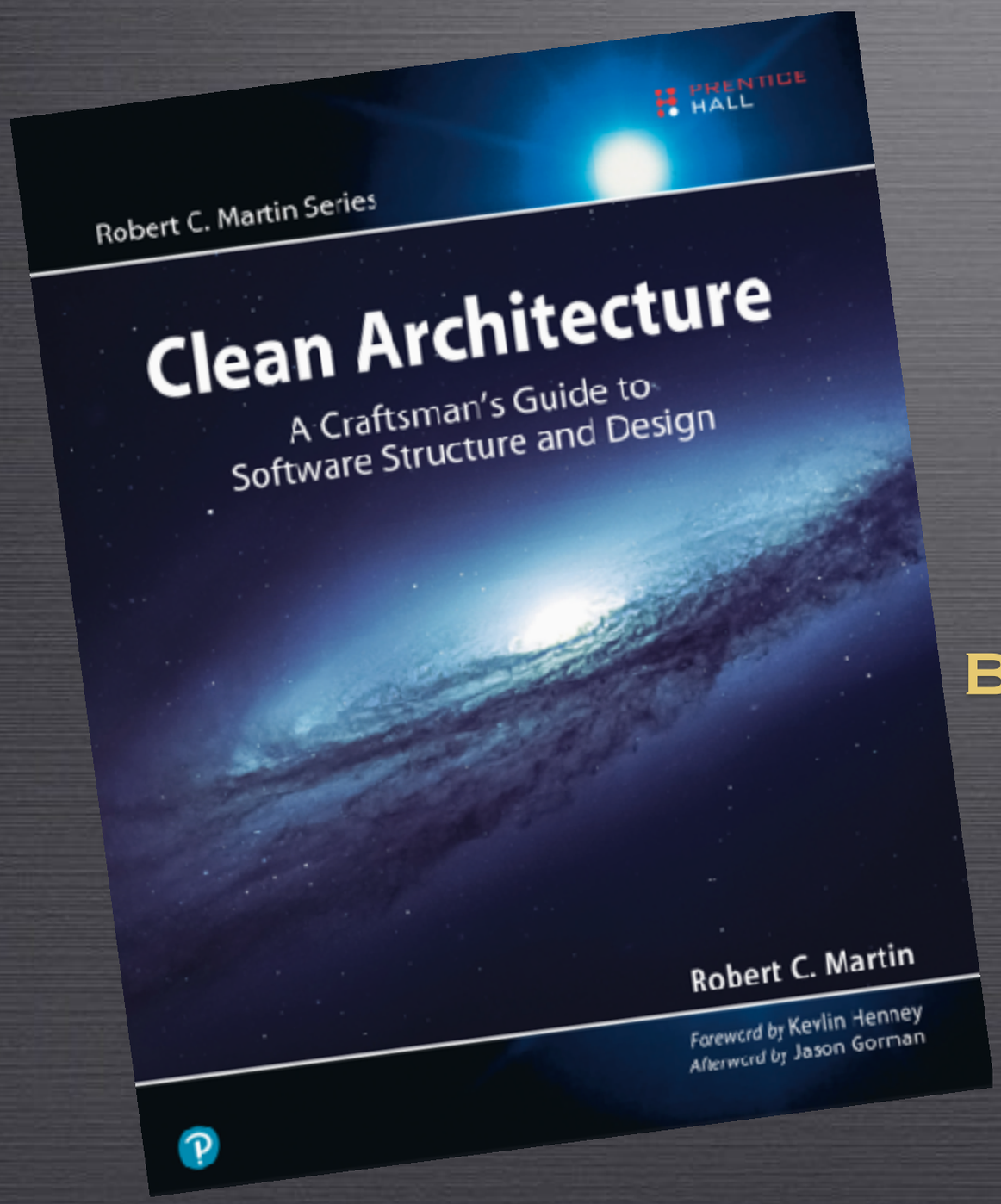
Clean Architecture



By Robert C. Martin
(Uncle Bob)

Welcome!

- Daily Schedule
- Introductions
  - Name
  - How long you have been coding?
  - Language
  - Why are you taking this course?

Introduction

- I've built lots of apps.
- The architecture rules are independent of every other variable.
- Twenty-two orders of magnitude in hardware. Yet software remains the same.
- Not HAL, just machines governed by timeless, changeless, rules.

Working vs. Right

- Getting software working is the easy part.
  - Getting it right is the hard part.
- Magic happens when you get it right.
  - Effort is minimized.
  - Functionality and flexibility are maximized.

What is Design and Architecture?

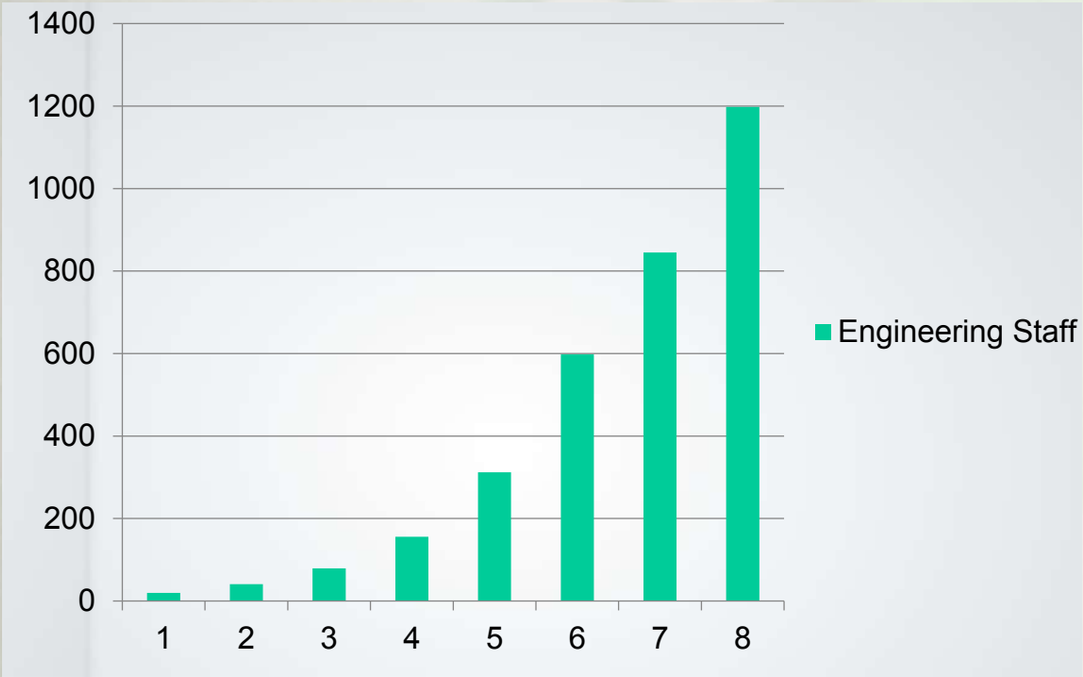
Design vs. Architecture

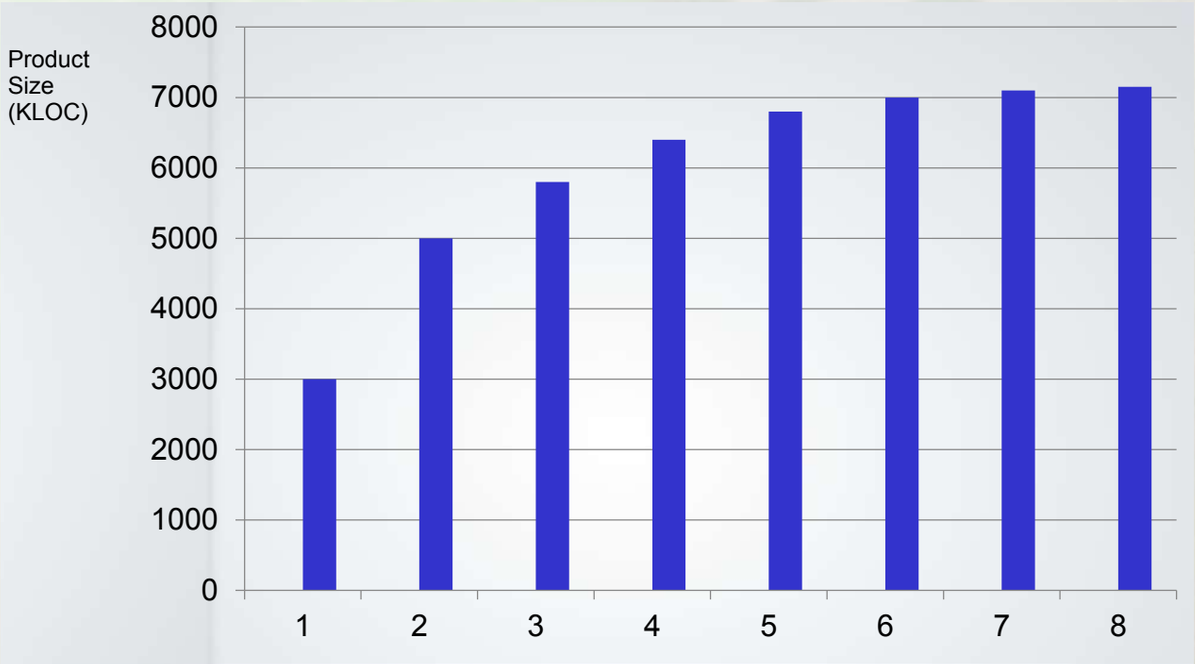
- No difference.
- Architect's House Plans.

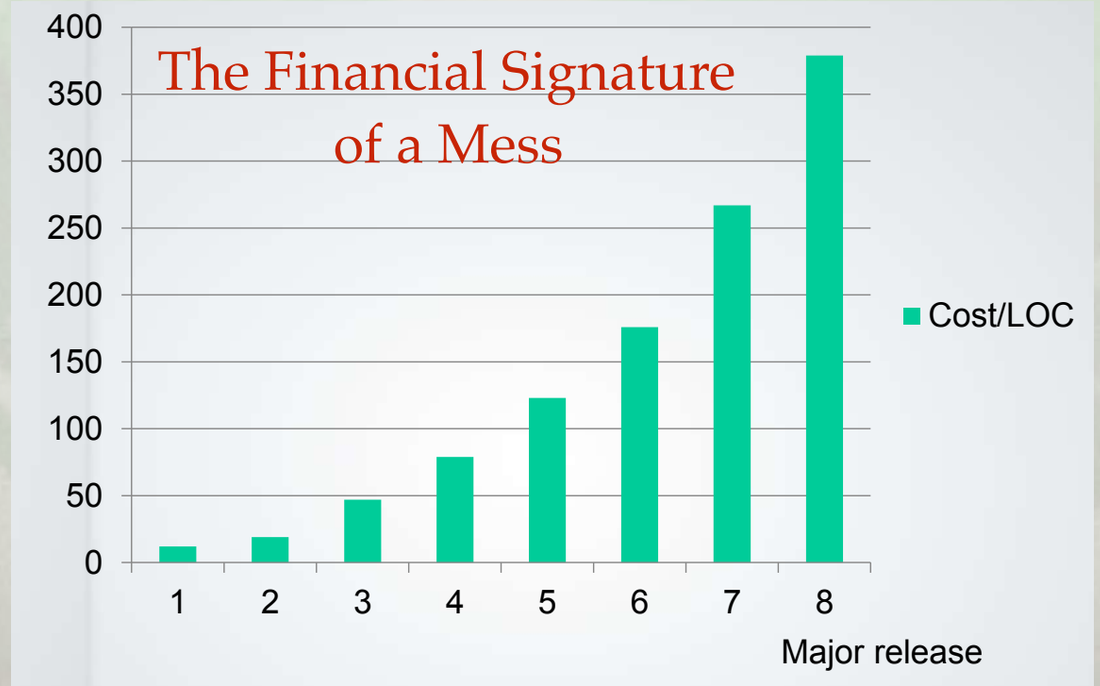
THE GOAL

- Minimize the human resources required to build and maintain software systems.
- The measure of design quality is simply the measure of effort required to meet the needs of the customer.
- If effort remains constant or shrinks with time then the design is good.
- If effort increases with each passing release, the design is bad.

Case Study of Bad Architecture

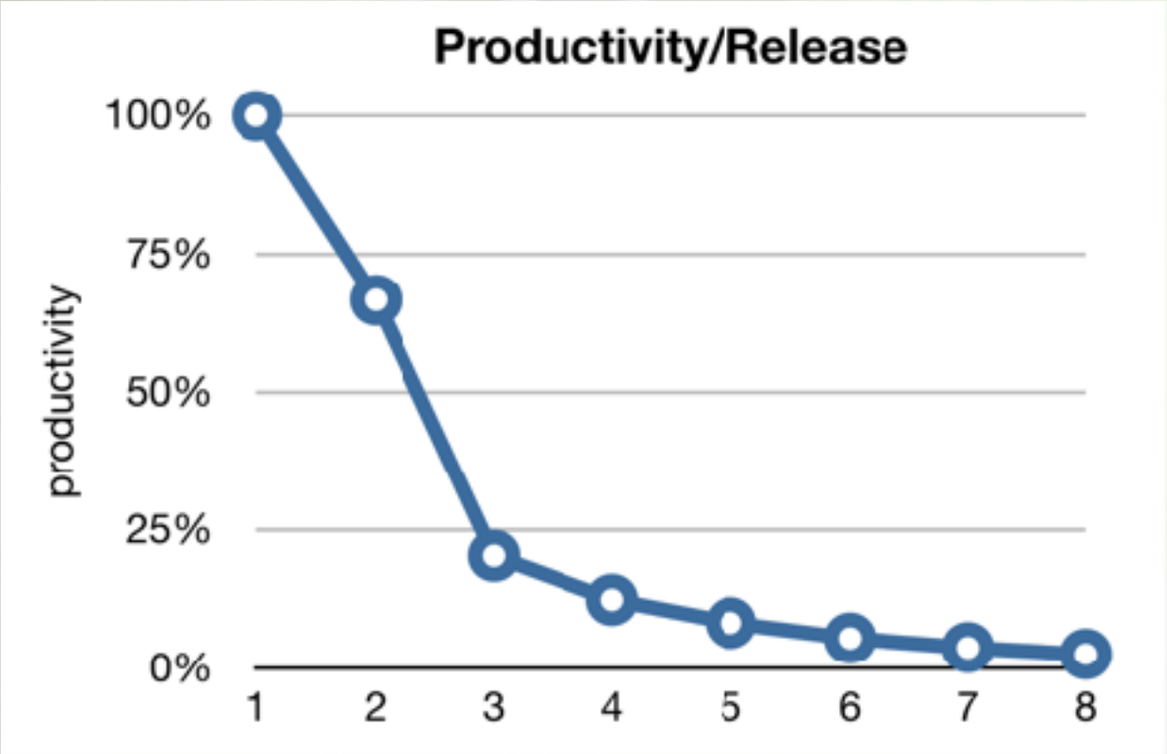


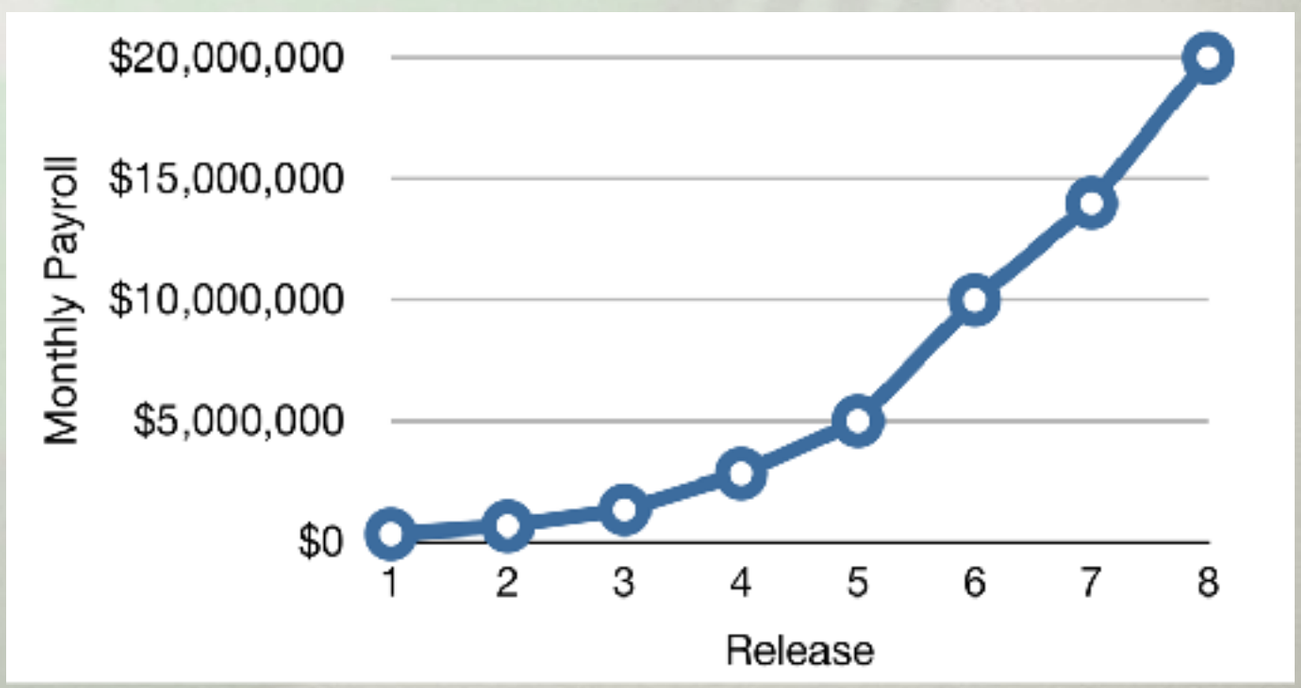




Courtesy: Jason Gorman

The Executive View

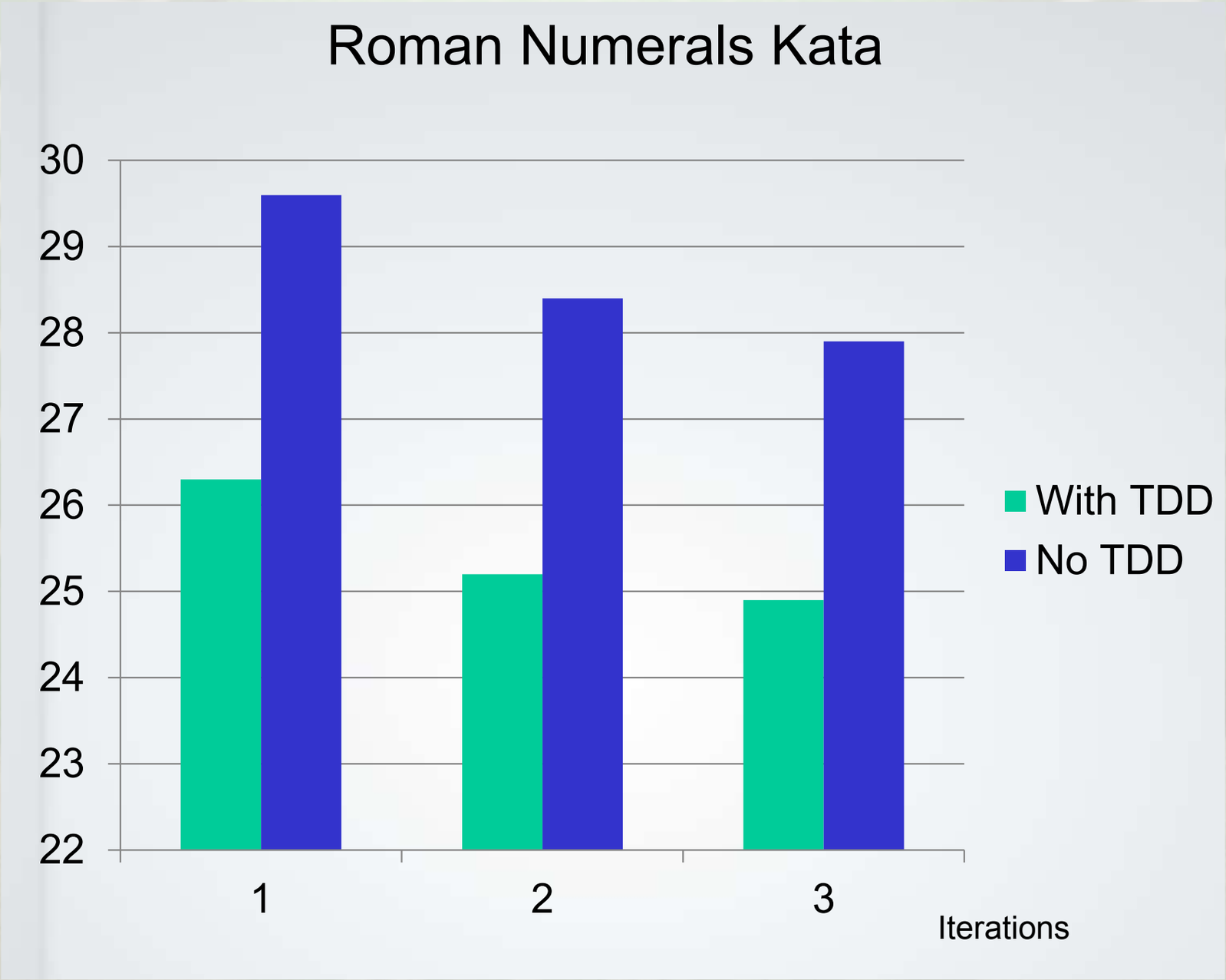




What Went Wrong

- Tortoise and the Hare. Aesop. 2600 ya.
  - Foolishness and overconfidence.
    - "We can remain productive and be messy. We can clean it up later."
- But the creeping mess of code that saps their productivity never sleeps, and never relents.
  - If given it's way it will reduce their productivity to zero in a matter of months.

Messes aren't faster even in the short term.



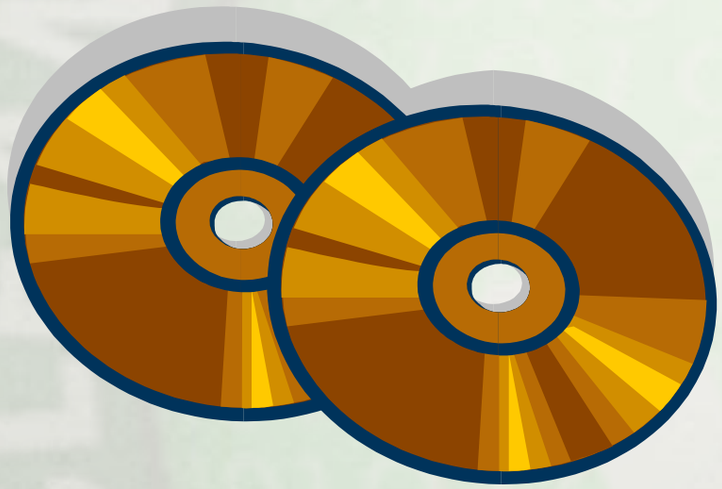
Courtesy: Jason Gorman

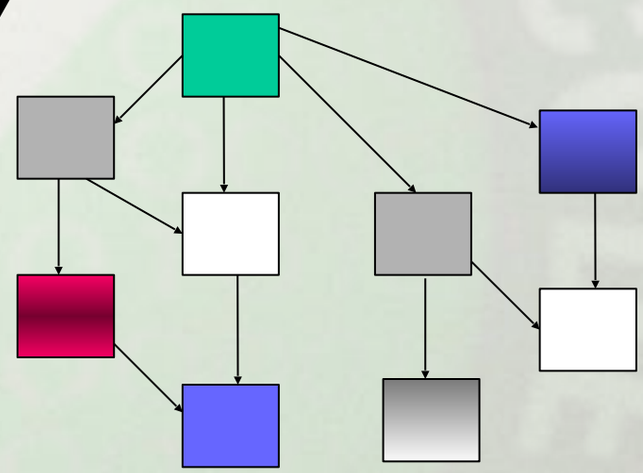
Solution to the Executive's Dilemma

- The only way to reverse the decline in productivity is to realize:
  - The only way to go fast, is to go well.
  - And set that expectation.

A tale of Two Values

The Two Values of Software





Behavior

- Help stakeholders make or save money.
  - By making machines behave in ways that satisfy the requirements.
- Many programmers believe this is the entirely of their job.
  - They are sadly mistaken.

Architecture

- Soft-Ware is a compound word.
- When stakeholders change their minds the change to the software should be simple and easy to make.
- The cost of change should be proportional to scope, not shape.

Scope vs. Shape

- Stakeholders believe that requirements and changes are complicated only by scope.
- Programmers see each new requirements as a puzzle piece that must be fit into an every more complex puzzle.
- Solution: Keep the puzzle simple.

The Greater Value

- If I give you a system that works perfectly, but is hard to change, it will stop working when the requirements change and will never work again.
- If I give you a system that does not work, but is easy to change, you can make it work, and keep it working.

Stakeholders want Changes

- But they never prioritize architecture.
- Eisenhower: *I have two kinds of problems, the urgent and the important. The urgent are not important, and the important are never urgent.*

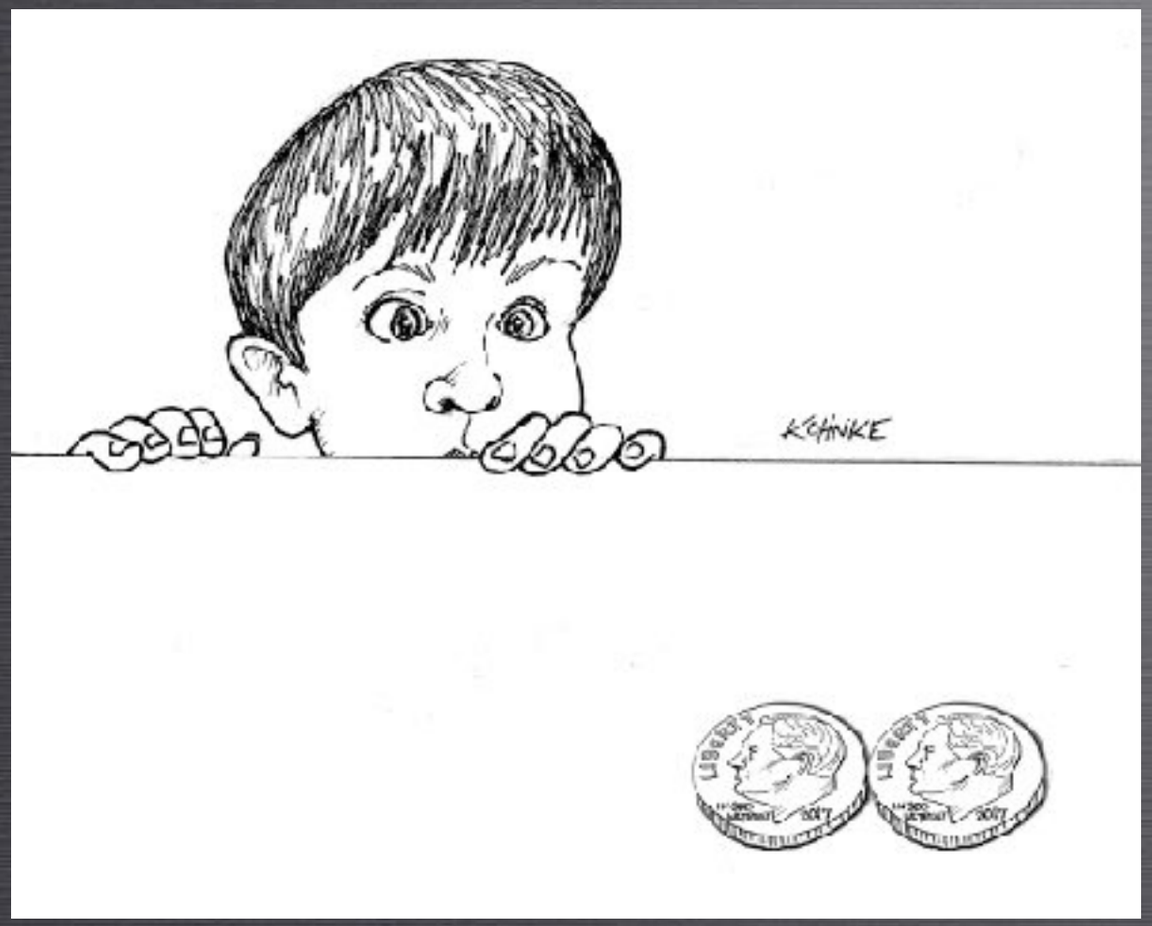
Urgency and Importance

IMPORTANT URGENT	IMPORTANT NOT URGENT
UNIMPORTANT URGENT	UNIMPORTANT NOT URGENT

- Behavior is Urgent, Architecture is Important
- Stakeholders prioritize Urgency.
- YOU are the engineers: responsible for Importance.

Fight for the Architecture

- It's always a struggle.
- YOU are a stakeholder too! Argue as equals.
- If architecture comes last, costs will grow. "Behind the Power Curve".



Starting with the Bricks:

Programming Paradigms

The Three Paradigms

- Structured Programming: Dijkstra, 1968
- Object-Oriented Programming: Dahl & Nygard, 1966
- Functional Programming, Church, 1936, McCarthy, 1958.

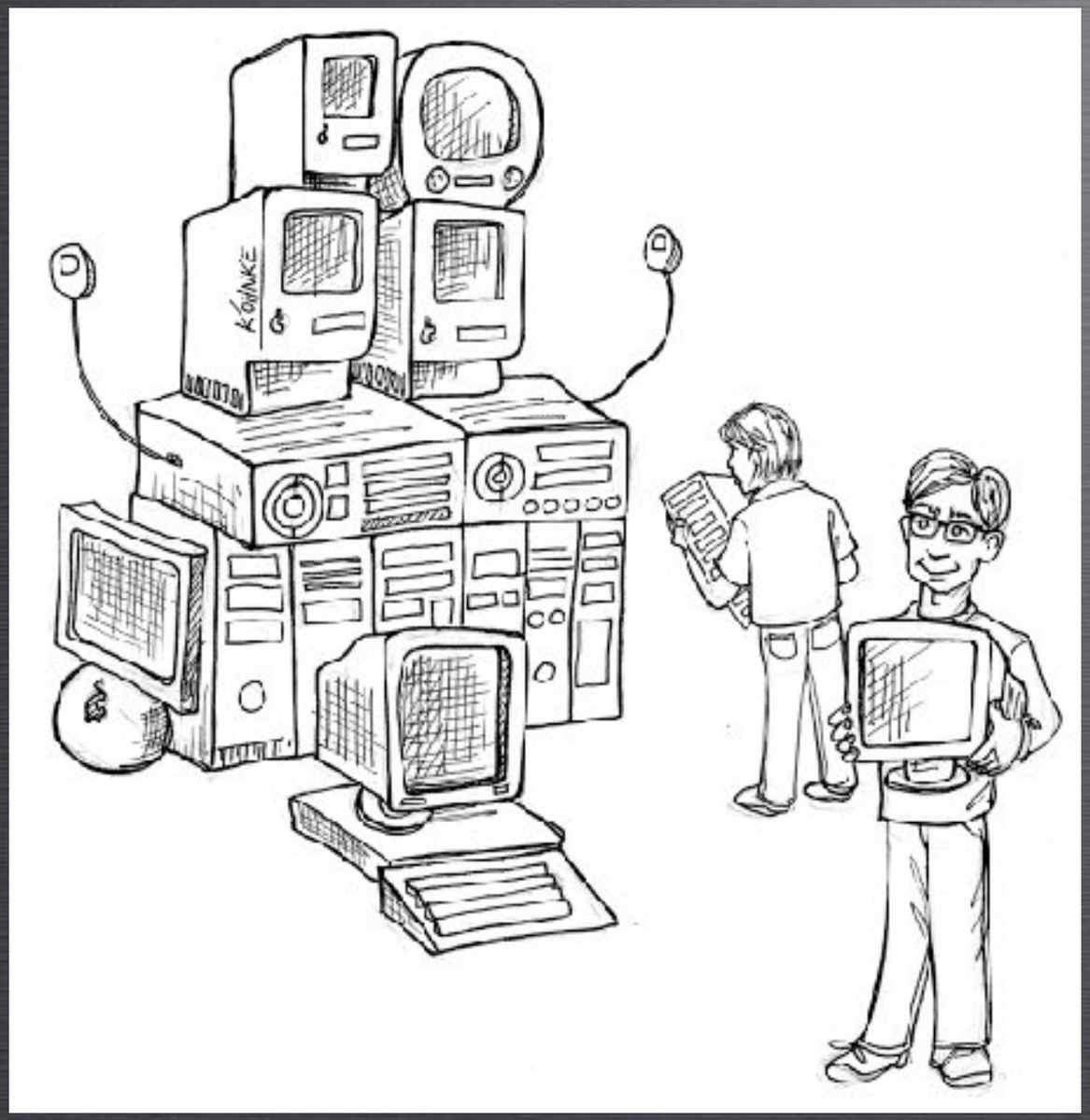
The Three Constraints

- SP:
  Imposes discipline on direct transfer of control.
- OOP:
  Imposes discipline on indirect transfer of control.
- FP:
  Imposes discipline on assignment.

- *Is there anything left to constrain?*

The Architectural Concerns

- SP:
  Algorithmic foundation of modules.
- OOP:
  Component Boundaries.
- FP:
  Data Location and Access.



Structured Programming

Edsger Wybe Dijkstra

- b. 1930 Rotterdam
- Graduated H.S. 1948 Highest possible marks in math, physics, chemistry and biology.
- Netherlands very first programmer in March, 1952.
- Decided in 1955 that software was greater intellectual challenge than theoretical physics.
- Married Maria Debets, 1957. Dutch authorities were unwilling to accept "programmer" as profession on marriage license, so he put: "Theoretical Physicist".
- Concerned that there was no science or discipline. His advisor told him he may be one of those who raise software to a science.
- Vacuum tubes, paper tape, punched cards. Fragile, slow, unreliable machines.

Proof

- Programming is hard. Programmers don't do it well.
- Dijkstra's solution: Employ the mathematical discipline of proof.
- Create a Euclidian hierarchy of postulates, theorems, and lemmas that programmers could draw from.

Goto

- Dijkstra showed, that programs with unrestrained **`goto`** could be unprovable.
- Böhm & Jacopini shows all programs could be constructed with sequence. selection and iteration.
- *Goto Statement Considered Harmful*
  Letter to editor of CACM, March, 1968

The War was Won

- The Flame war before the internet.
- The gradual acceptance.
- The enforcement by language.

Functional Decomposition

- SP forces programs to be decomposed into independent provable units: *functions*.
- *Structured Analysis* - DeMarco
- *Structured Design* - Page-Jones
- All due to the quest for *provability*.

The Quest Failed

- We don't write proofs. Dijkstra's dream never materialized.
- There is no Euclidean hierarchy of postulates, theorems, and lemmas.
- The mathematical discipline fizzled.
- But mathematical proofs are not the only way to show correctness.

Science to the Rescue

- Math is a positive discipline. Things are proven correct.
- Science is a negative discipline. Things are proven incorrect.
- A Scientific Theory is never “correct”. But the possible incorrectness can be very narrowly constrained.

The Scientific Method

- *"Testing shows the presence, not the absence, of bugs."* Dijkstra.
- Just like a scientific theory. Experiments show incorrectness, not correctness.
- Software is more of a science than a mathematics.

The software scientific method

- Decompose the problem down into a series of provable/falsifiable units.
- Surround those units with enough tests to acceptably constrain their incorrectness.
- Test Driven Development.

Object Oriented Programming

What is OO?

- Combination of data and function?
  - is `o.f()` really superior to `f(o)`?
- A strategy to model the real world?
  - I've got a bridge to sell you.
- Encapsulation, Inheritance, Polymorphism.

The Perfect Encapsulation of C.

```
point.h
-----

struct Point;
struct Point* makePoint(double x, double y);
double distance (struct Point *p1, struct Point *p2);

point.c
--------

#include "point.h"
#include <stdlib.h>
#include <math.h>

struct Point {
  double x,y;
};

struct Point* makepoint(double x, double y) {
  struct Point* p = malloc(sizeof(struct Point));
  p->x = x;
  p->y = y;
  return p;
}

double distance(struct Point* p1, struct Point* p2) {
  double dx = p1->x - p2->x;
  double dy = p1->y - p2->y;
  return sqrt(dx*dx+dy*dy);
}
```

The Broken Encapsulation of C++

Horrible Hacks

```
point.h
-------

class Point {
public:
  Point(double x, double y);
  double distance(const Point& p) const;

private:
  double x;
  double y;
};

point.cc
---------

#include "point.h"
#include <math.h>

Point::Point(double x, double y)
: x(x), y(y)
{}

double Point::distance(const Point& p) const {
  double dx = x-p.x;
  double dy = y-p.y;
  return sqrt(dx*dx + dy*dy);
}
```

Java and C# just gave up.

Inheritance in C

```
namedPoint.h
------------

struct NamedPoint;
struct NamedPoint* makeNamedPoint(double x,
                                  double y,
                                  char* name);
void setName(struct NamedPoint* np, char* name);
char* getName(struct NamedPoint* np);

namedPoint.c
------------

#include "namedPoint.h"
#include <stdlib.h>

struct NamedPoint {
  double x,y;
  char* name;
};

struct NamedPoint* makeNamedPoint(double x,
                                  double y,
                                  char* name) {
  struct NamedPoint* p = malloc(
    sizeof(struct NamedPoint));
  p->x = x;
  p->y = y;
  p->name = name;
  return p;
}
```

```
void setName(struct NamedPoint* np, char* name) {
  np->name = name;
}

char* getName(struct NamedPoint* np) {
  return np->name;
}

main.c
------

#include "point.h"
#include "namedPoint.h"
#include <stdio.h>

int main(int ac, char** av) {
  struct NamedPoint* origin =
    makeNamedPoint(0.0, 0.0,"origin");
  struct NamedPoint* upperRight =
    makeNamedPoint(1.0, 1.0, "upperRight");

  printf("distance=%f\n",
    distance((struct Point*) origin,
             (struct Point*) upperRight));
}
```

Score so far

- OO gets negative points for encapsulation.
- OO gets half a point for Inheritance.

Polymorphism in C

```
#include <stdio.h>

void copy() {
  int c;
  while ((c=getchar()) != EOF)
    putchar(c);
}
```

getchar & putchar are polymorphic!

Five pointers to Functions

Like a C++ Vtable

```
struct FILE {
  void (*open)(char* name, int mode);
  void (*close)();
  int (*read)();
  void (*write)(char);
  void (*seek)(long index, int mode);
};
```

Final score
What OO gave us?

- Encapsulation: -1
- Inheritance: 0.5
- Polymorphism: 0

Except for one little thing...

Pointers to Functions are Dangerous

- To get polymorphism right in C, everybody had to follow the conventions *perfectly*.
- All the tables had to be initialized correctly.
- All functions had to be called through the tables.
- One mistake and…

So, although we had polymorphism in C, we didn't use it.

OO MADE POLYMORPHISM SAFE.

