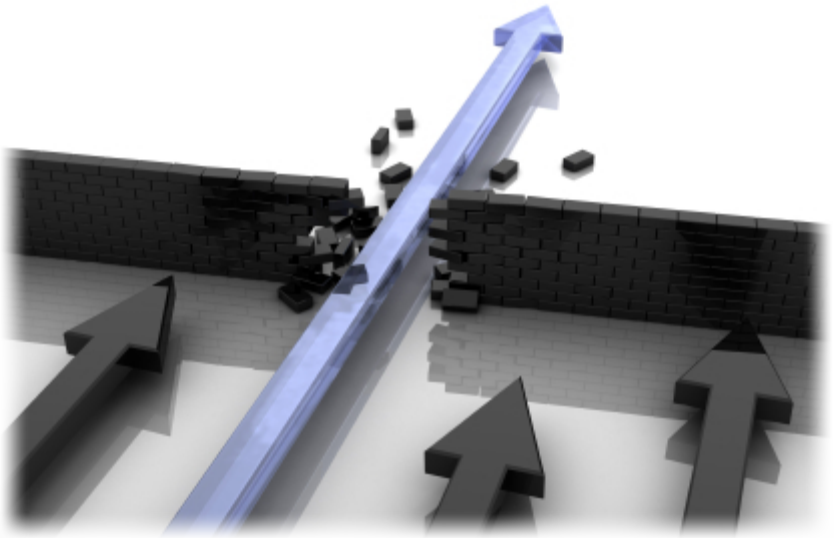
emergent design & evolutionary architecture



NEAL FORD software architect / meme wrangler

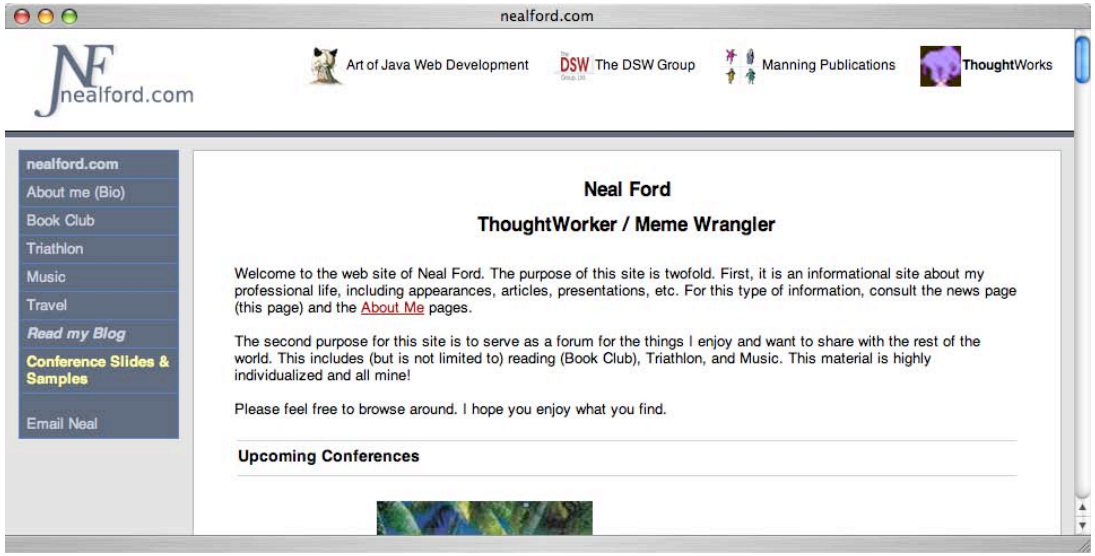
ThoughtWorks

nford@thoughtworks.com
3003 Summit Boulevard, Atlanta, GA 30319
www.nealford.com
www.thoughtworks.com
memeagora.blogspot.com

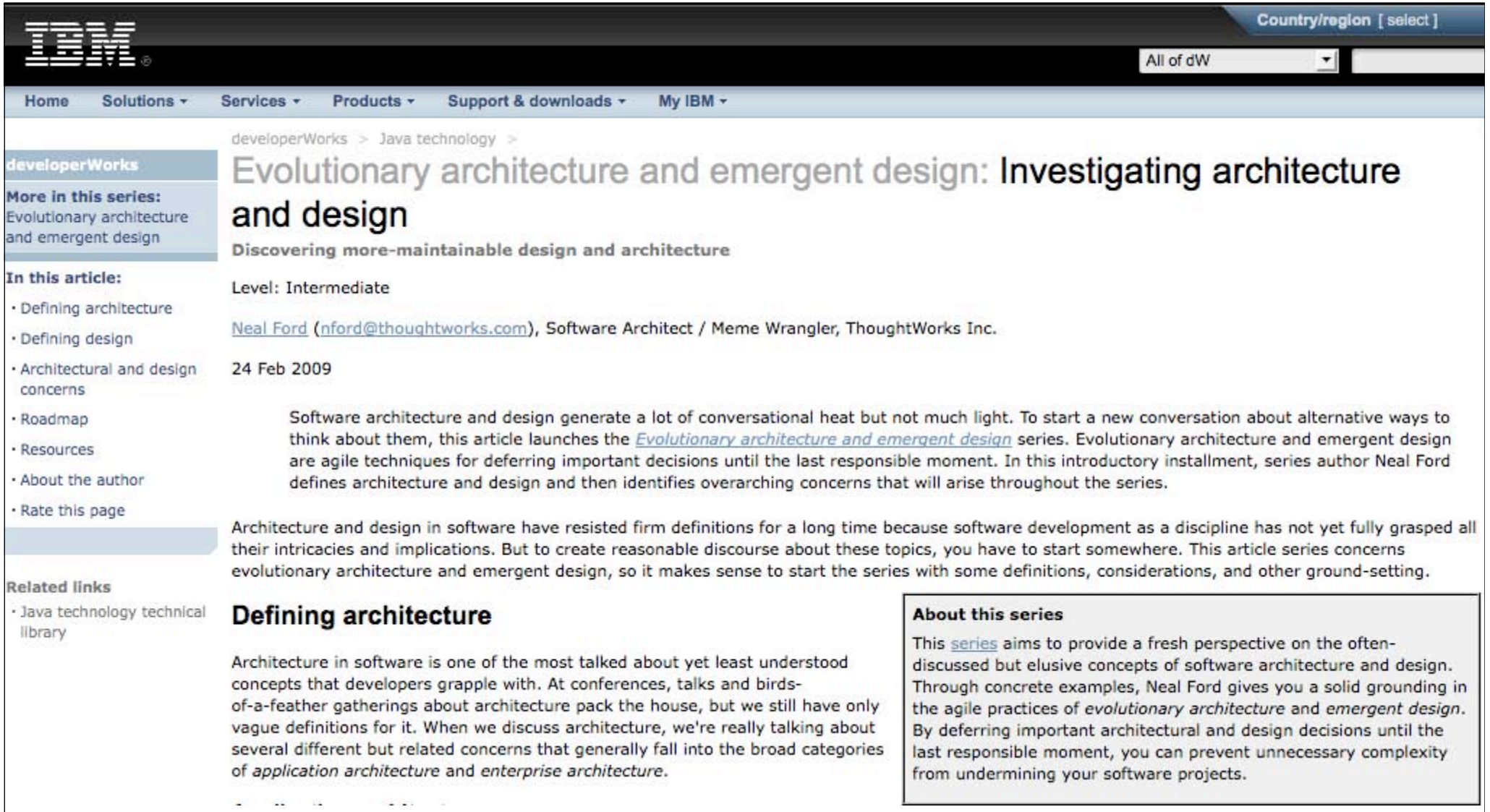
housekeeping

ask questions anytime

download slides from nealford.com



download samples from `github.com/nealford`



IBM

Country/region [select]

All of dW

Home Solutions Services Products Support & downloads My IBM

developerWorks > Java technology >

developerWorks

More in this series:
Evolutionary architecture and emergent design

In this article:

- Defining architecture
- Defining design
- Architectural and design concerns
- Roadmap
- Resources
- About the author
- Rate this page

Related links

- Java technology technical library

Evolutionary architecture and emergent design: Investigating architecture and design

Discovering more-maintainable design and architecture

Level: Intermediate

Neal Ford (nford@thoughtworks.com), Software Architect / Meme Wrangler, ThoughtWorks Inc.

