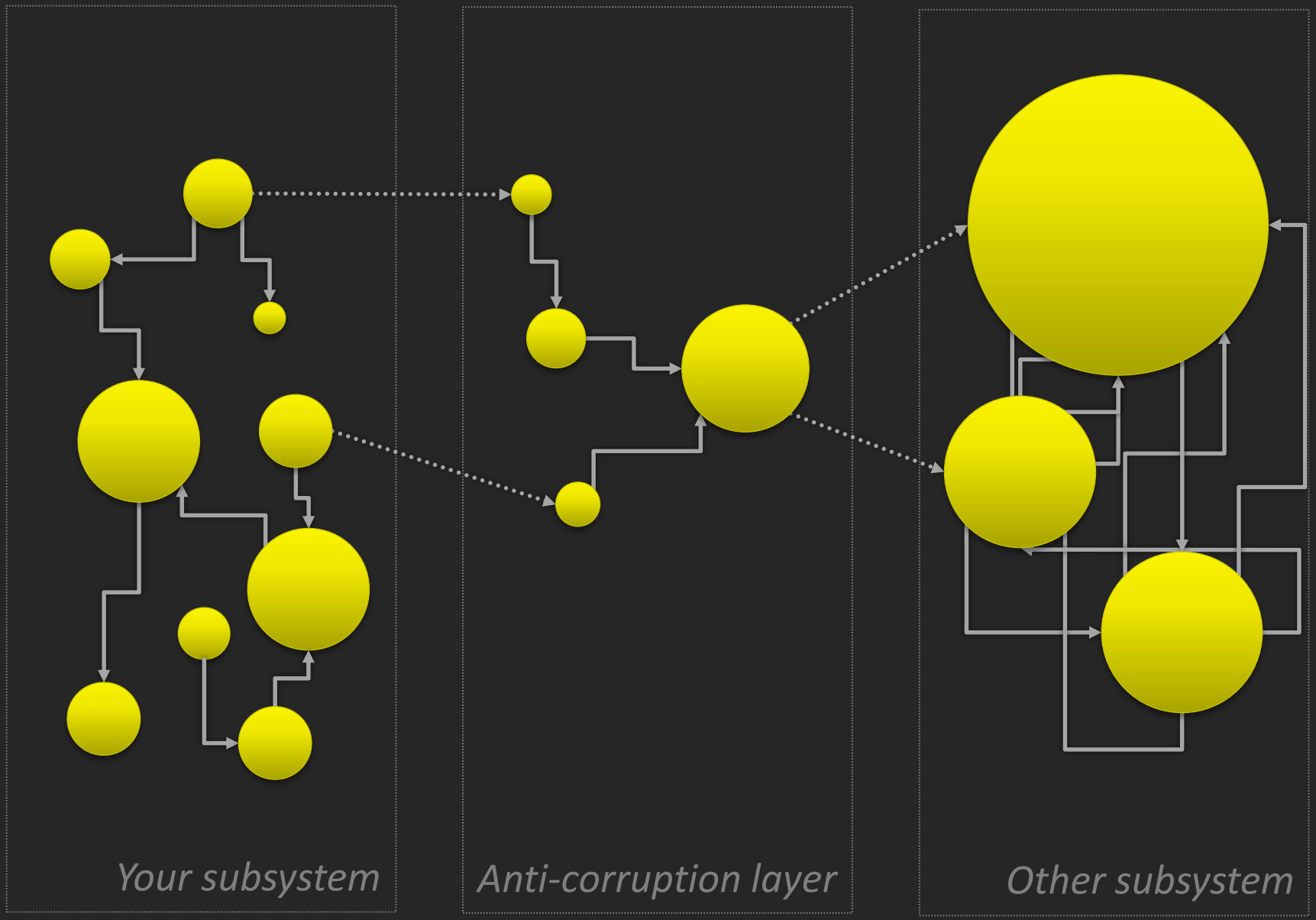
for querying

```
public interface ICustomerRepository
{
    IEnumerable<Customer> GetCustomersSatisfying(
        ISpecification<Customer> spec);
}
```

```
var goldCustomerSpec = new GoldCustomerSpecification();

var customers = this.customerRepository
    .GetCustomersSatisfying(goldCustomerSpec);
```

Querying for objects that match some specification

Anticorruption Layer


Your subsystem
Anti-corruption layer
Other subsystem

“Any 3rd party system that I have to integrate with was written by a drunken monkey typing with his feet.”