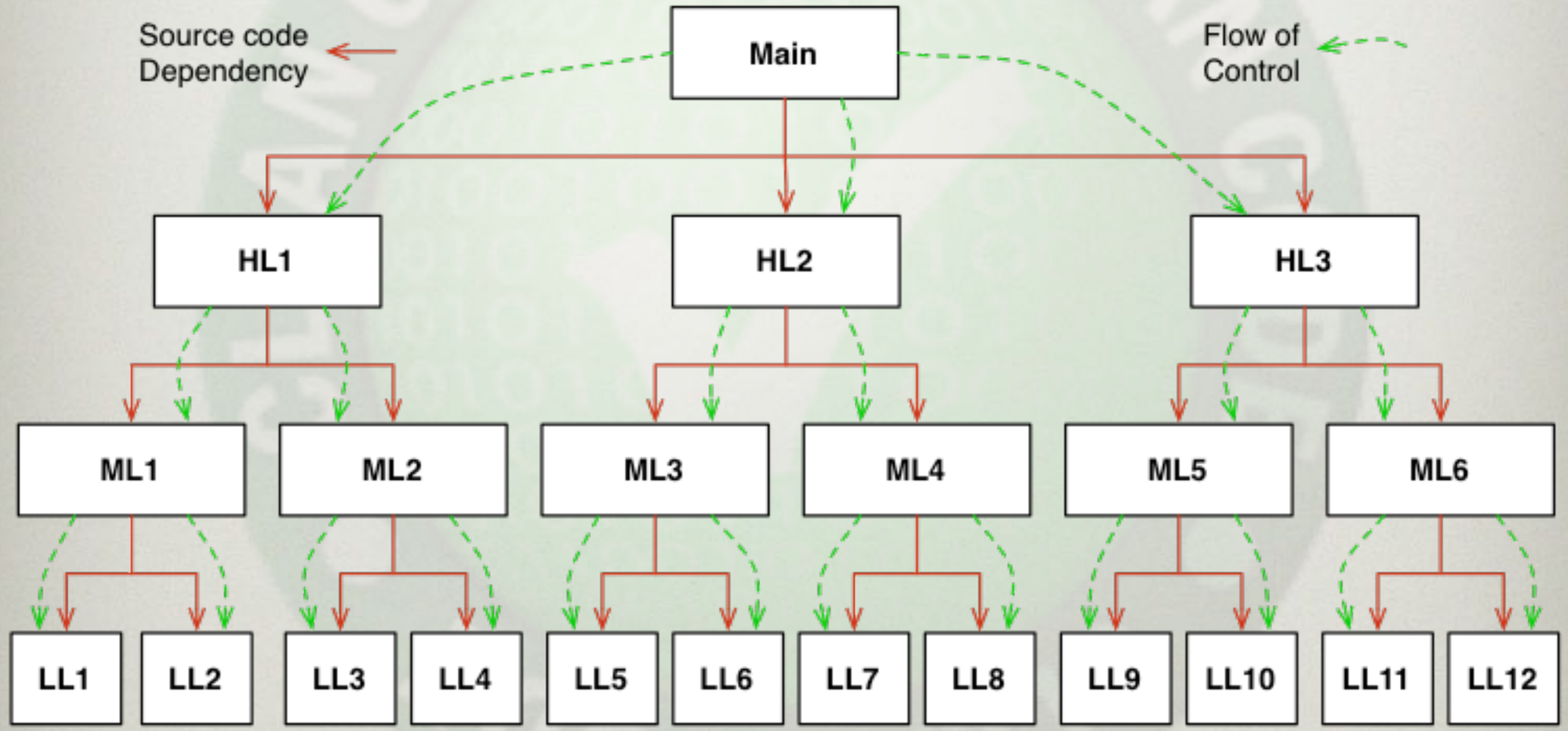
Why is this important?

The Power of Polymorphism

```
#include <stdio.h>

void copy() {
  int c;
  while ((c=getchar()) != EOF)
    putchar(c);
}
```

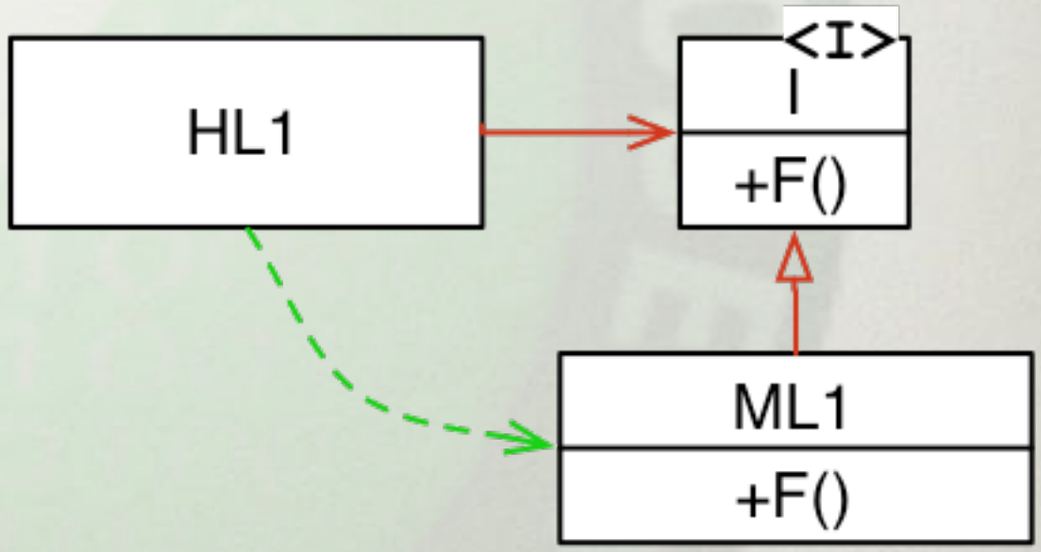
- Device independence. Devices are *plugins*.
- Polymorphism allows independent plugins for anything, not just devices.

Procedural Source Code Dependencies



OO Dependency Inversion

- OO Languages provide save, convenient, polymorphism.
- This means that *any* source code dependency, no matter where, can be inverted.
- Thus architects have *absolute control* over the direction of their dependencies.



This is power!

What can you do with that power?

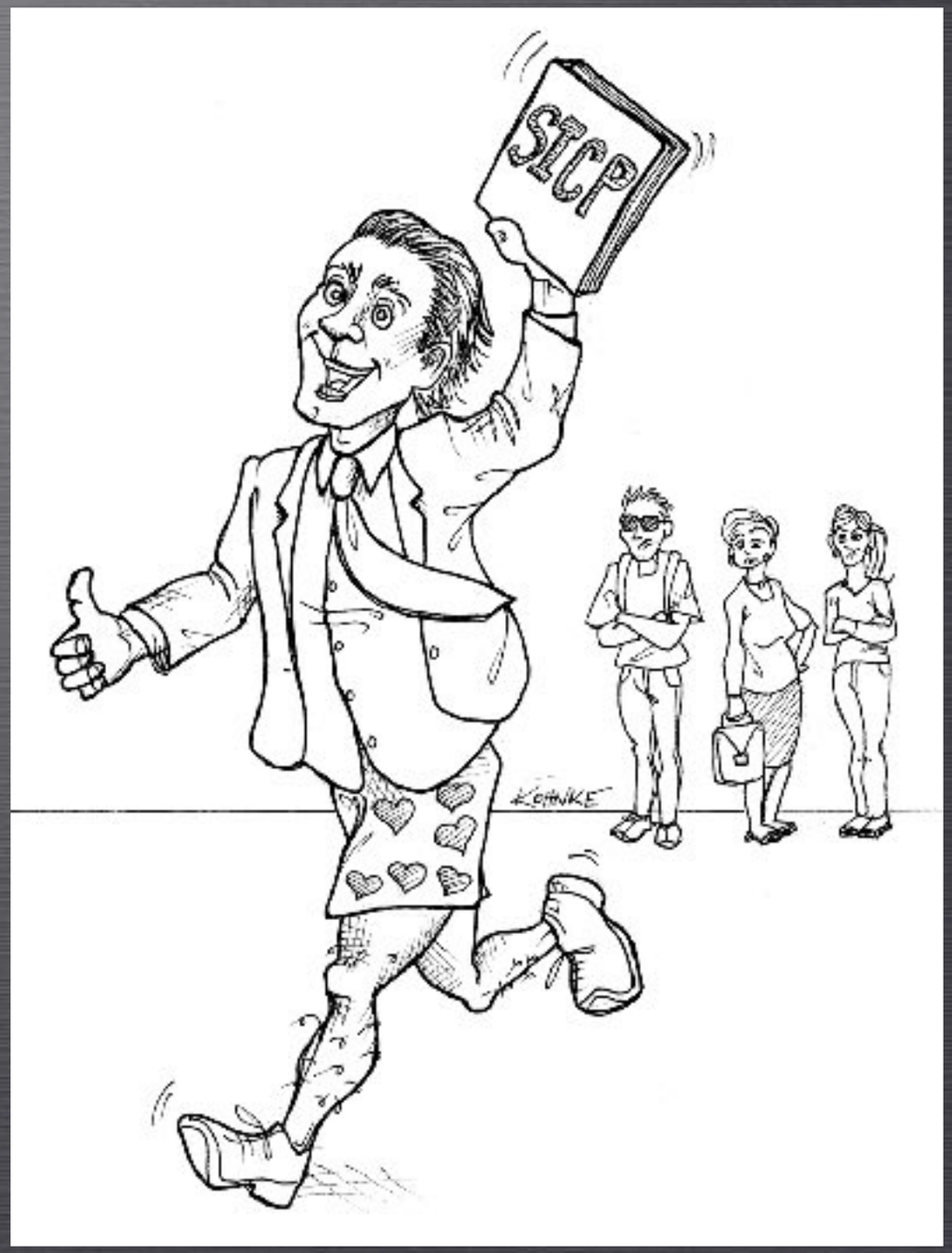
- For example, you could rearrange the source code dependencies in your system so that the UI and Database were *plugins* to the business rules.
- Independent Deployability!

So what is OO?

- The ability, through the use of polymorphism, to gain absolute control over every source code dependency in the system.
- This allows a plugin architecture in which high level modules are independent of low level details.

And all because we imposed a little discipline on pointers to functions...

Functional Programming



Squares of Integers

Java Procedural

```
public class Squint {
  public static void main(String args[]) {
    for (int i=0; i<25; i++)
      System.out.println(i*i);
  }
}
```

Clojure Functional

```
(println (take 25 (map (fn [x] (* x x)) (range))))
```

Clojure Functional Broken down

```
(println ;_____________________ Print
  (take 25 ;___________________ the first 25
    (map (fn [x] (* x x)) ;__ squares
      (range))))  ;___________ of Integers
```

Infinite List of integers

Mutable Variables

```
public class Squint {
  public static void main(String args[]) {
    for (int i=0; i<25; i++)
      System.out.println(i*i);
  }
}
```

Mutable Variable

```
(println (take 25 (map (fn [x] (* x x)) (range))))
```

No Mutable Variables

FP is programming without mutable variables.

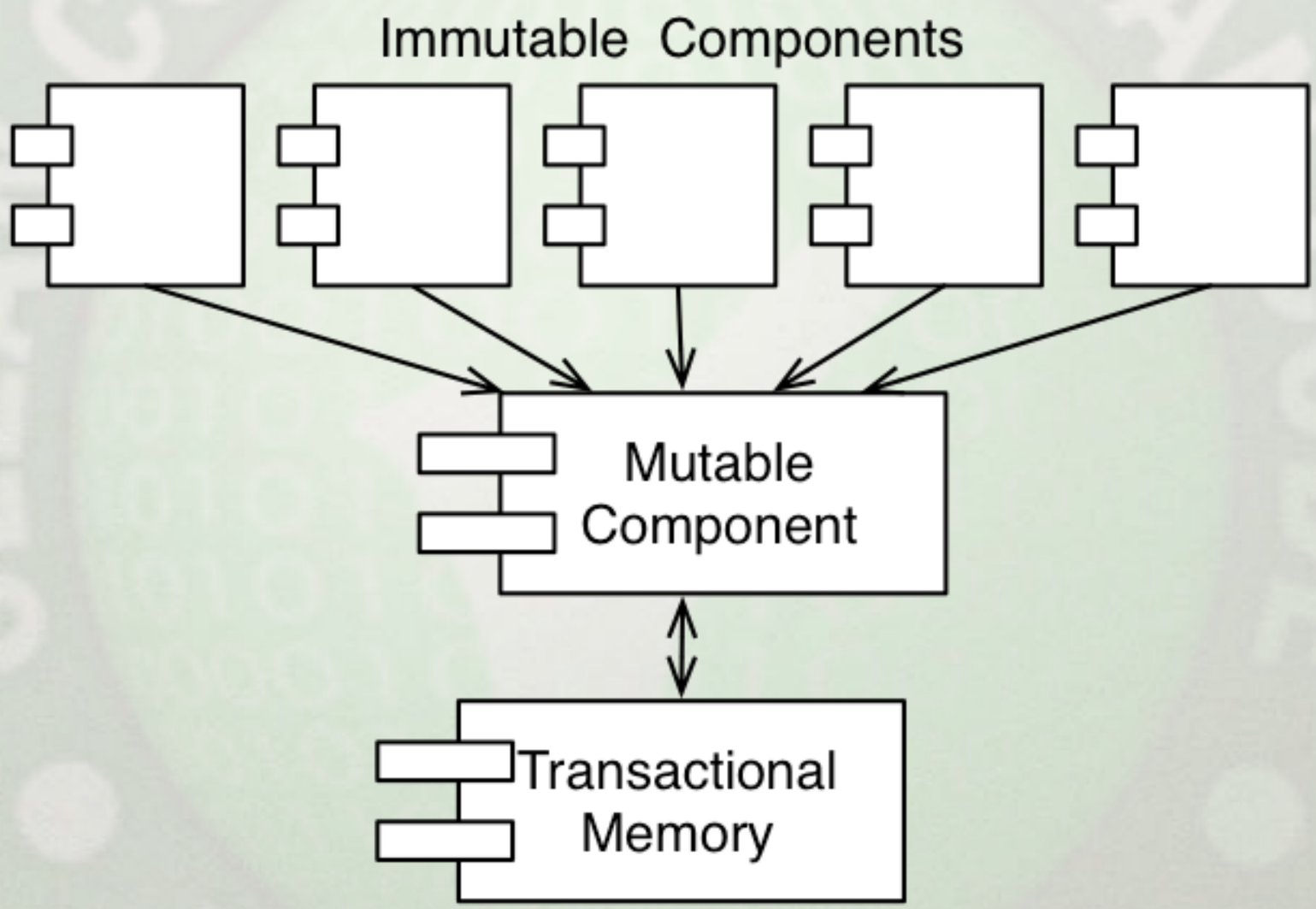
Mutability and Architecture

- All race conditions, deadlocks, and concurrent update problems are caused by *mutable variables*.
- All the problems we face in concurrent, multi-threaded, multi-processor applications, cannot happen without mutability.

The cost of immutability

- If we had infinite memory and processor power, then immutability would be practical.
- Since we don't (though we *almost* do) we will often have to compromise.

Segregation of Mutability




```
(def counter (atom 0)) ; initialize counter to 0
(swap! counter inc)    ; safely increment counter.
```

Event Sourcing

- Imagine a banking application without account records.
- Instead we keep a record of all transactions.
- Account queries simply apply all transactions.
- No updates, or deletes. CR~~UD~~.

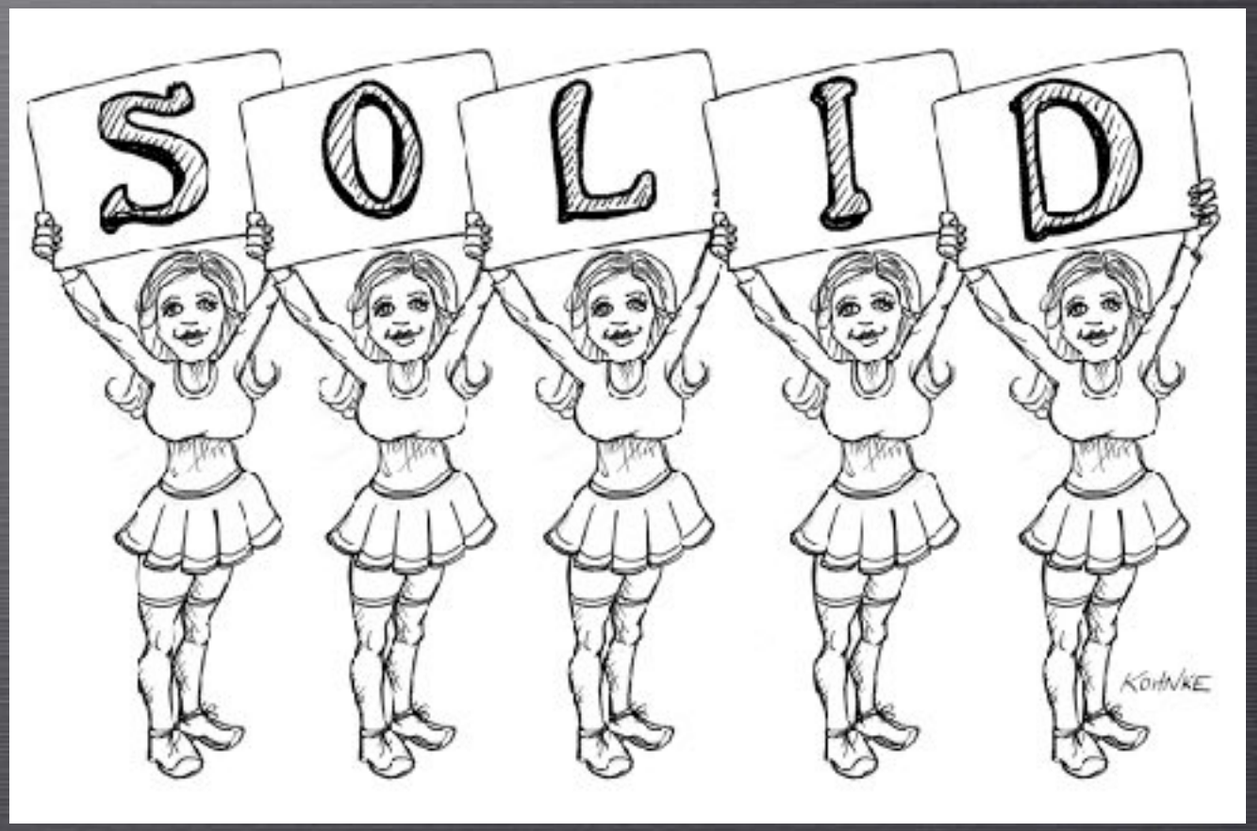
Paradigm Conclusion

- SP: *Discipline imposed upon direct transfer of control.*
- OO: *Discipline imposed upon indirect transfer of control.*
- FP: *Discipline imposed upon assignment (mutability).*
- What we have learned over the last 70 years is what *not* to do.

Unwelcome fact?

- Software is not a rapidly advancing technology. The rules are the same as 1946. The tools and hardware have changed. The essence remains the same.
- Software is composed of sequence, selection, and iteration. Nothing more. Nothing less.

Design Principles



The SOLID principles

- Help us arrange functions and data into classes.
- Help us arrange the interconnections between those classes.
- “class” is simply a coupled group of functions and data in any language.

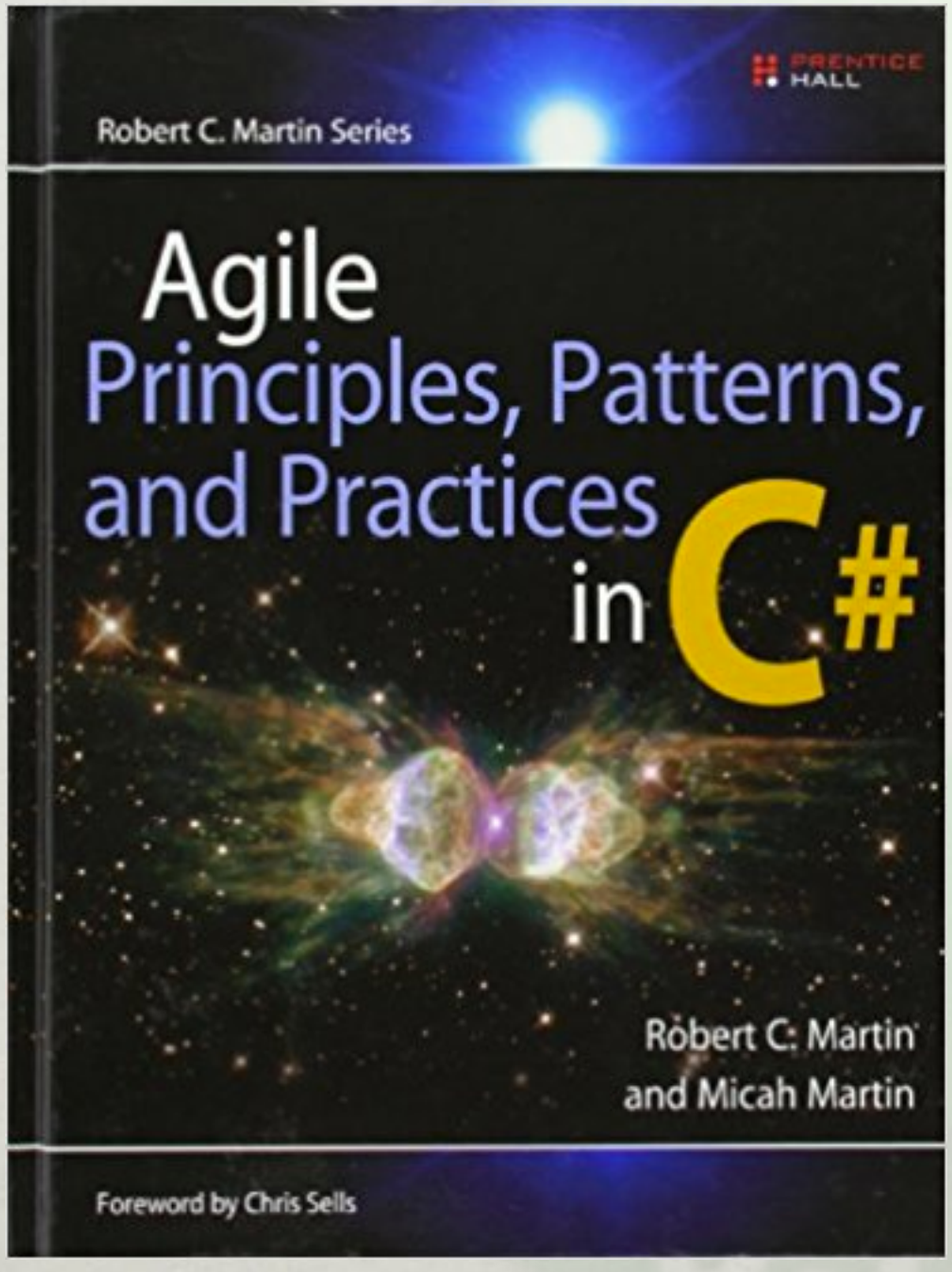
The Goal of SOLID

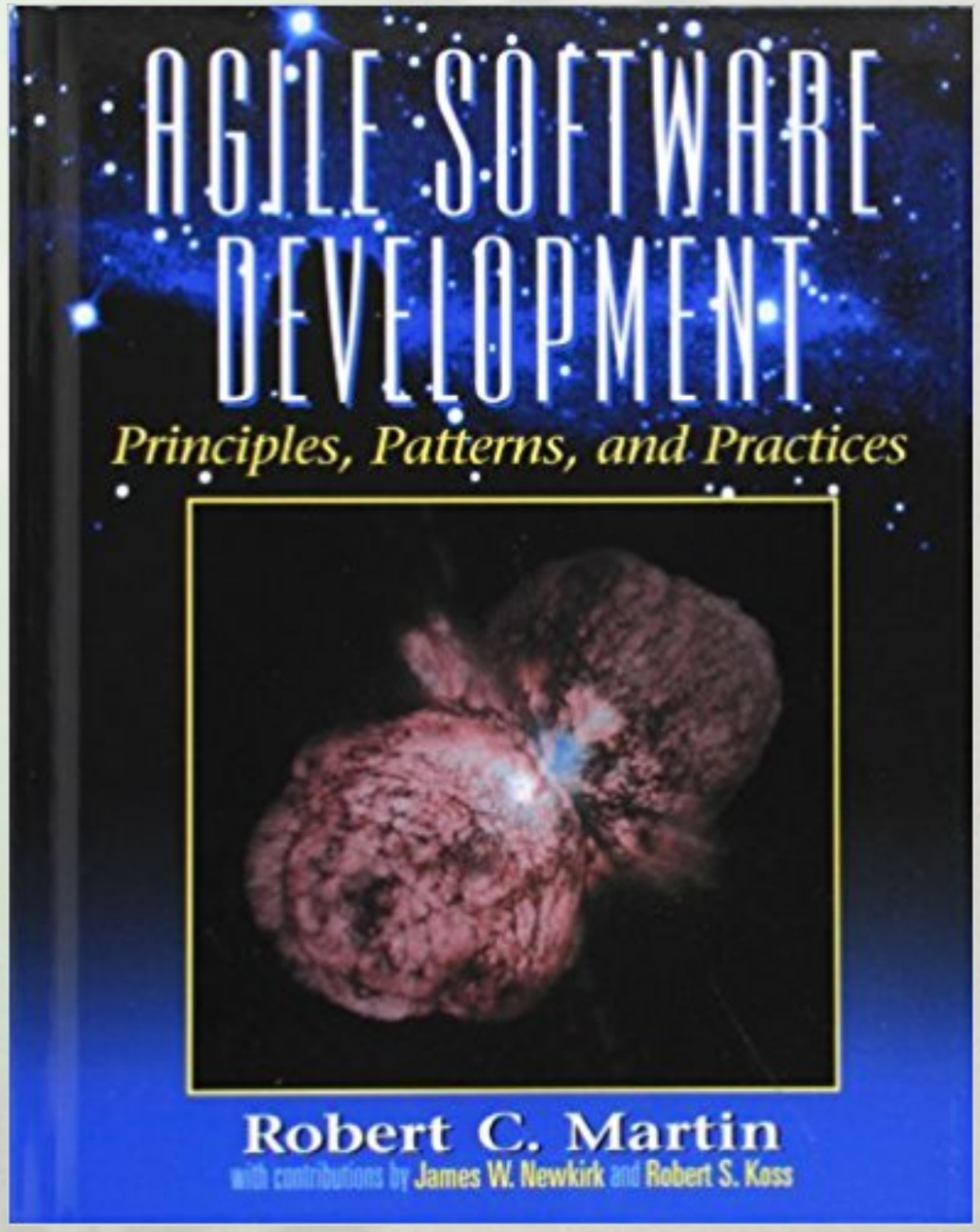
- The creation of mid-level software structure that:
  - Tolerate change.
  - Are easy to understand.
  - Are the basis of reusable components.

SOLID Overview

- SRP: *The Single Responsibility Principle*
- OCP: *The Open-Closed Principle*
- LSP: *The Liskov Substitution Principle*
- ISP: *The Interface Segregation Principle*
- DIP: *The Dependency Inversion Principle*

The Definitive Source



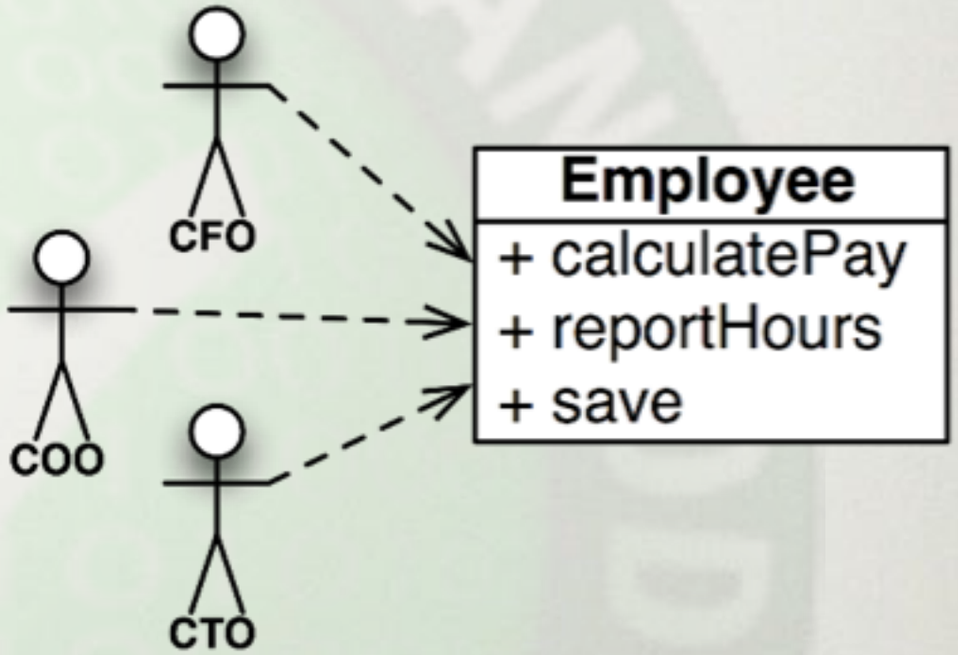


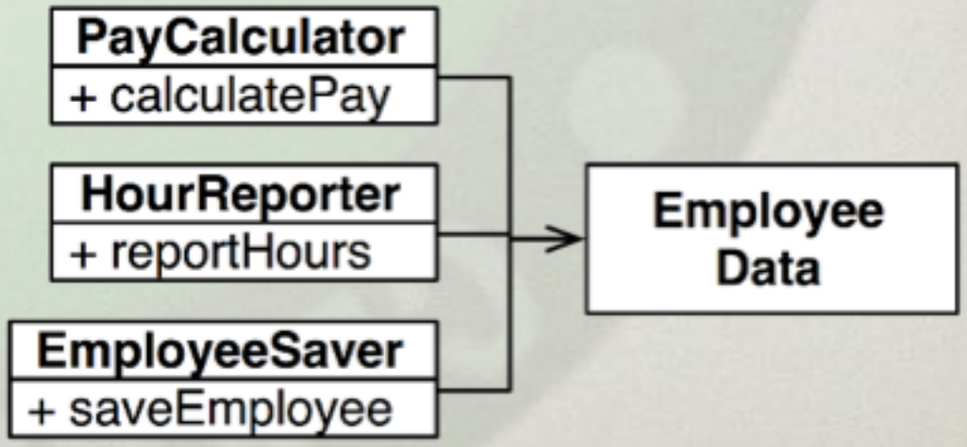
SRP

The Single Responsibility Principle

SRP

- A module should be responsible to one, and only one, actor.
- Separate functions that change for different reasons and at different times.





OCP

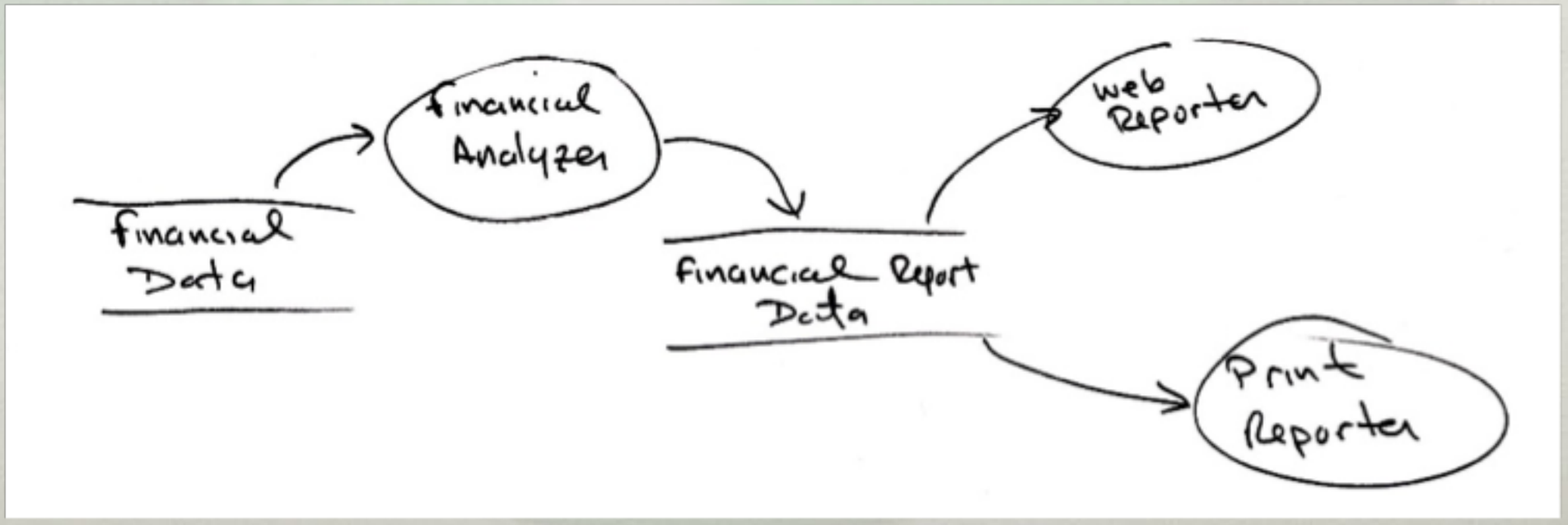
The Open-closed Principle

OCP

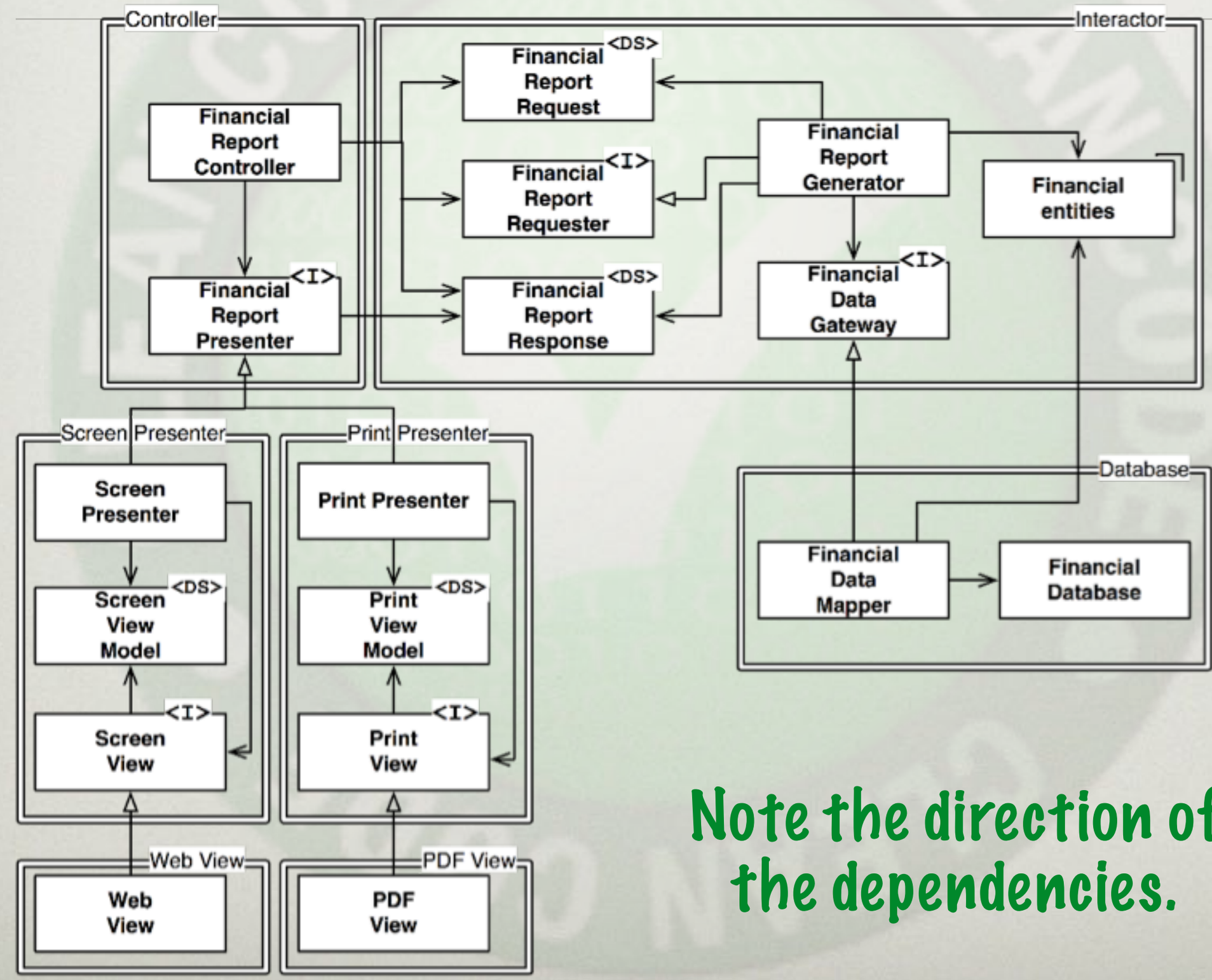
- Coined in 1988 by Bertrand Meyer
- A software artifact should be open for extension but closed for modification.

A Thought Experiment

- A financial system that presents a report on a website.
- New feature: print that report.
- Solution: Separate report data from report format.



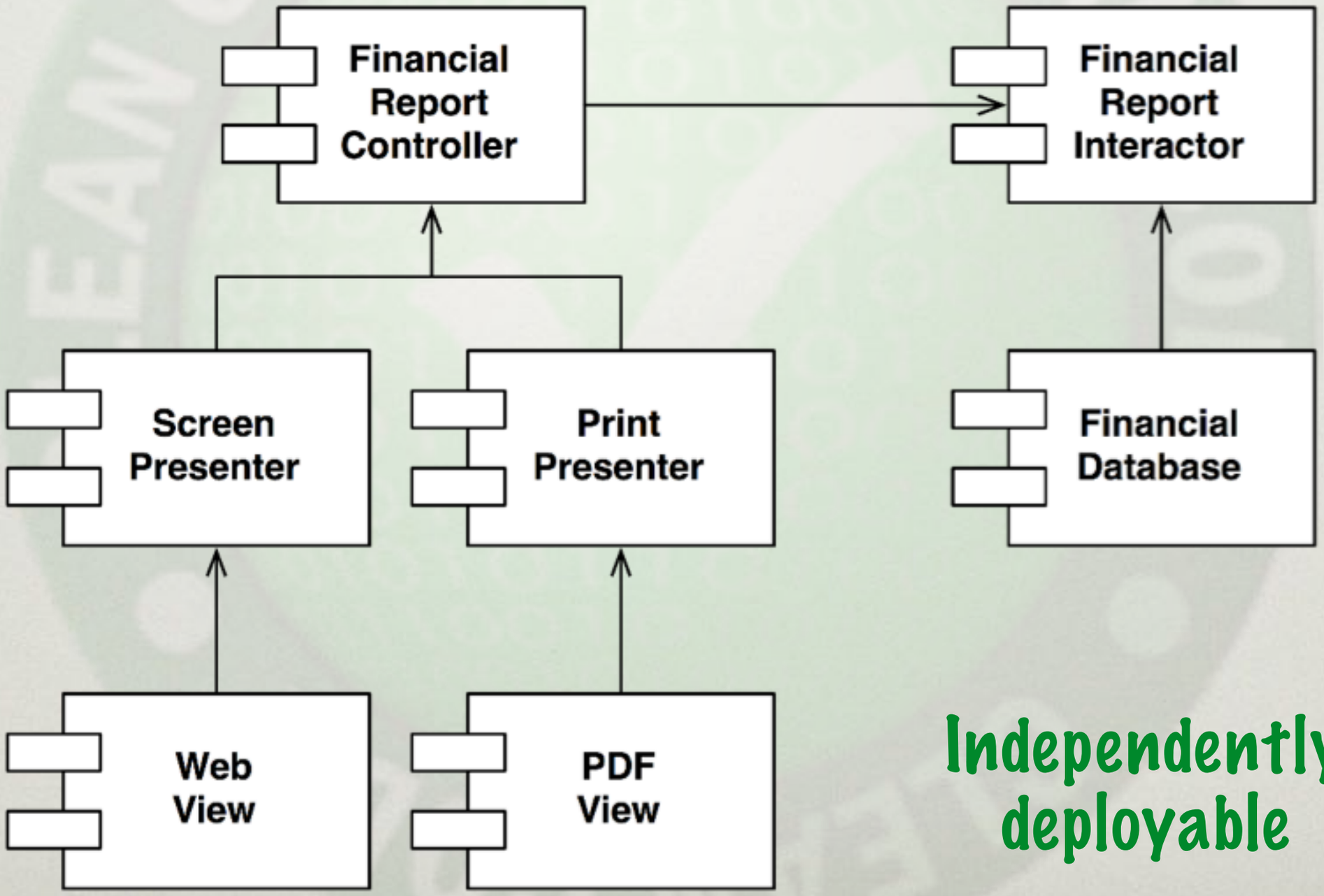
Design of Financial Report system.