24 Feb 2009

Software architecture and design generate a lot of conversational heat but not much light. To start a new conversation about alternative ways to think about them, this article launches the *Evolutionary architecture and emergent design* series. Evolutionary architecture and emergent design are agile techniques for deferring important decisions until the last responsible moment. In this introductory installment, series author Neal Ford defines architecture and design and then identifies overarching concerns that will arise throughout the series.

Architecture and design in software have resisted firm definitions for a long time because software development as a discipline has not yet fully grasped all their intricacies and implications. But to create reasonable discourse about these topics, you have to start somewhere. This article series concerns evolutionary architecture and emergent design, so it makes sense to start the series with some definitions, considerations, and other ground-setting.

Defining architecture

Architecture in software is one of the most talked about yet least understood concepts that developers grapple with. At conferences, talks and birds-of-a-feather gatherings about architecture pack the house, but we still have only vague definitions for it. When we discuss architecture, we're really talking about several different but related concerns that generally fall into the broad categories of *application architecture* and *enterprise architecture*.

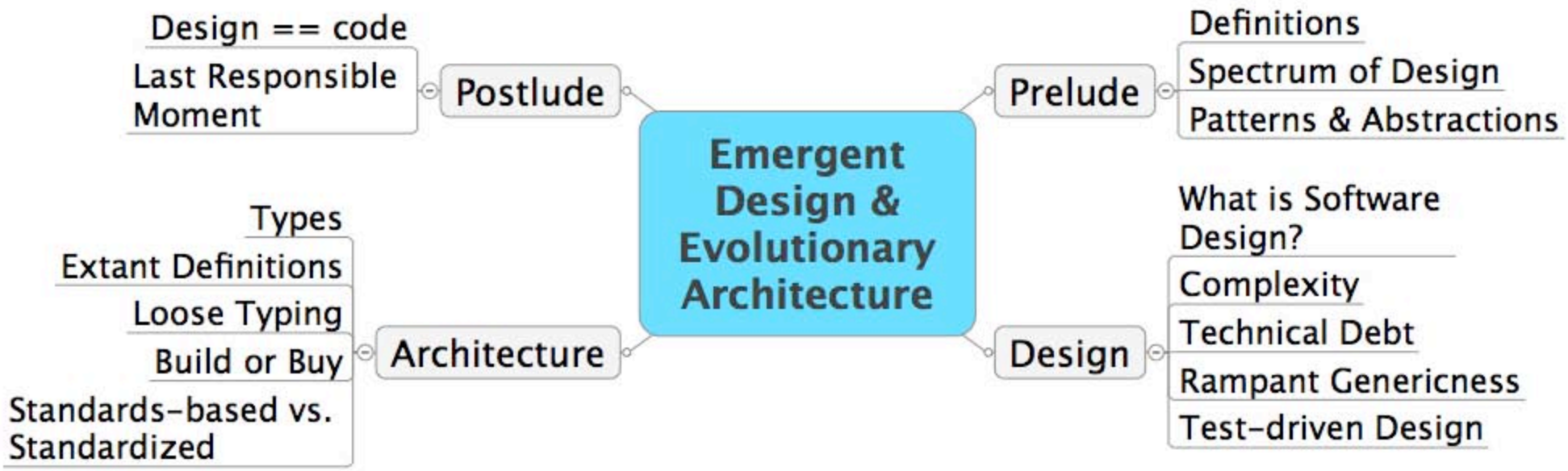
About this series

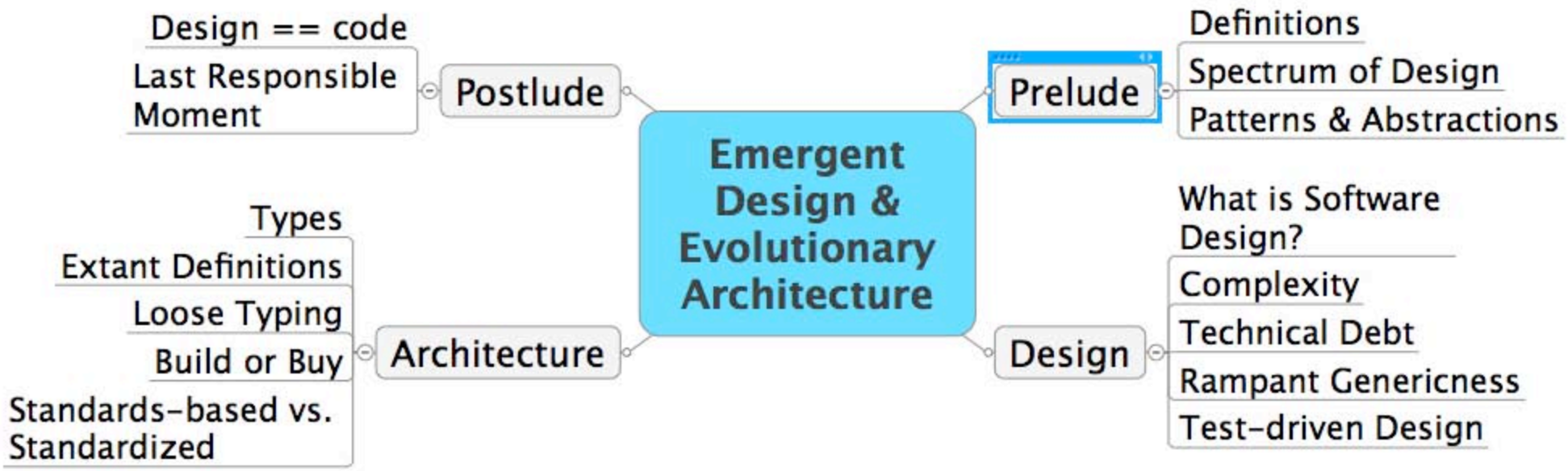
This series aims to provide a fresh perspective on the often-discussed but elusive concepts of software architecture and design. Through concrete examples, Neal Ford gives you a solid grounding in the agile practices of *evolutionary architecture* and *emergent design*. By deferring important architectural and design decisions until the last responsible moment, you can prevent unnecessary complexity from undermining your software projects.

http://www.ibm.com/developerworks/java/library/j-eaed1/index.html?S_TACT=105AGX02&S_CMP=EDU

http://tinyurl.com/nf-ead

agenda


Emergent Design & Evolutionary Architecture
Postlude
Design == code
Last Responsible Moment
Prelude
Definitions
Spectrum of Design
Patterns & Abstractions
Architecture
Types
Extant Definitions
Loose Typing
Build or Buy
Standards-based vs. Standardized
Design
What is Software Design?
Complexity
Technical Debt
Rampant Genericness
Test-driven Design