Oren Eini aka Ayende

Bounded Context

```
public class Lead
{
    public IEnumerable<Opportunity> Opportunities { get; }
    public Person Contact { get; }
}
```

```
public class Client
{
    public IEnumerable<Invoice> GetOutstandingInvoices();
    public Address BillingAddress { get; }
    public IEnumerable<Order> PurchaseHistory { get; }
}
```

```
public class Customer
{
    public IEnumerable<Ticket> Tickets { get; }
}
```

Dependency Injection

```
public interface INotificationService
{
    void Notify(Employee employee, string message);
}
```

An interface defines the model

```
public class EmailNotificationService : INotificationService
{
    void Notify(Employee employee, string message)
    {
        var message = new MailMessage(employee.Email, message);
        this.smtpClient.Send(message);
    }
}
```

Far away, a concrete class satisfies it

```
public class LeaveService
{
    private readonly INotificationService notifications;

    public LeaveService(INotificationService notifications)
    {
        this.notifications = notifications;
    }

    public void TakeLeave(Employee employee, DateTime start,
                          DateTime end)
    {
        // do stuff

        this.notifications.Notify(employee, "Leave approved.");
    }
}
```

Dependencies are injected at runtime

Persistence Ignorance

“…ordinary classes where you focus on the business problem at hand without adding stuff for infrastructure-related reasons… nothing else should be in the Domain Model.”

```
public class Customer
{
    public int Id { get; private set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public IEnumerable<Address> Addresses { get; }
    public IEnumerable<Order> Orders { get; }

    public Order CreateOrder(ShoppingCart cart)
    {
        ...
    }
}
```

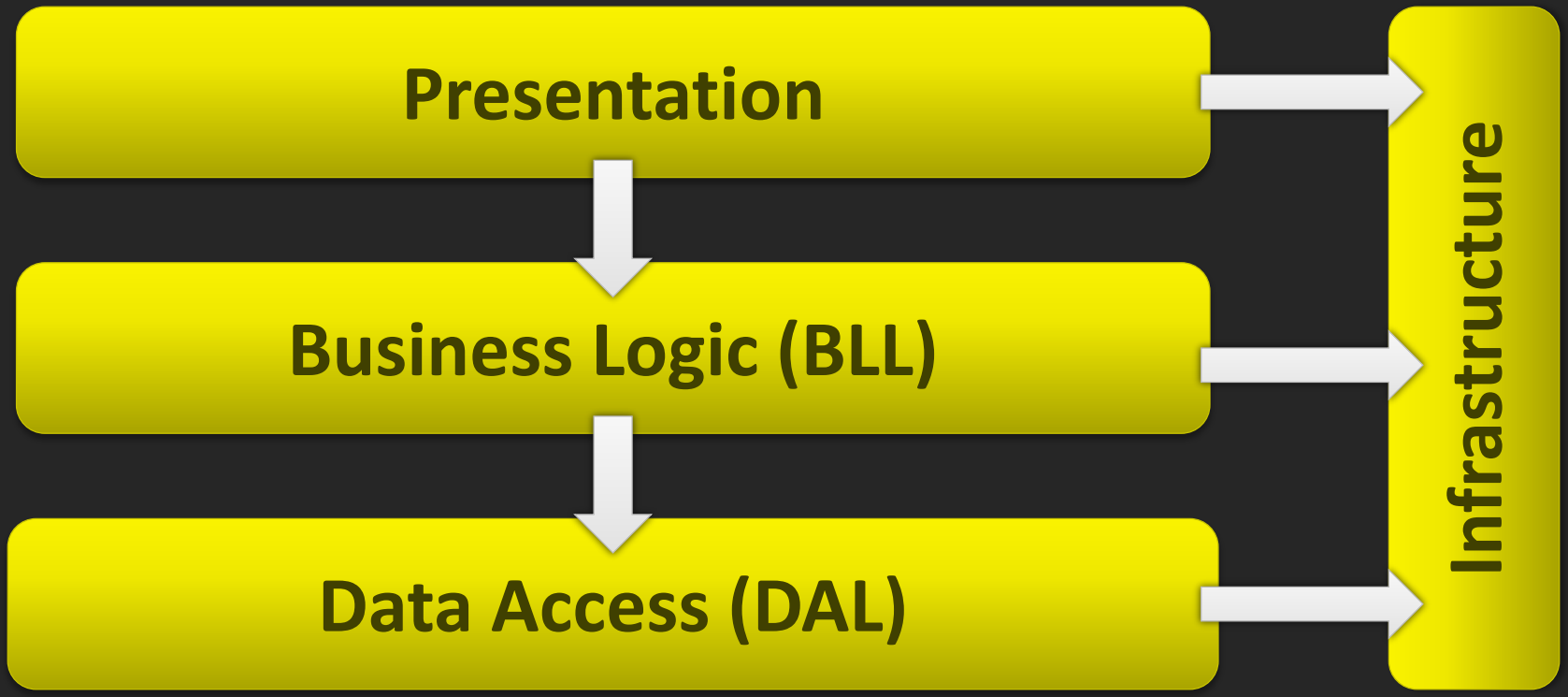
Plain Old CLR Object (POCO)

```
[global::System.Data.Objects.DataClasses.EdmEntityTypeAttribute(NamespaceName="AdventureWorksLTModel",
Name="Customer")]
    [global::System.Runtime.Serialization.DataContractAttribute(IsReference=true)]
    [global::System.Serializable()]
    public partial class Customer : global::System.Data.Objects.DataClasses.EntityObject
    {
        [global::System.Data.Objects.DataClasses.EdmScalarPropertyAttribute(EntityKeyProperty=true,
IsNullable=false)]
        [global::System.Runtime.Serialization.DataMemberAttribute()]
        public int CustomerID
        {
            get
            {
                return this._CustomerID;
            }
            set
            {
                this.OnCustomerIDChanging(value);
                this.ReportPropertyChanging("CustomerID");
                this._CustomerID =
global::System.Data.Objects.DataClasses.StructuralObject.SetValidValue(value);
                this.ReportPropertyChanged("CustomerID");
                this.OnCustomerIDChanged();
            }
        }
        private int _CustomerID;
        partial void OnCustomerIDChanging(int value);
        partial void OnCustomerIDChanged();
```