Architectural Boundaries

Note the direction of the dependencies.

Component Breakdown



Independently deployable

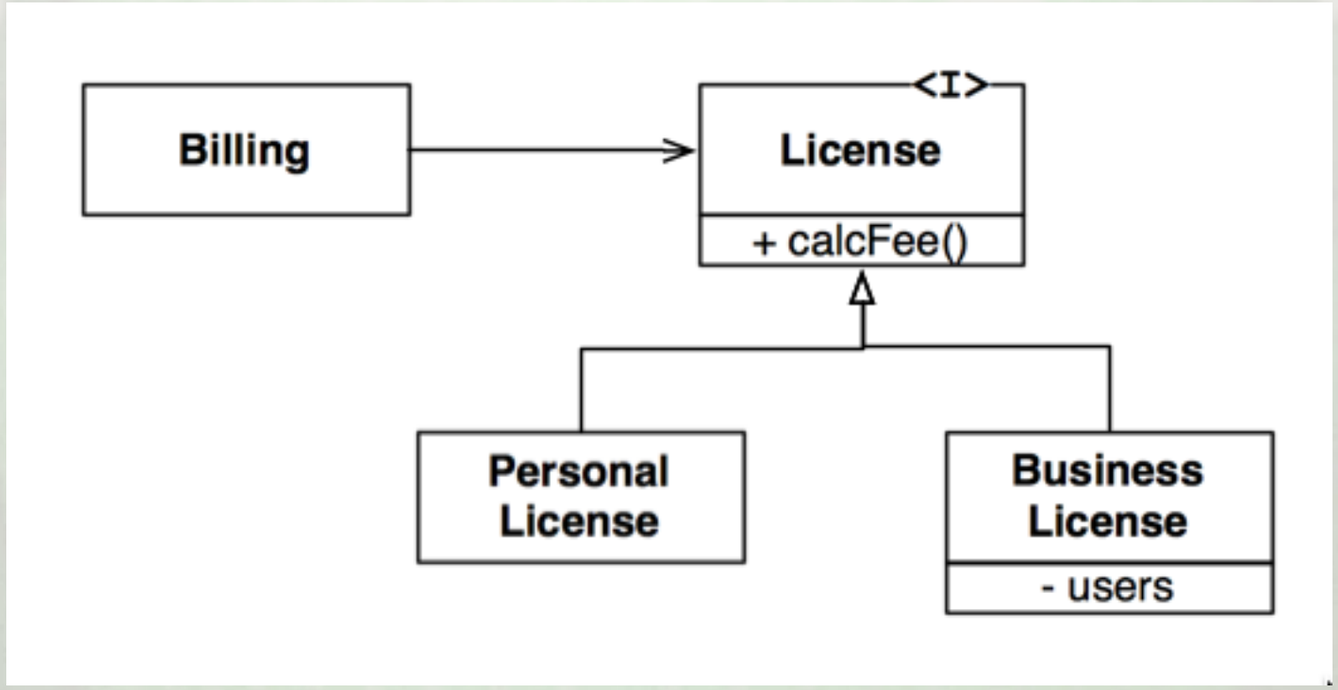
LSP

The Liskov Substitution Principle

Barbara Liskov — 1988

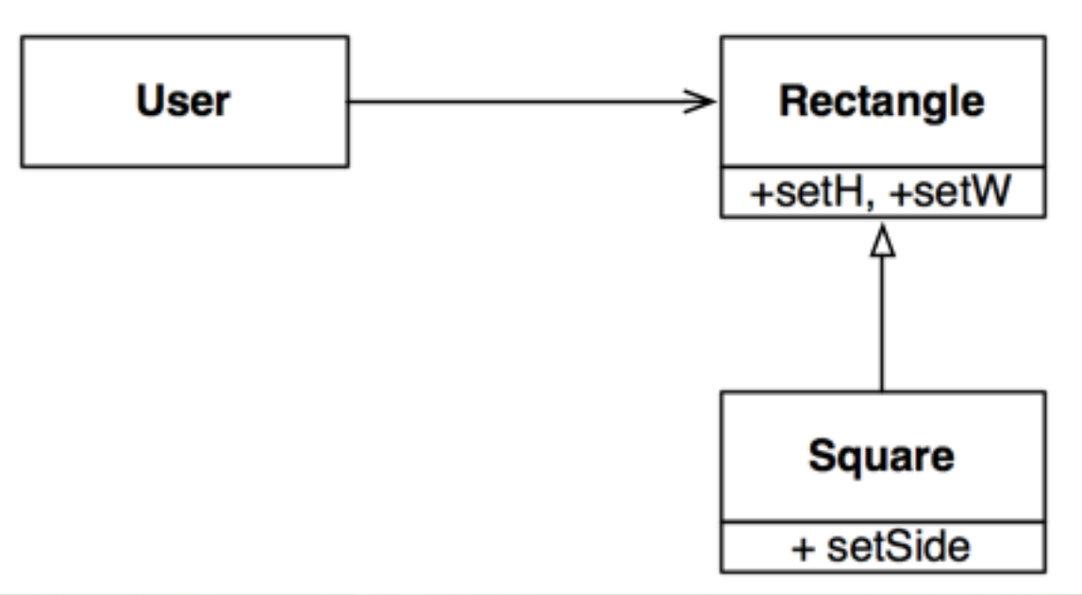
What is wanted here is something like the following substitution property: If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T.

Guiding the use of Inheritance



Substitutable

The Square/Rectangle problem



Not Substitutable

Rest Example
Taxi Dispatch Integrator

purplecab.com
/driver/Bob
/pickupAddress/24 Maple St.
/pickupTime/153
/destination/ORD

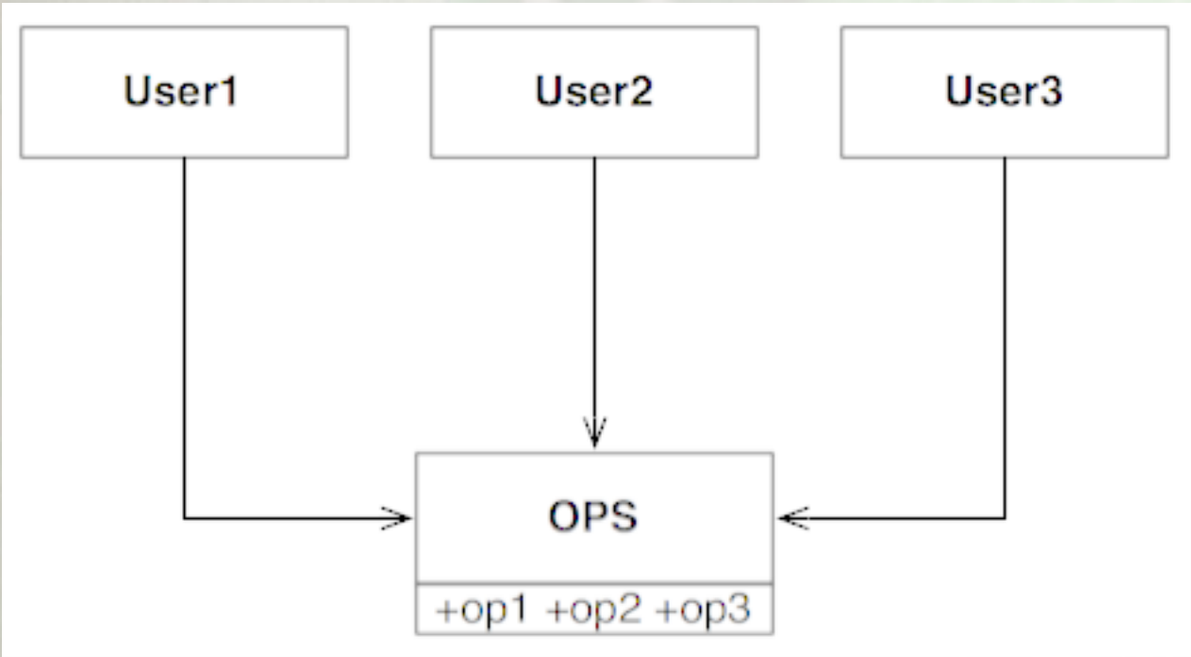
acmecab.com
/driver/Bob
/pickupAddress/24 Maple St.
/pickupTime/153
/dest/ORD

Not Substitutable

ISP

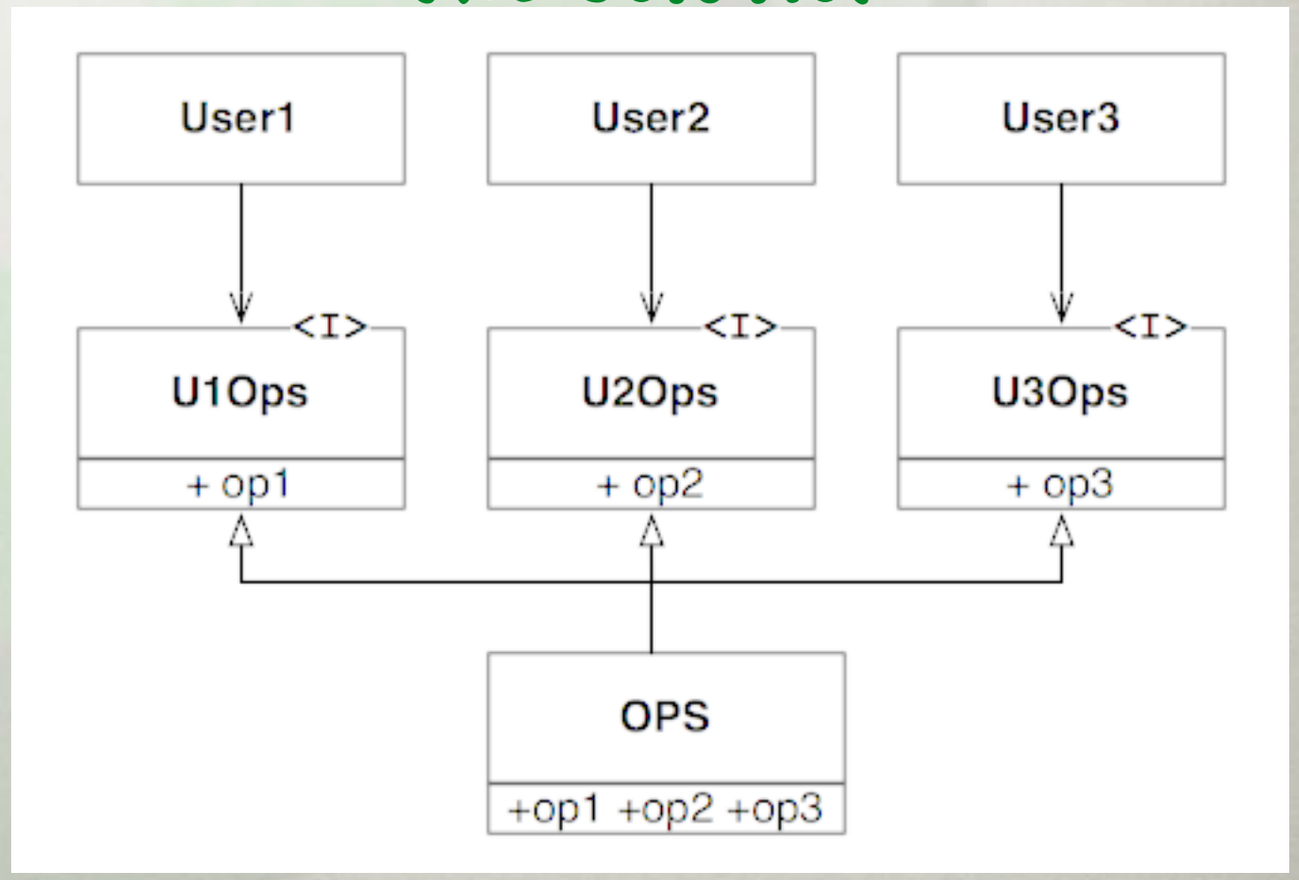
The Interface Segregation Principle

Over-dependence

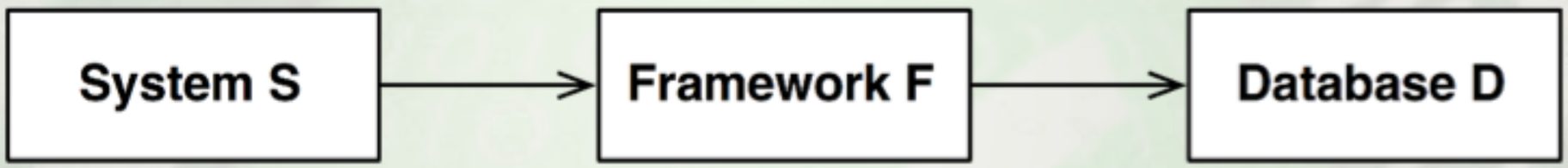


The Problem

The Solution



Don't depend on more than you need.

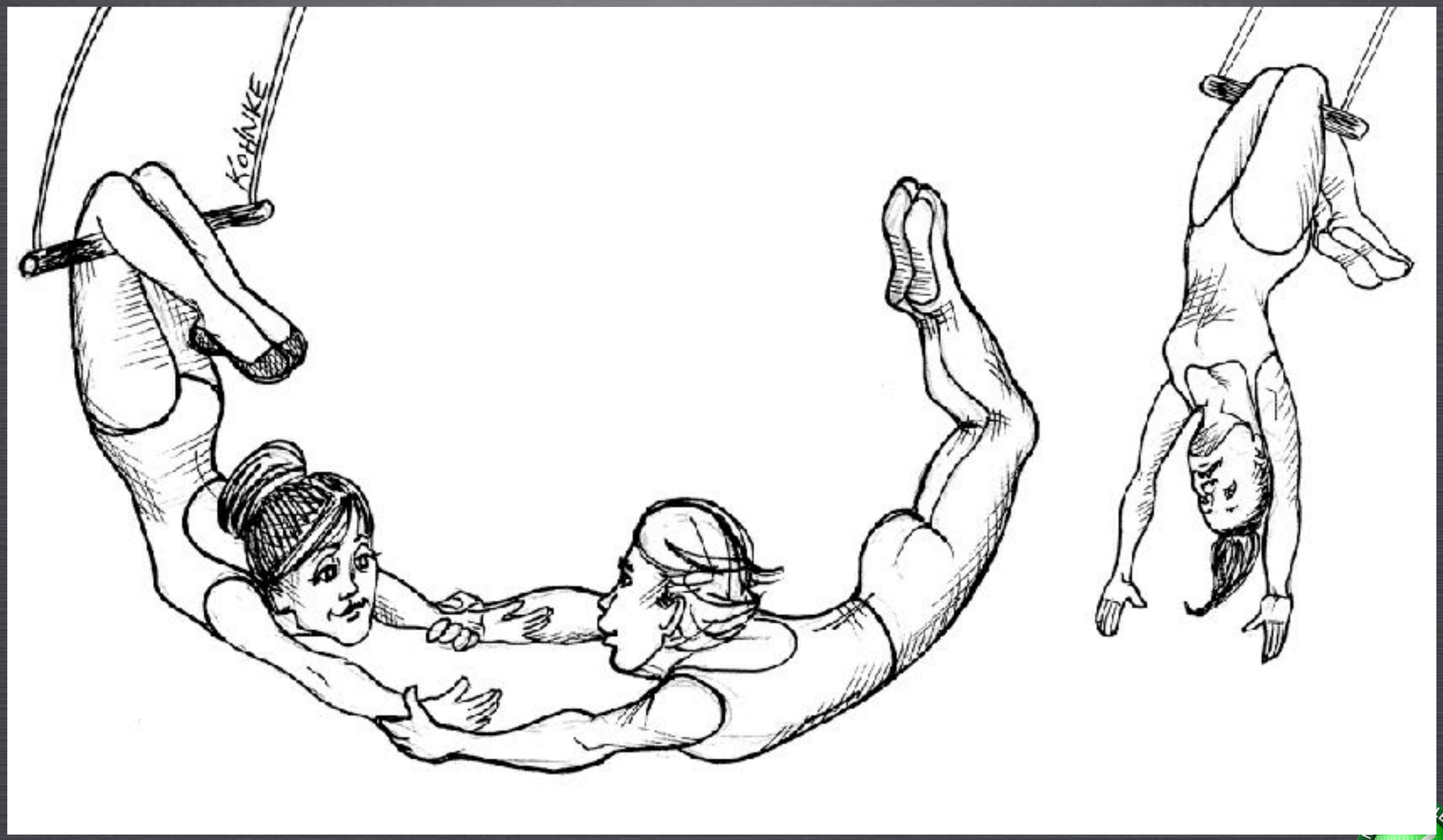


Suppose D contains features that F does not use and therefore S does not care about.

Changes to those features in D may cause failures in both F and D.

DIP

The Dependency Inversion Principle



Depend on Abstractions

- In statically typed languages, depend only on abstract classes or interfaces.
- In dynamically typed language don't depend on concrete classes.
- Unrealistic in every case. **`String`**, for example. But **`string`** is stable.

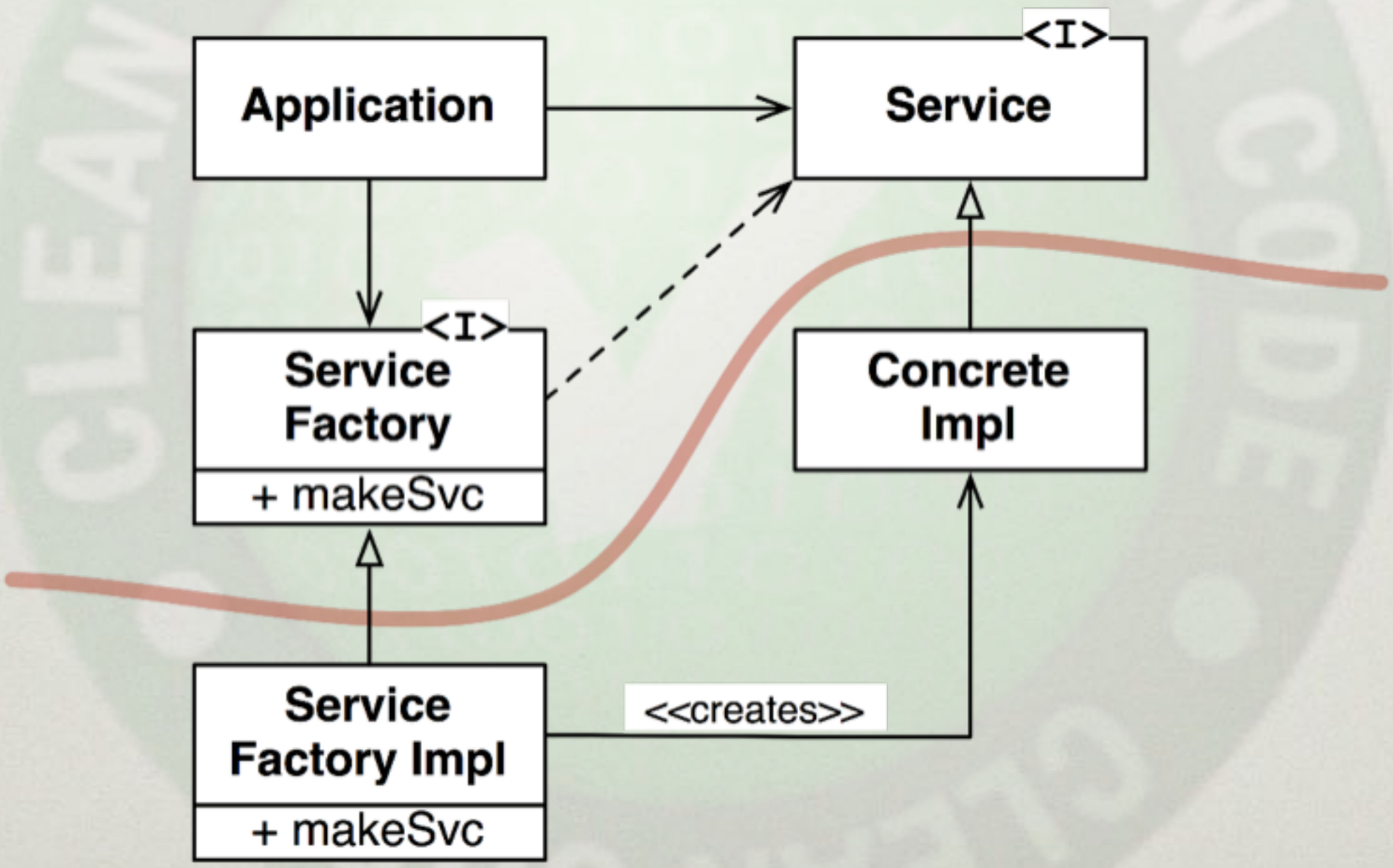
Stable Abstractions

- Every change to an abstract class or interface corresponds to a change to the implementations.
- Changes to implementation do not necessarily mean changes to the interface
- Abstractions should be stable.

Depending on stable abstractions

- Don't refer to, or derive from, volatile concrete modules.
- Don't override concrete functions.
- Don't mention the name of anything concrete and volatile.
- What about the **`new`** keyword?

Abstract Factories



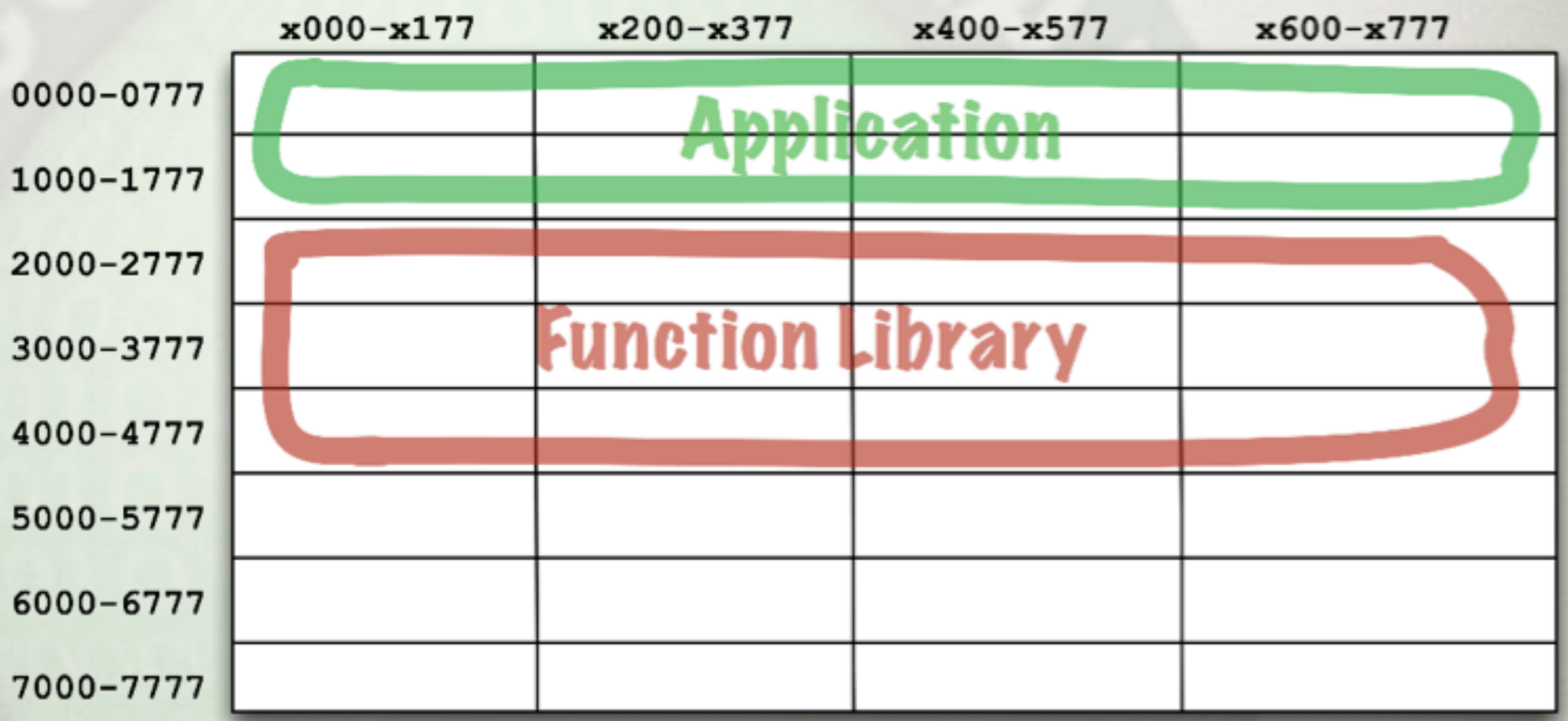
Components

Components are:

- Units of *independent* deployment.
- The smallest entities that can be deployed as part of a system.
- .JAR, .DLL, GEM, etc.
- Components are linked, statically or dynamically, into an executable.

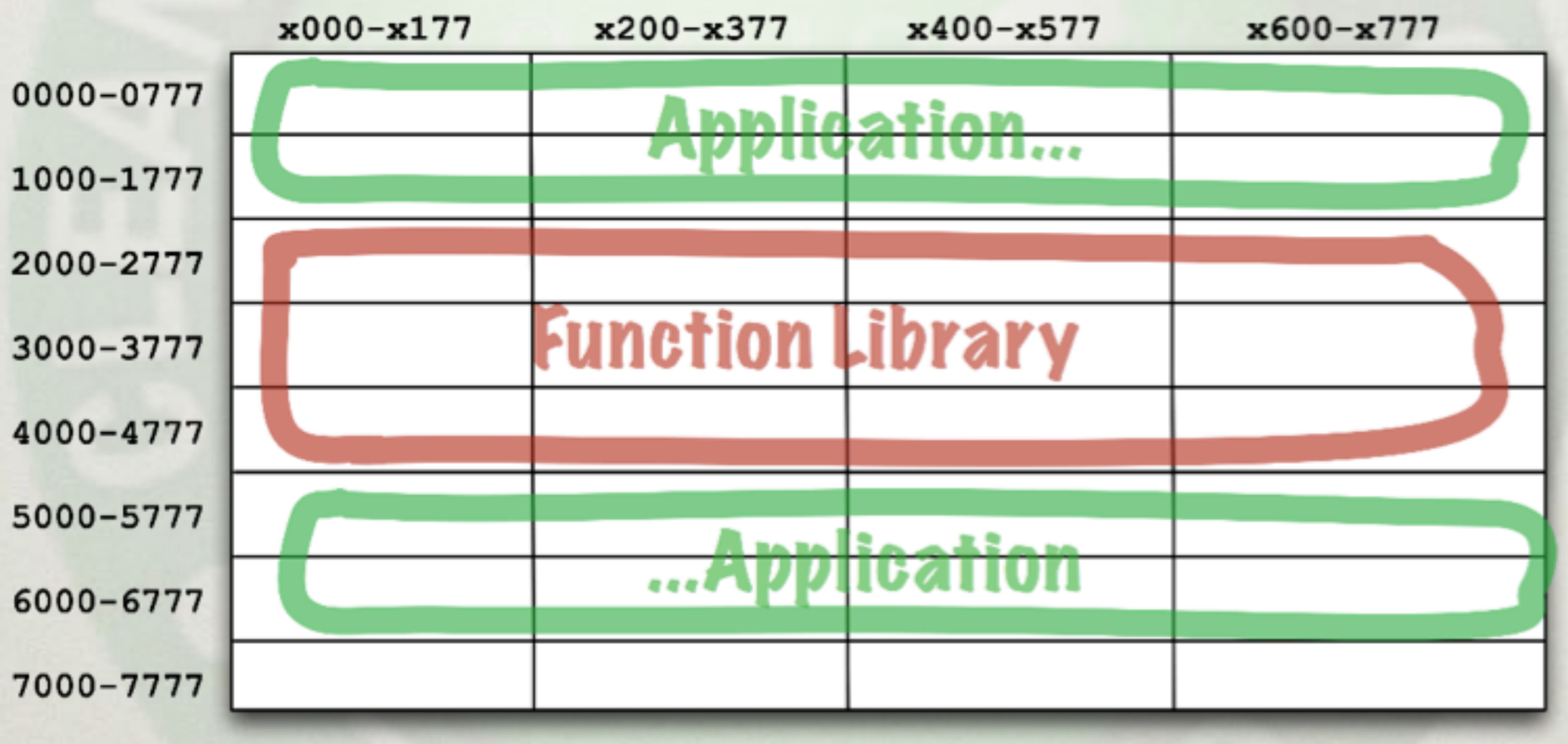
A Brief History of Components

```
          *200

          TLS
START,    CLA
          TAD  BUFR
          JMS  GETSTR
          CLA
          TAD  BUFR
          JMS  PUTSTR
          JMP  START
BUFR,     3000

GETSTR,   0
          DCA  PTR
NXTCH,    KSF
          JMP  .-1
          KRB
          DCA I PTR
          TAD I PTR
          AND  K177
          ISZ  PTR
          TAD  MCR
          SZA
          JMP  NXTCH
K177,     177
MCR,      -15
```



Absolute Addresses

As the applications grew...



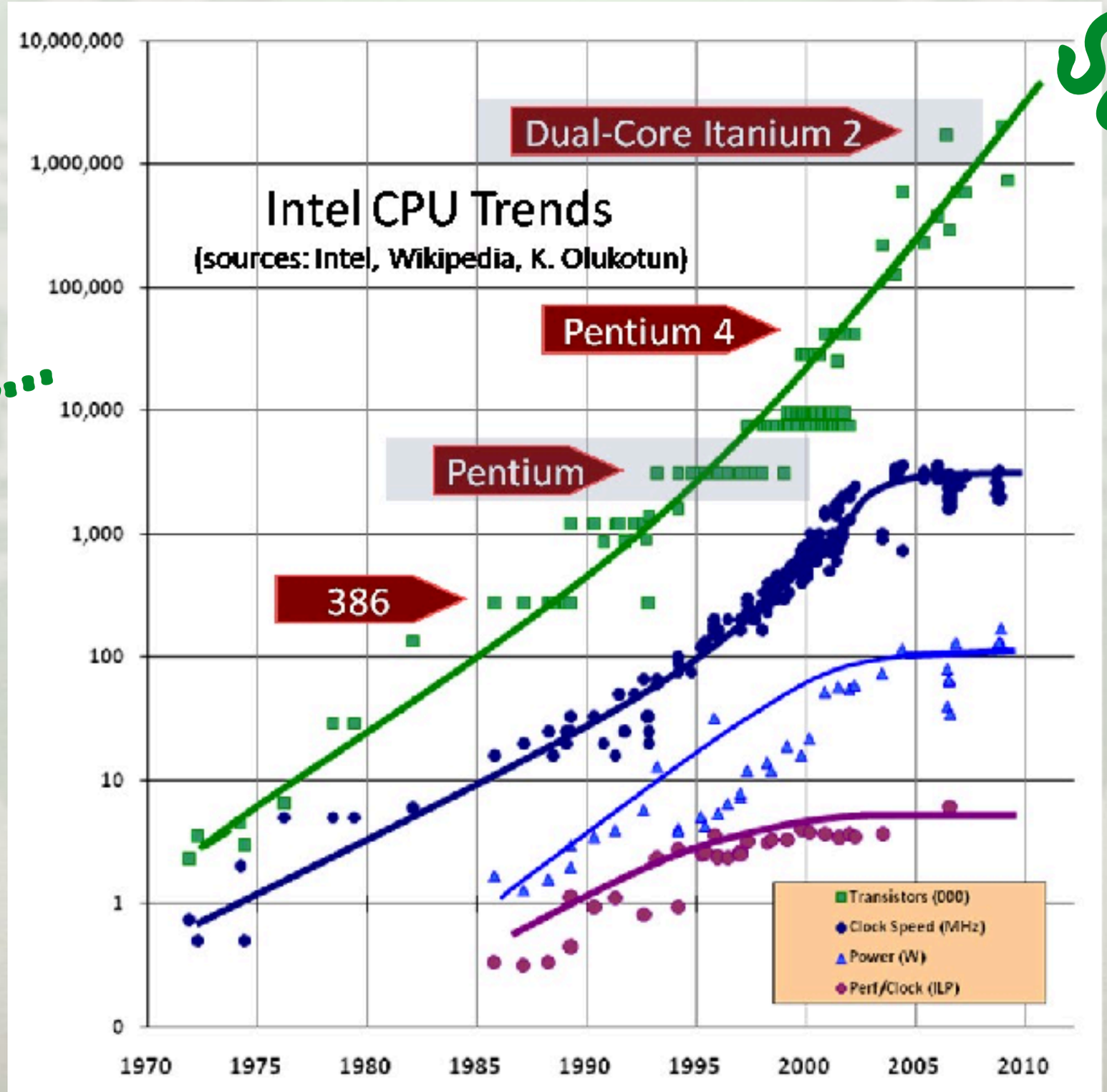
Murphy's law:
Programs grow to fill available space and time.

Interim Solutions to battle Murphy:

- Relocatability
- Linking loaders — got slow.
- Linkers. — Better, still slow.

But everything got faster, smaller, cheaper

But then came Moore...



So Quickly!

Disks

MASS RANDOM ACCESS DATA STORAGE UNIT

CYBAN CE PACK
200 MB

WD
WD BLUE
1TB SATA
PC SSD
Solid State Drive

Murphy loses!

- Murphy: A program will grow to fill all available space and time.
- Moore, time and space will double every 18 months.
- Moore won!
- Mid ‘90s: Link time shrank back down to load time.

True Dynamically Loadable Components.

- Active X. DLLs.
- Shared libraries.
- JAR files.
- Dynamic linking rules everywhere.

Component Cohesion

Principles that govern component innards.

- REP, CCP, CRP. The criteria for deciding which classes and modules to group together, and which to keep separate.

REP: Release/Reuse Equivalence.

- Maven et. al. The era of software reuse has truly begun.
- In order to reuse, you must release.
- Releasing is a human, clerical, activity.
- Classes and modules “reused” together.
- Weak principle without support of other two.

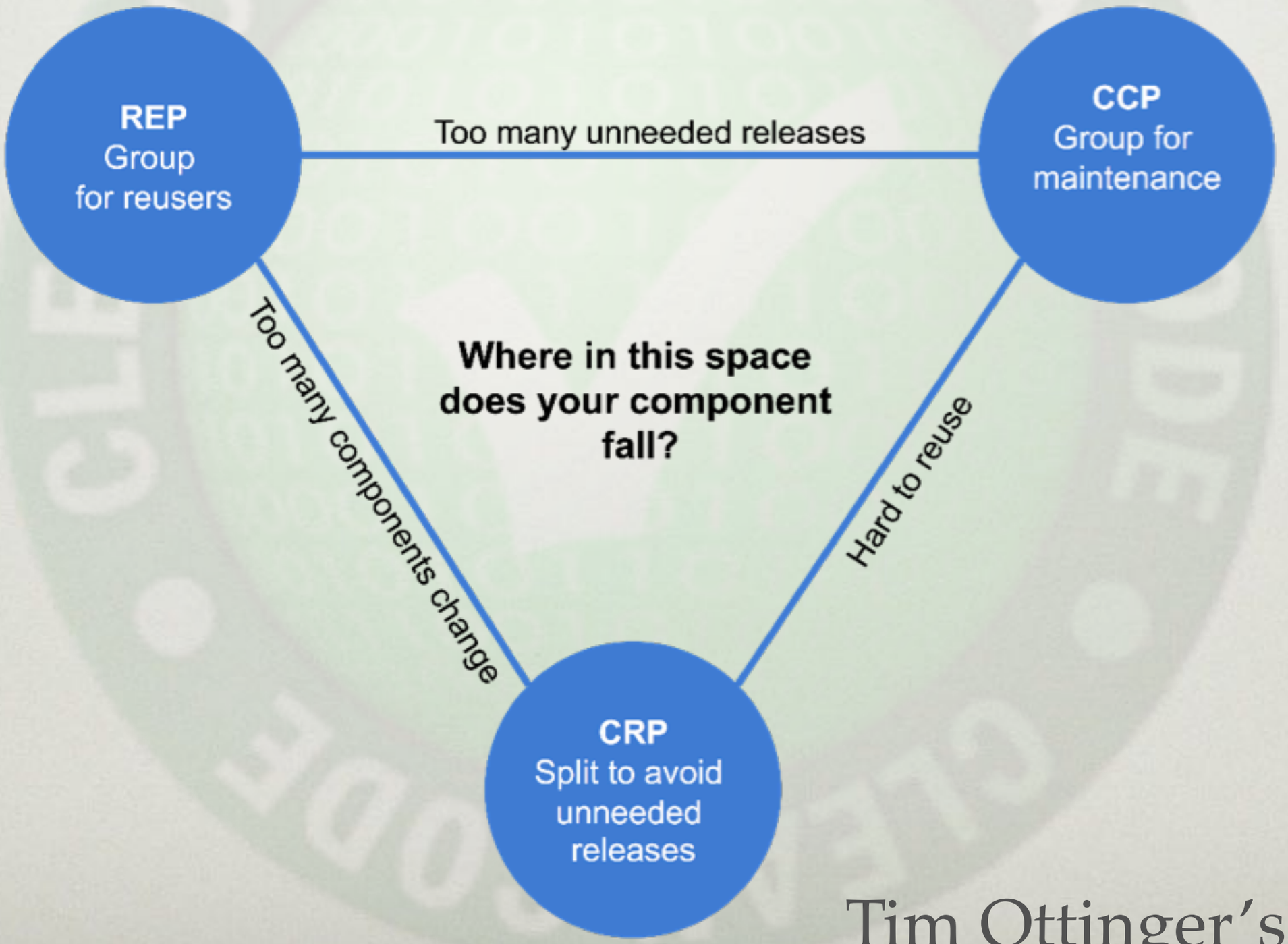
CCP: Common Closure Principle

- Gather together those things that change for the same reasons and at the same time.
- Separate those things that change for different reasons or at different times.
- SRP for components.
- Closure is in the sense of OCP.