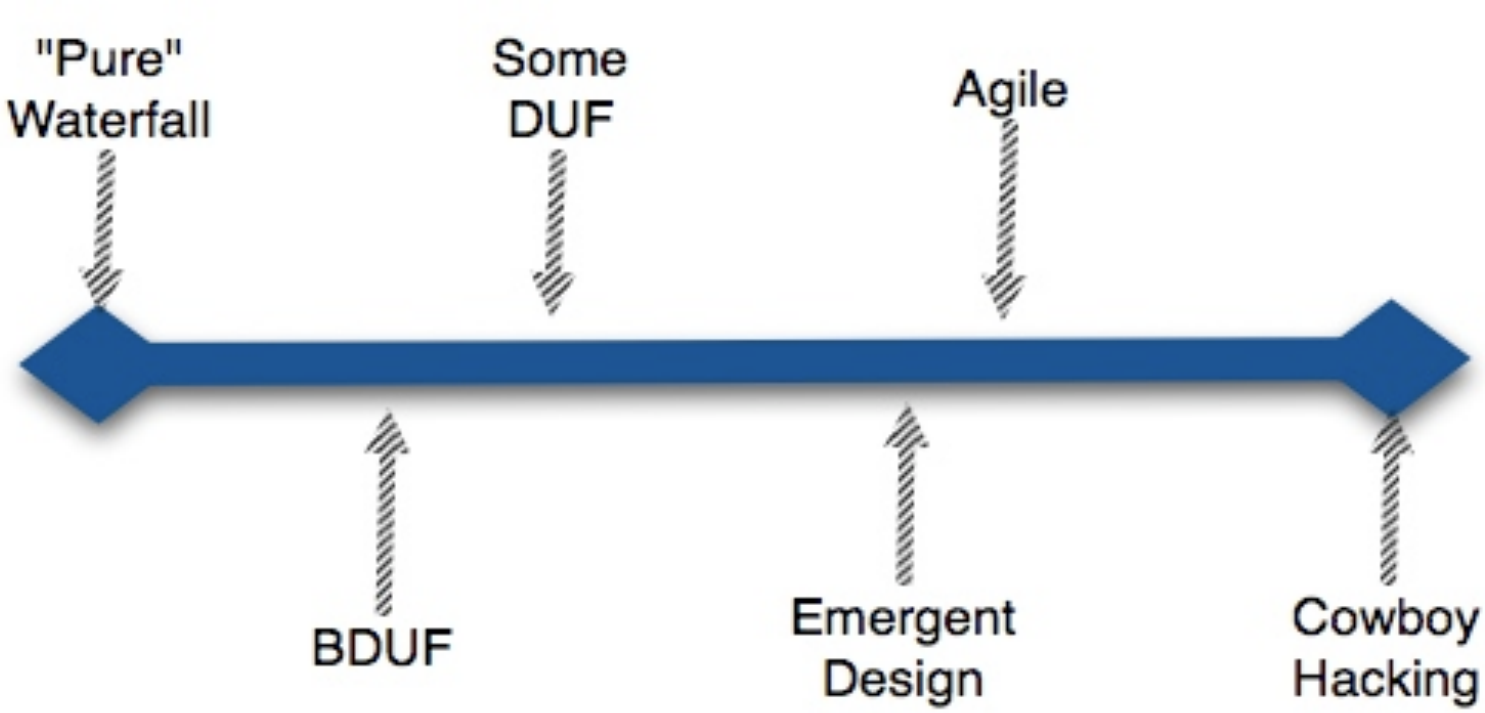

Emergent Design & Evolutionary Architecture
Postlude
Design == code
Last Responsible Moment
Prelude
Definitions
Spectrum of Design
Patterns & Abstractions
Architecture
Types
Extant Definitions
Loose Typing
Build or Buy
Standards-based vs. Standardized
Design
What is Software Design?
Complexity
Technical Debt
Rampant Genericness
Test-driven Design

things to think about

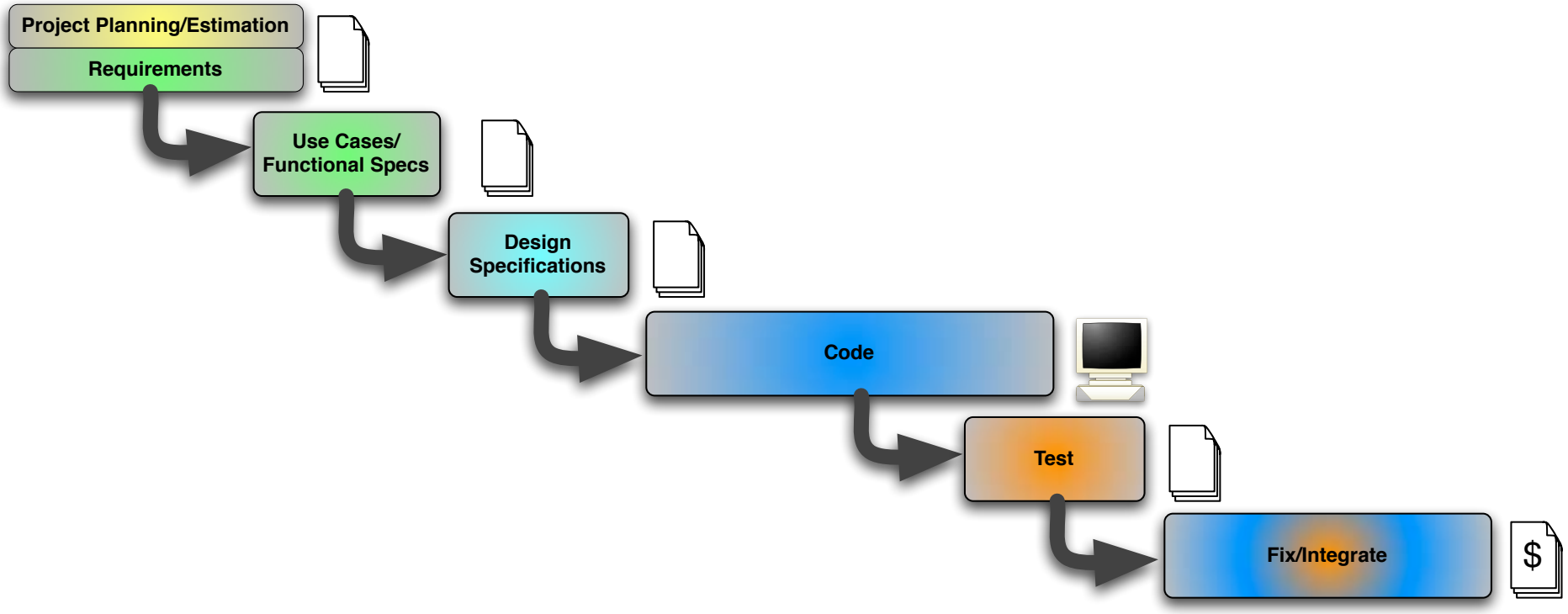
Emergent, a.
[L. emergens, p. pr. of emergere.]

1. Rising or emerging out of a fluid or anything that covers or conceals; issuing; coming to light. [1913 Webster]
2. Suddenly appearing; arising unexpectedly; calling for prompt action; urgent. [1913 Webster]