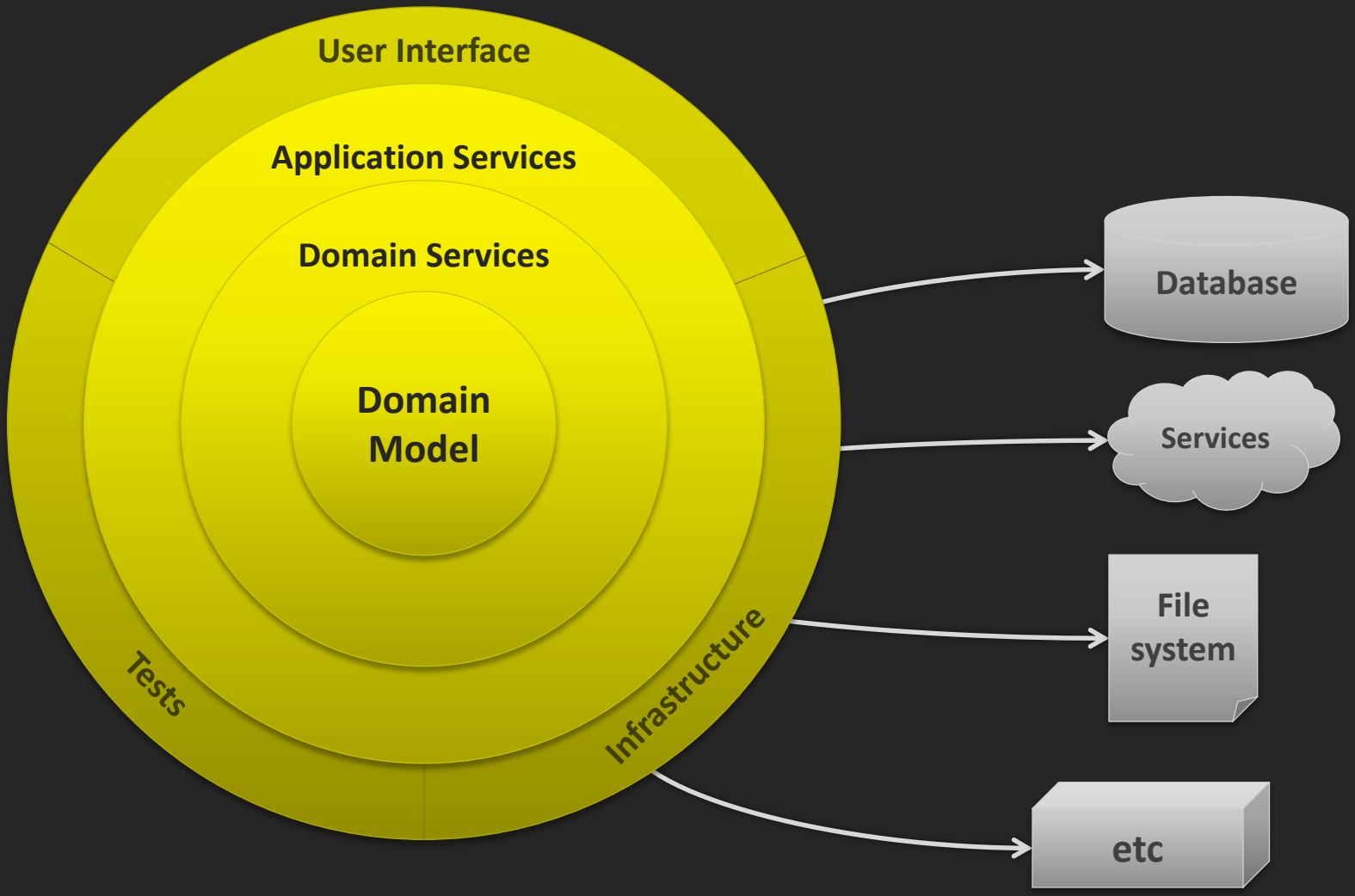
This is not a POCO.