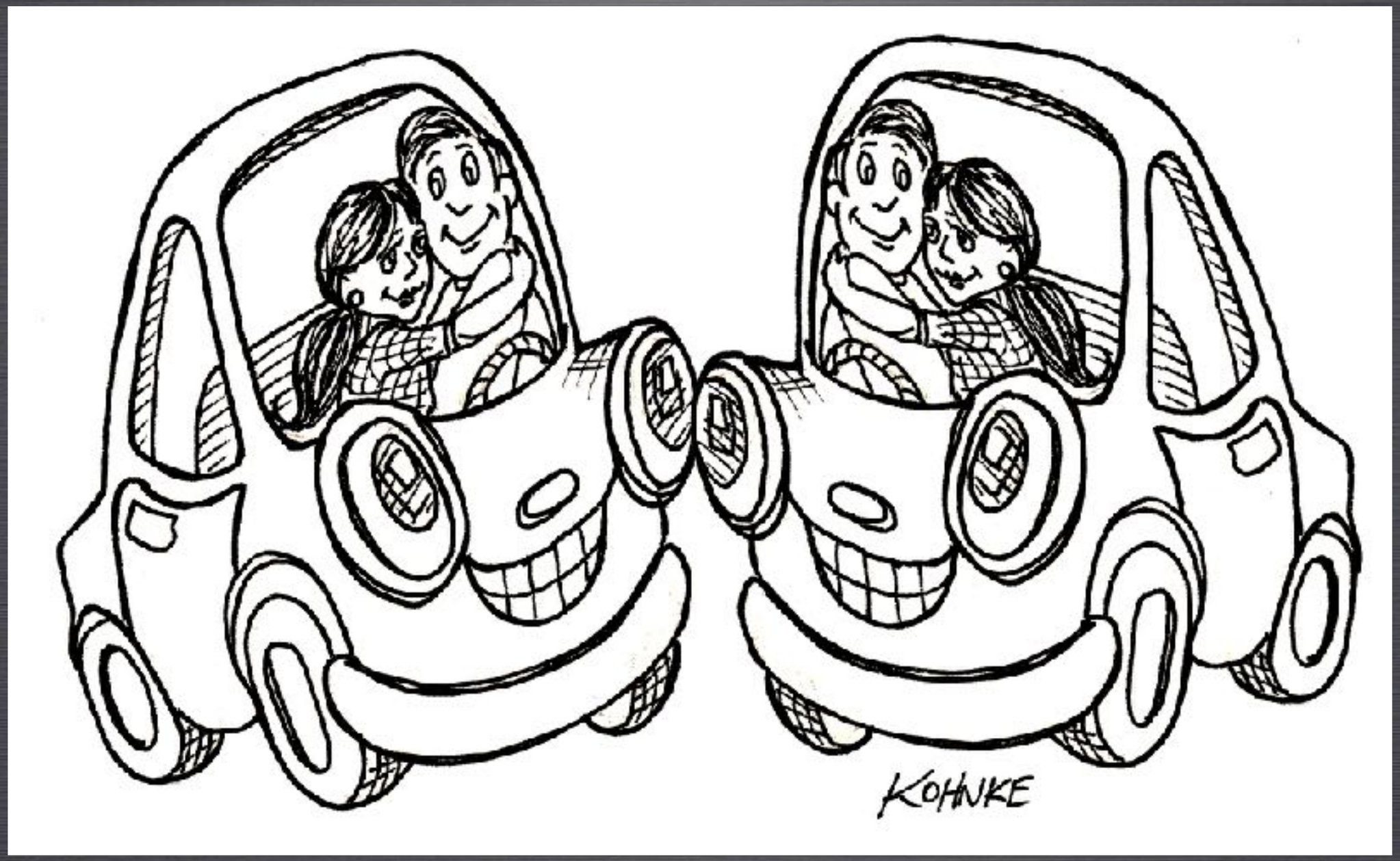
CRP: Common Reuse Principle

- Don't force the users of a component to depend on things they don't need.
- Negative cohesion. CRP tells us what *not* to include in a component.
- Related to ISP.

Cohesion Principle Tension



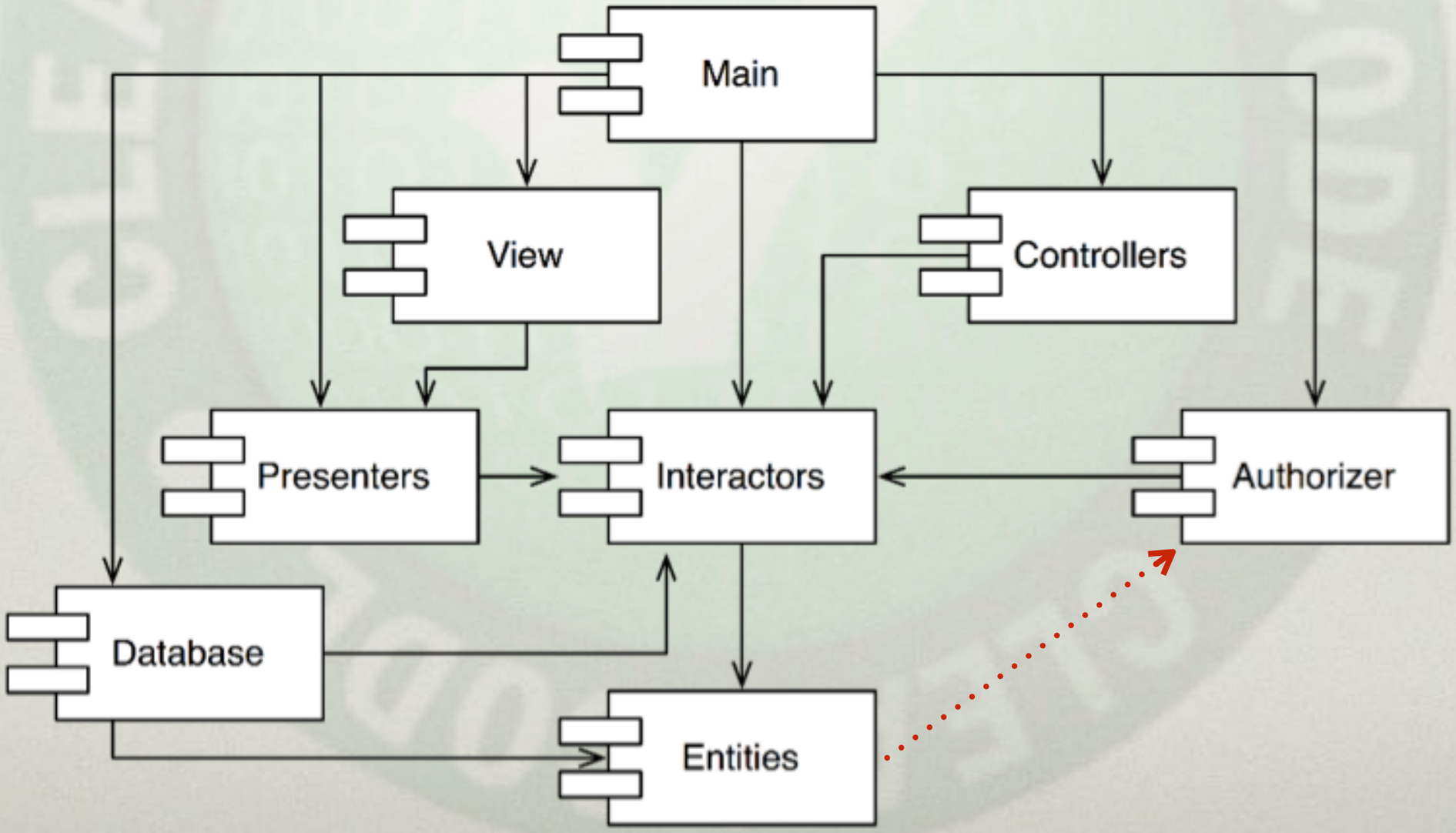
Tim Ottinger's idea.



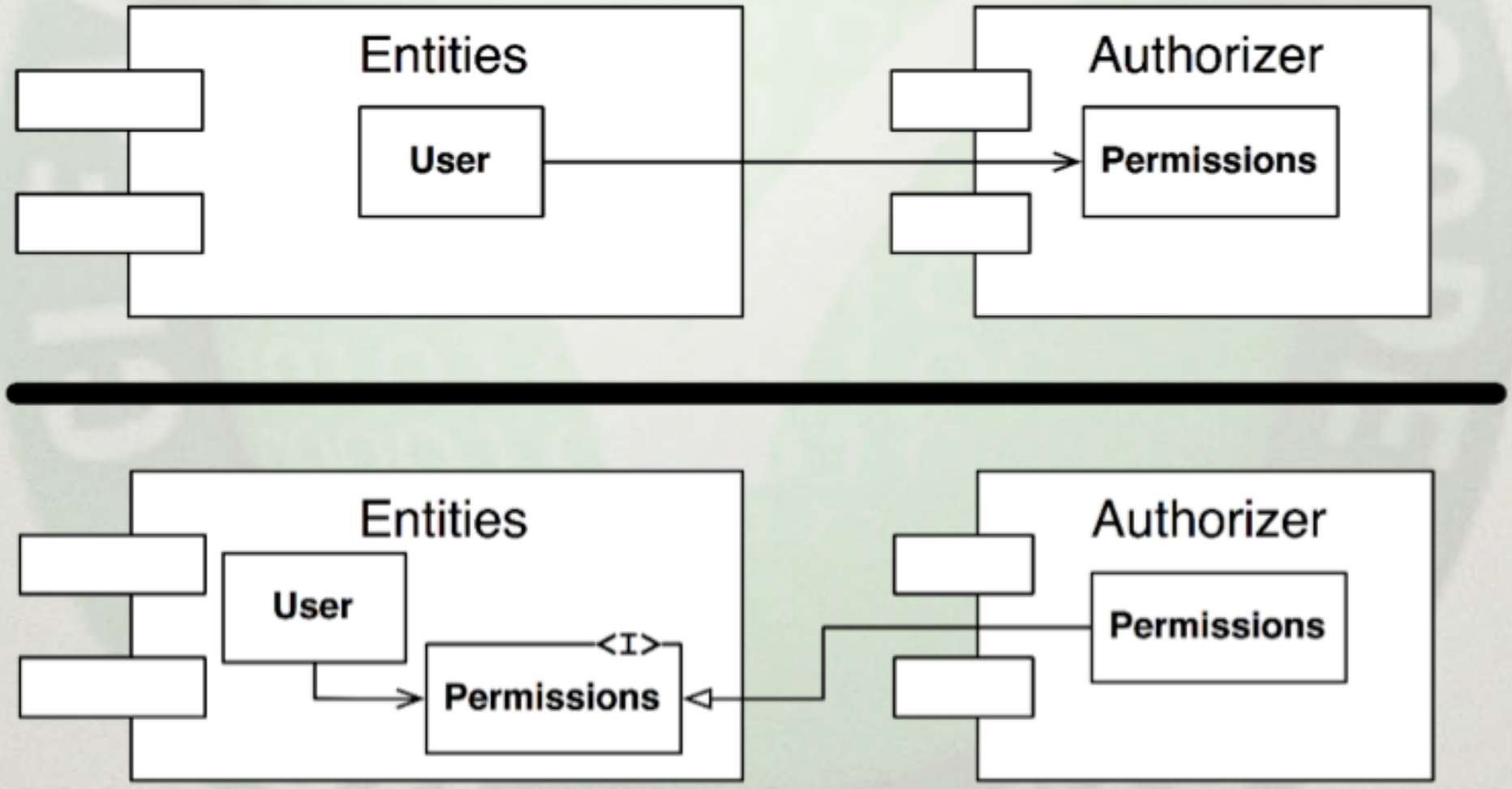
Component Coupling

ADP: Acyclic Dependencies Principle

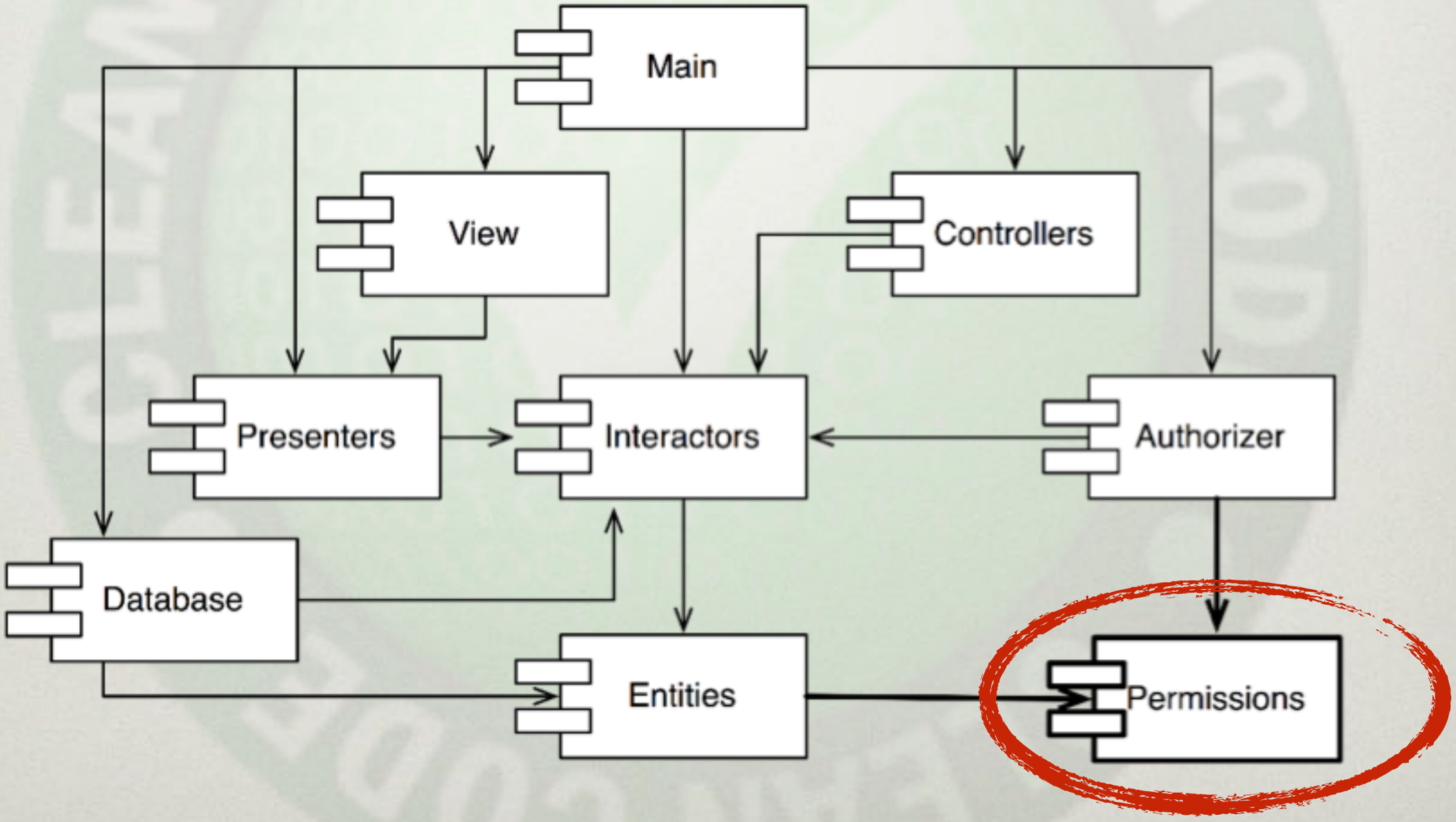
- The Morning After
- The Weekly Build



Use DIP to solve

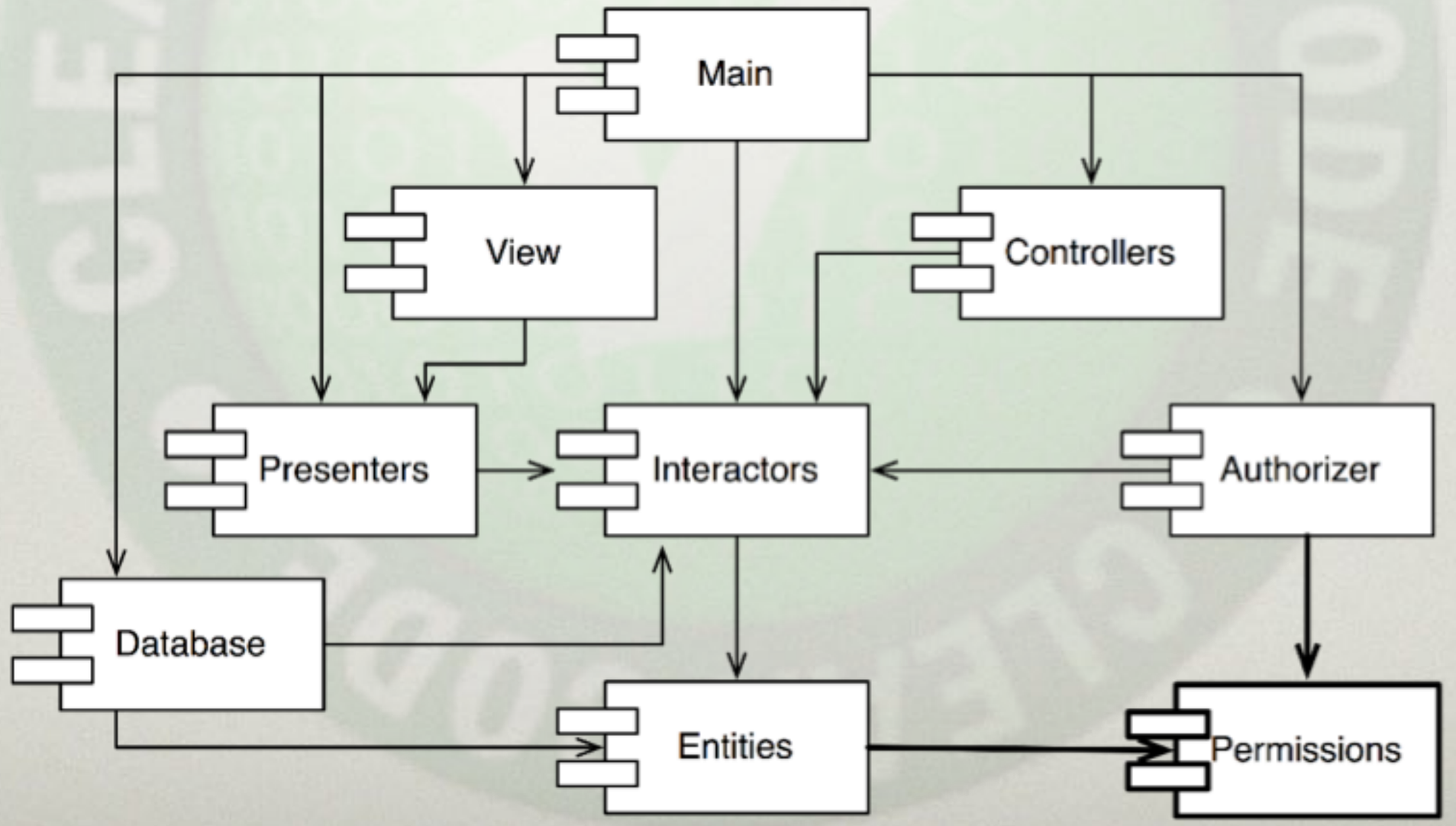


Create new component to solve.



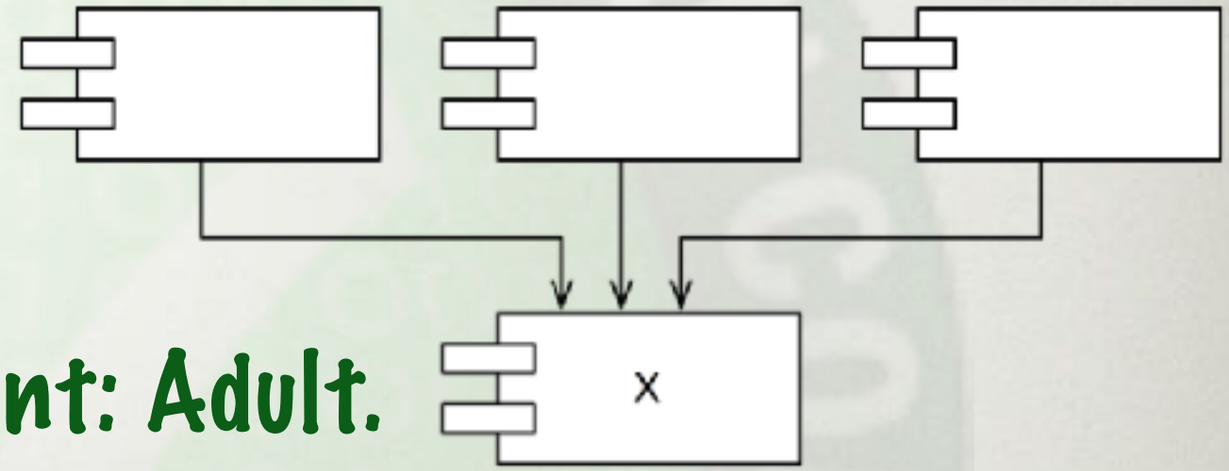
Not Top-Down Design

- The component graph evolves over time.
- It is not entirely the product of an up front design.



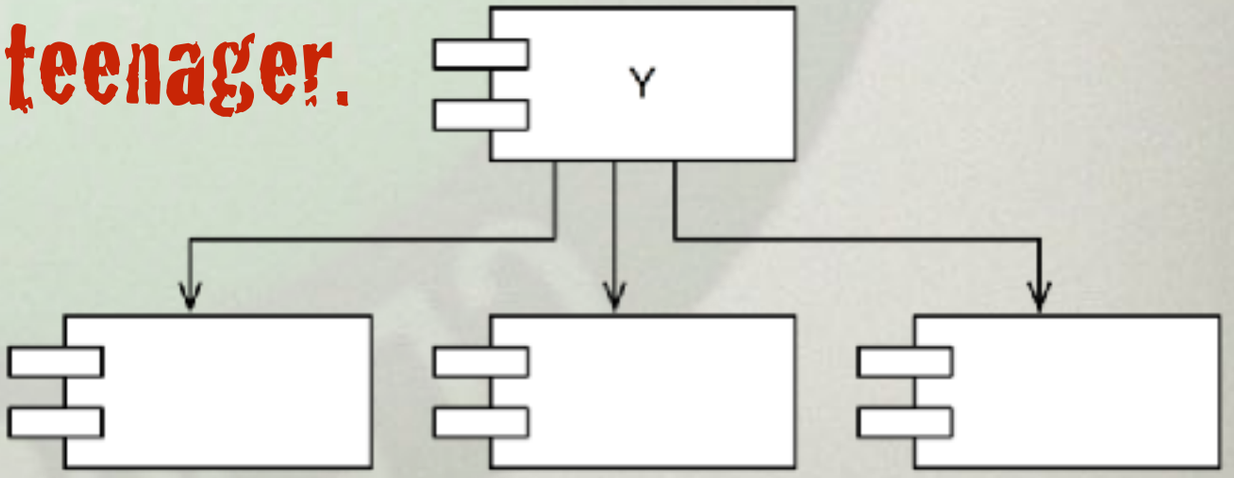
SDP: Stable Dependencies Principle

- What is Stability?

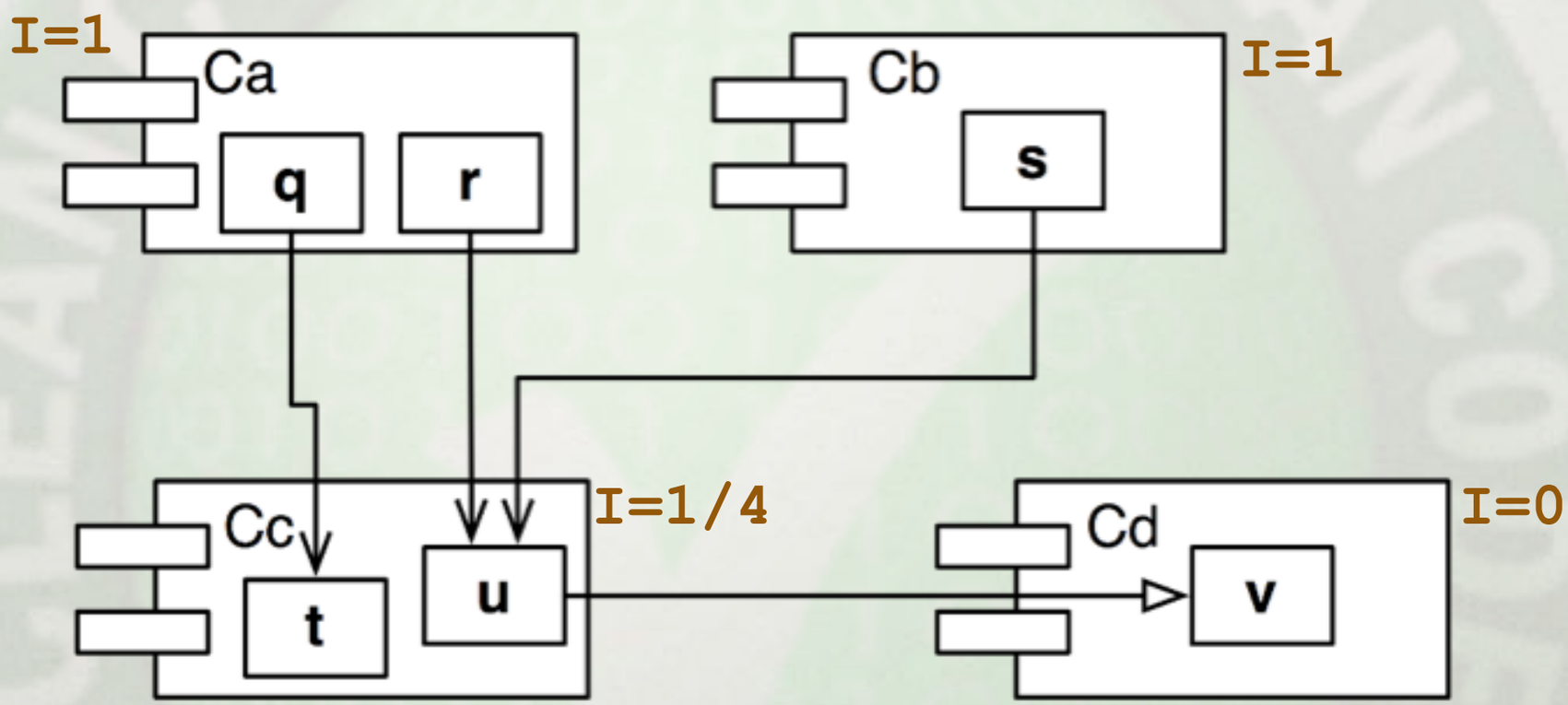


Responsible, Independent: Adult.

Irresponsible, dependent, teenager.



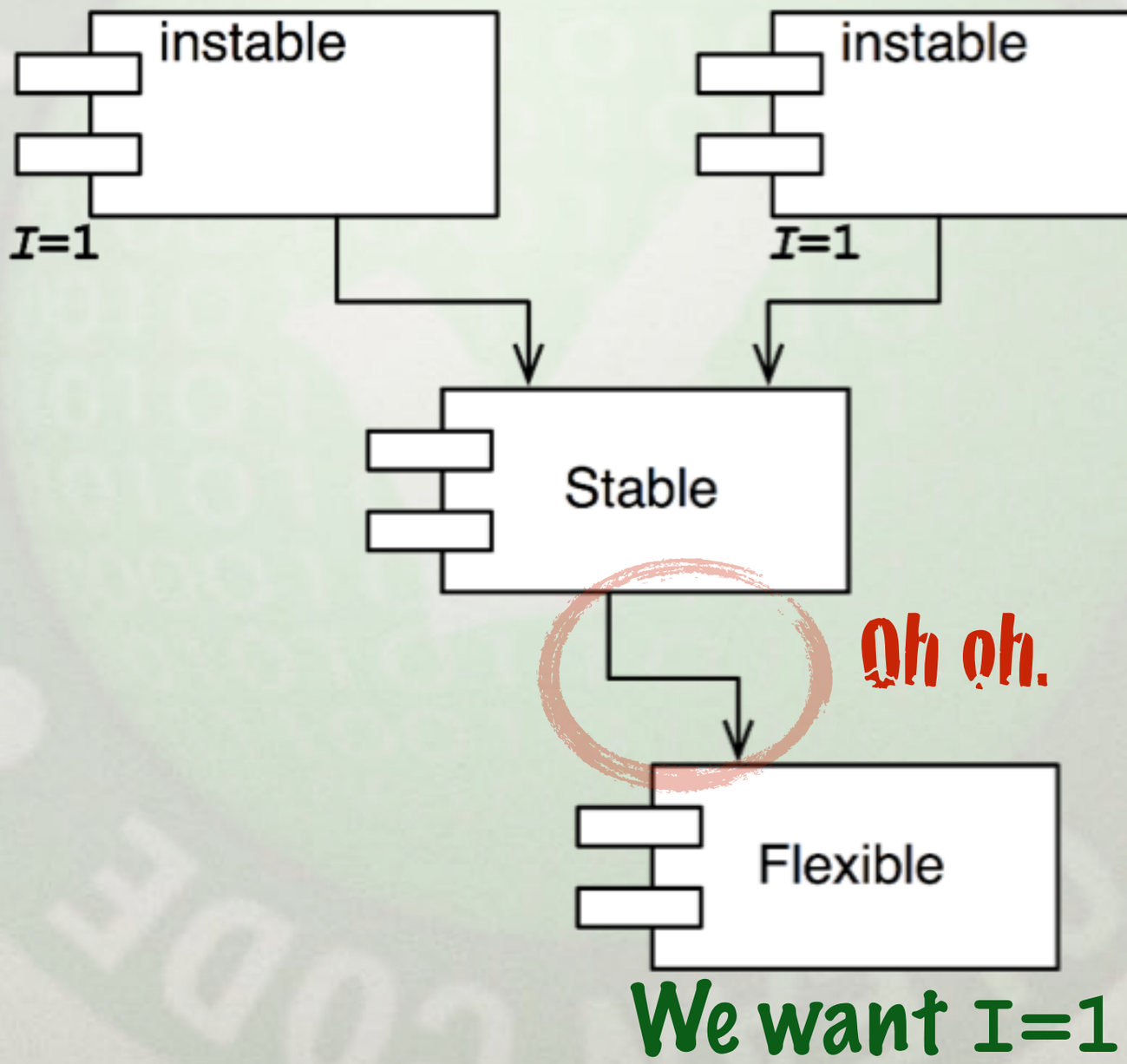
Instability Metric




```
I = FanOut / (FanOut + FanIn)
[0,1]
```

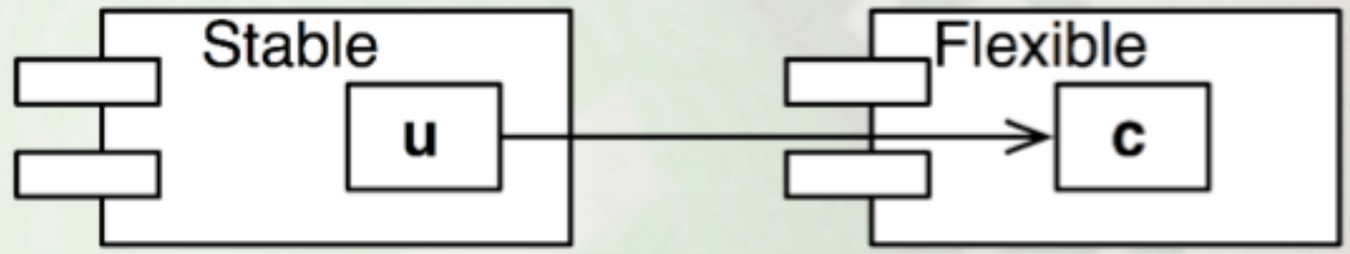
SDP: Depend in the direction of *decreasing* `I`.

An Example of an SDP violation

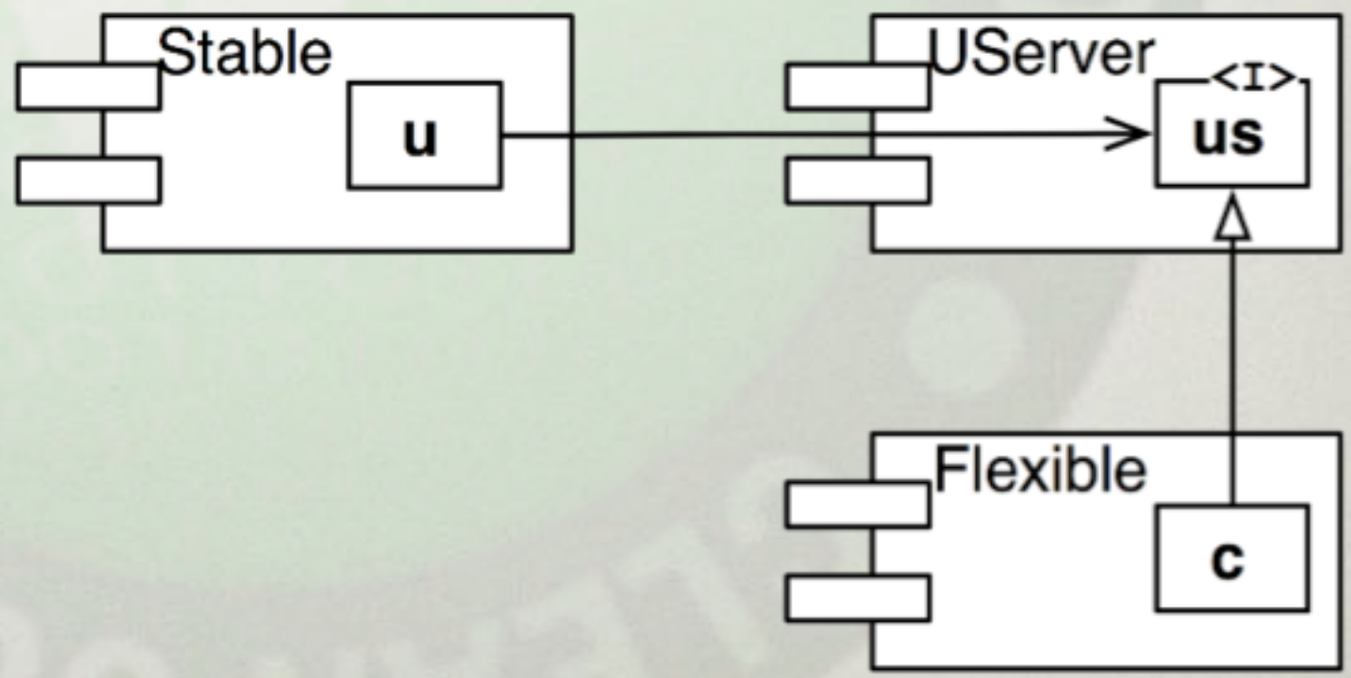


DIP solution

Given:



DIP to solve:



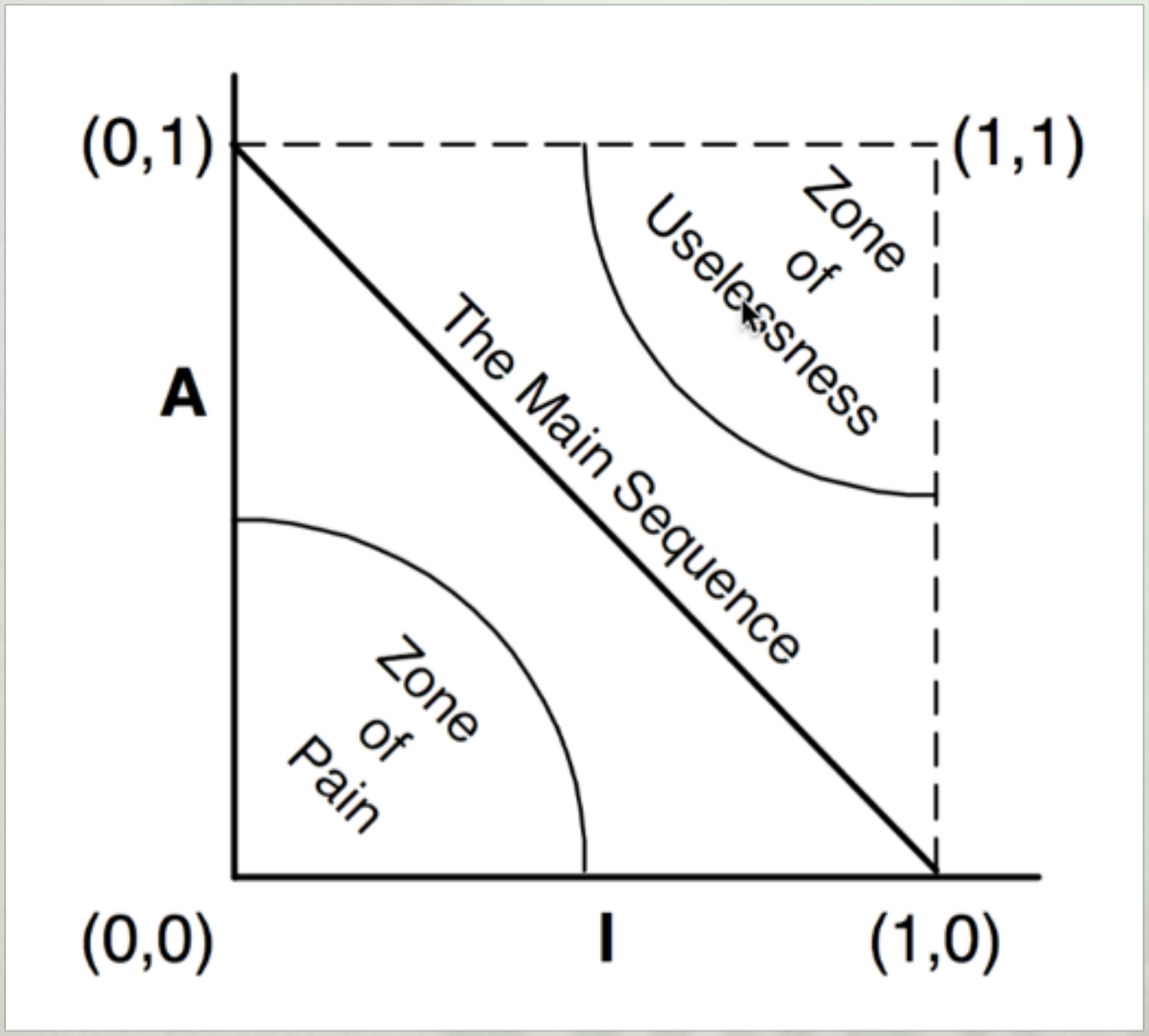
SAP: Stable Abstractions Principle

- We put high level policy into stable components. (DIP).
- That makes high level policy inflexible.
- Unless we invoke the OCP.
- Abstract components are open for extension!