spectrum of design

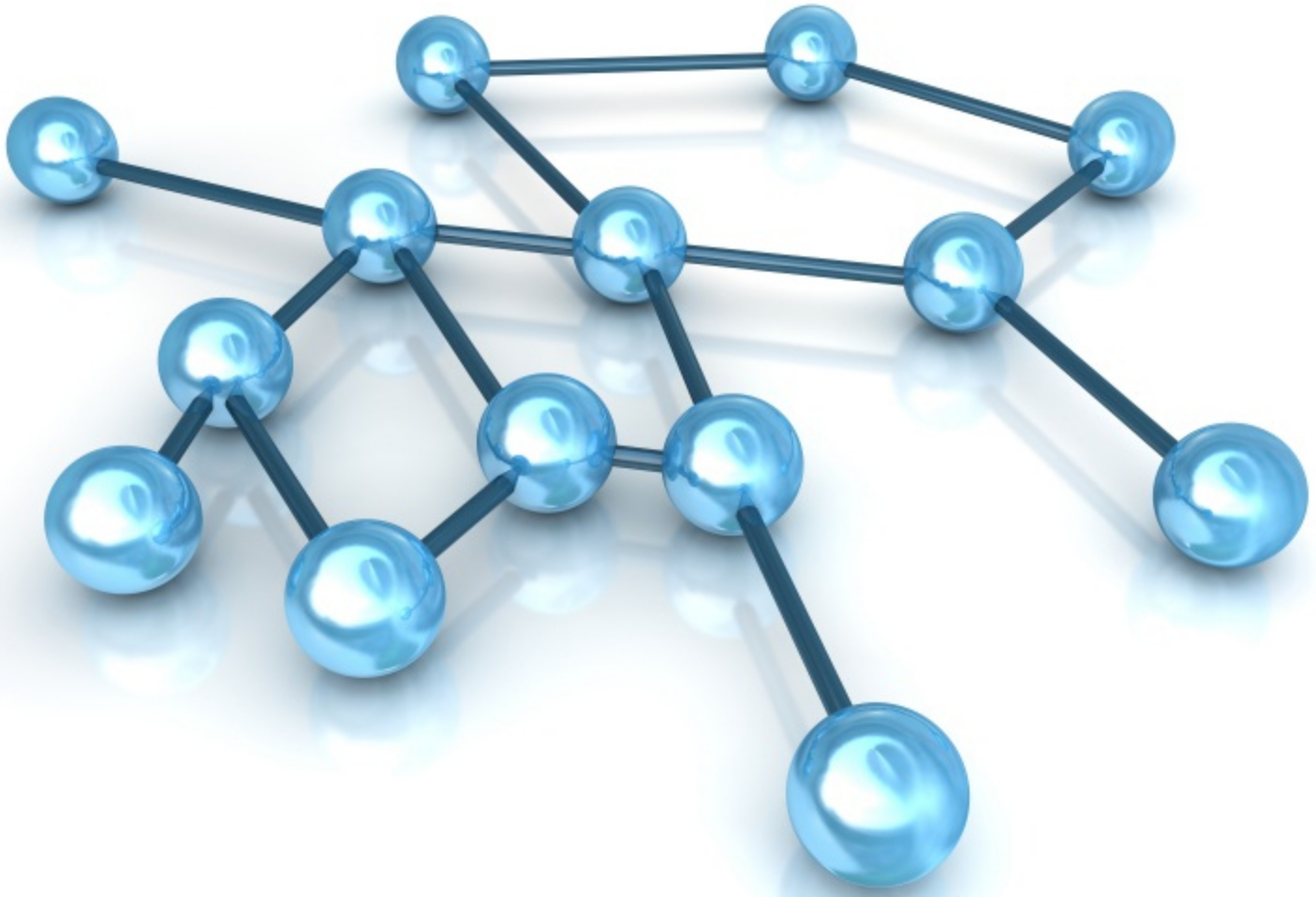


big design up front




who thought this
was a good idea?

emerging design



finding abstractions & patterns

Patterns

patterns == nomenclature

patterns describe effective abstractions

good abstractions disappear

the simpler the substrate, the easier the abstraction

abstractions leak

leaky abstractions

All non-trivial abstractions, to some degree, are leaky. joel spolsky

file system in java

javascript libraries

o/r mapping

ActiveRecord in ruby on rails !

abstracting too early

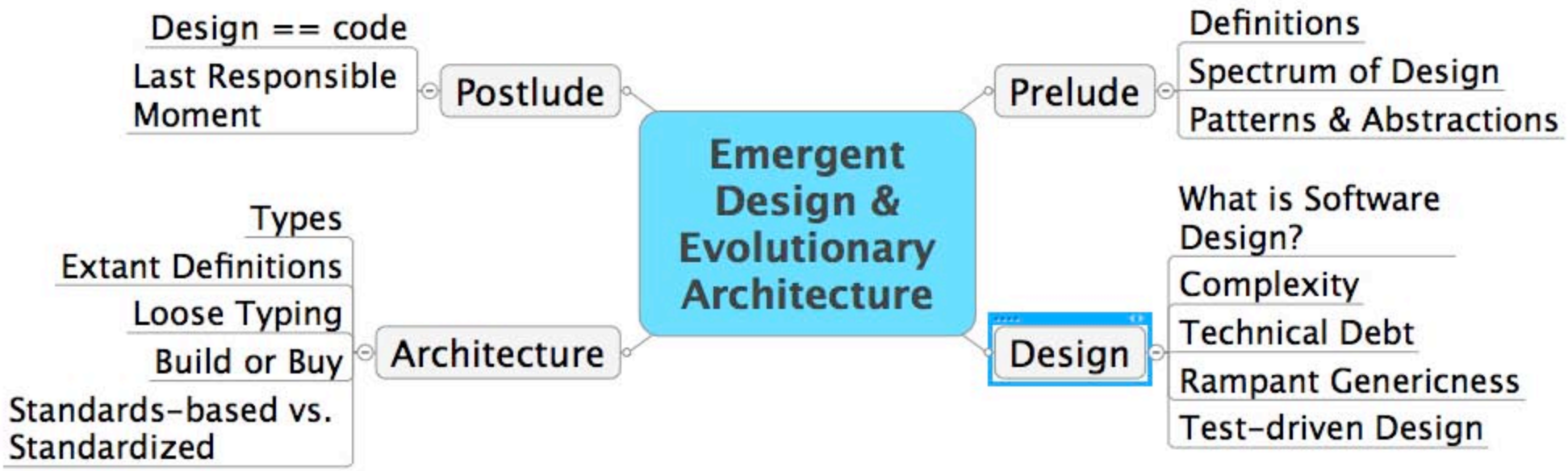
speculation without facts

yagni

business processes change radically and often

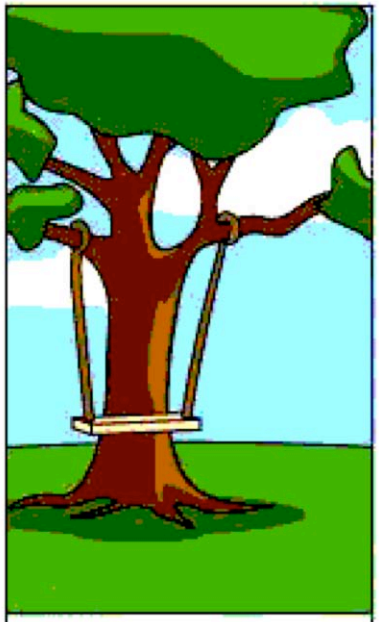
experience helps

spike solutions


Emergent Design & Evolutionary Architecture
Postlude
Design == code
Last Responsible Moment
Prelude
Definitions
Spectrum of Design
Patterns & Abstractions
Architecture
Types
Extant Definitions
Loose Typing
Build or Buy
Standards-based vs. Standardized
Design
What is Software Design?
Complexity
Technical Debt
Rampant Genericness
Test-driven Design