Architecture

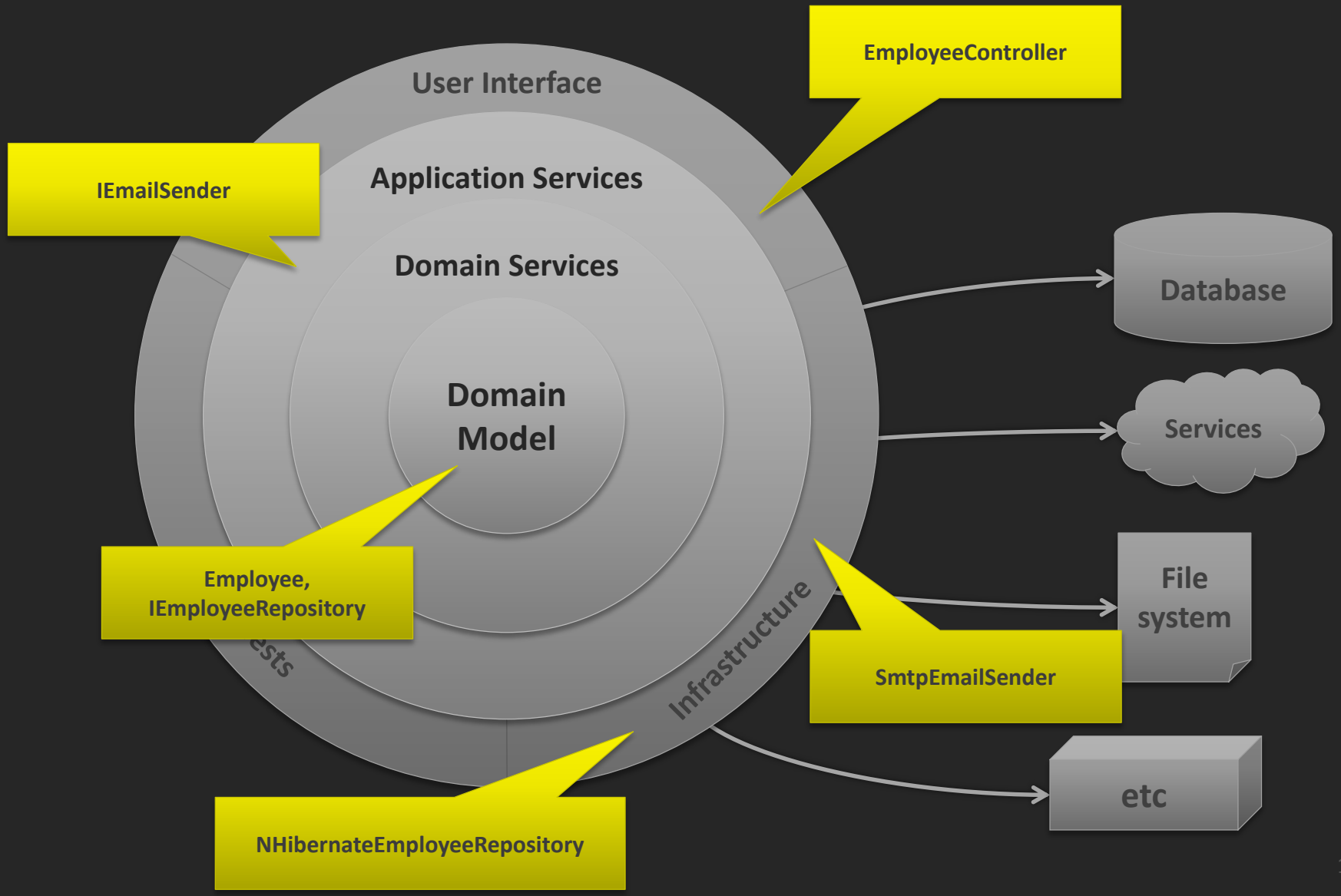
Traditional Architecture



Onion Architecture



Onion Architecture



Validation

Validation Examples

Input validation

Business domain

Is the first name filled in?

Is the e-mail address format valid?

Is the first name less than 255 characters long?

Is the chosen username available?

Is the password strong enough?