Measuring Abstractness

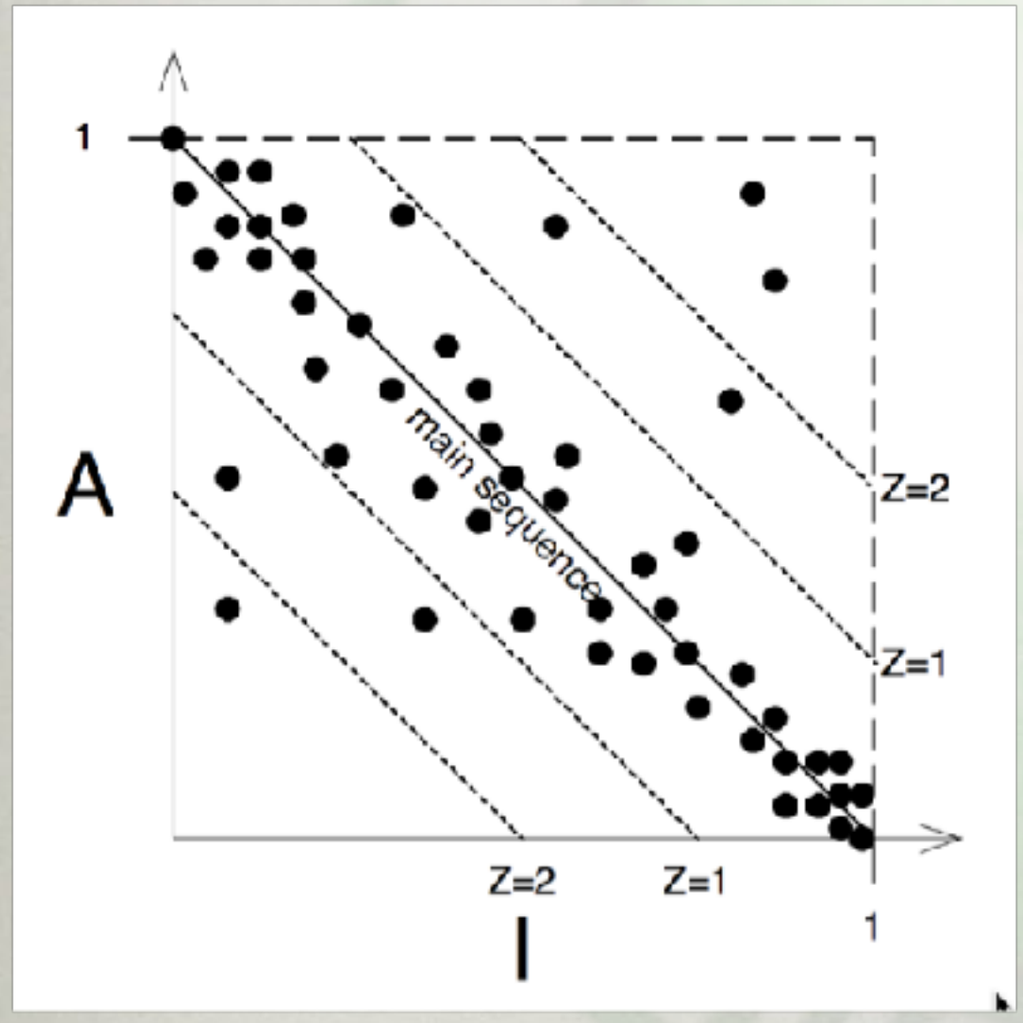
- Nc: Number of classes in component.
- Na: Number that are abstract.
- A = Na/Nc : [0,1]
- SAP: Depend in the direction of *increasing A.*

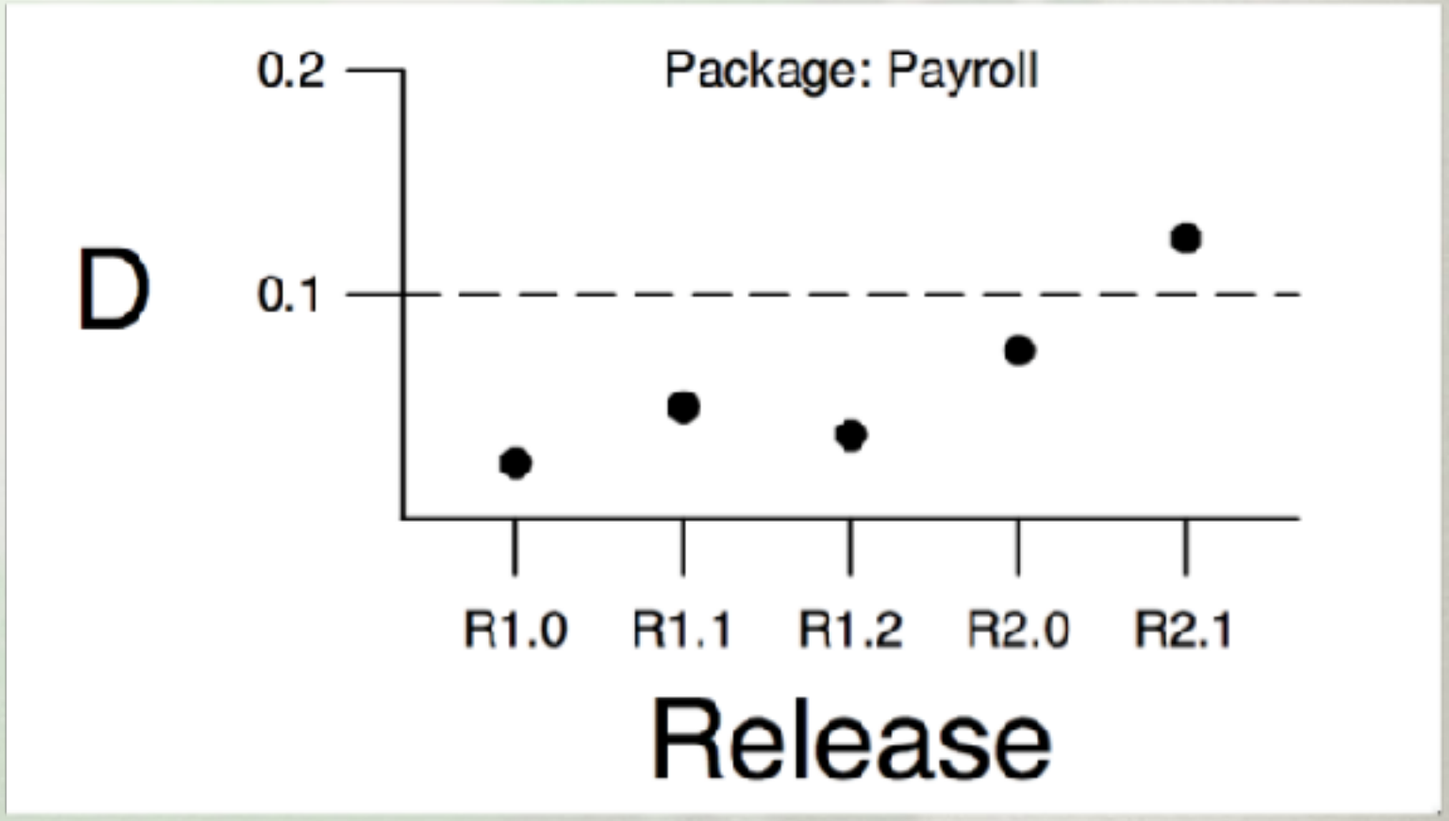
The Main Sequence

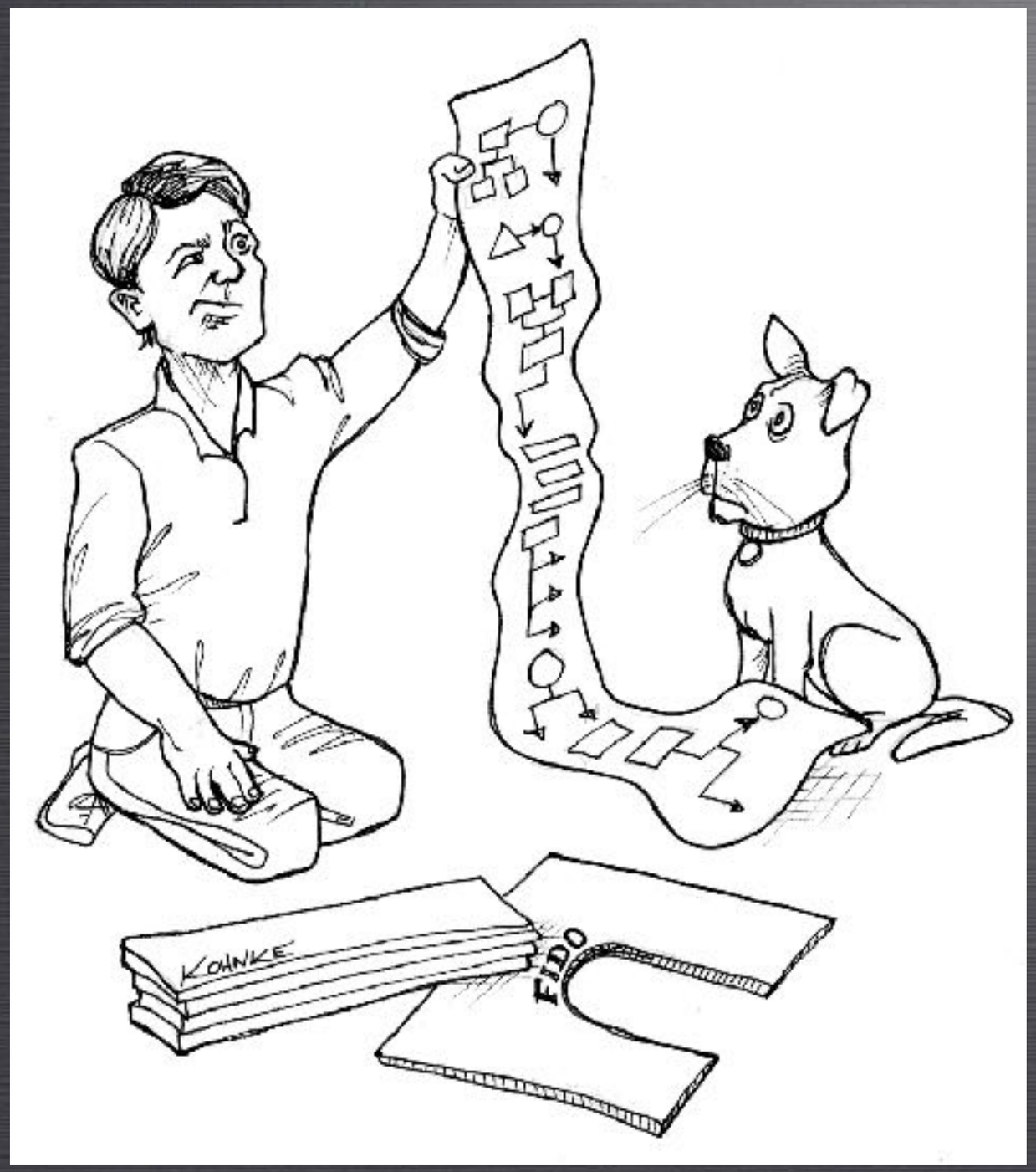


$$D=|A+I-1|$$
$$[0,1]$$

Uses of D


1
A
main sequence
Z=2
Z=1
Z=2
Z=1
1
I


Package: Payroll
0.2
D
0.1
R1.0
R1.1
R1.2
R2.0
R2.1
Release



What is Architecture?

Architect

- Conjures visions of power, mystery, weighty decisions and deep technical prowess.
- First of all, an architect is a programmer; and continues to write code so that they experience, first hand, the consequences of their decisions.

Architecture

- The shape given to a system: the arrangement and connections between its components.
- Purpose: Facilitate the development, deployment, operation, and maintenance of that system.
- Strategy: Leave as many options open as possible.

Keeping Options Open

- Systems can be decomposed into high level policy and low level detail.
- The options you leave open are the low level details. e.g. IO, Web, DB, Servers, etc.
- Create the shape that recognizes high level policy as the most essential elements.
- While making details irrelevant to that policy.
- Allow decisions about details to be delayed and deferred.

For Example

- In the early days of development it is not necessary to choose:
  - A database.
  - A web server.
  - REST
  - DI
  - GUI framework.
  - etc.

The longer you leave options open, the more information you will have to help make informed decisions about them later.

More examples

- Device Independence.
- Junk Mail -DSpool.
- Physical disk addresses: 705

Independence

Goal

- Independent developability.
- Independent deployability.

Separation of Concerns

- Decouple layers: UI from DB.
- Decouple use cases.
- Decouple frameworks.
- But don't commit to a decoupling mode: e.g. services μ or otherwise.
  - Keep that option open.

Decoupling Modes

- Source level. (Gems, Monolith)
  - One address space, function calls.
- Deployment level (DLL, Jar)
  - Multiple address spaces, single machine, function calls, IPC.
- Service level
  - Multiple address spaces, Sockets

Decoupling mode evolves

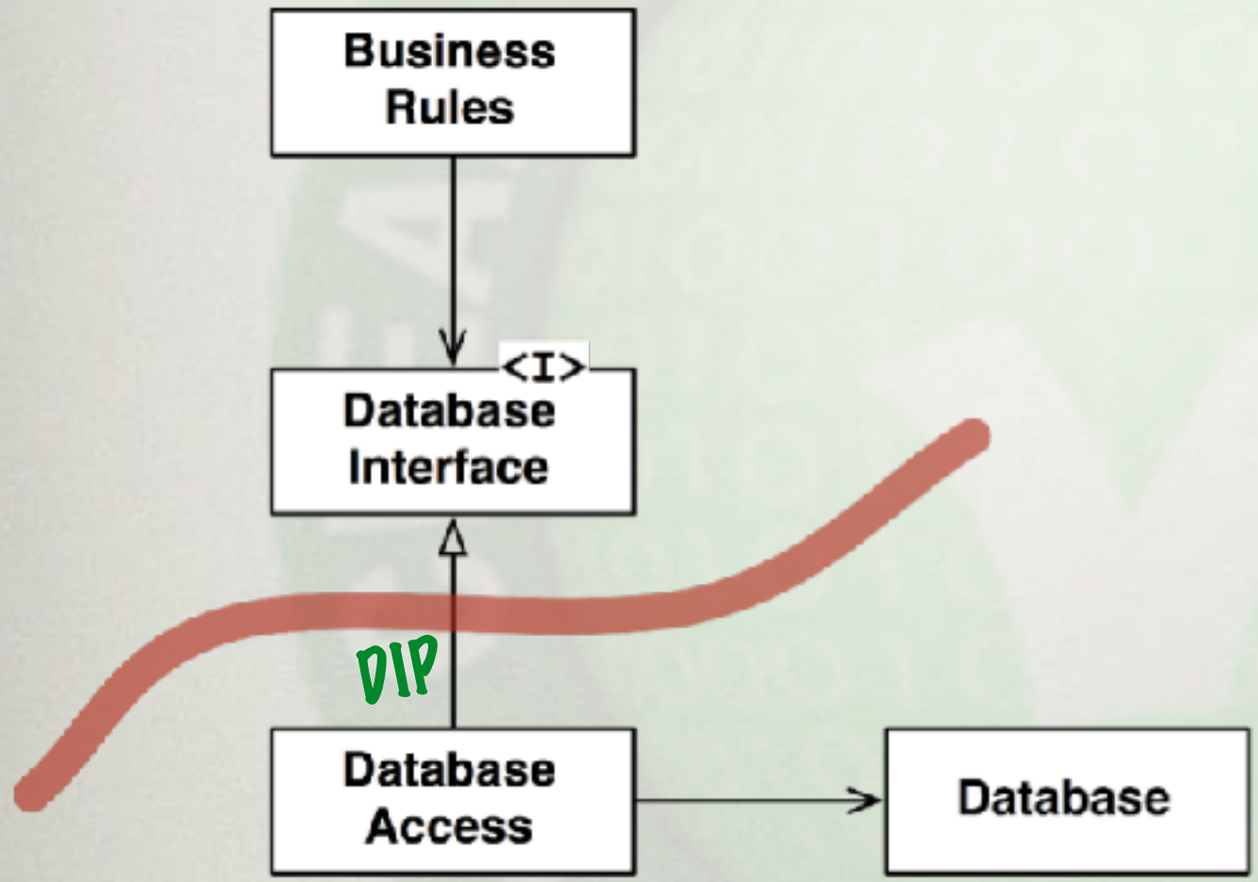
- A system may start as a monolith
- Grow to become a set of independently deployed jars.
- Shift to μServices
- Slide back to Jars
- and even back to a monolith.
- Architecture should not care.

Boundaries

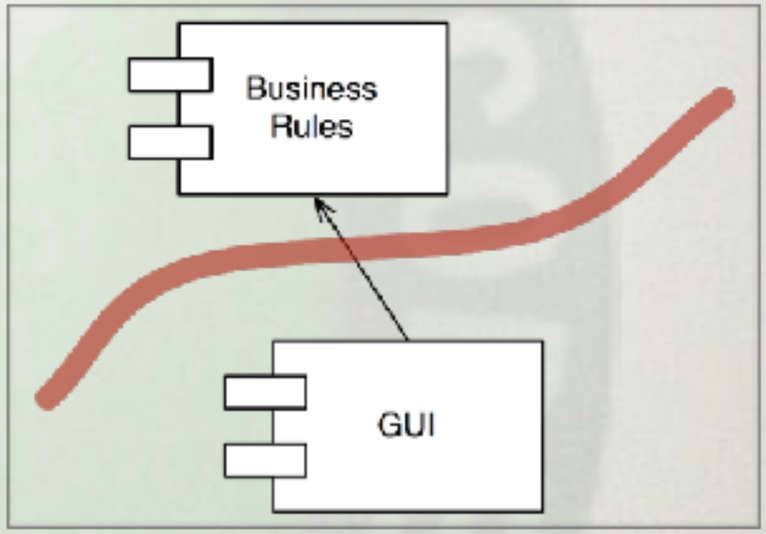
The Art of Drawing Lines

- Boundaries that separate high level policy decisions that have to be made now, from low level details that can be deferred.
- Sad Story of P (Layers gone amok) , and W (services gone amok).
- FitNesse.

Which lines to draw? Where to draw them?

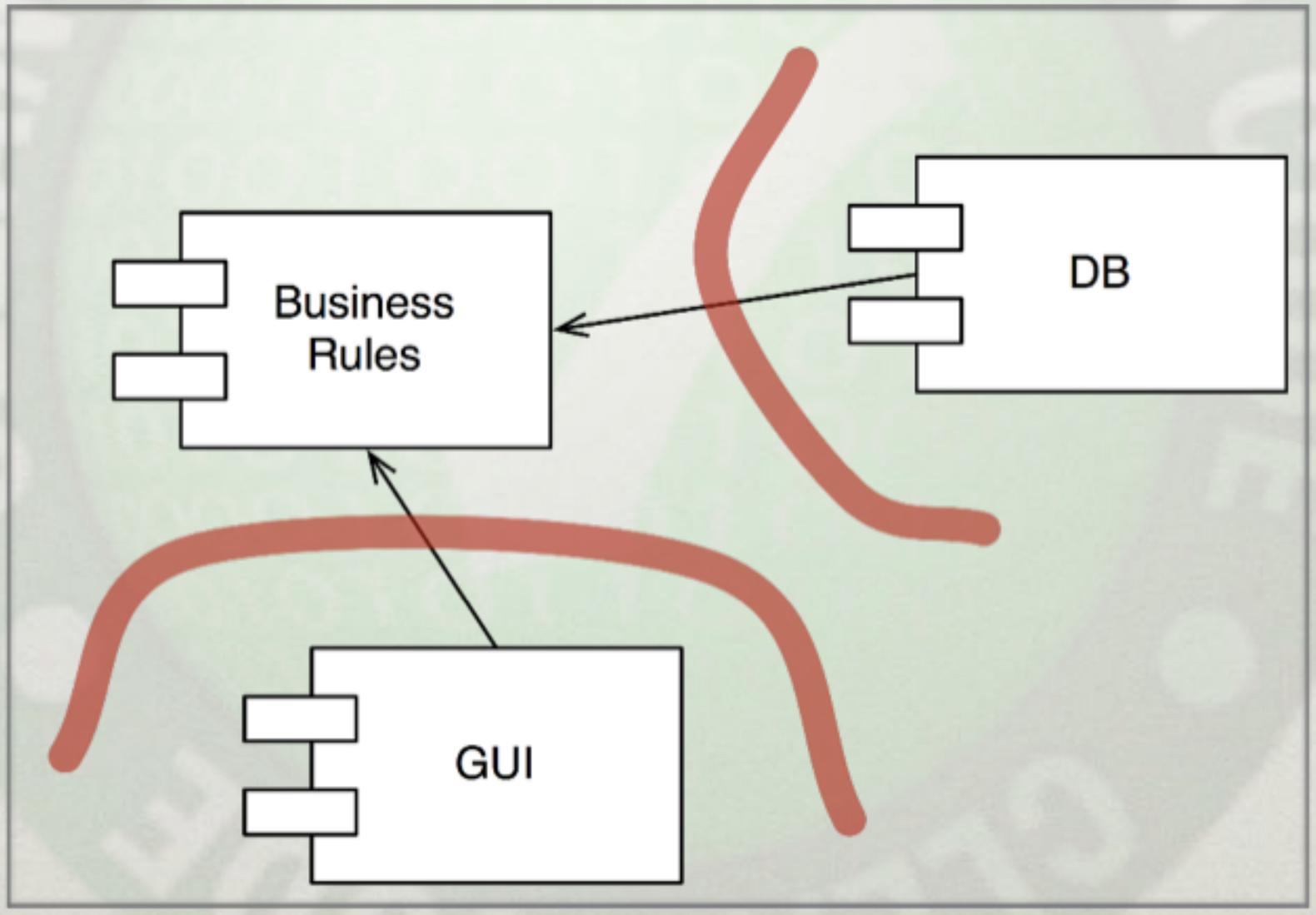


No one knows about this

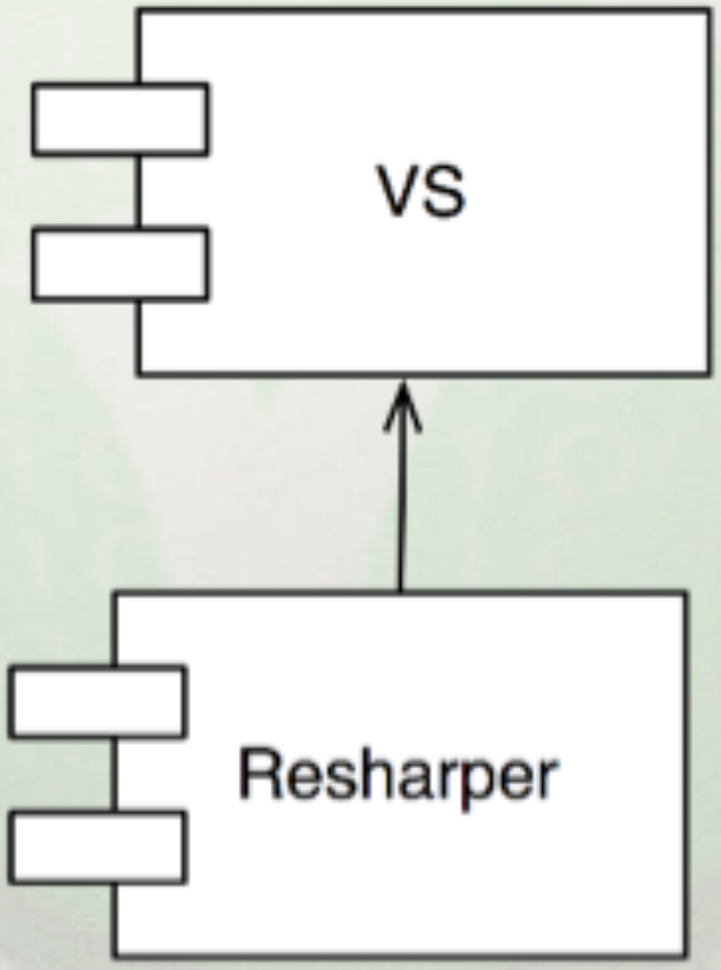


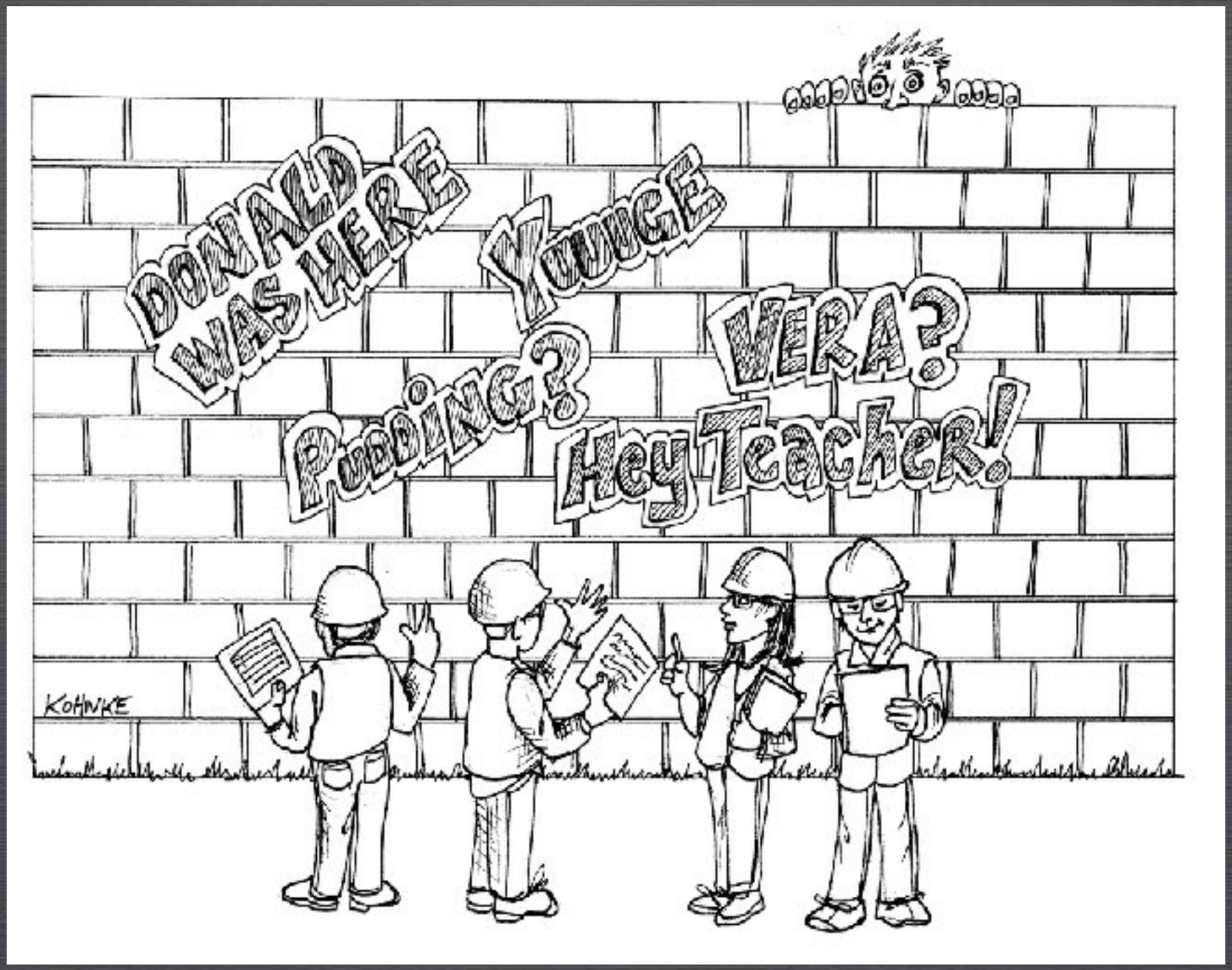
Dependencies cross boundary pointing towards High Level Policy.

Plugin Architecture



The Plugin Argument





Boundary Anatomy

Boundary Crossings

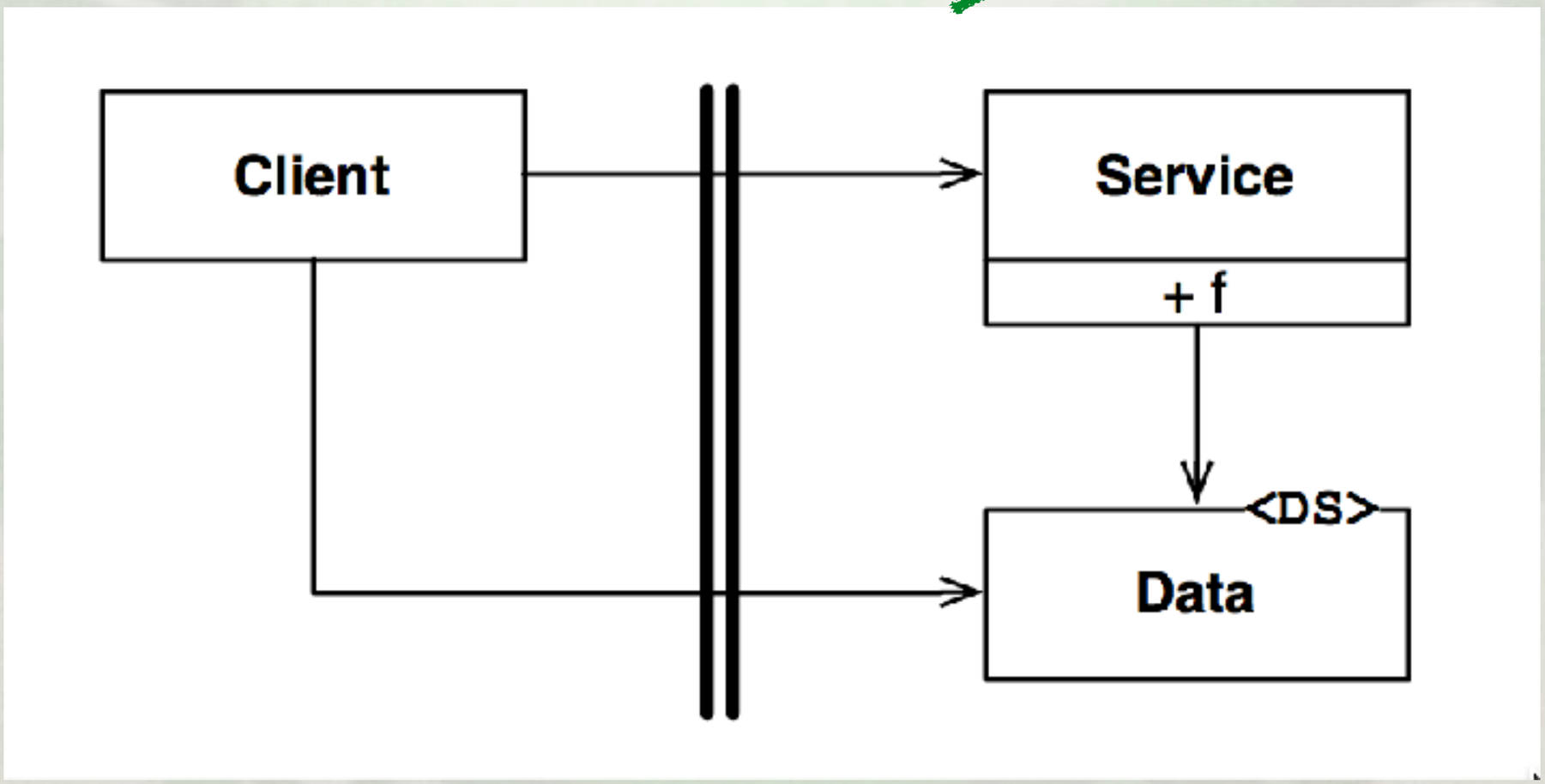
- One function calls another across a boundary.
- The trick is to manage the *source code* dependencies to prevent unnecessary recompiles and redeployments.

The Dreaded Monolith

- Simplest and most common form of boundary
- Disciplined segregation of functions and data within a single address space.
- Either statically or dynamically linked.
- Visible during development, not deployment.

Low level detail calls High level Policy

Flow of control

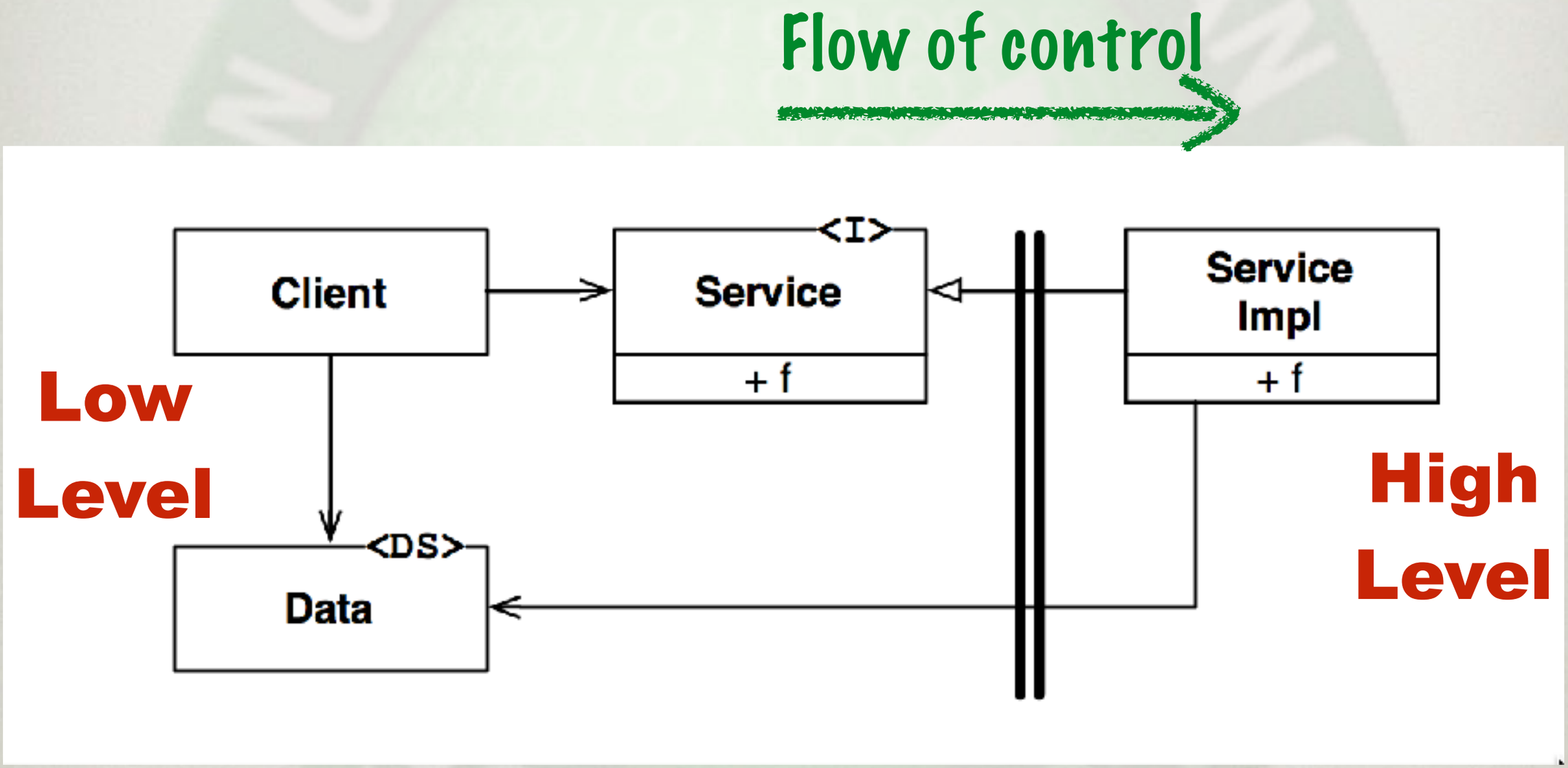


Low Level

High Level

Simplest of all boundary crossings

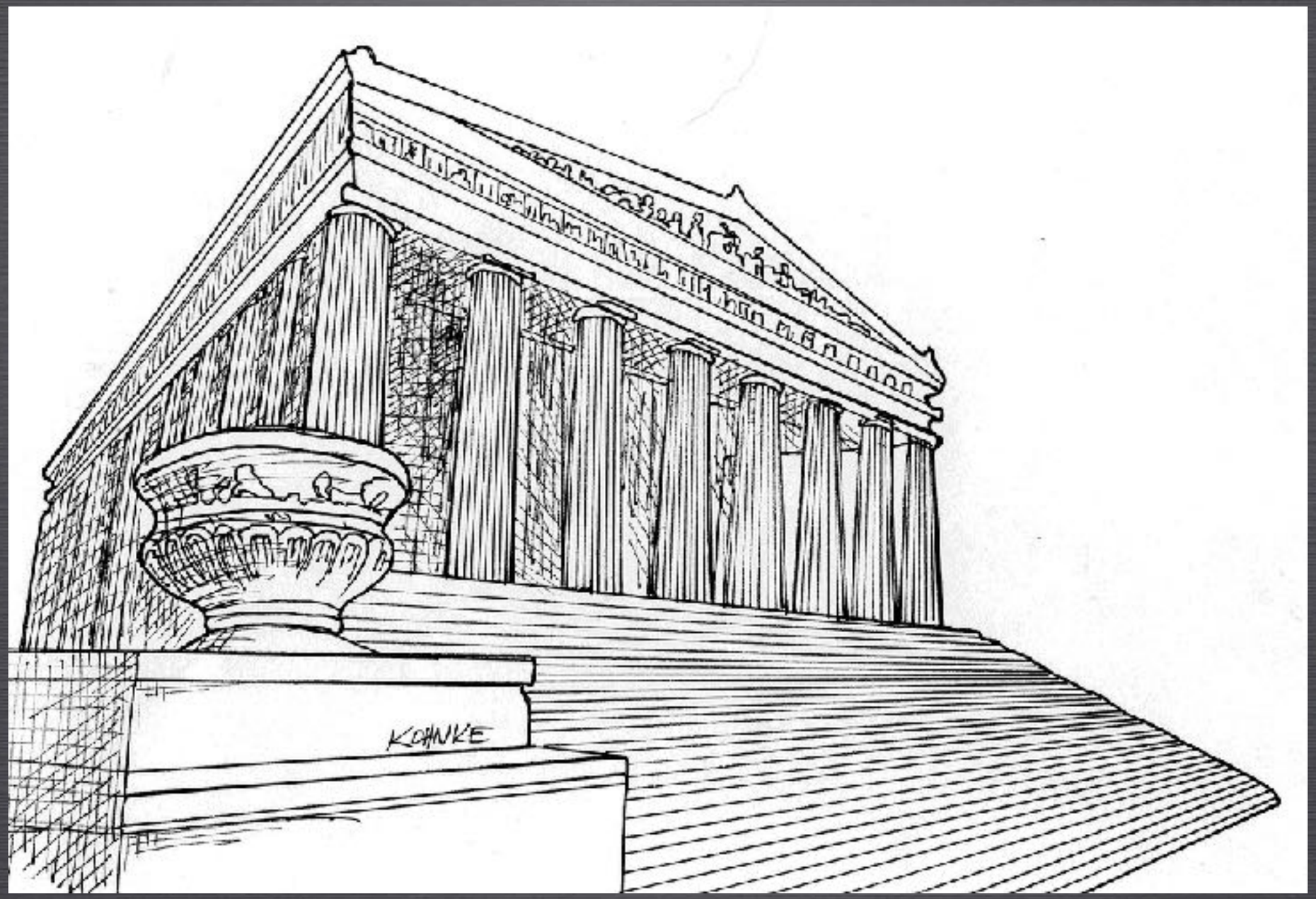
High Level Policy calls Low Level Service



Uses dynamic polymorphism

Deployment Boundaries

- JAR/DLL/GEM (often still monoliths)
- Threads. (often still monoliths)
- Local Processes (single processor, IPC)
- Services (network communication)



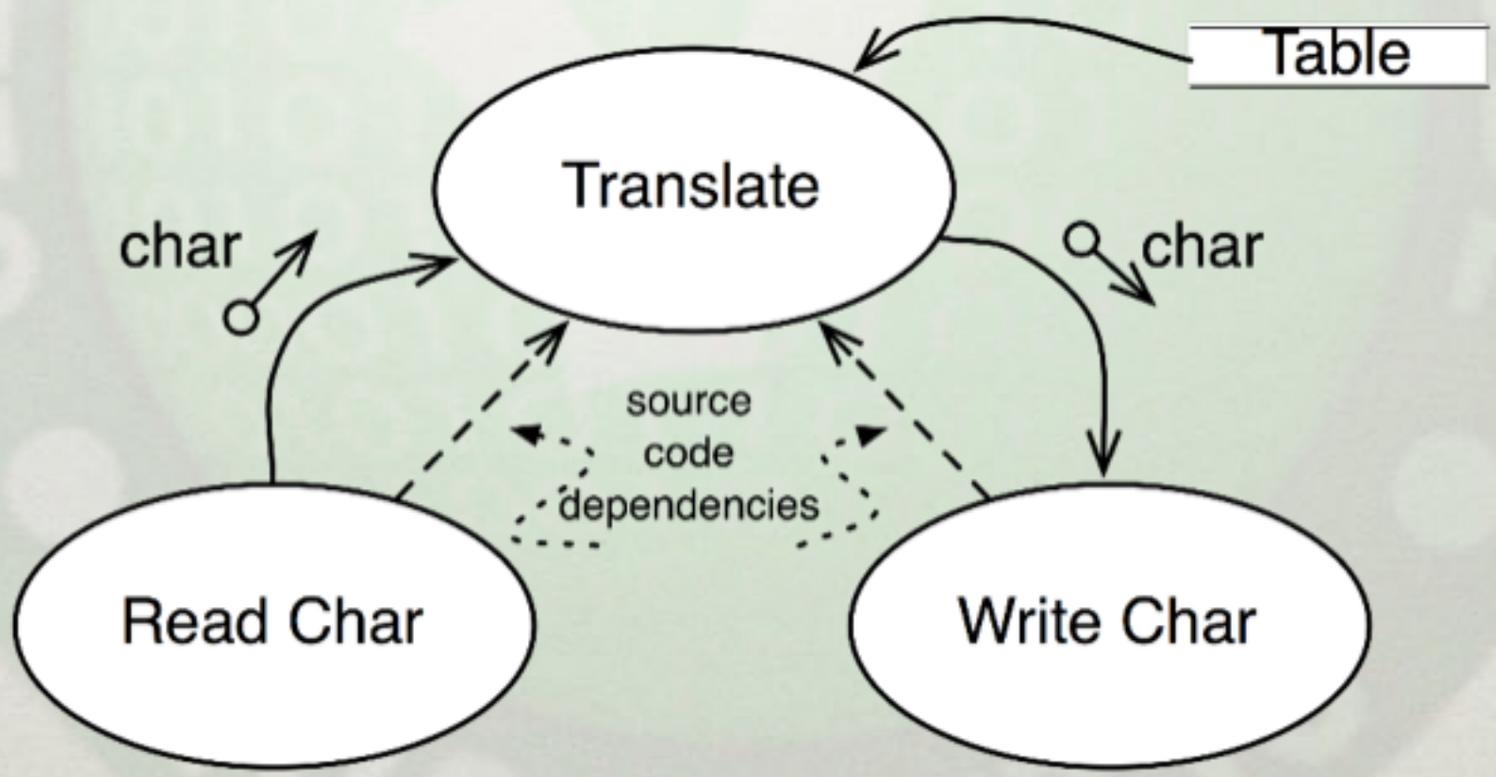
Policy and Level

Policy

- Software systems are statements of policy at many levels.
- A program is the policy by which inputs get translated to outputs.
- Part of the art of developing software architecture is carefully separating policies by level.