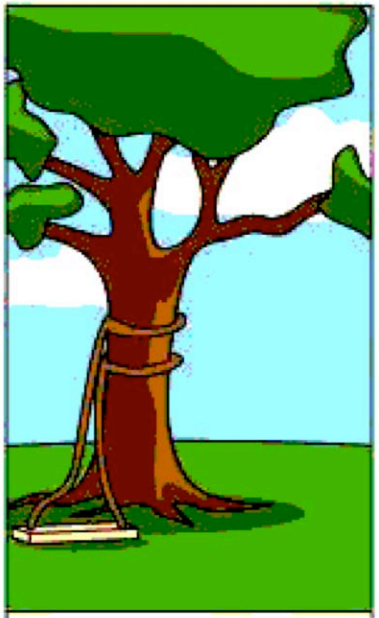

emergent
design

How the customer explained it

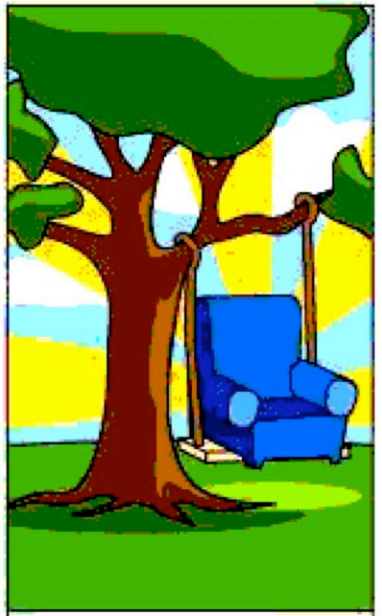


How the Project Leader understood it

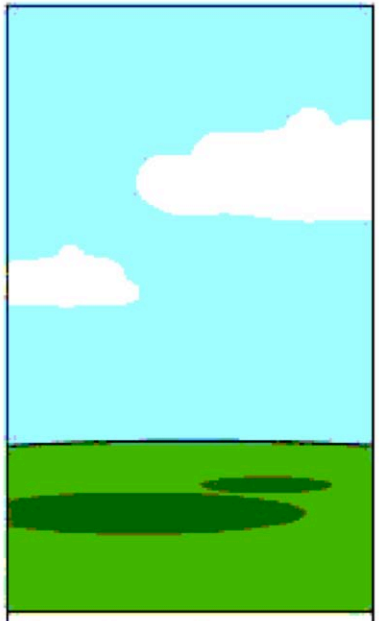
How the Analyst designed it



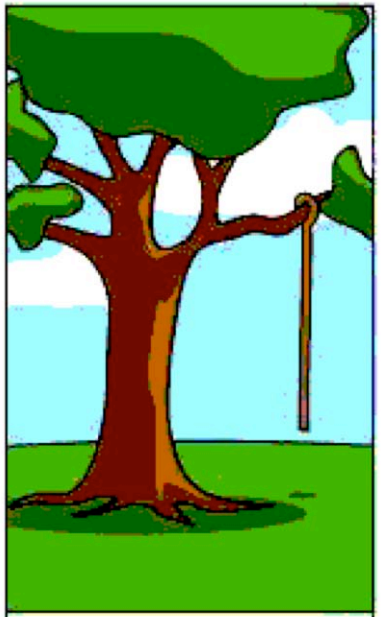
How the Programmer wrote it



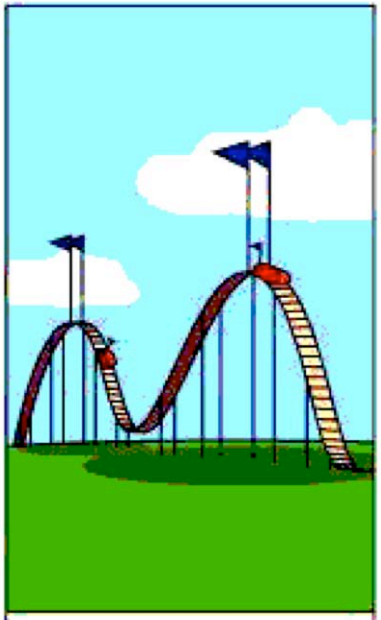
How the Business Consultant described it



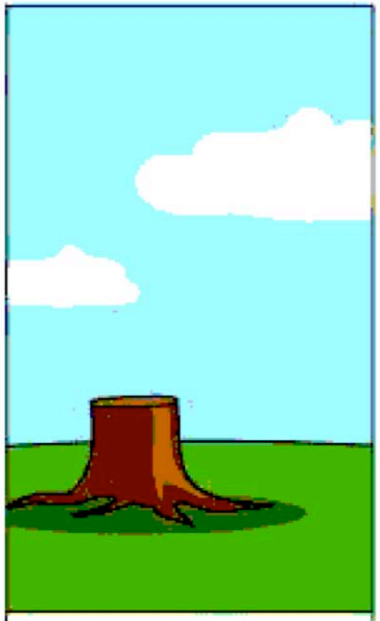
How the project was documented



What operations installed



How the customer was billed



How it was supported

What the customer really needed

“what is software design?”

Jack C. Reeves
fall 1992, c++ journal

http://www.developerdotstar.com/mag/articles/reeves_design.html

software “engineering”

“The final goal of any engineering activity is some type of documentation”

“When the design effort is complete, the design documentation is turned over to the manufacturing team.”

what is the design document in software?

the source code

source == design

"...software is cheap to build. It does not qualify as inexpensive; it is so cheap it is almost free".

manufacturing == build process

"...software design is easy to create, at least in the mechanical sense."

"Given that software designs are relatively easy to turn out, and essentially free to build, an unsurprising revelation is that software designs tend to be incredibly large and complex."

source == design

"...it is cheaper and simpler to just build the design and test it than to do anything else."

"The overwhelming problem with software development is that everything is part of the design process."

"Coding is design, testing and debugging are part of design, and what we typically call software design is still part of design."

"Software may be cheap to build, but it is incredibly expensive to design."