Is the requested book available, or already out on loan?

Is the customer eligible for this policy?

```
public class PersonRepository : IPersonRepository
{
    public void Save(Person customer)
    {
        if (!customer.IsValid())
            throw new Exception(...)
    }
}
```

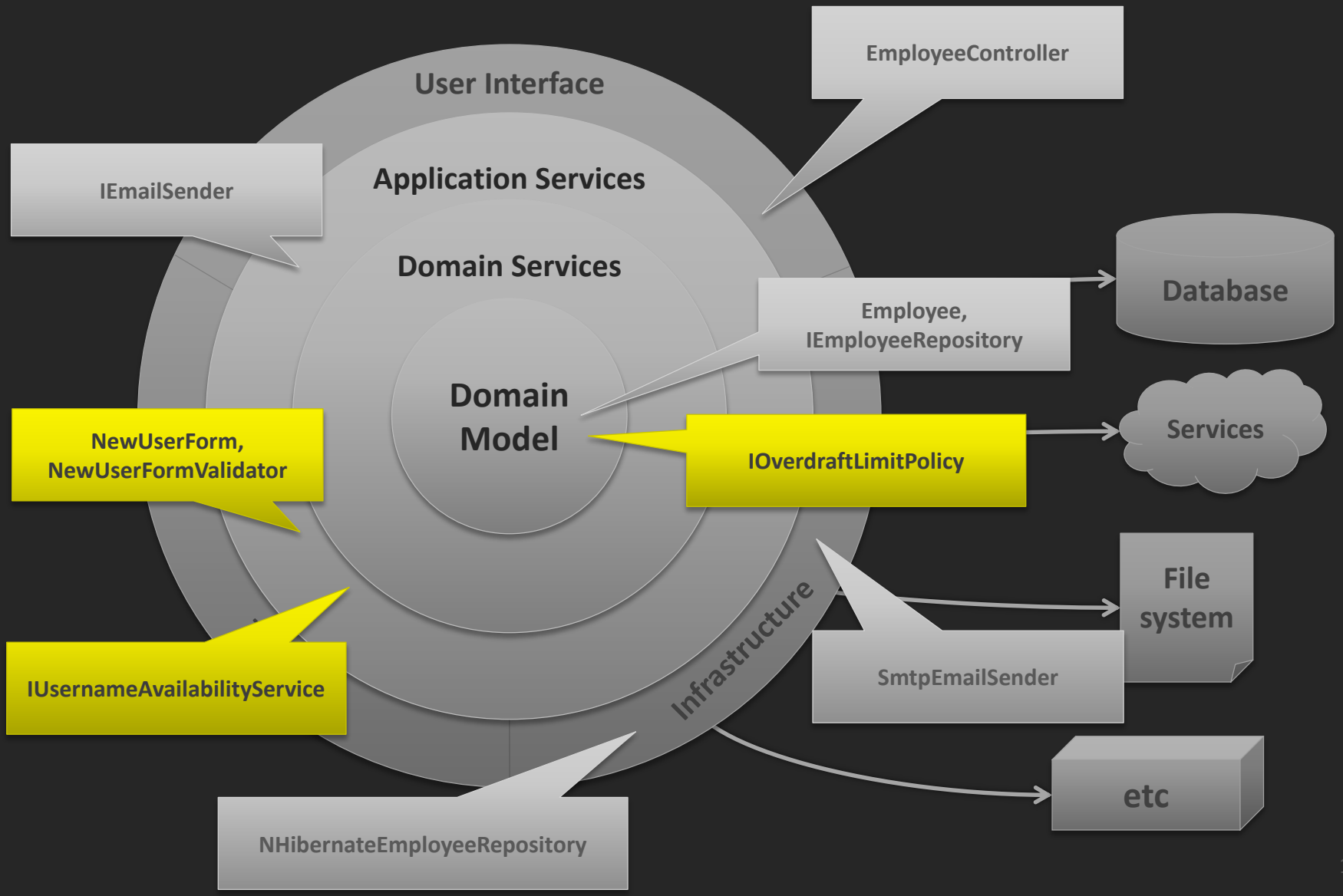
validation and persistence anti-patterns