Level

- Definition: Highest level is farthest from inputs and outputs.



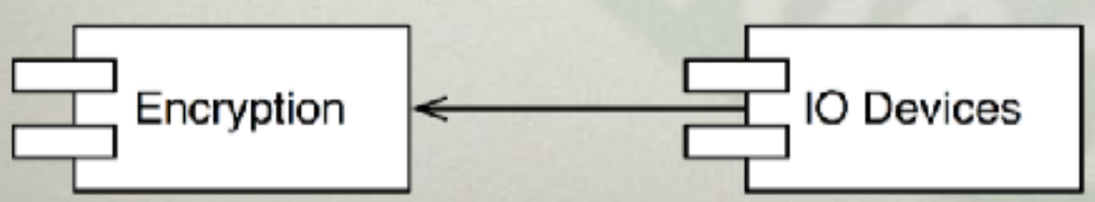
Data flows and source code dependencies don't always point in the same direction.

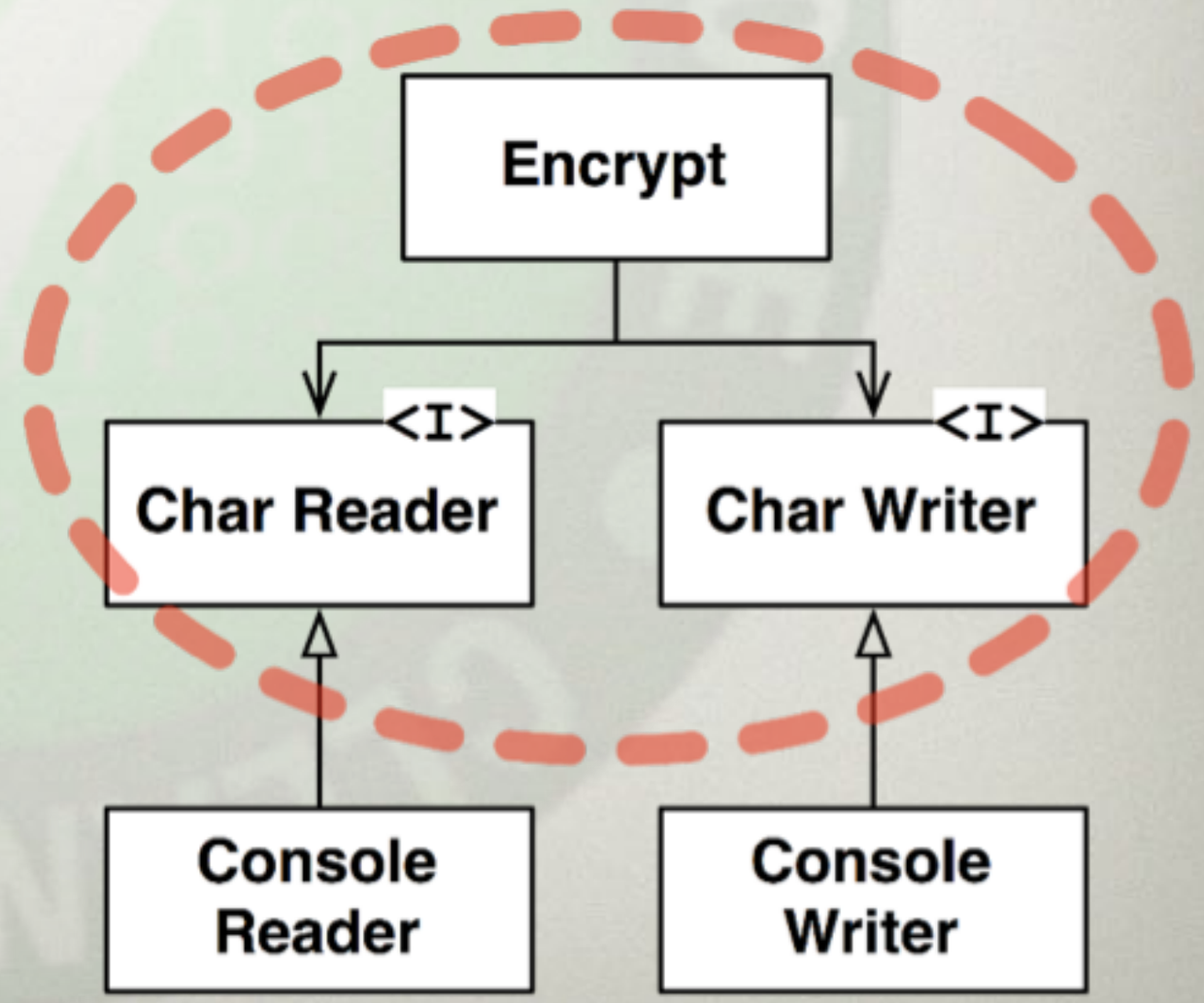
Level Implementation

```
function encrypt() {
  while(true)
    writeChar(translate(readChar()));
}
```

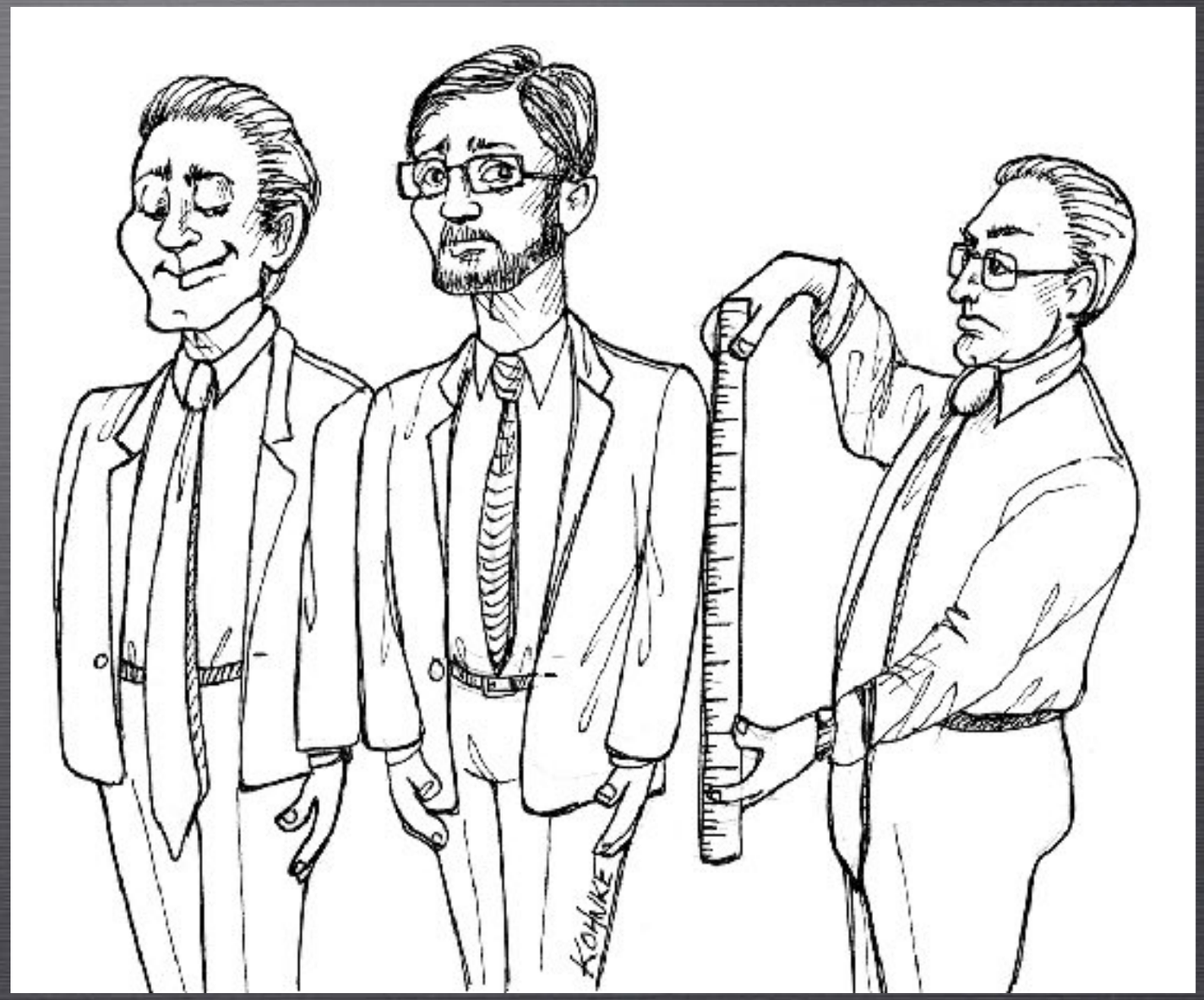
Bad Implementation

Good Implementation





Business Rules

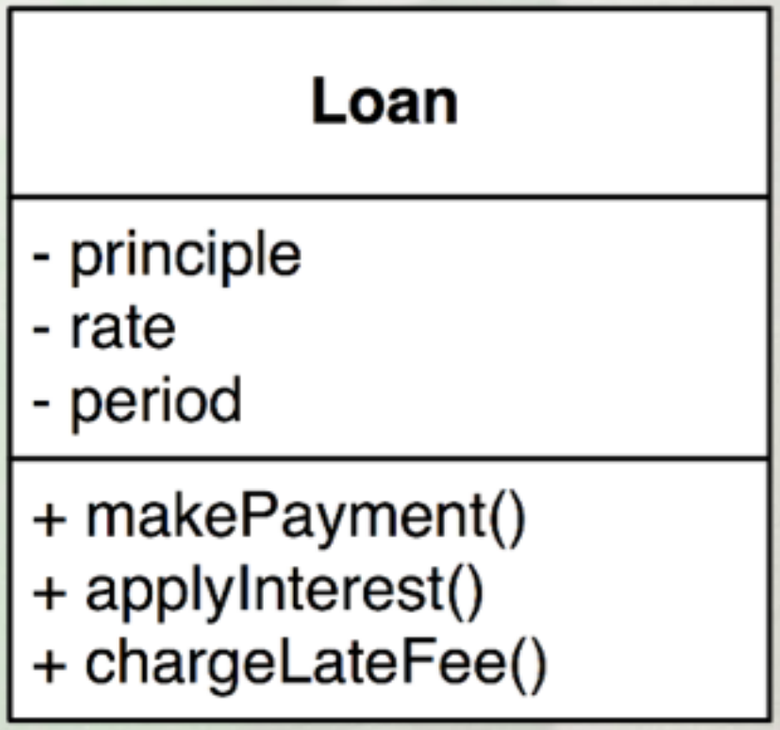


Critical Business Rules

- Critical Business Rules make or save the company money. They are business policies independent of automation.
- The data associated with CBRs are Critical Business Data which are also independent of automation.

Entities

- Encapsulate a small set of Critical Business Rules that operate upon Critical Business Data.
- Sometimes known as business objects.



Use Cases

- Business rules and data that are intrinsic to automation. e.g. the flow of an interaction.
- Describe how and when Entities are invoked.
- Application Specific Business rules.

Gather Contact Info for New Loan

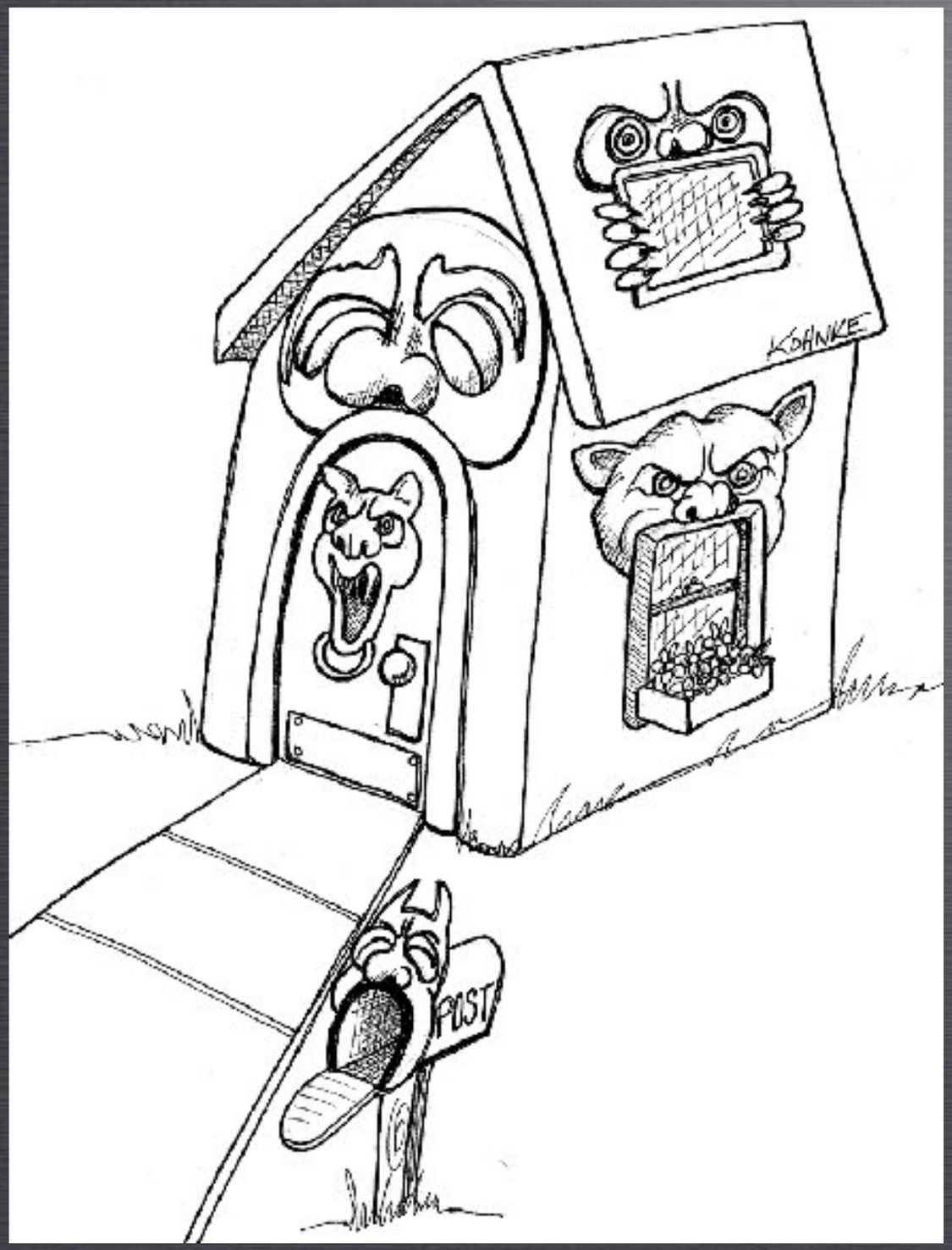
Input: Name, Address, Birthdate, D.L. #, SSN, etc.
Output: Same info for readback + credit score.

Primary Course:

1. Accept and validate name.
2. Validate address, birthdate, D.L.#, SSN, etc.
3. Get credit score.
4. If credit score is < 500 activate Denial.
5. Else create Customer and activate Loan Estimation.

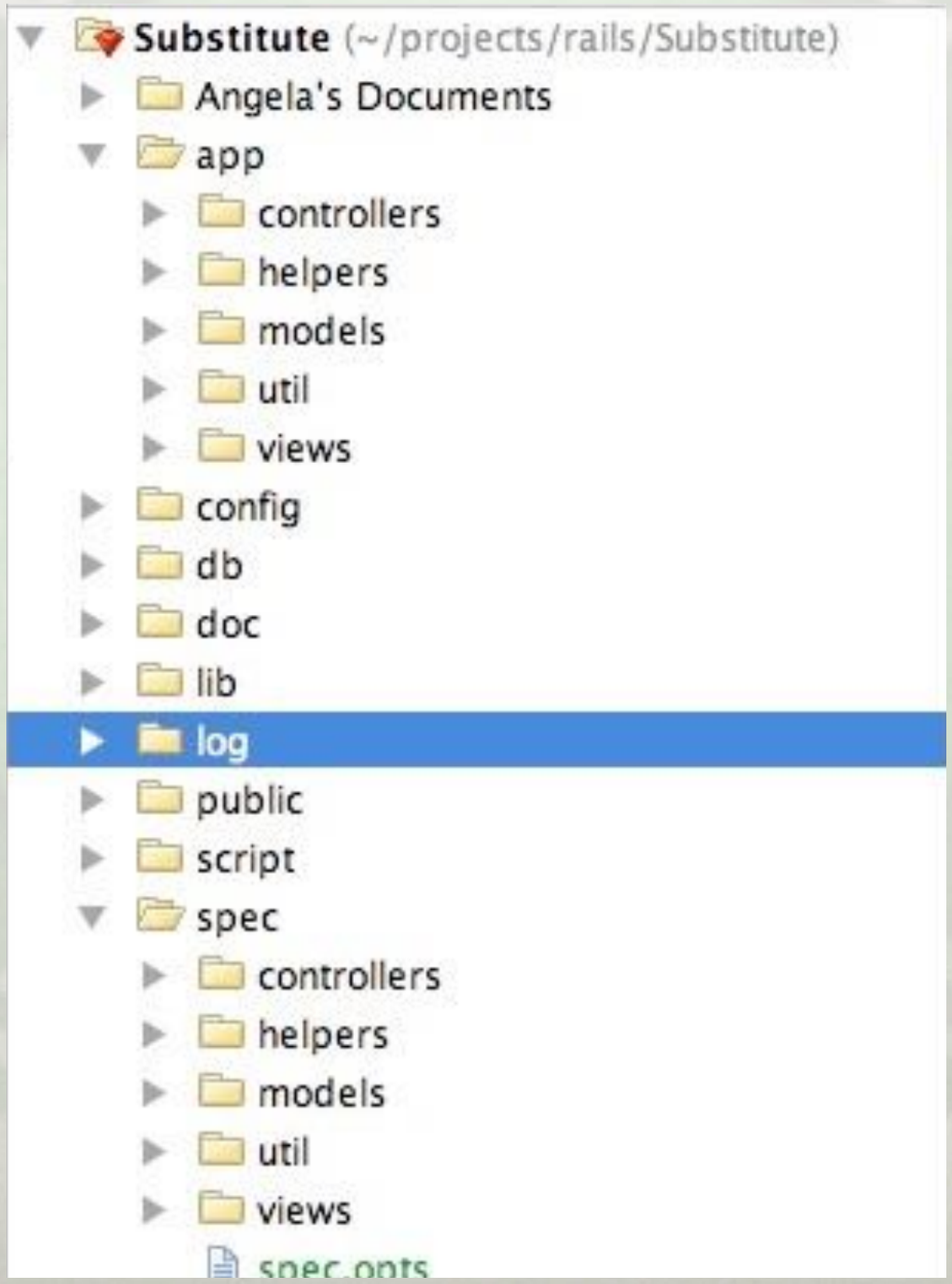
Request/Response models

- Use cases are responsible for receiving and decoding requests.
- Controlling the dance of the entities.
- Gathering and organizing the response data.



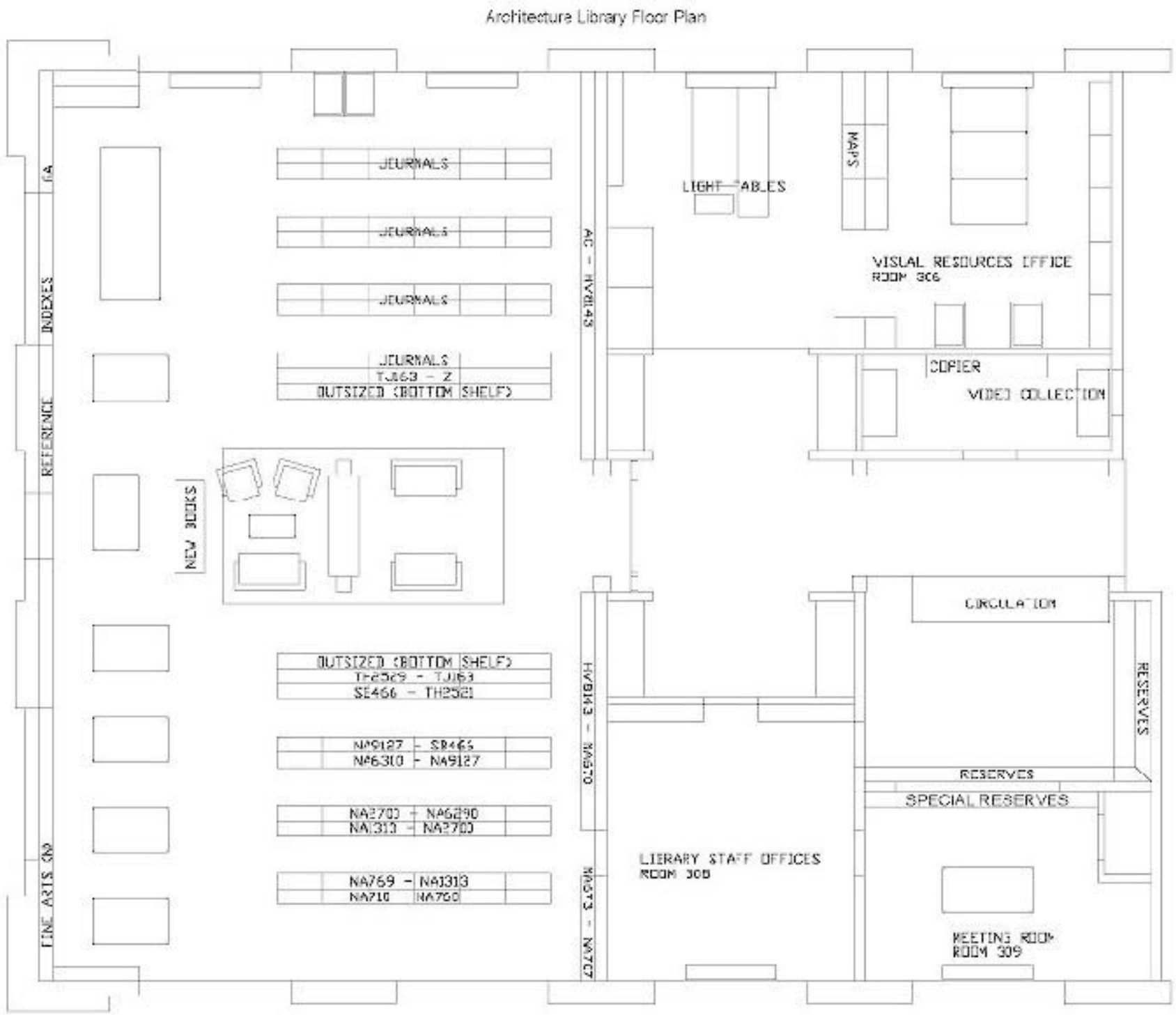
Screaming Architecture

A Rails App


Substitute (~/projects/rails/Substitute)
Angela's Documents
app
controllers
helpers
models
util
views
config
db
doc
lib
log
public
script
spec
controllers
helpers
models
util
views
spec.opts

The Web is a Delivery Mechanism!

Library

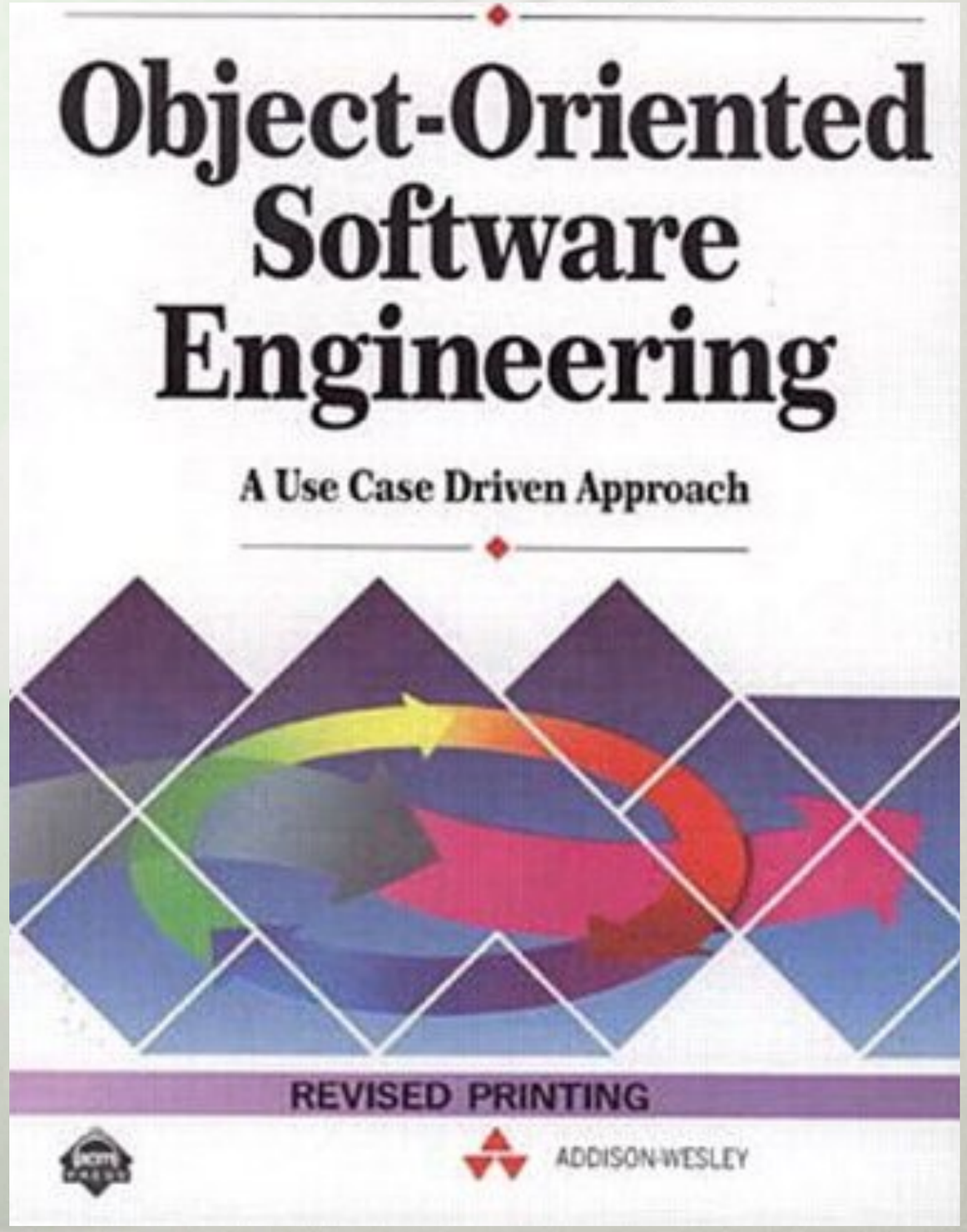

Architecture Library Floor Plan
JOURNALS
JOURNALS
JOURNALS
JOURNALS
TJ163 - Z
OUTSIZED (BOTTOM SHELF)
NEW BOOKS
OUTSIZED (BOTTOM SHELF)
TH2529 - TJ163
SE466 - TH2521
NA9127 - SB466
NA6310 - NA9127
NA2700 - NA6290
NA1313 - NA2700
NA769 - NA1313
NA710 NA760
INDEXES
REFERENCE
FINE ARTS 3N
AC - HVAC43
HVAC43 - NA370
NA373 - NA707
LIGHT TABLES
MAPS
VISUAL RESOURCES OFFICE
ROOM 306
COPIER
VIDEO COLLECTION
CIRCULATION
RESERVES
RESERVES
SPECIAL RESERVES
LIBRARY STAFF OFFICES
ROOM 308
MEETING ROOM
ROOM 309

Church

CLASS ROOM
CLASS ROOM
CLASS ROOM
CLASS ROOM
STORAGE
MEN
WOMEN
STORAGE
SUNDAY SCHOOL SUPERINTENDENT
MEN
WOMEN
STAGE
STORAGE MAINTENANCE
MECHANICAL
CLASS ROOM
CLASS ROOM
CLASS ROOM
CLASS ROOM
YOUTH PASTOR
DRY ROOM
DRY ROOM
SOUND
KITCHENETTE
NURSERY
CONFERENCE ROOM
FOYER– FELLOWSHIP
COVERED DRIVE THROUGH
OFFICE
OFFICE STORAGE
ASSOC. PASTOR
ELEV.
SENIOR PASTOR

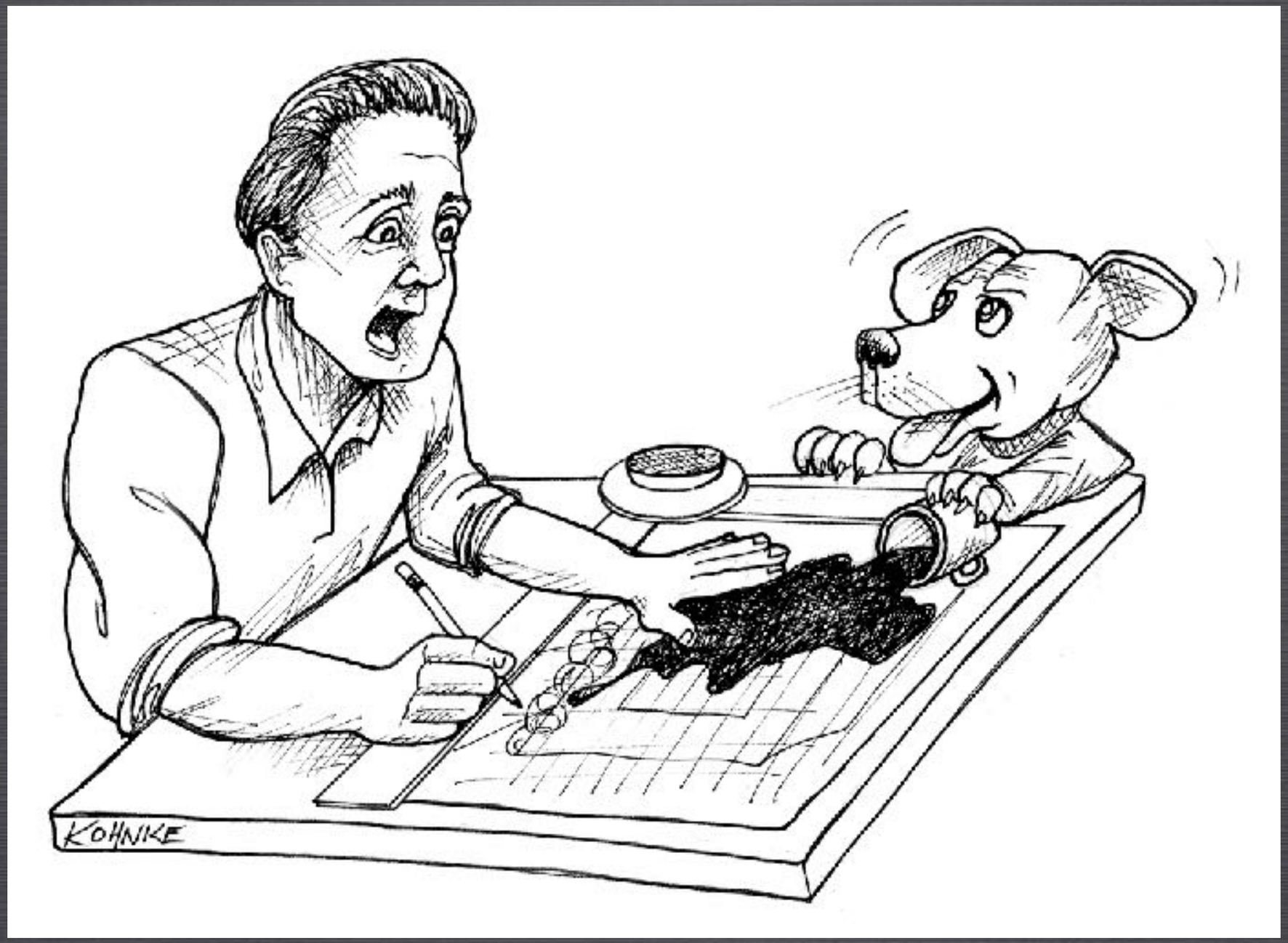
Architecture is about intent

What Happened?



Scream

- A good architecture screams its intent through the organization of its use cases.
- It is about testable use cases, not tools, not frameworks, not languages, not platforms.



The Clean Architecture

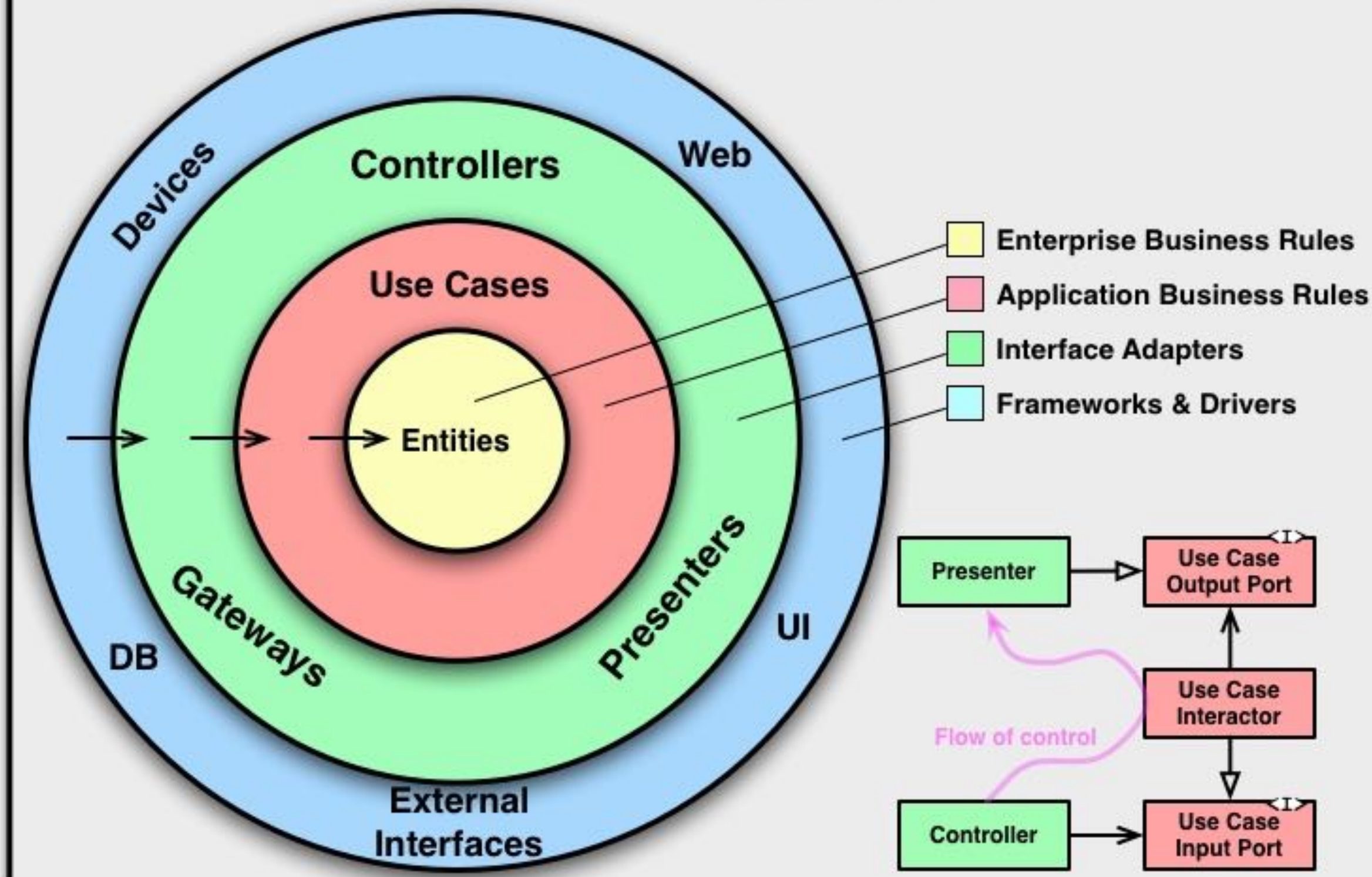
Clean Architecture Styles

- Hexagonal Architecture (Cockburn)
- Ports and Adapters (Freeman, Pryce)
- DCI (Coplien & Reenskaug)
- BCE (Jacobson)

Attributes of Clean Architecture

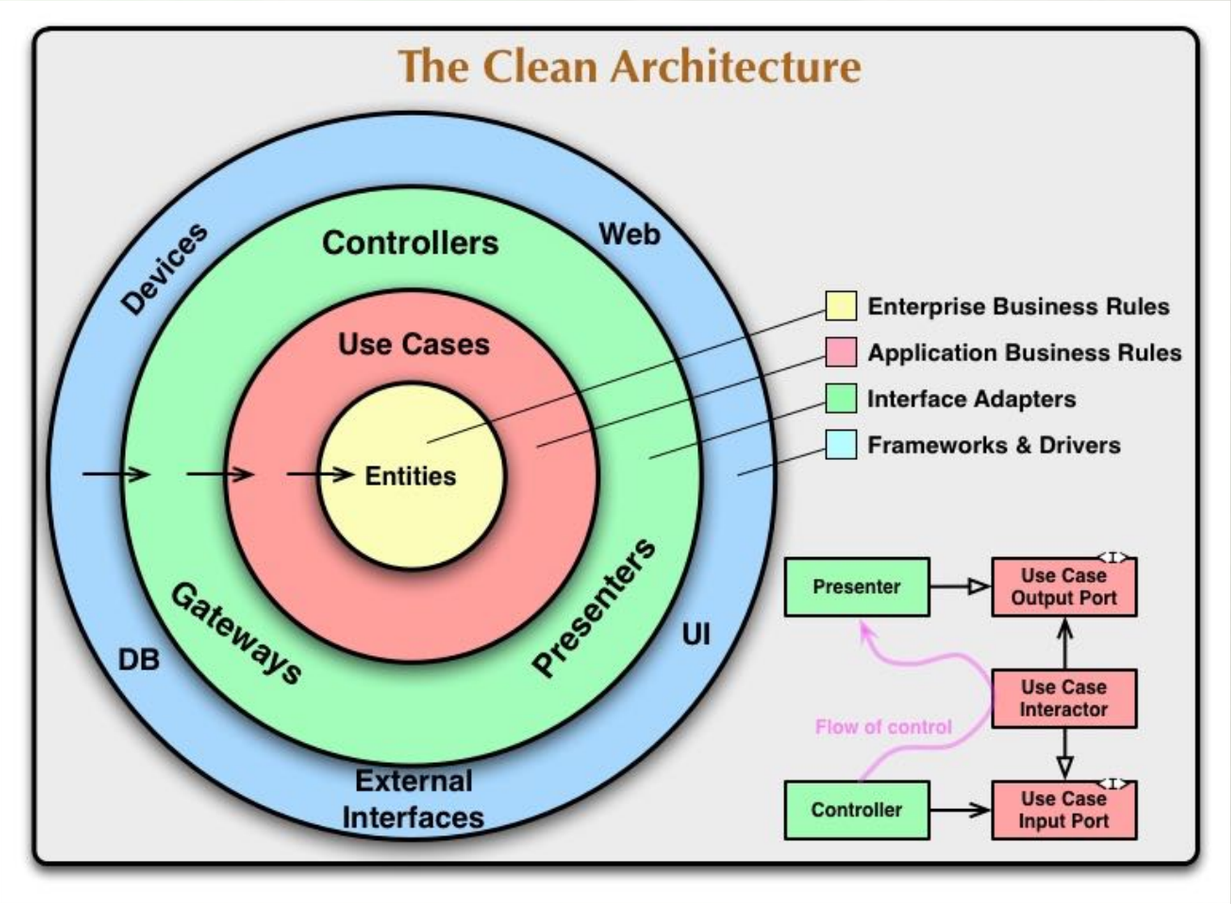
- Independent of Frameworks
- Testable
- Independent of UI
- Independent of Database
- Independent of any external agency.