things that obscure emergent design

complexity

essential complexity

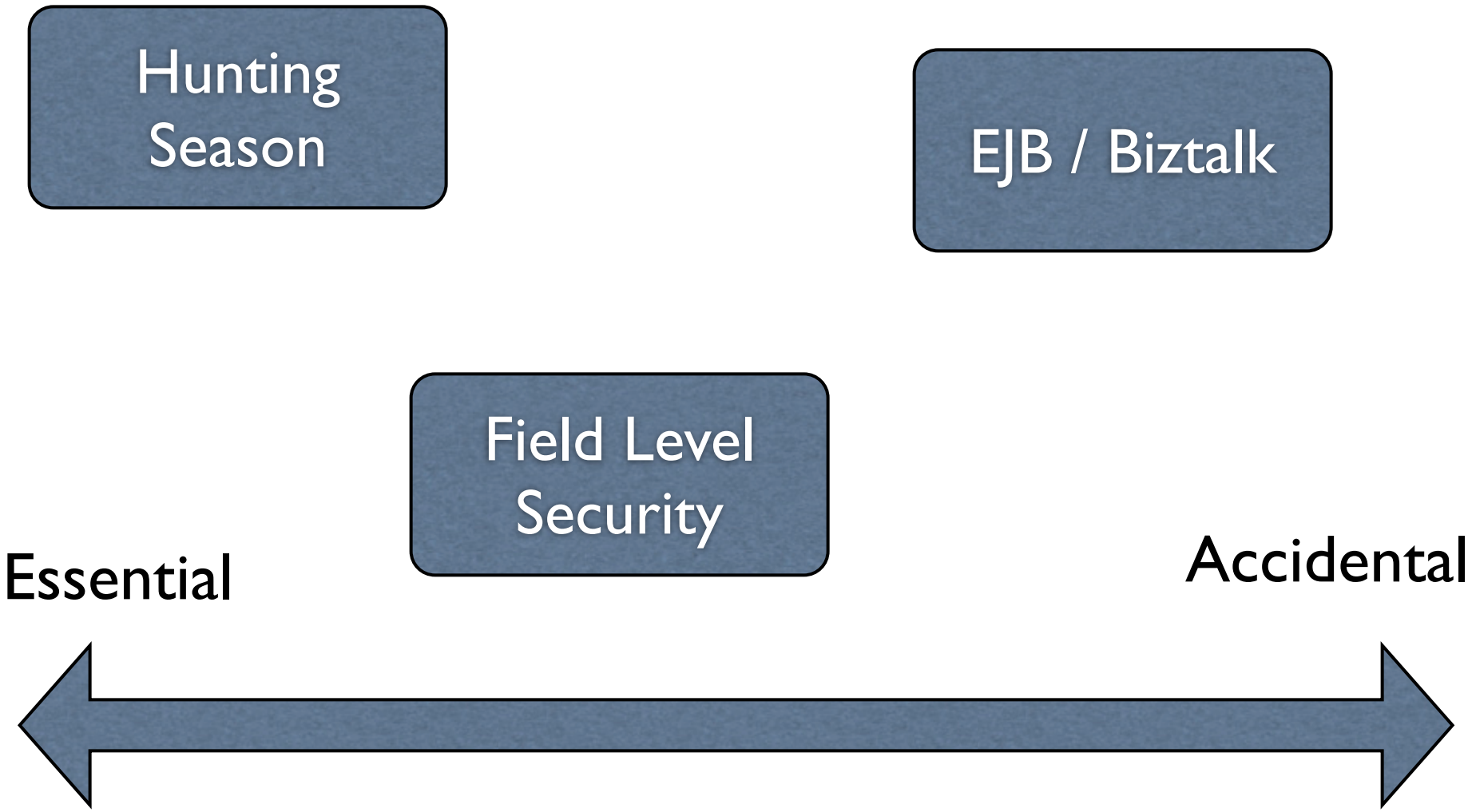
inherent complexity

accidental complexity

all the externally imposed ways that software becomes complex

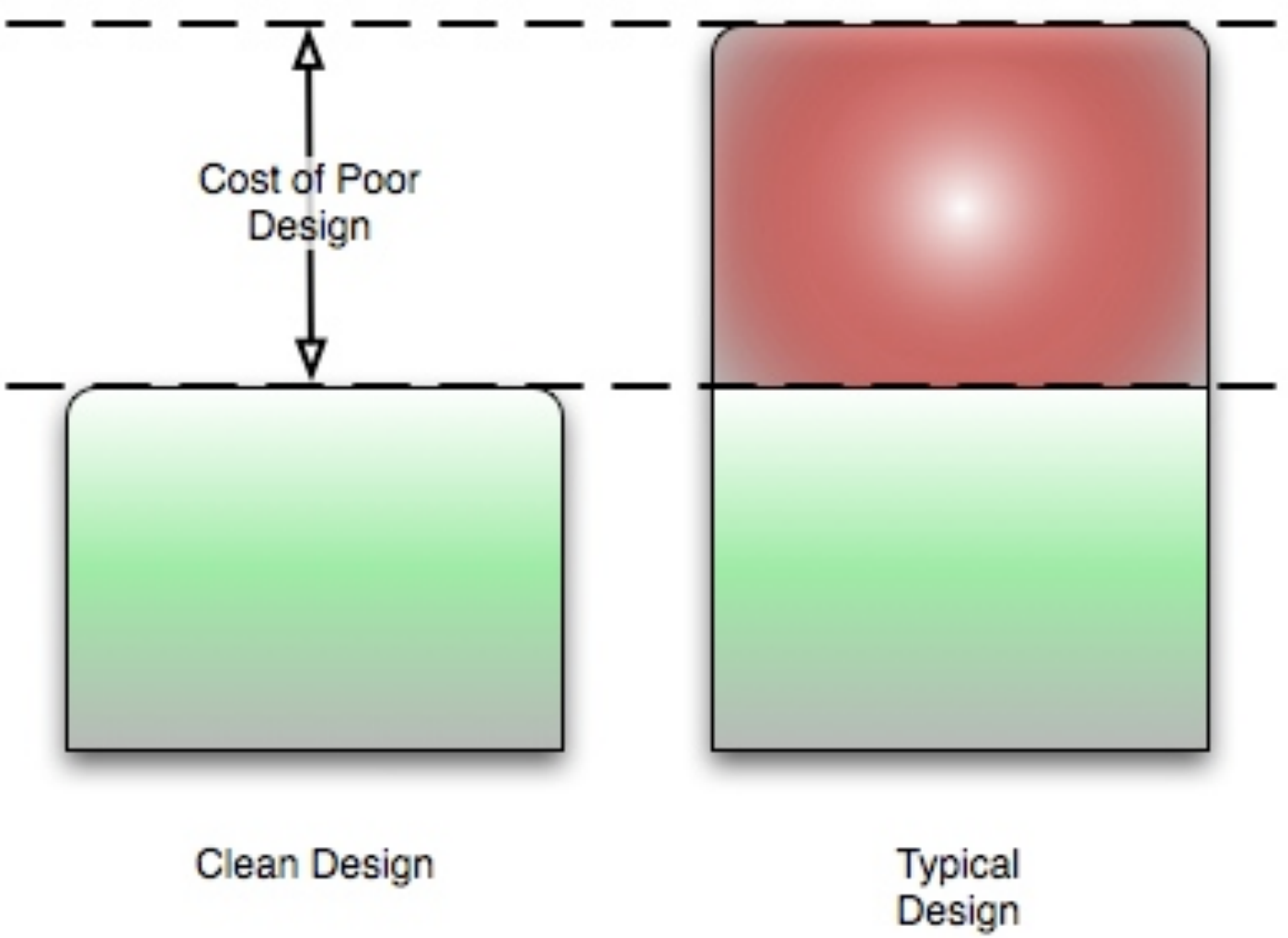
essential vs. accidental complexity

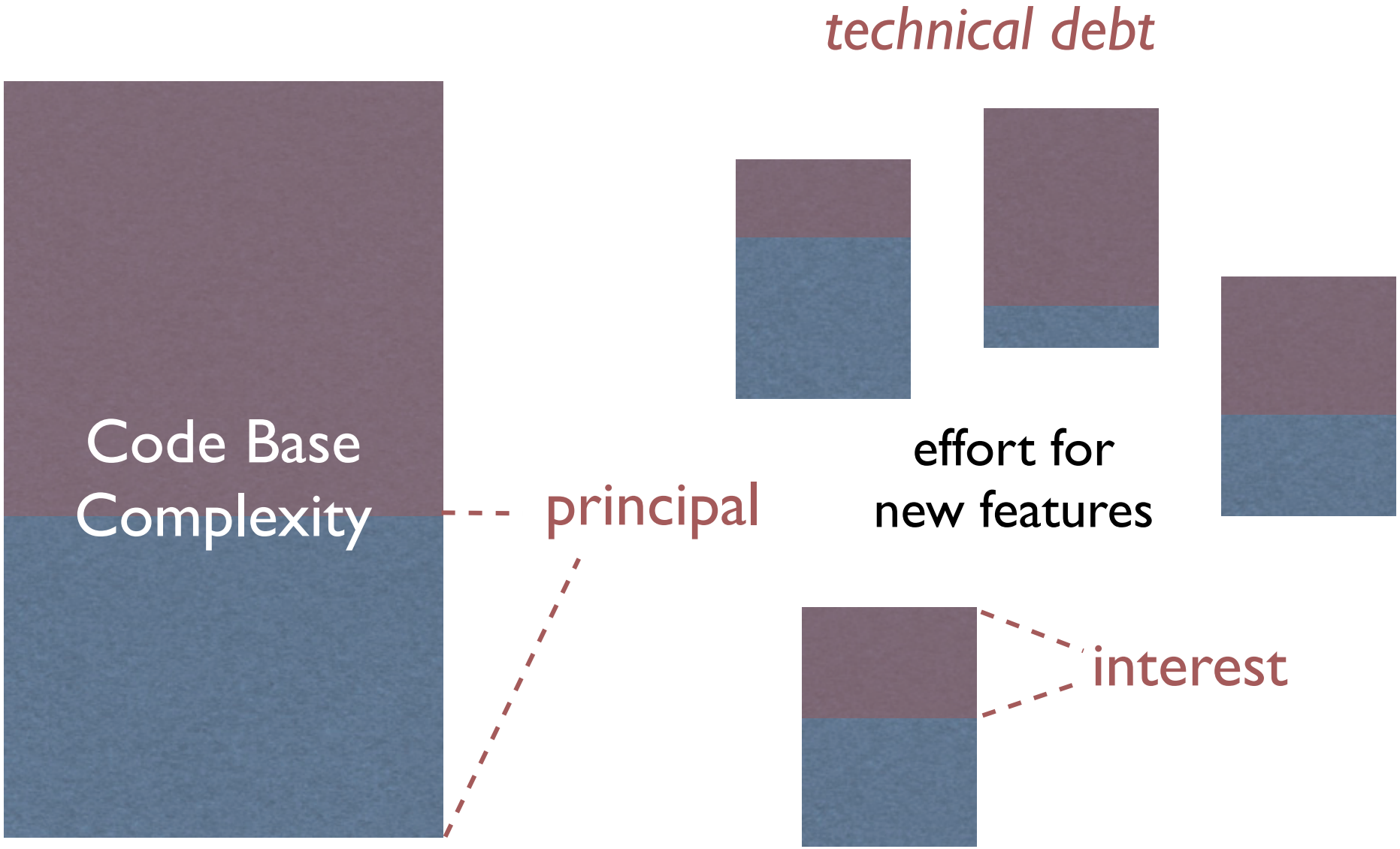
examples


Hunting Season
EJB / Biztalk
Field Level Security
Essential
Accidental


technical
debt

technical debt



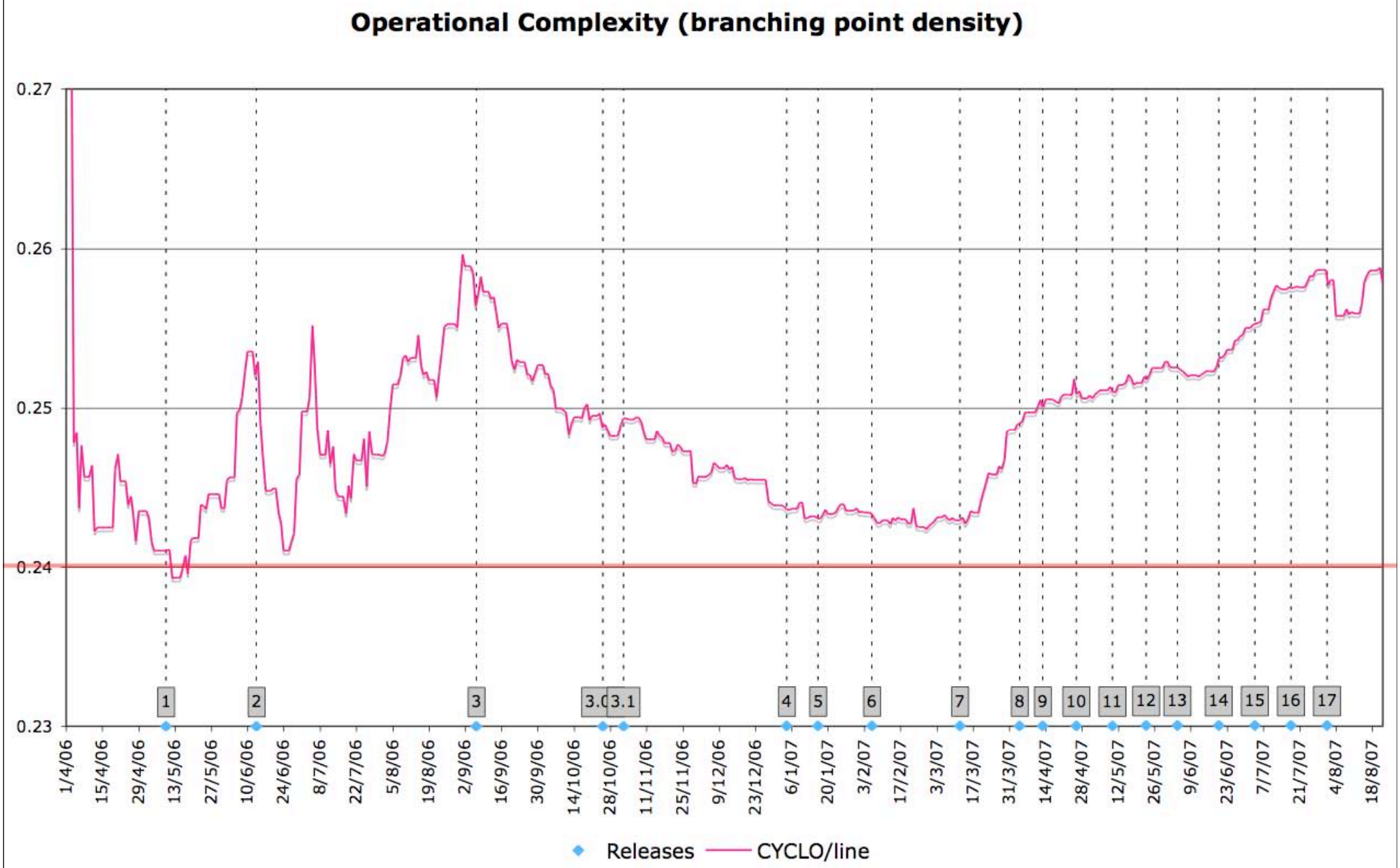

technical debt
Code Base
Complexity
principal
effort for
new features
interest

negotiating repayment

you must convince someone technical debt exists...

...start a conversation about repayment

demonstration trumps discussion


Operational Complexity (branching point density)
0.27
0.26
0.25
0.24
0.23
1
2
3
3.0
3.1
4
5
6
7
8
9
10
11
12
13
14
15
16
17
1/4/06
15/4/06
29/4/06
13/5/06
27/5/06
10/6/06
24/6/06
8/7/06
22/7/06
5/8/06
19/8/06
2/9/06
16/9/06
30/9/06
14/10/06
28/10/06
11/11/06
25/11/06
9/12/06
23/12/06
6/1/07
20/1/07
3/2/07
17/2/07
3/3/07
17/3/07
31/3/07
14/4/07
28/4/07
12/5/07
26/5/07
9/6/07
23/6/07
7/7/07
21/7/07
4/8/07
18/8/07
Releases
CYCLO/line


rampant genericness

genericness

often result of over engineering

“if we build lots of layers for extension, we can easily build more onto it later”

increases software entropy

accidental complexity

generic obfuscation

IT Service Delivery Model & Consolidation Phases

State of New Mexico
ITC Architecture Committee

Consolidation and Standardization Phases:

- First Phase
- Second Phase