The golden rule for validation:
The Domain Model is always
in a valid state

```
public class NewUserFormValidator : AbstractValidator<NewUserForm>
{
    IUsernameAvailabilityService usernameAvailabilityService;

    public NewUserFormValidator()
    {
        RuleFor(f => f.Email).EmailAddress();

        RuleFor(f => f.Username).NotEmpty().Length(1, 32)
            .WithMessage("Username must be between 1 and 32 characters");

        RuleFor(f => f.Url).Must(s => Uri.IsWellFormedUriString(s))
            .Unless(f => String.IsNullOrEmpty(f.Url))
            .WithMessage("This doesn't look like a valid URL");

        RuleFor(f => f.Username)
            .Must(s => this.usernameAvailabilityService.IsAvailable(s))
            .WithMessage("Username is already taken");
    }
}
```

separation of validation concerns with FluentValidation

Where validation fits

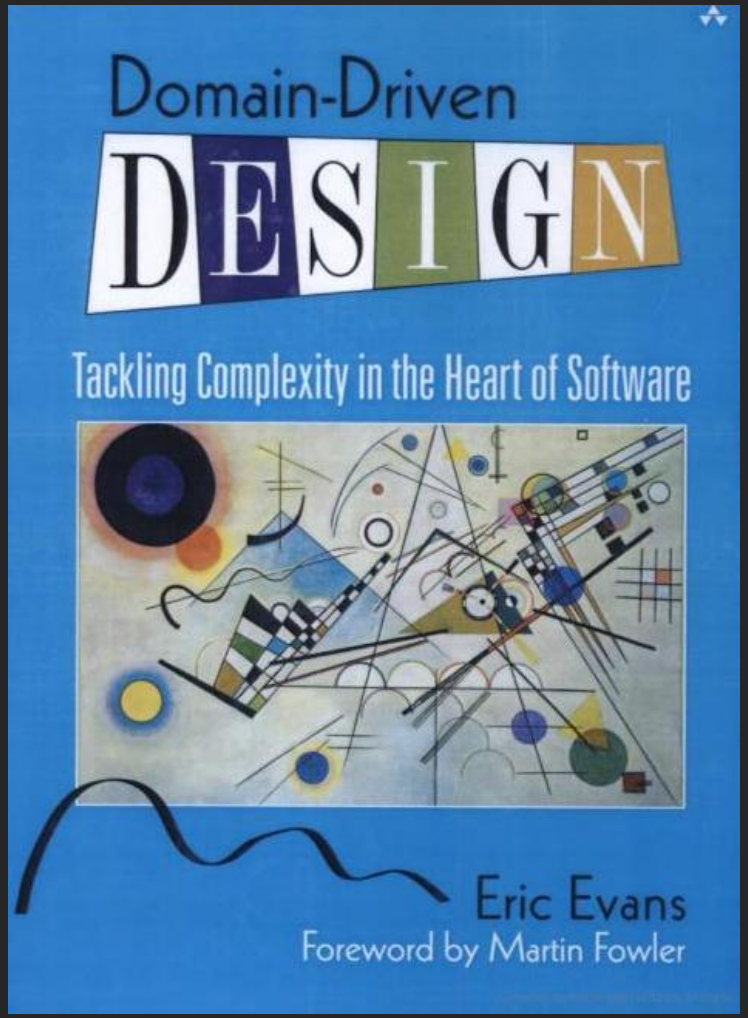


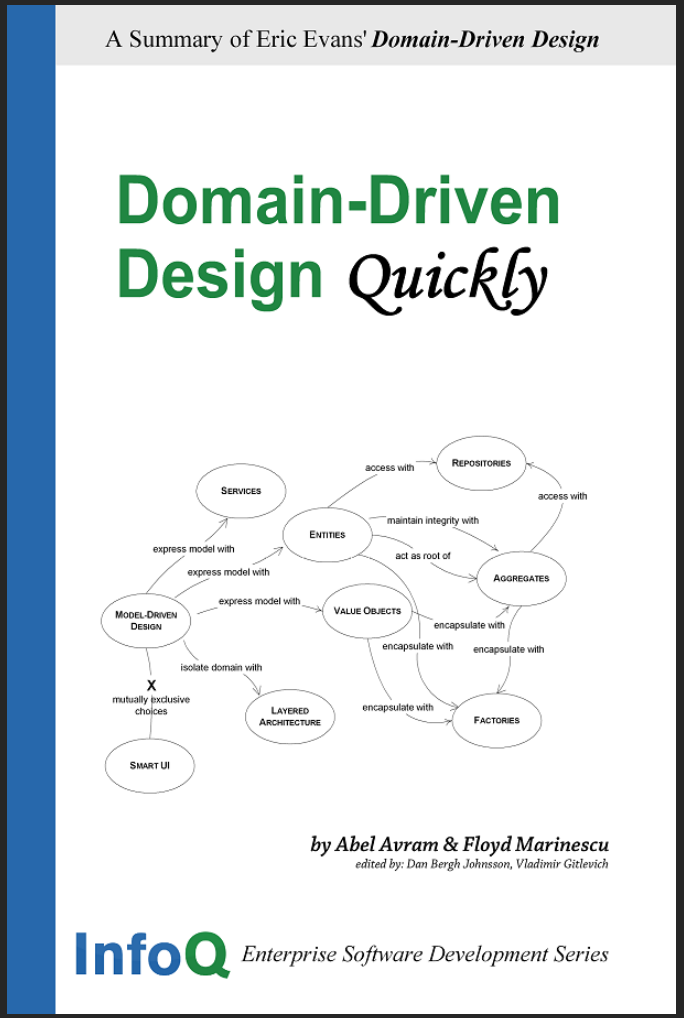

Challenges

When DDD isn’t appropriate


Benefits

Books


Domain-Driven
DESIGN
Tackling Complexity in the Heart of Software
Eric Evans
Foreword by Martin Fowler


A Summary of Eric Evans' Domain-Driven Design