The Clean Architecture

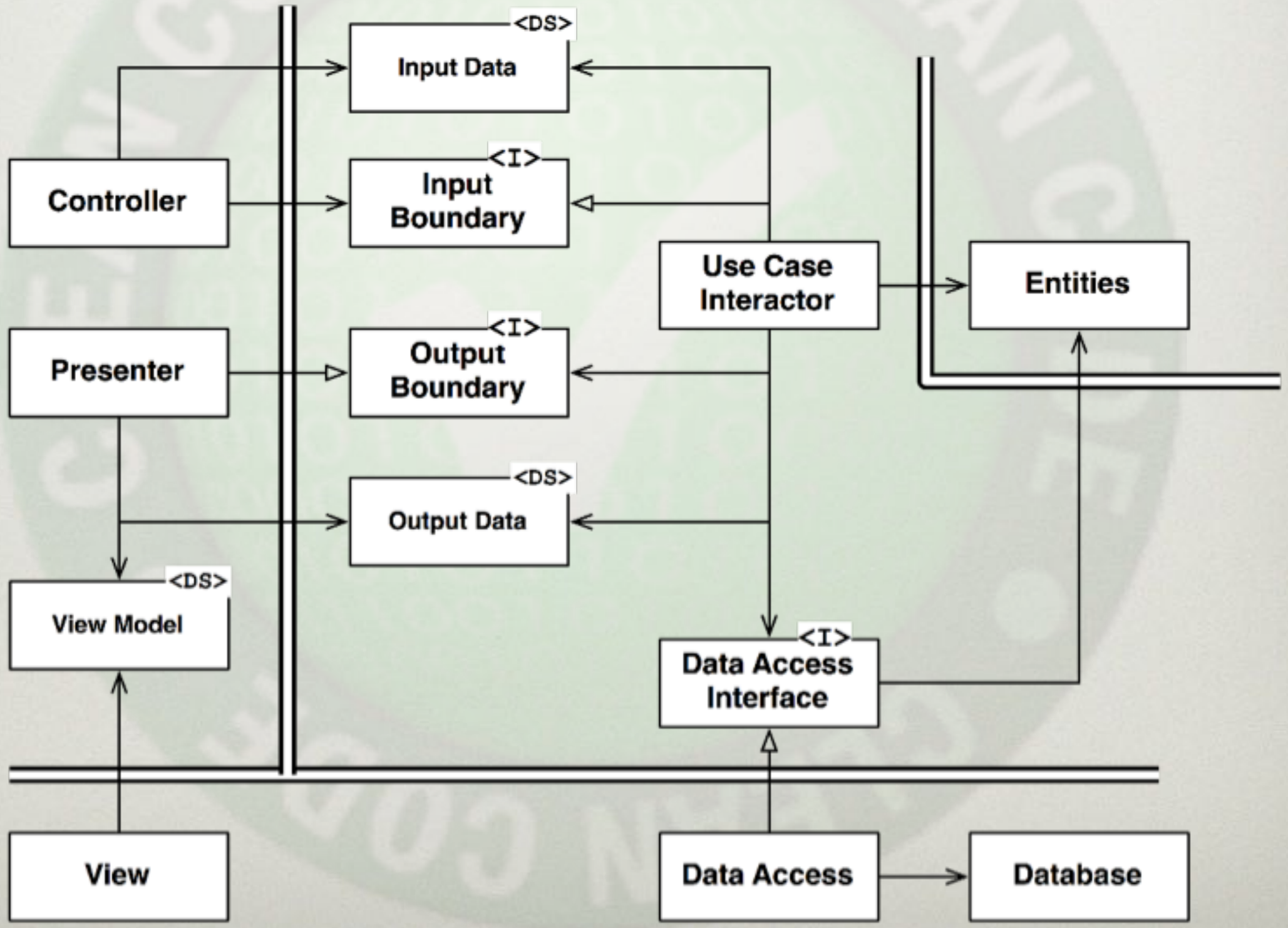


The Dependency Rule

- Source code dependencies must only point inward, towards higher level policies.



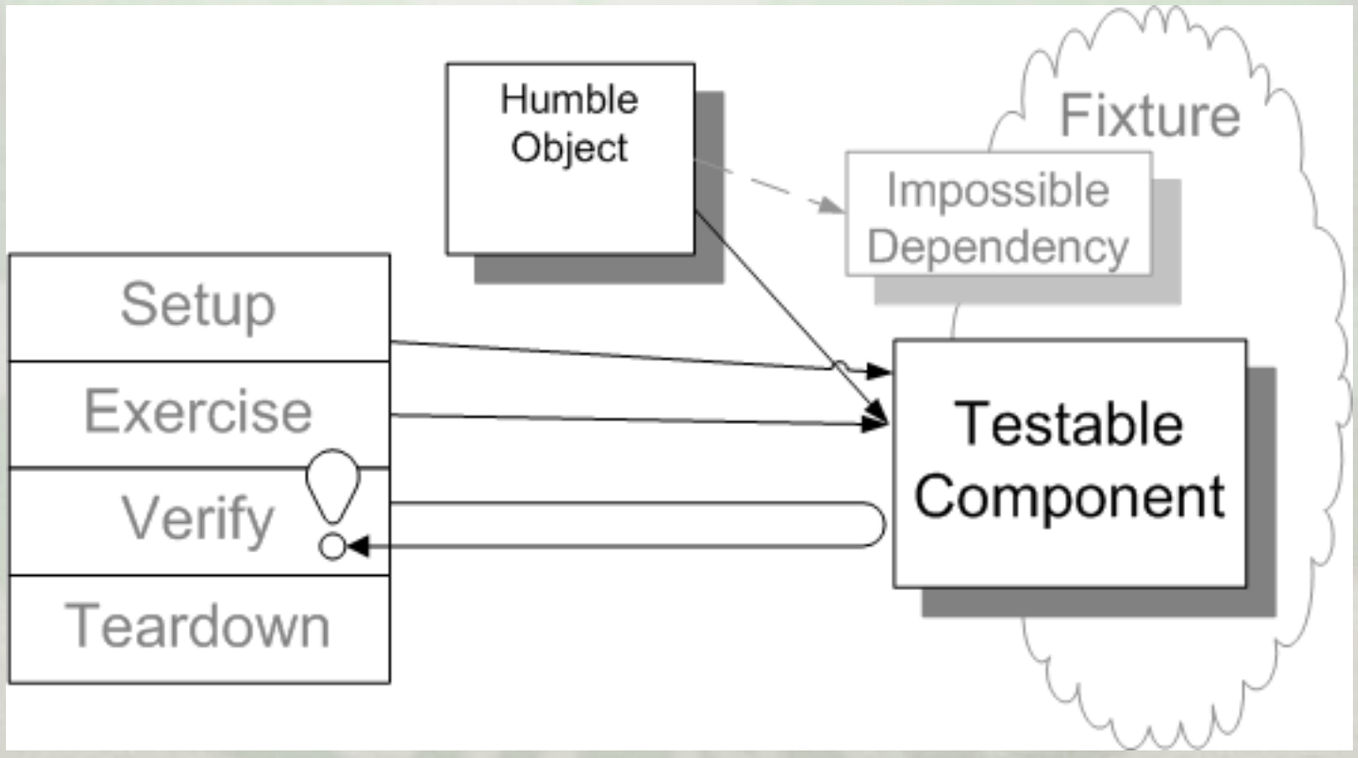
Typical Scenario



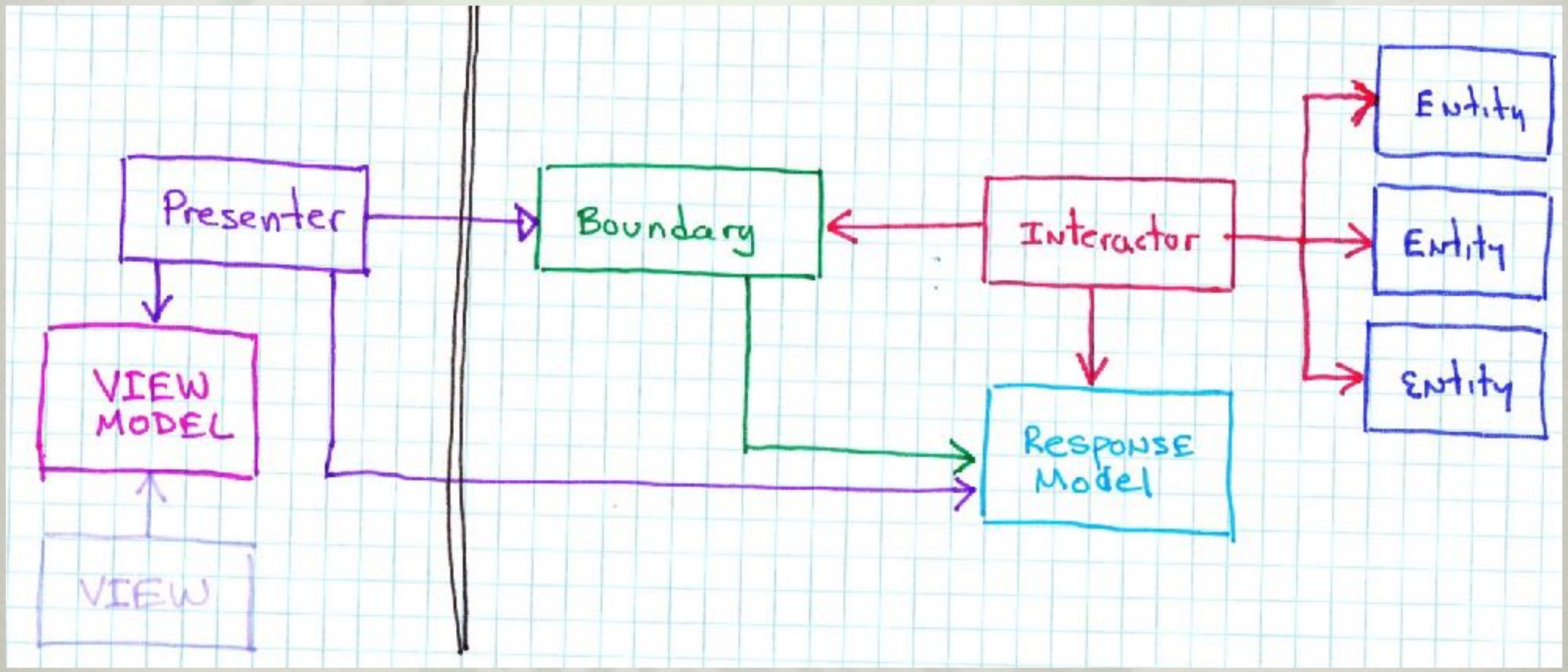
Presenters and Humble Objects

The Humble Object Pattern

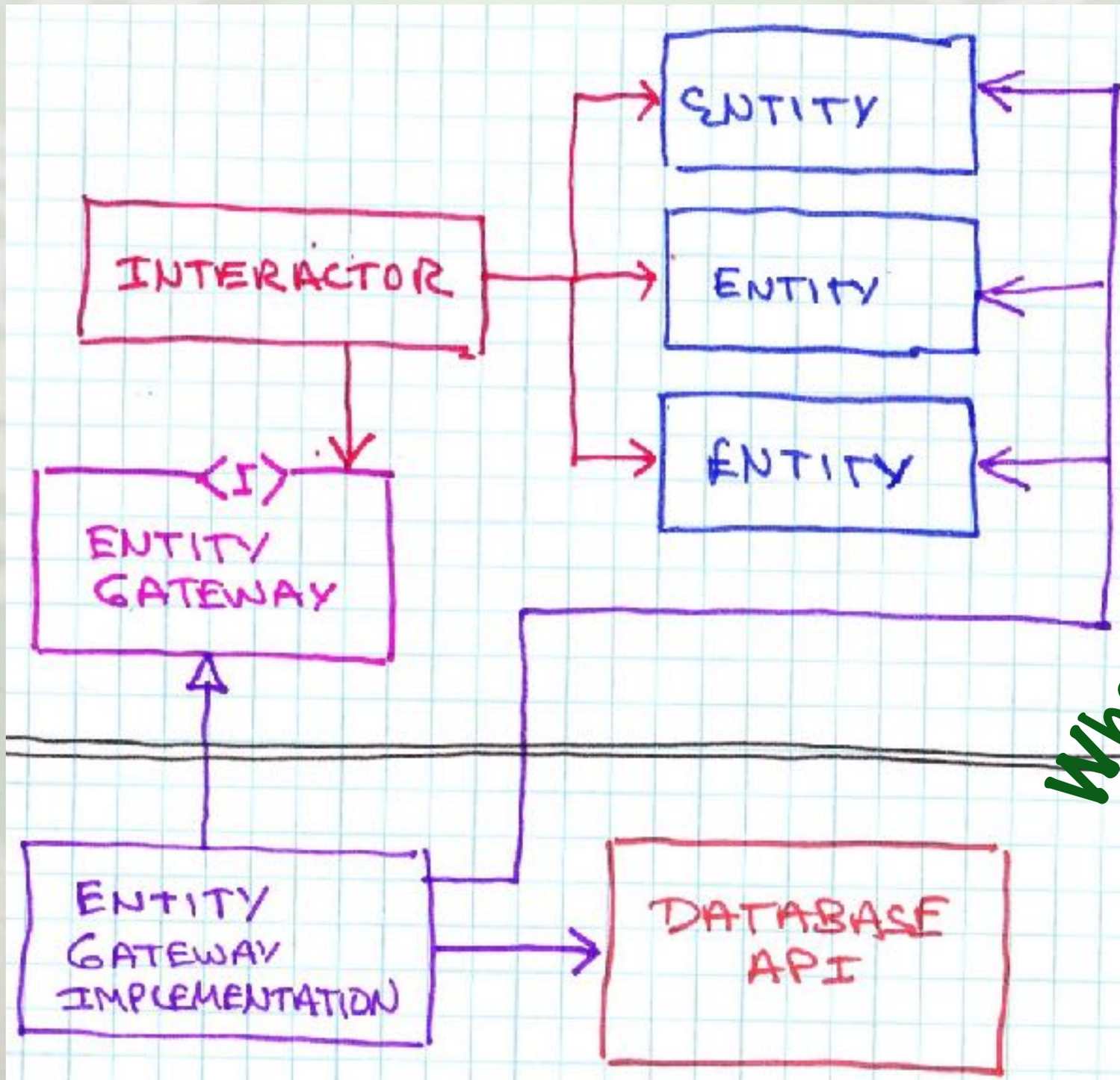
- XUnit Patterns, Meszaros.
  - Separate and humble hard to test behaviors.



Presenters and Views

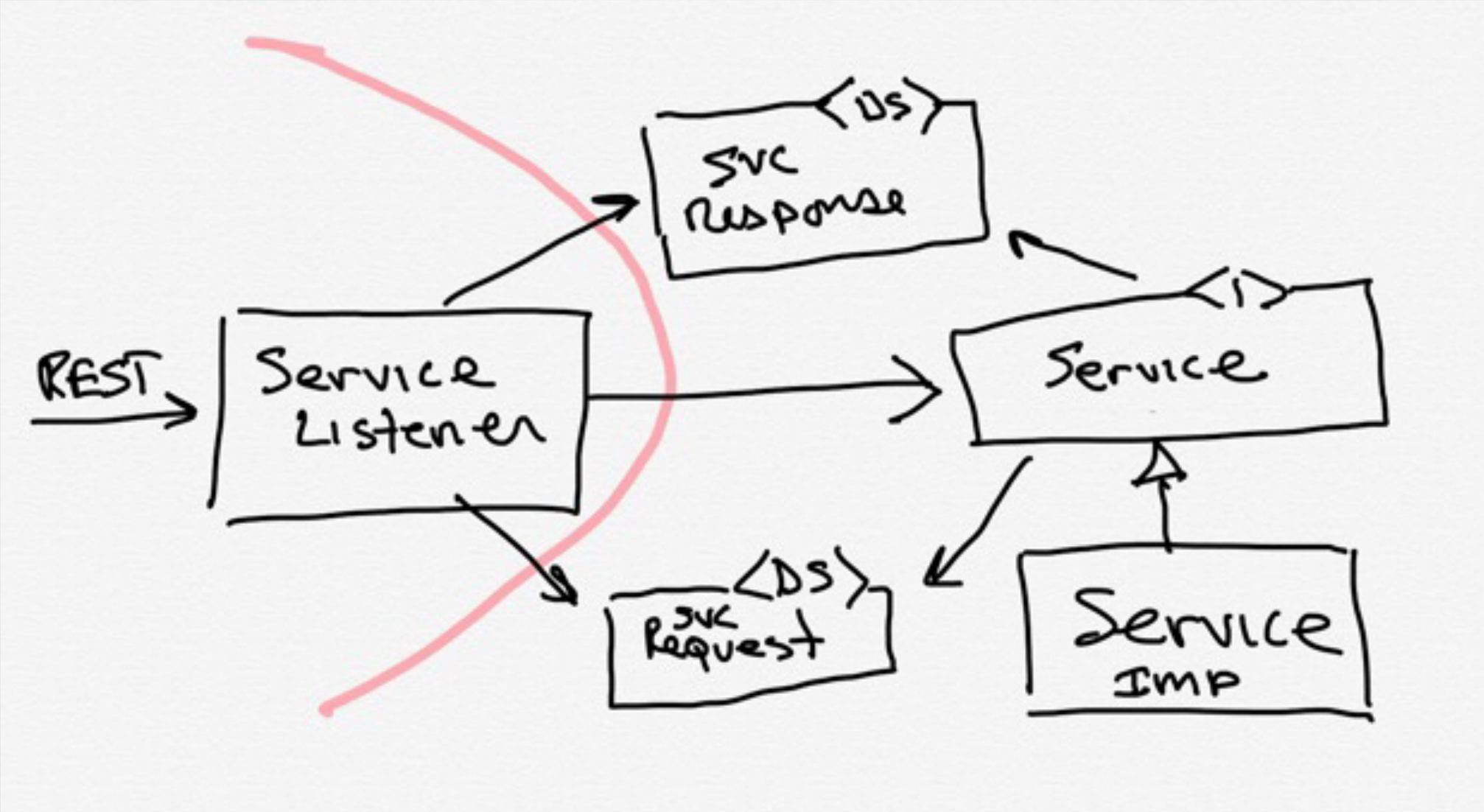


Database Gateways



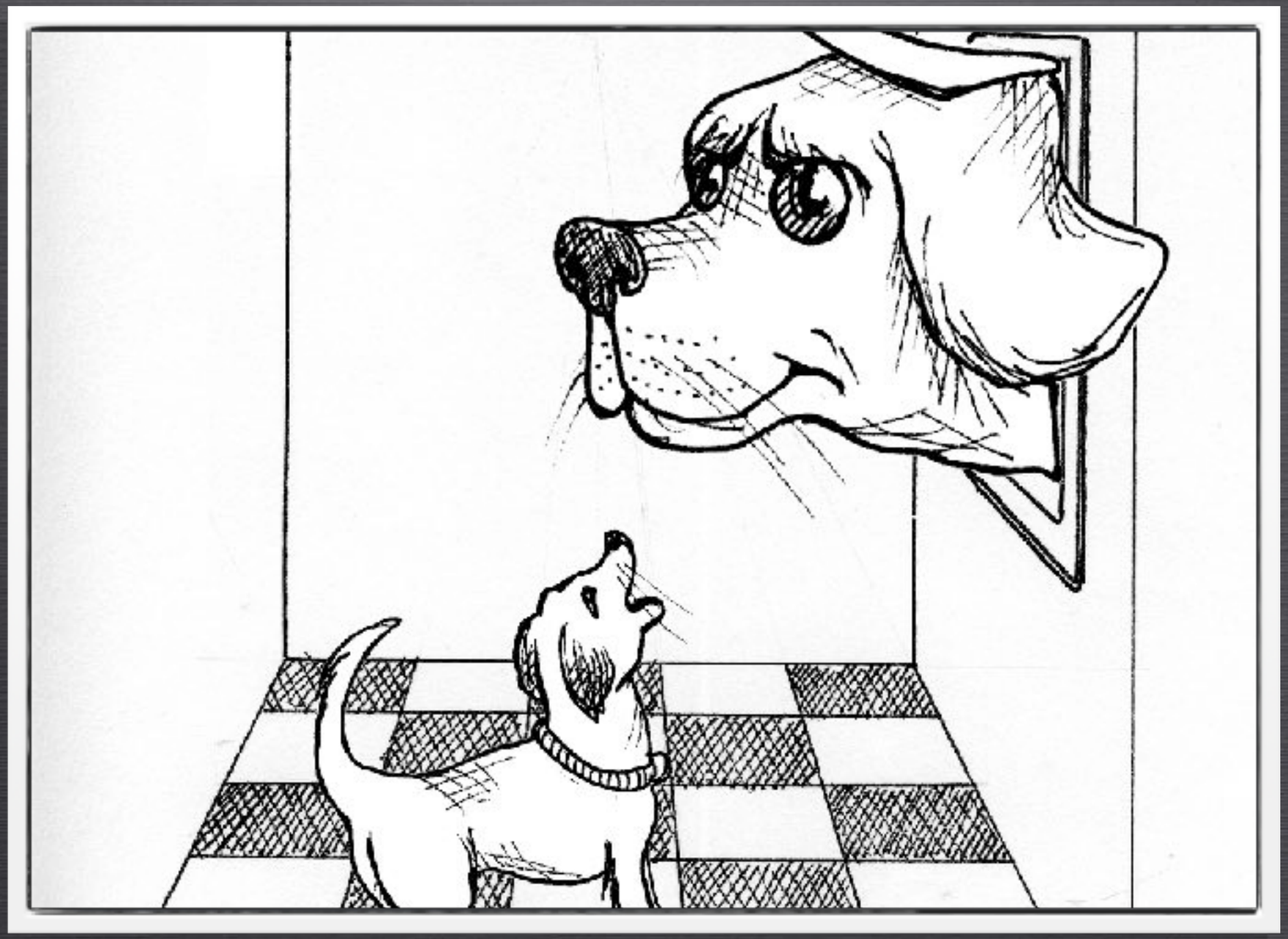
Where is the ORM?

Service Listeners



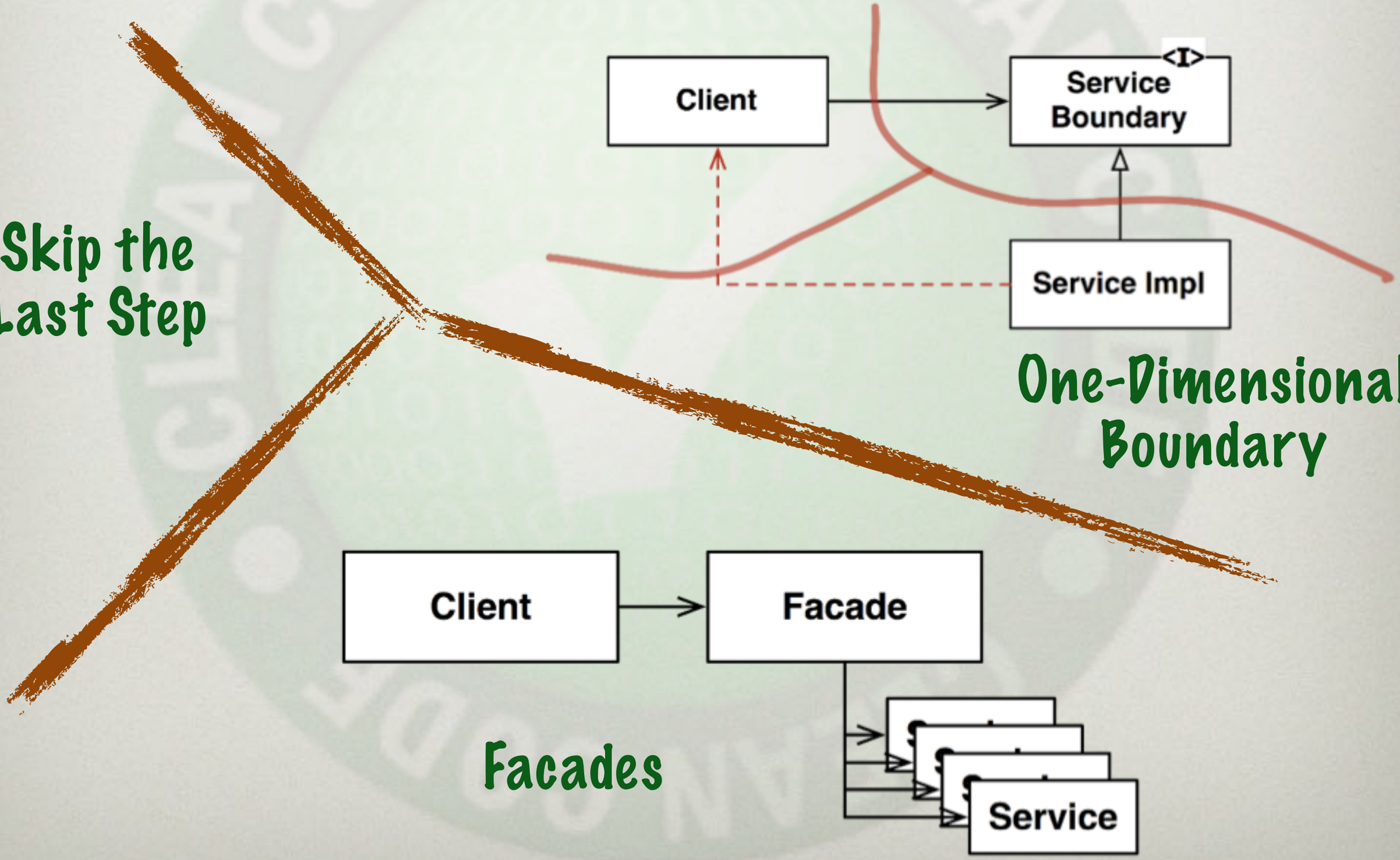
Testing and Architecture

A good architecture is a testable architecture.



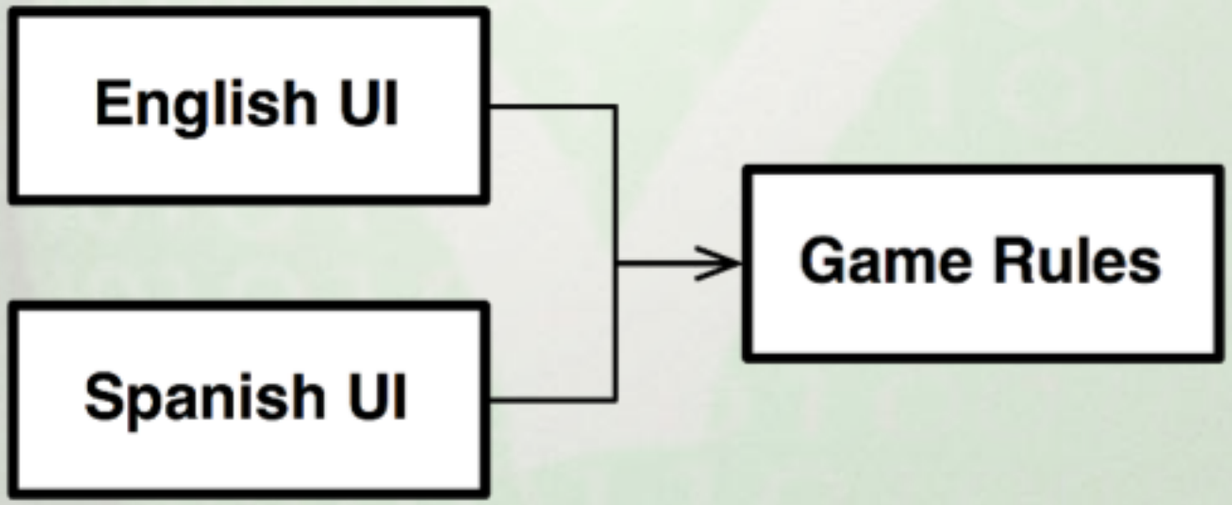
Partial Boundaries

Partial Boundaries



Layers and Boundaries

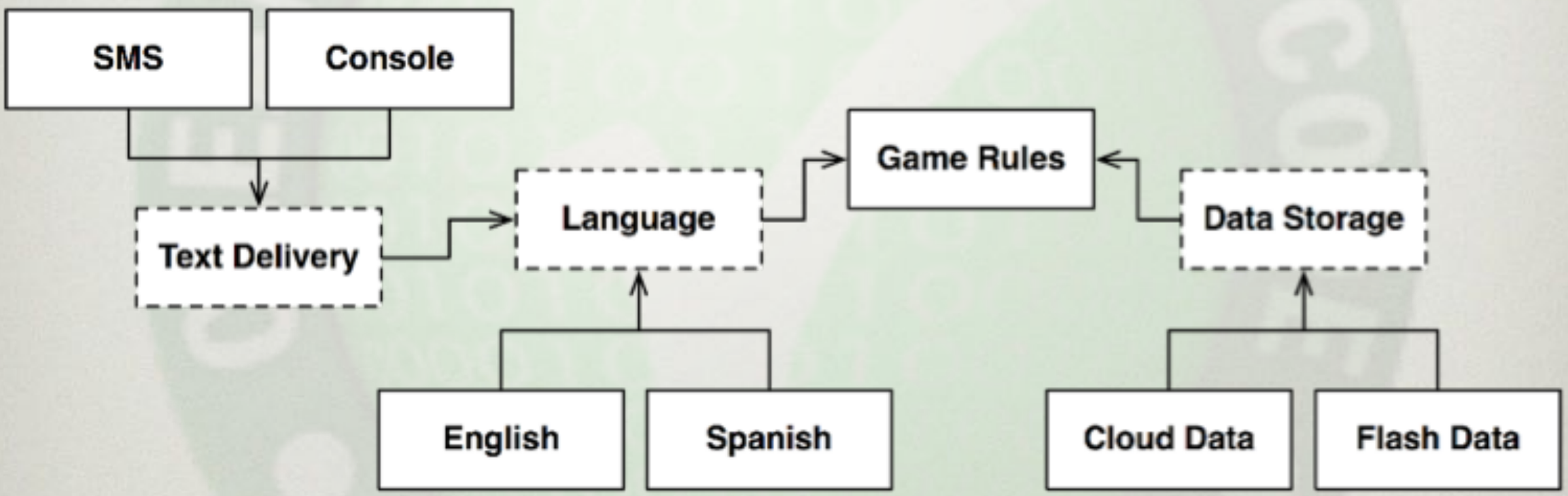
Multi-lingual Hunt the Wumpus



Storage Agnostic

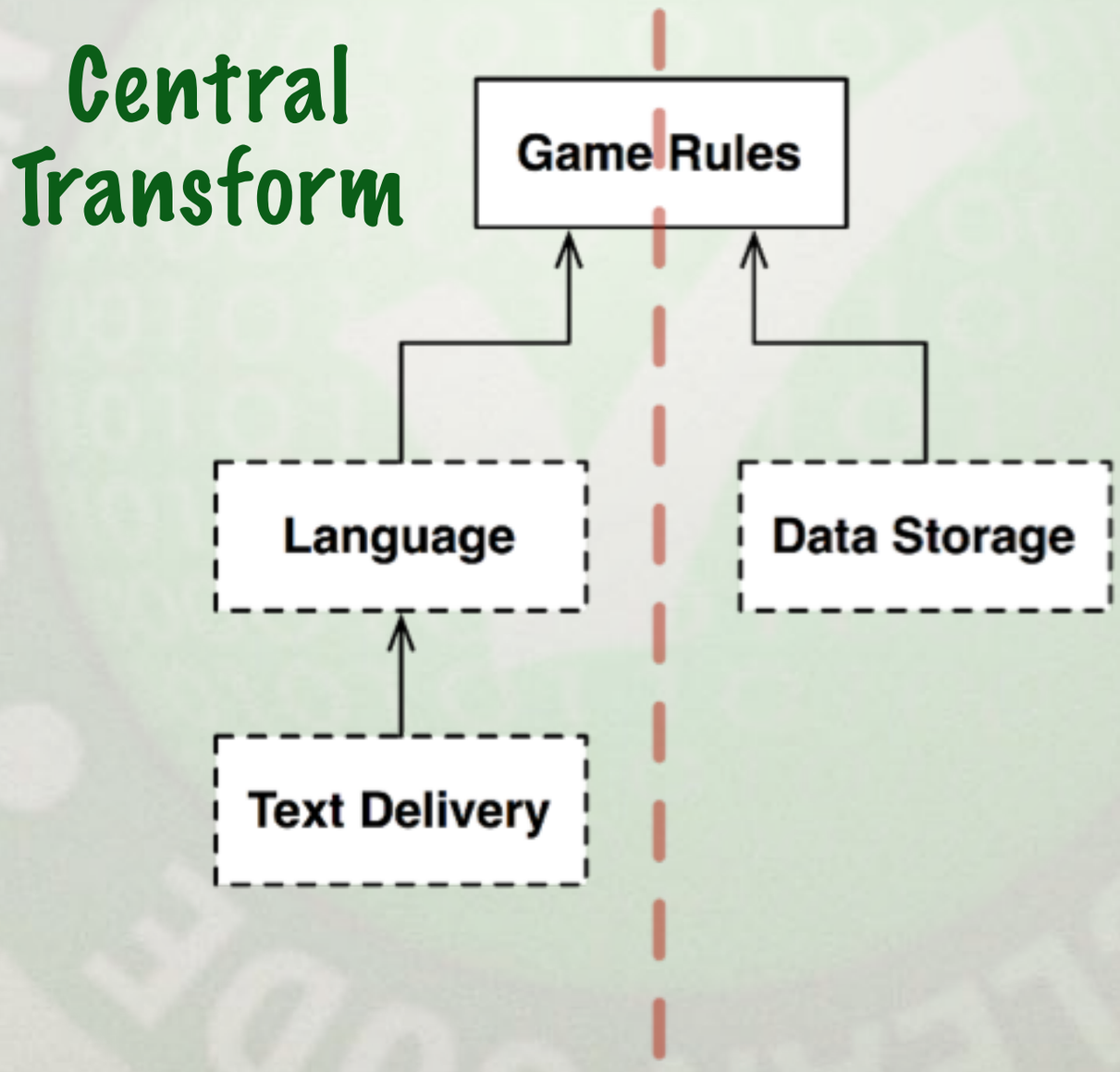
Text Delivery Agnostic

Layers and Levels

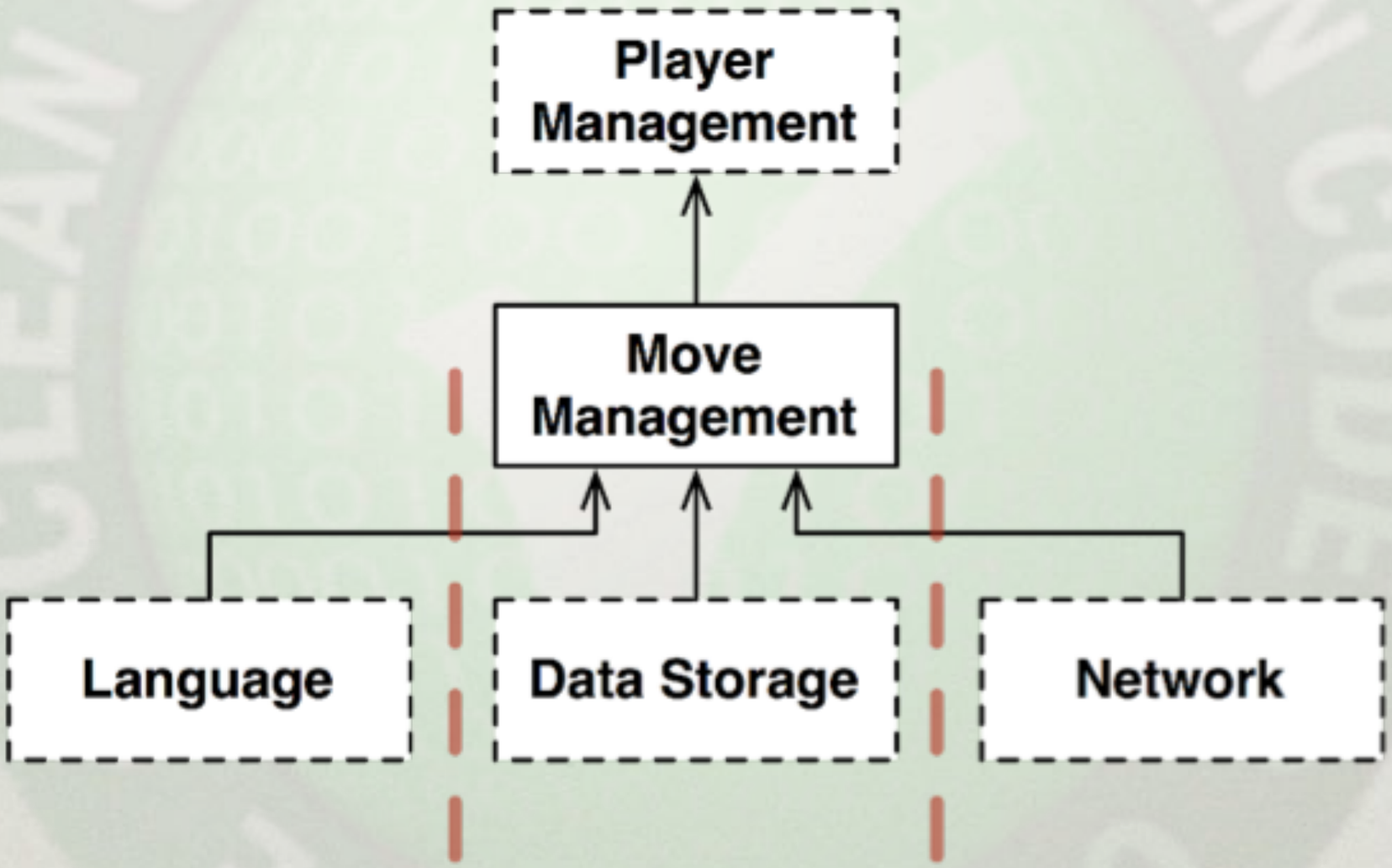


Where are the Boundaries?