- Future
- Not Considered

State of New Mexico Vision and Mission

IT Consolidation & Quality Efforts

Technical Domains

Business Domains

Constituent Svcs

- Human Services
- Medicaid Mgmt
- Child Support Sys
- Health
- Public Health Sys
- Taxation
- Tax Mgmt & Admin
- Children & Fam
- Fam&Chil Tracking
- Labor
- Unempl Ins

Justice

- Courts
- Court Case Mgmt
- Corrections
- Criminal Mgmt
- Public Safety
- Criminal Justice

Resource Mgmt

- Transportation
- Roads Inventory
- Environment
- Env Permits
- Cultural Affairs
- Artifact Mgmt
- State Engineer
- Waters

Govt Ops

- General Svcs
- Risk Mgmt
- Finance / Admin
- Personnel
- CIO
- Sec of State
- Voter Registration

Education

- Public Eduction
- Distict Reporting
- Higher Educ

Apps & Data Mgmt Domains

- HIPAA
- Eligibility
- Doc Mgmt Sys
- Case Mgmt
- Justice Info Share
- GIS
- Licensing
- Dist Learn
- Stud Info Sys

Applications Domain

- Portal
- SHARE (HR/Accounting/Payroll)
- Constituent Relationship Management (One Stop)
- eMail

Middleware Domain

- Collaboration Tools
- Middleware
- Data Interchange

Data Mgmt Domain

- Enterprise Reporting
- Data Warehouse
- Database Management

System Management Domain

- Helpdesk
- Asset Management
- Desktop Support

Platform Domain

- Desktop
- Storage Management
- Shared Server Space
- Backup Management
- Data Centers

Info Protection /Security Domain

- Disaster Recovery/Business Continuity
- Identity Management
- Security

Network Domain

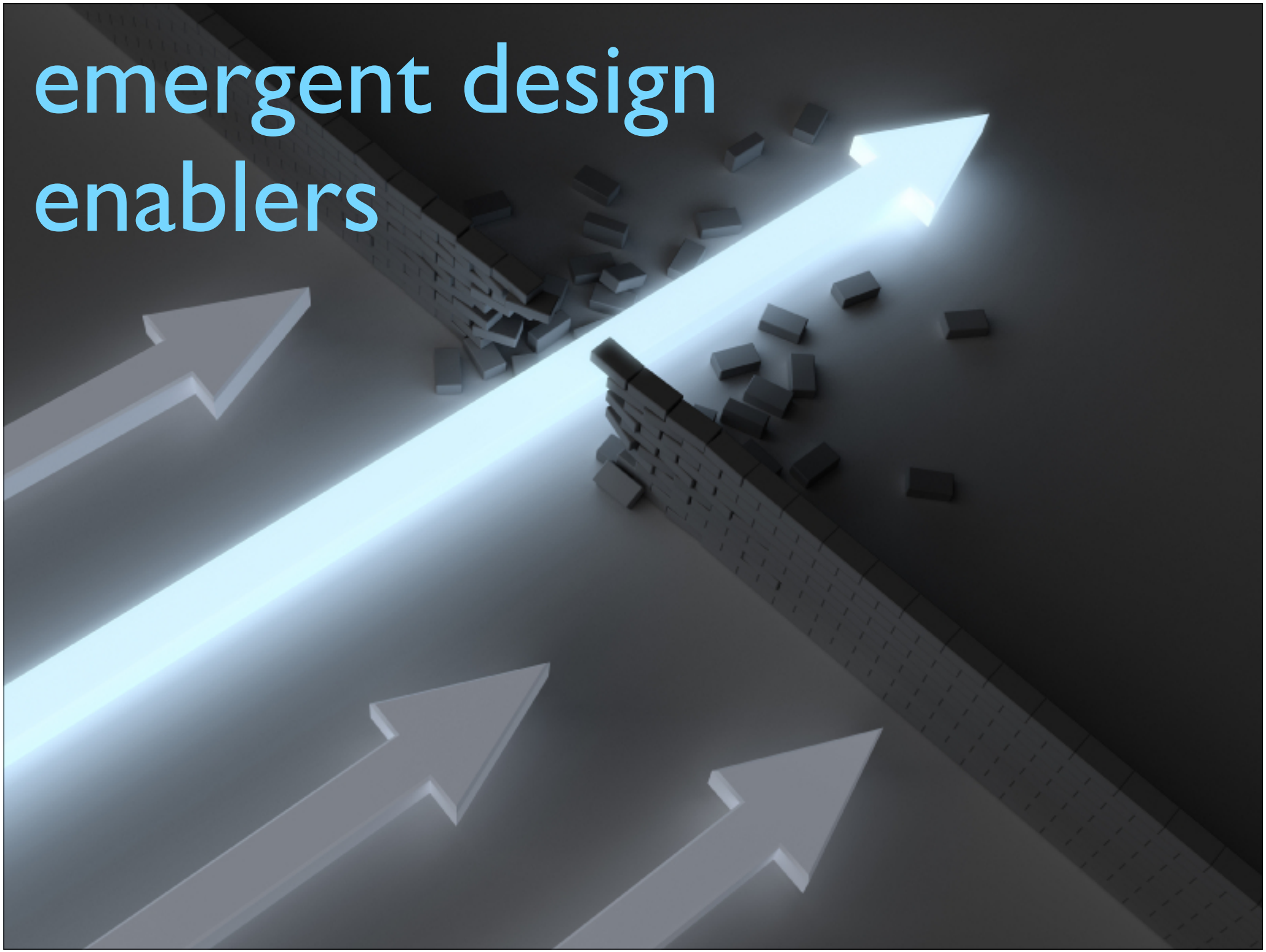
- Network - LAN
- Network - WAN (MagNET)