Domain-Driven Design Quickly
Services
Repositories
Entities
Aggregates
Model-Driven Design
Value Objects
Layered Architecture
Factories
Smart UI
access with
maintain integrity with
act as root of
express model with
isolate domain with
encapsulate with
mutually exclusive choices
by Abel Avram & Floyd Marinescu
edited by: Dan Bergh Johnsson, Vladimir Gitlevich
InfoQ Enterprise Software Development Series

Links

Domain Driven Design mailing list

- http://tech.groups.yahoo.com/group/domaindrivendesign/

ALT.NET mailing list

- http://tech.groups.yahoo.com/group/altdotnet/

DDD Step By Step

- http://dddstepbystep.com/

Domain Driven Design Quickly (e-book)

- http://www.infoq.com/minibooks/domain-driven-design-quickly

“Any fool can write code that a computer can understand. Good programmers write code that humans can understand.”

Martin Fowler

דוגמא

- https://github.com/andreazevedo/petstore-ddd-csharp

סיכום

- מודלים לתהליכי ושלבי פיתוח תוכנה
  - תיכון וארכיטקטורה
- נושאים נוספים
  - תפקידו של הארכיטקט?
  - תבניות תיכון
  - Event Sourcing
- Coursera, Mobile Cloud Computing with Android, review of the many patterns Android is made of