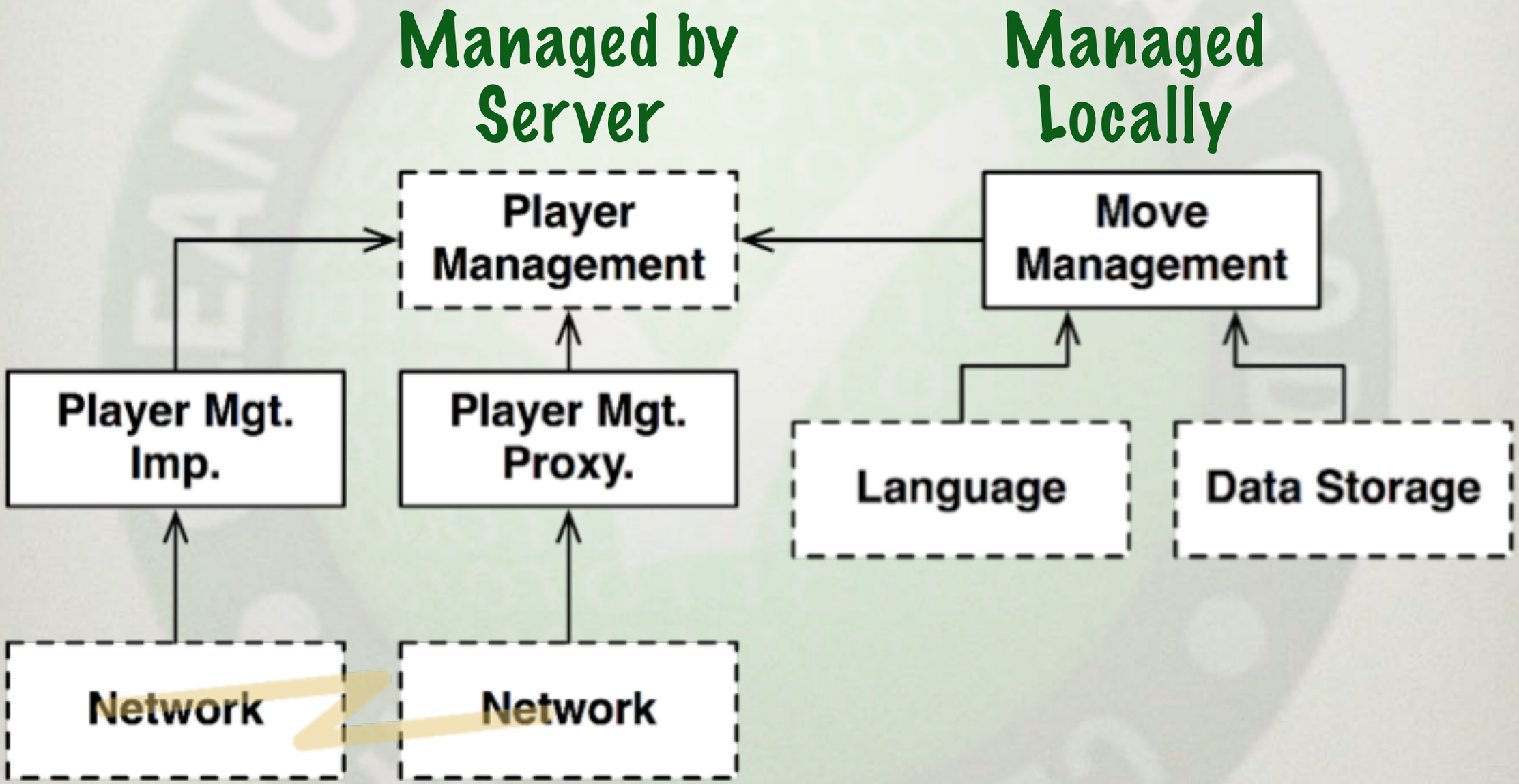
Two Data Streams. UI to Datastore



Three Streams MUD

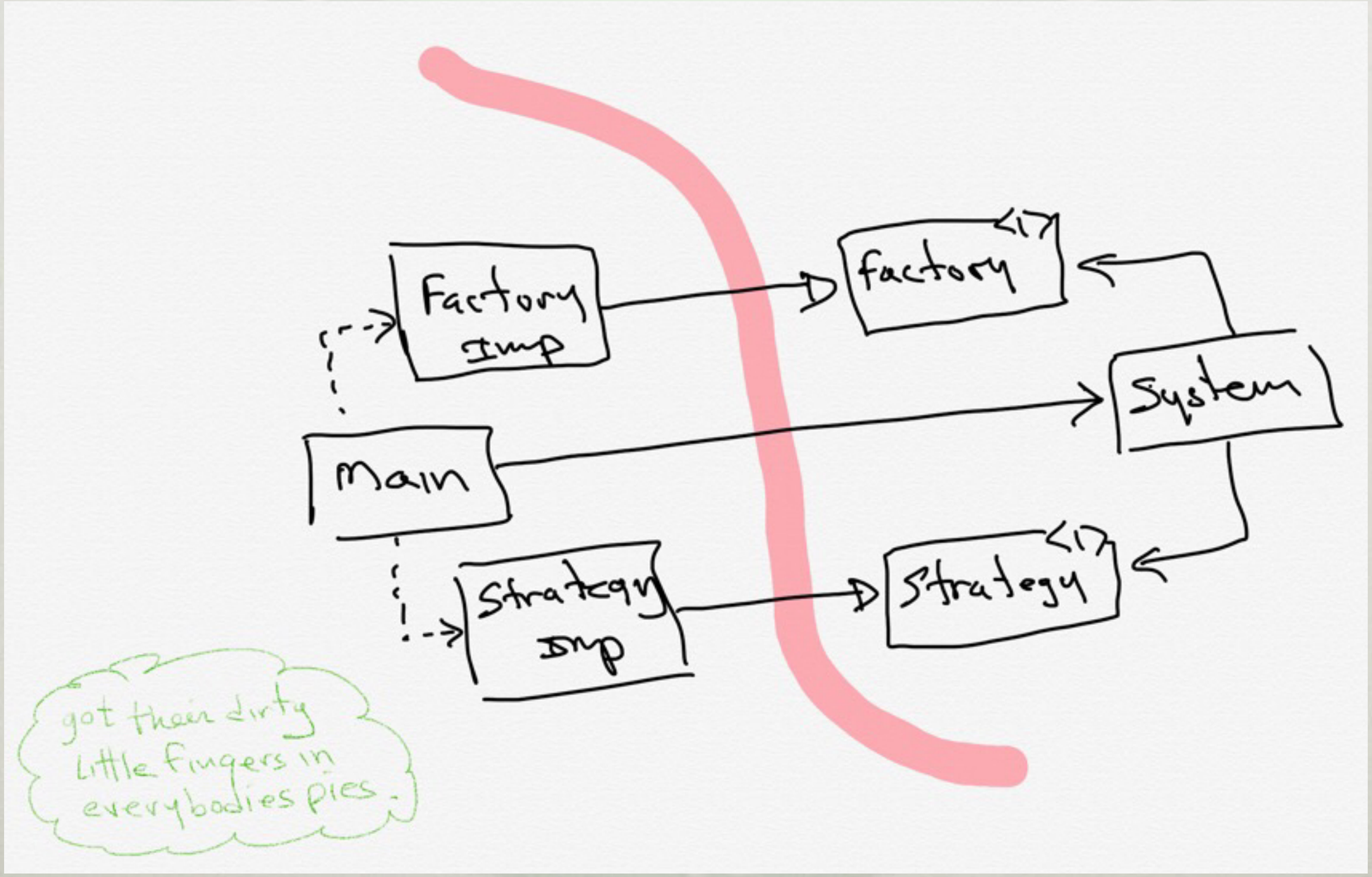


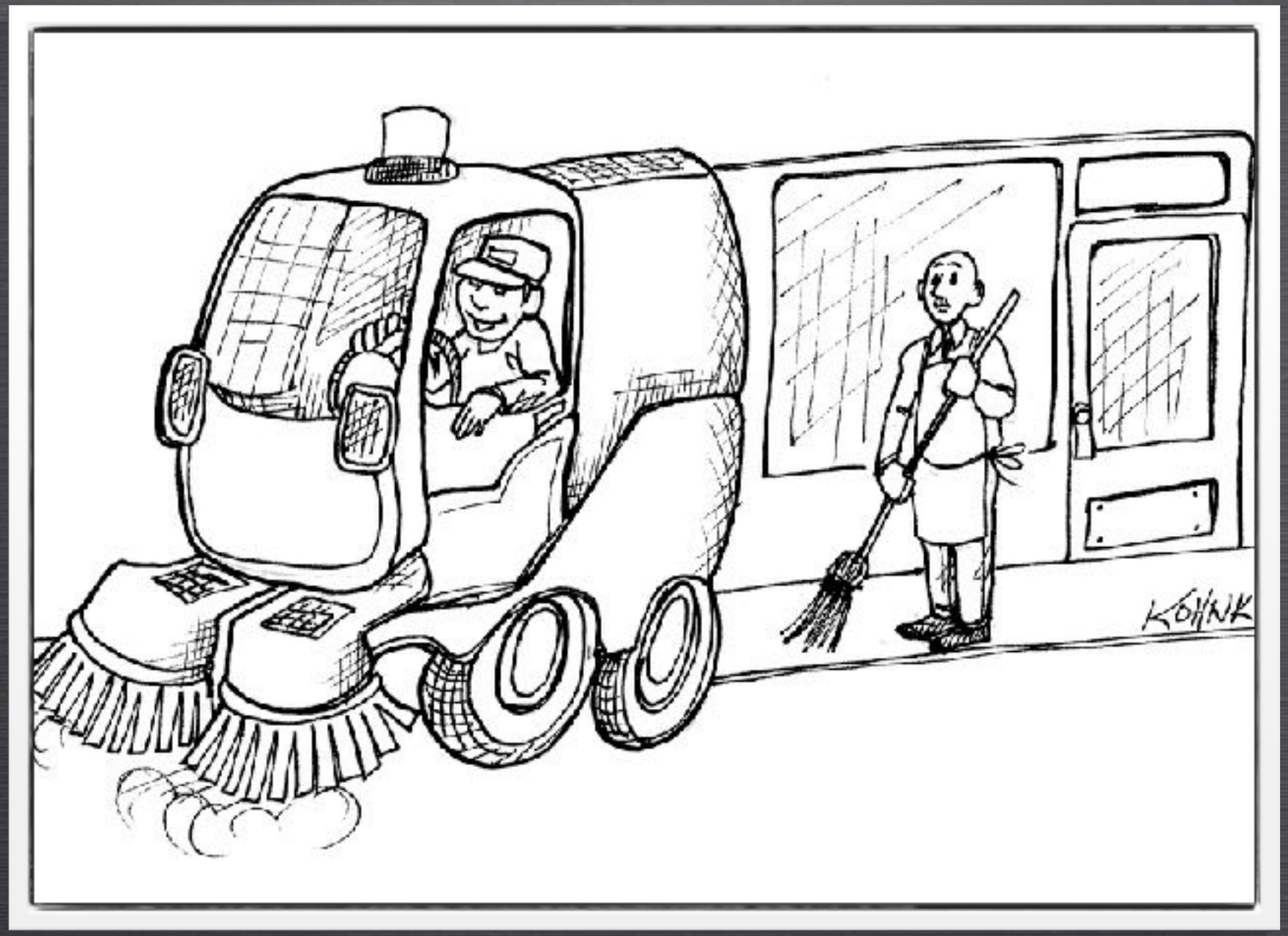
Streams within Streams
μservice MMUD



The Main Component

The dirtiest component of all.





Services great and small

Service Architecture

- Services are not an architecture, they are an implementation.
- True of μservices too.
- Just fancy ways to call functions.

The Decoupling Fallacy

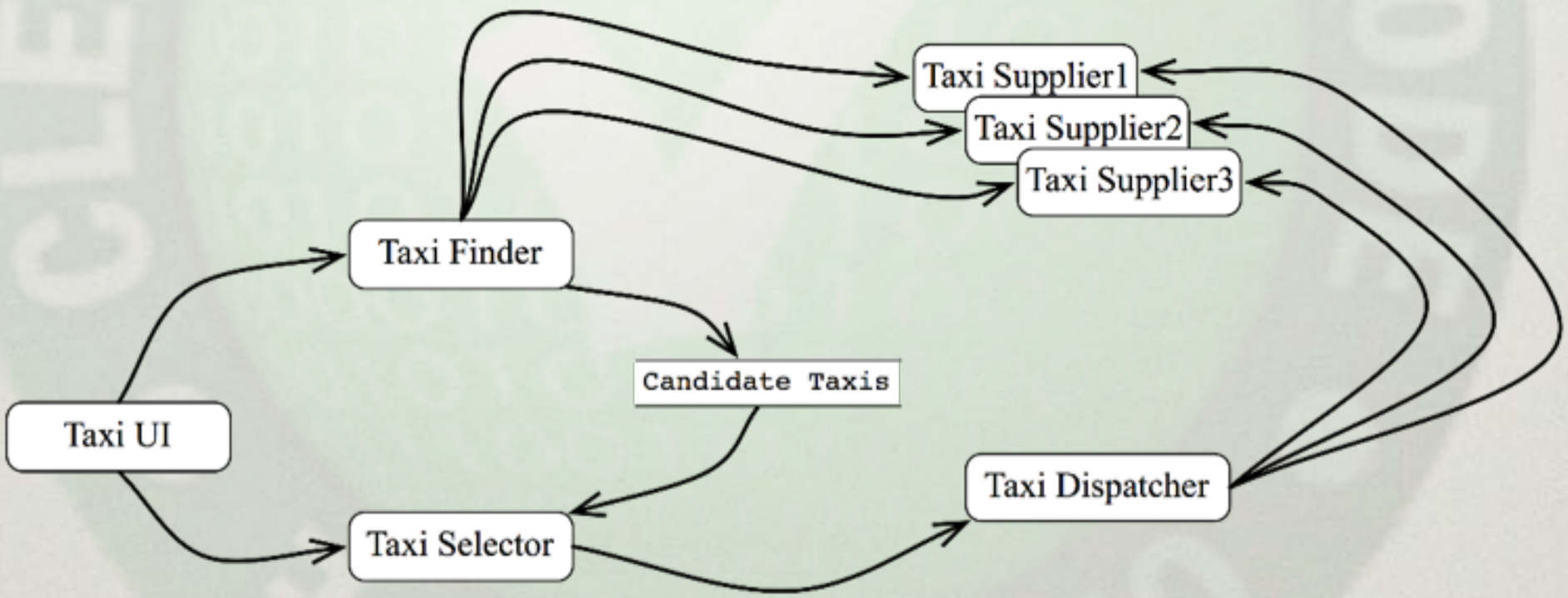
- Decoupling is often touted as an advantage.
- Little true decoupling, because they share interfaces and data.
- A change to the data or interface in one, forces changes in others.
  - Even if they use REST.

The Independent Deployment Fallacy

- Another supposed benefit is independent deployment and scalability.
- But this depends upon decoupling…
- Services *can* be decoupled, just like functions. But this has to be *designed*.

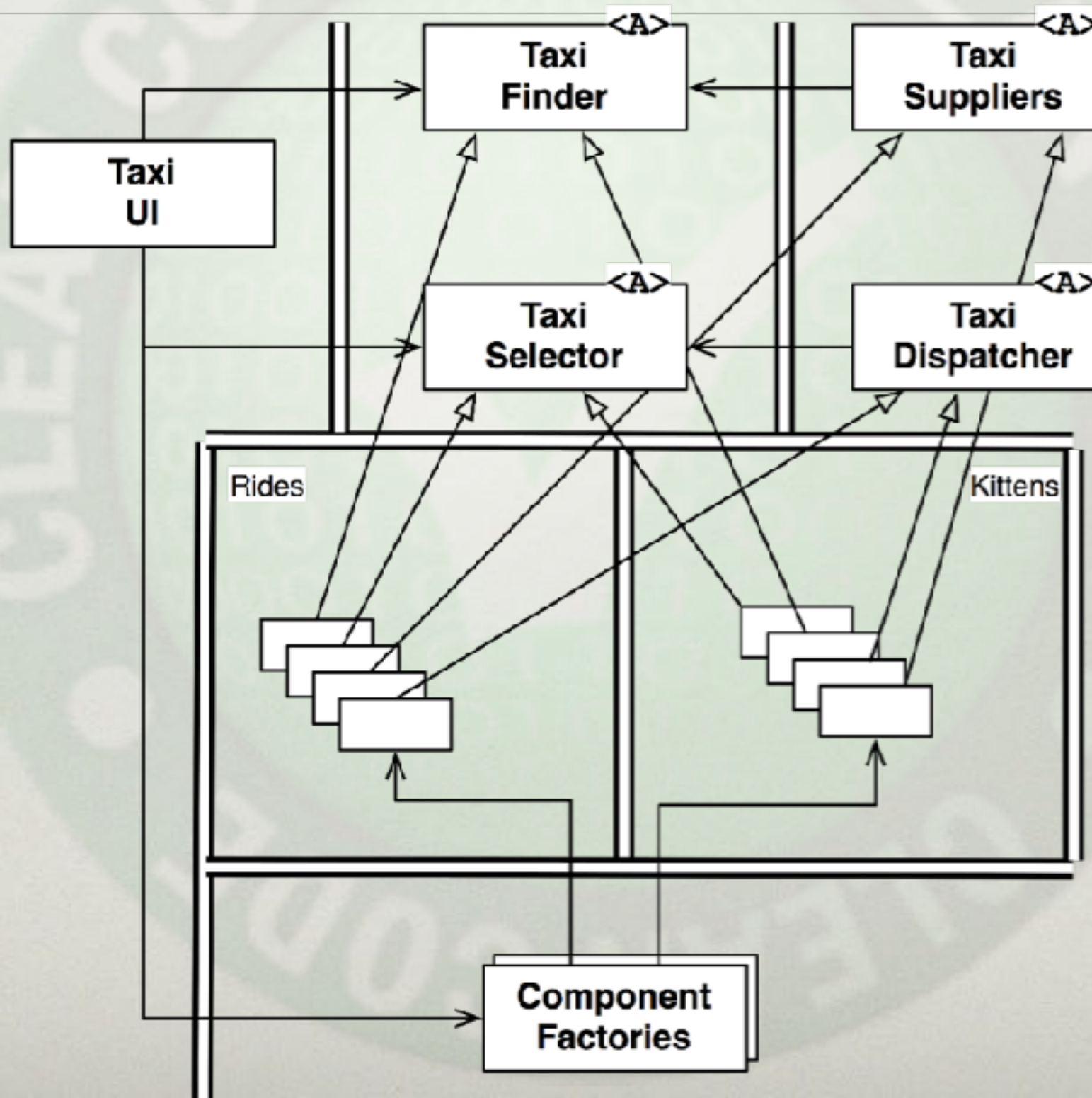
The "Kitty" problem

µService implementation of Taxi Dispatch App.

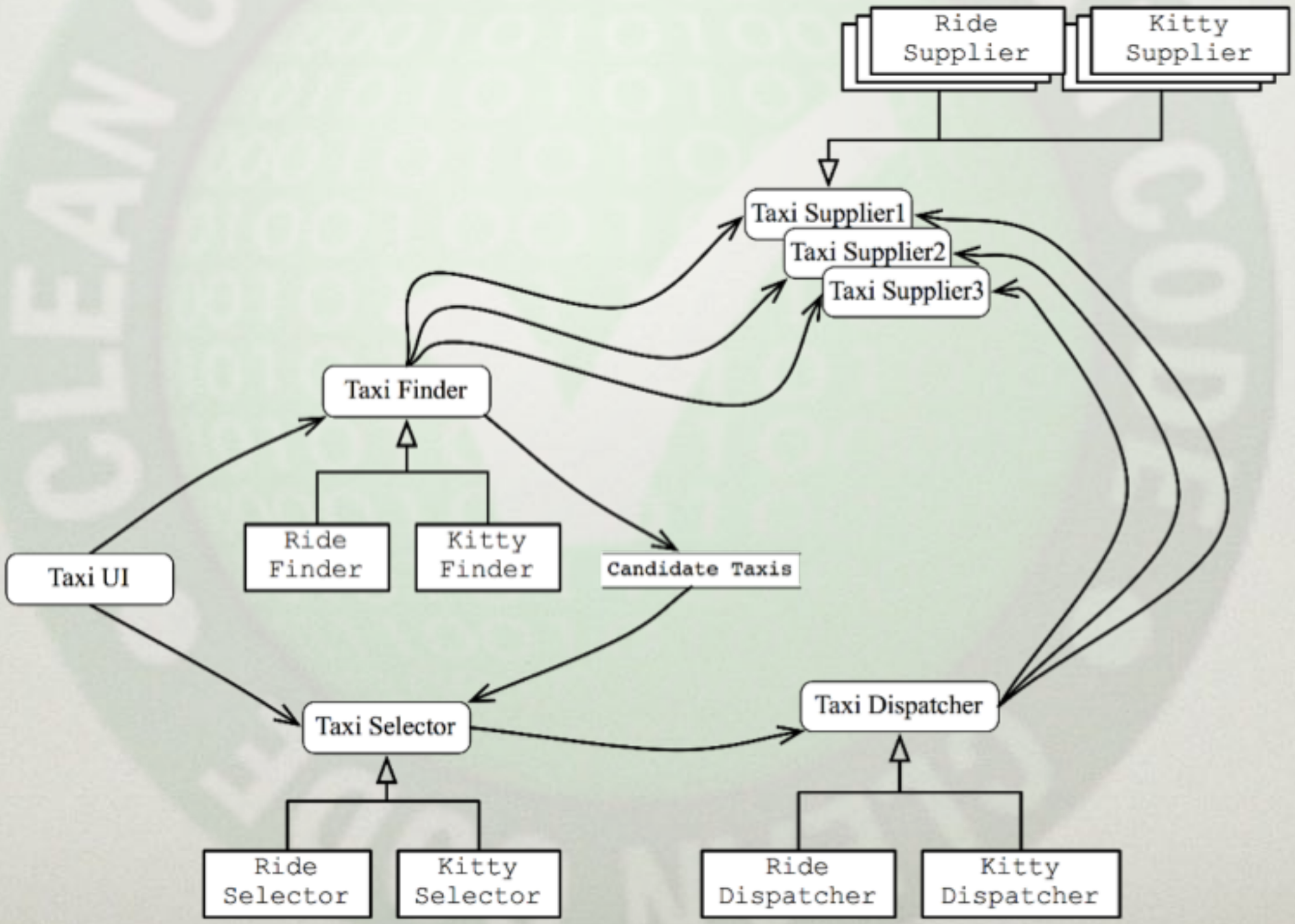


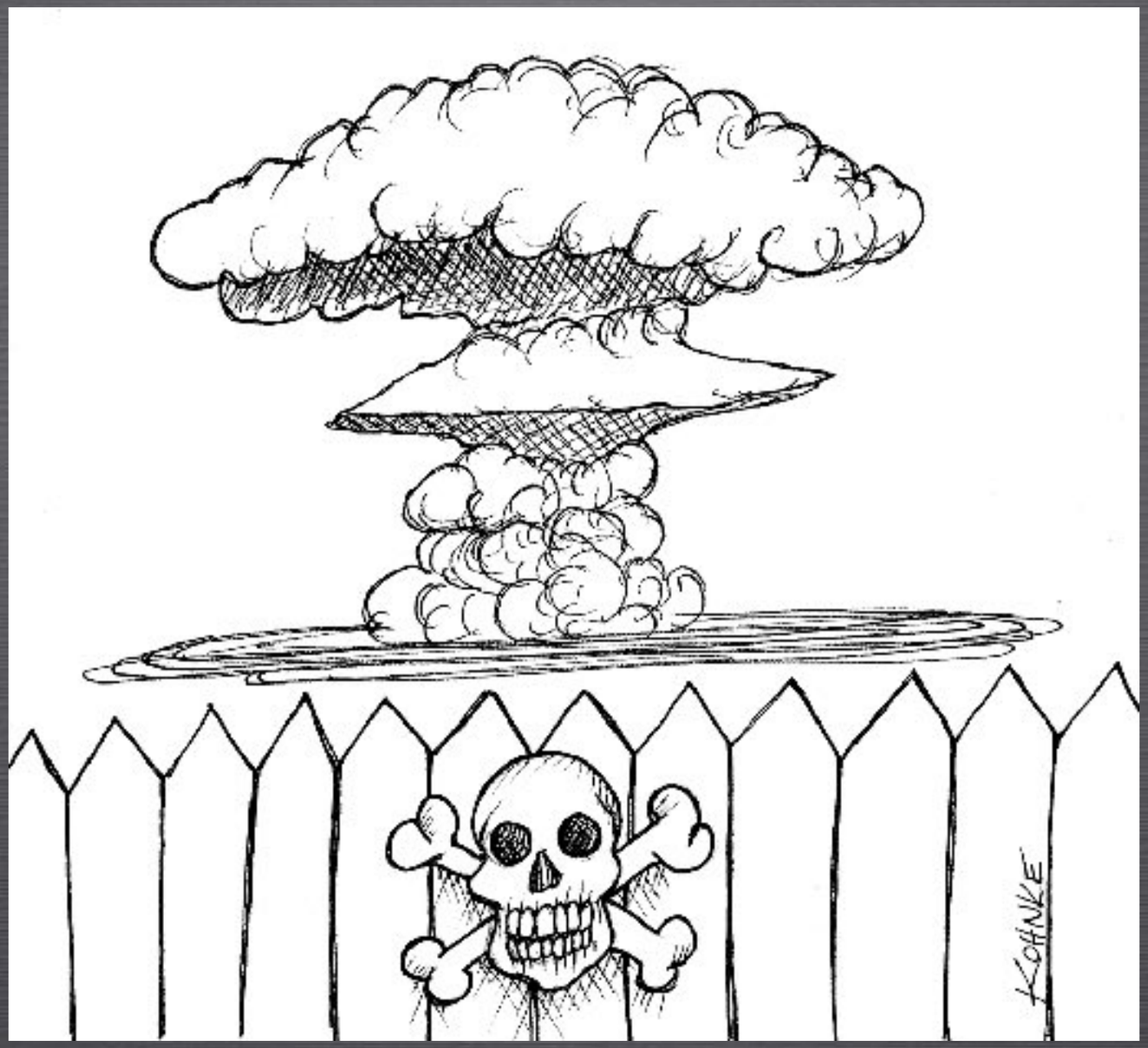
Deliver kittens to users.

Objects to the Rescue



Component based Services





The Test Boundary

Tests as System Components

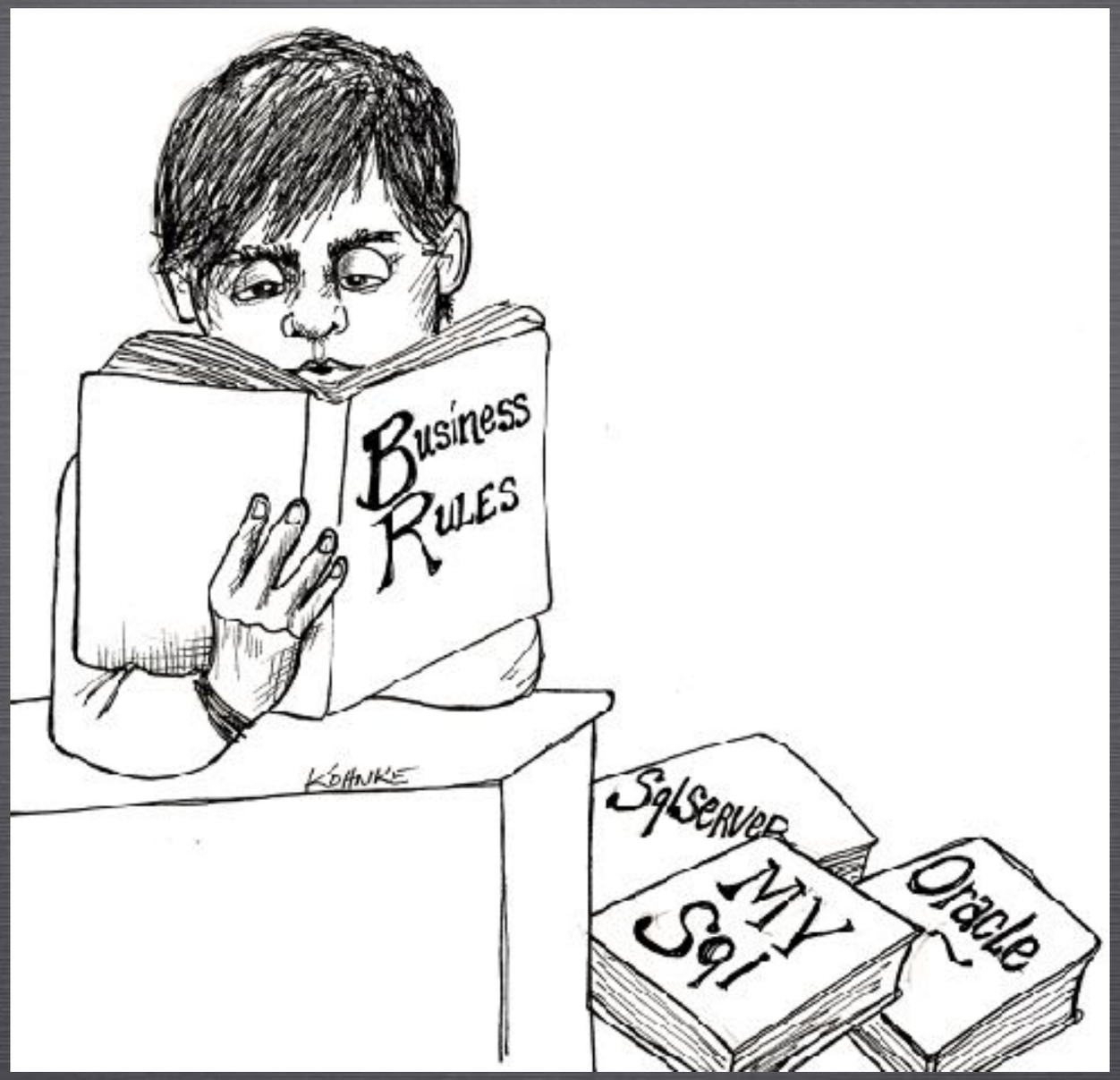
- Tests are designed.
- Tests follow the Dependency Rule
- Tests are independently deployable
- Tests are independently developable
- Tests are the most isolated components

Design for Testability

- Tests are part of the system, and need to be designed.
- Decoupling is, as always, the issue.
- The Fragile Test Problem.
- Fragile tests make the system rigid.

The Testing API

- Specific System API for tests.
  - Superpowers.
  - Avoid Structural Coupling of Tests
  - Security issues.



The Database is a Detail

The Origin of Databases

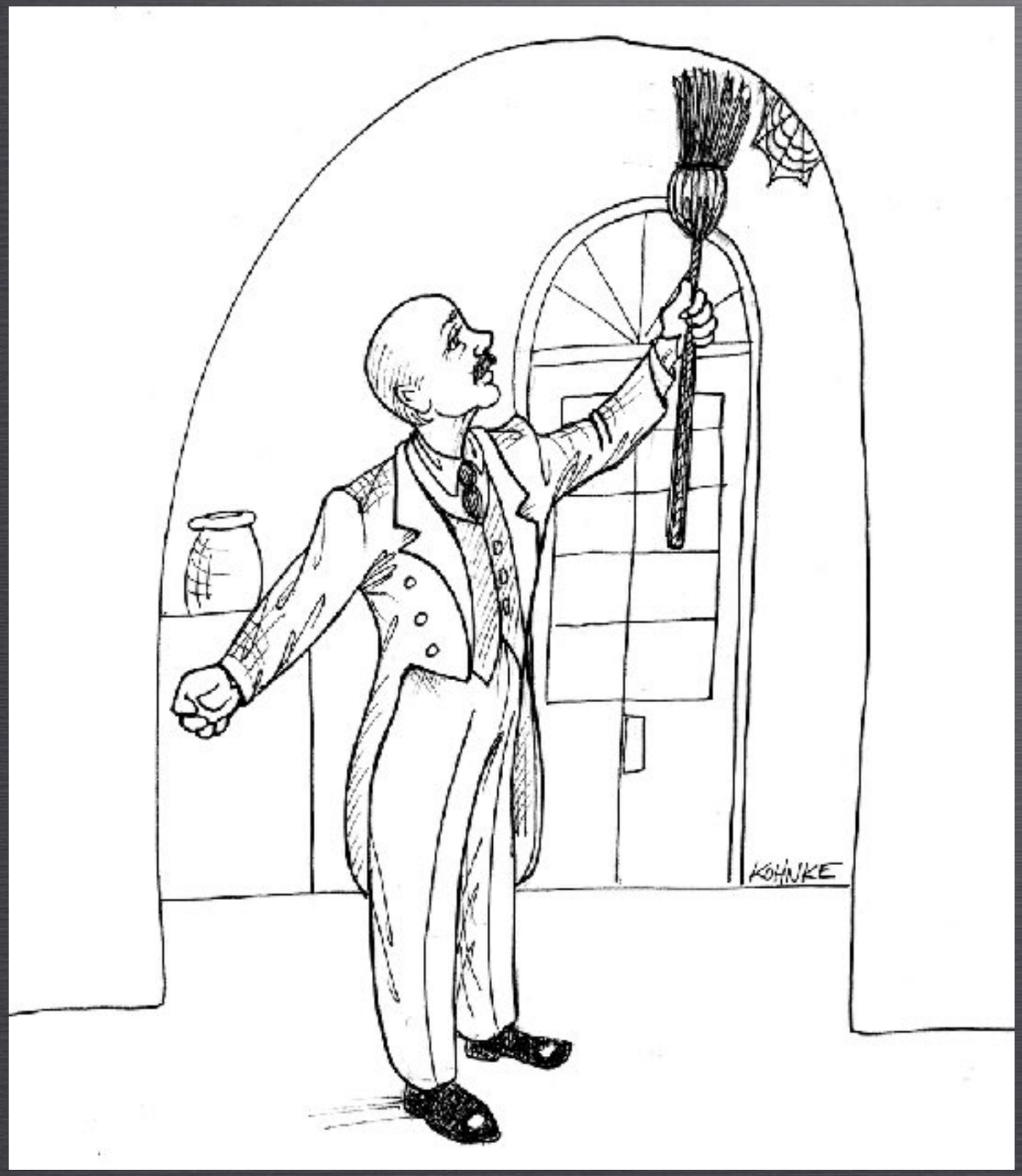
Memorex Mark XI Disc Pack 1970's about 200 Mb (IBM3336-11) PDP-10

What if there were no Disks?

- Bye bye disks!
- Without disks would you organize data into relational tables?
- Or would you use lists, binary trees, and hash maps?
- What if we had a virtually infinite amount of *persistent* RAM.

Details

- Disks are details because they are going away.
- Relational databases are details because we won't need them.
- Yes, relational algebra is wonderful but does not require database systems.
- And we really don't need SQL. !!!!!



The Web is a Detail

The Web Changed Everything.

- Except that it didn't actually change anything.
- The WEB is an I/O device.
- And we like device independence.

The Endless Pendulum

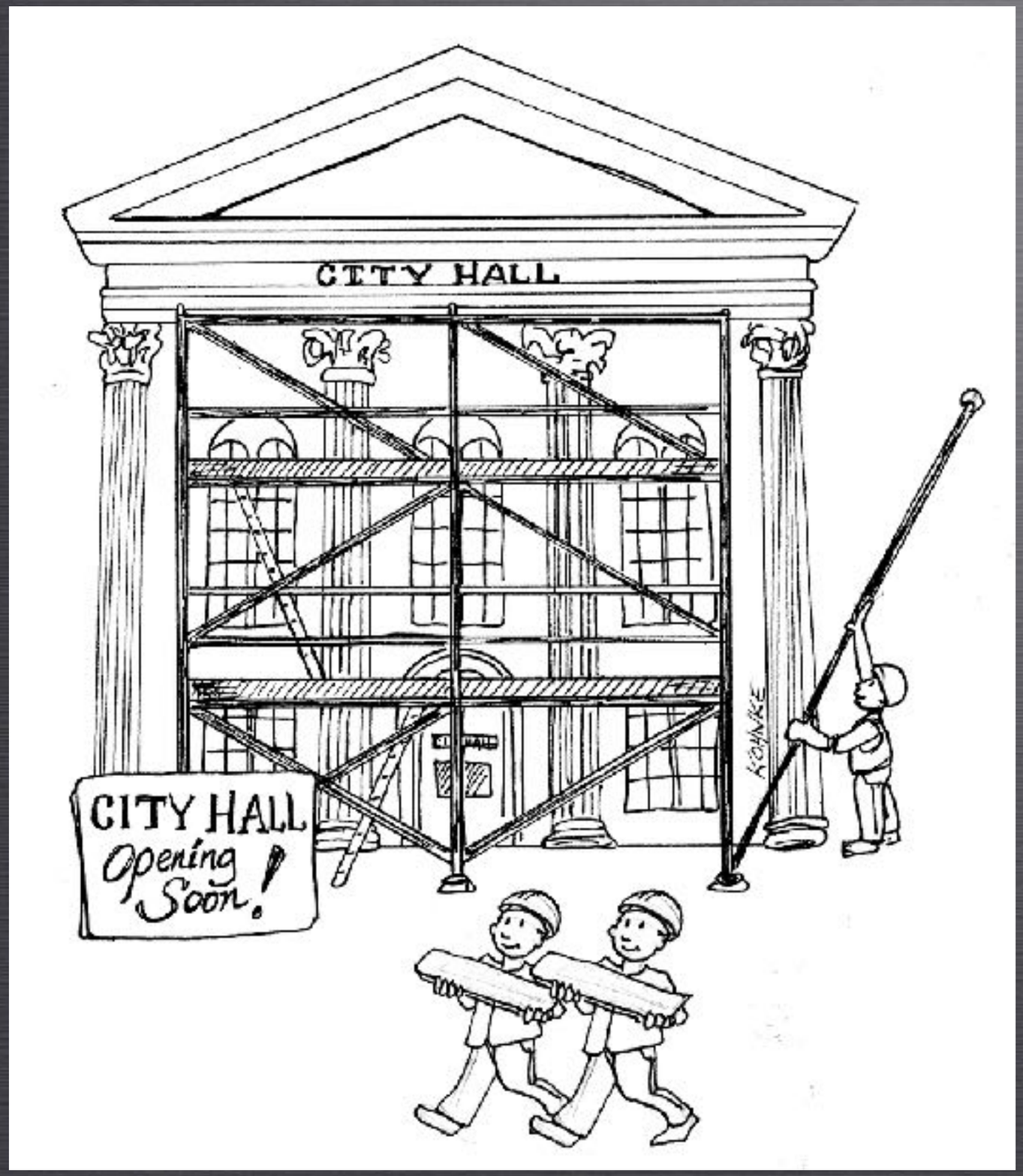
- Batch
- 3270
- Text terminals
- HTML
- Applets
- Web 2.0
- GTK

- AJAX
- Angular
- …

We just can't make up our minds about where we want the intelligence.

Independence

- The WEB is an unstable IO device.
- Changing all the time.
- What do we do with unstable IO devices?
- We decouple our code from them.
- Because they are details.



Frameworks are Details

Framework Authors

The Asymmetric Marriage

The Risks

- Framework architectures are seldom clean.
- Early help gradually transitions to impediment as complexity increases.
- Framework may evolve away from your need.
- New and better frameworks appear all the time.

Solution

- Don’t marry the framework right away — date it for awhile.
- Use it. Just don’t couple to it. Keep it at arm’s length, behind a boundary.
- Don’t derive from framework base classes.
- Don’t let framework artifacts into your business rules.
- Try to get the milk without buying the cow.