Agency Strategic Architectures

Enterprise Business Architectures

Enterprise Technical Architectures

emergent design enablers



testing as a design tool

confidence against unanticipated side effects

regression testing

understandable (executable) documentation

executable intent

protection between API boundaries

test driven design

more about design than testing

design will emerge from tests

atomic understanding of intent

better abstractions

less accidental complexity


case study:
perfect
numbers

perfect number:

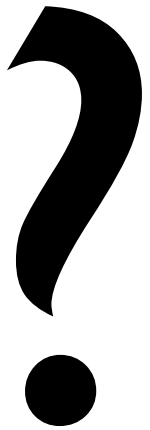
∑ of the factors == number
(not including the number)

∑ of the factors - # == #

```java
public class PerfectNumberFinder1 {
    public static boolean isPerfect(int number) {
        // get factors
        List<Integer> factors = new ArrayList<Integer>();
        factors.add(1);
        factors.add(number);
        for (int i = 2; i < number; i++)
            if (number % i == 0)
                factors.add(i);

        // sum factors
        int sum = 0;
        for (int n : factors)
            sum += n;

        // decide if it's perfect
        return sum - number == number;
    }
}
```

```java
private static Integer[] PERFECT_NUMS = {6, 28, 496, 8128, 33550336};

@Test public void test_perfection() {
    for (int i : PERFECT_NUMS)
        assertTrue(PerfectNumberFinder1.isPerfect(i));
}

@Test public void test_non_perfection() {
    List<Integer> expected = new ArrayList<Integer>(
            Arrays.asList(PERFECT_NUMS));
    for (int i = 2; i < 100000; i++) {
        if (expected.contains(i))
            assertTrue(PerfectNumberFinder1.isPerfect(i));
        else
            assertFalse(PerfectNumberFinder1.isPerfect(i));
    }
}
```

```java
public static boolean isPerfect(int number) {
    // get factors
    List<Integer> factors = new ArrayList<Integer>();
    factors.add(1);
    factors.add(number);
    for (int i = 2; i < number; i++)
        if (number % i == 0)
            factors.add(i);

    // sum factors
    int sum = 0;
    for (int n : factors)
        sum += n;

    // decide if it's perfect
    return sum - number == number;
}
```



```
public class PerfectNumberFinder2 {
    public static boolean isPerfect(int number) {
        // get factors
        List<Integer> factors = new ArrayList<Integer>();
        factors.add(1);
        factors.add(number);
        for (int i = 2; i <= sqrt(number); i++)
            if (number % i == 0) {
                factors.add(i);
                factors.add(number / i);
            }

        // sum factors
        int sum = 0;
        for (int n : factors)
            sum += n;

        // decide if it's perfect
        return sum - number == number;
    }
}
```

whole number square roots

```
public class PerfectNumberFinder2 {
    public static boolean isPerfect(int number) {
        // get factors
        List<Integer> factors = new ArrayList<Integer>();
        factors.add(1);
        factors.add(number);
        for (int i = 2; i <= sqrt(number); i++)
            if (number % i == 0) {
                factors.add(i);
                // account for whole-number square roots
                if (number / i !=  i)
                    factors.add(number / i);
            }

        // sum factors
        int sum = 0;
        for (int n : factors)
            sum += n;

        // decide if it's perfect
        return sum - number == number;
    }
}
```

Classifier

```
public class Classifier6 {
    private Set<Integer> _factors;
    private int _number;

    public Classifier6(int number) {
        if (number < 1)
            throw new InvalidNumberException(
            "Can't classify negative numbers");
        _number = number;
        _factors = new HashSet<Integer>();
        _factors.add(1);
        _factors.add(_number);
    }

    private boolean isFactor(int factor) {
        return _number % factor == 0;
    }
```

```
public Set<Integer> getFactors() {
    return _factors;
}

private void calculateFactors() {
    for (int i = 2; i < sqrt(_number) + 1; i++)
        if (isFactor(i))
            addFactor(i);
}

private void addFactor(int factor) {
    _factors.add(factor);
    _factors.add(_number / factor);
}

private int sumOfFactors() {
    int sum = 0;
    for (int i : _factors)
        sum += i;
    return sum;
}
```

design implications

```
for (int i = 2; i <= sqrt(number); i++)
    if (number % i == 0) {
        factors.add(i);
        // account for whole-number square roots
        if (number / i !=  i)
            factors.add(number / i);
    }
```

vs.

perfect number finder

```
for (int i = 2; i <= sqrt(number); i++)
    if (number % i == 0) {
        factors.add(i);
        // account for whole-number square roots
        if (number / i !=  i)
            factors.add(number / i);
    }
```

classifier

```
private void calculateFactors() {
    for (int i = 2; i < sqrt(_number) + 1; i++)
        if (isFactor(i))
            addFactor(i);
}

private void addFactor(int factor) {
    _factors.add(factor);
    _factors.add(_number / factor);
}
```

tdd vs test-after

test after doesn't expose design flaws early

example: wrong abstraction level

tdd forces you to think about every little thing

encourages refactoring what's not right

refactoring

collective code ownership

fix broken windows whenever you see them

regularly fix obsolescent abstractions

prudently refactor aggressively

code should get stronger with age

expressiveness matters

if code is design, readable design matters

complex languages hurt readability

most comments don't help

- not executable
- always (potentially) out of date

idiomatic pattern without closures

```
public void addOrderFrom(ShoppingCart cart, String userName,
                         Order order) throws Exception {
    setupDataInfrastructure();
    try {
        add(order, userKeyBasedOn(userName));
        addLineItemsFrom(cart, order.getOrderKey());
        completeTransaction();
    } catch (Exception condition) {
        rollbackTransaction();
        throw condition;
    } finally {
        cleanUp();
    }
}
```

without closures

```
public void wrapInTransaction(Command c) throws Exception {
    setupDataInfrastructure();
    try {
        c.execute();
        completeTransaction();
    } catch (Exception condition) {
        rollbackTransaction();
        throw condition;
    } finally {
        cleanUp();
    }
}

public void addOrderFrom(final ShoppingCart cart, final String userName,
                         final Order order) throws Exception {
    wrapInTransaction(new Command() {
        public void execute() {
            add(order, userKeyBasedOn(userName));
            addLineItemsFrom(cart, order.getOrderKey());
        }
    });
}
```

with closures (groovy)

```
public class OrderDbClosure {
    def wrapInTransaction(command) {
      setupDataInfrastructure()
      try {
        command()
        completeTransaction()
      } catch (RuntimeException ex) {
        rollbackTransaction()
        throw ex
      } finally {
        cleanUp()
      }
    }

    def addOrderFrom(cart, userName, order) {
      wrapInTransaction {
        add order, userKeyBasedOn(userName)
        addLineItemsFrom cart, order.orderKey
      }
    }
}
```

with closures (ruby)

```ruby
def wrap_in_transaction
    setup_data_infrastructure
    begin
      yield
      complete_transaction
    rescue ex
      rollback_transaction
      throw ex
    ensure
      clean_up
    end
end

def add_order_from cart, user_name, order
    wrap_in_transaction do
      add order, user_key_based_on(user_name)
      add_line_items_from cart, order.order_key
    end
end
```

abstraction styles

imperative

- structured / modular
- object-oriented

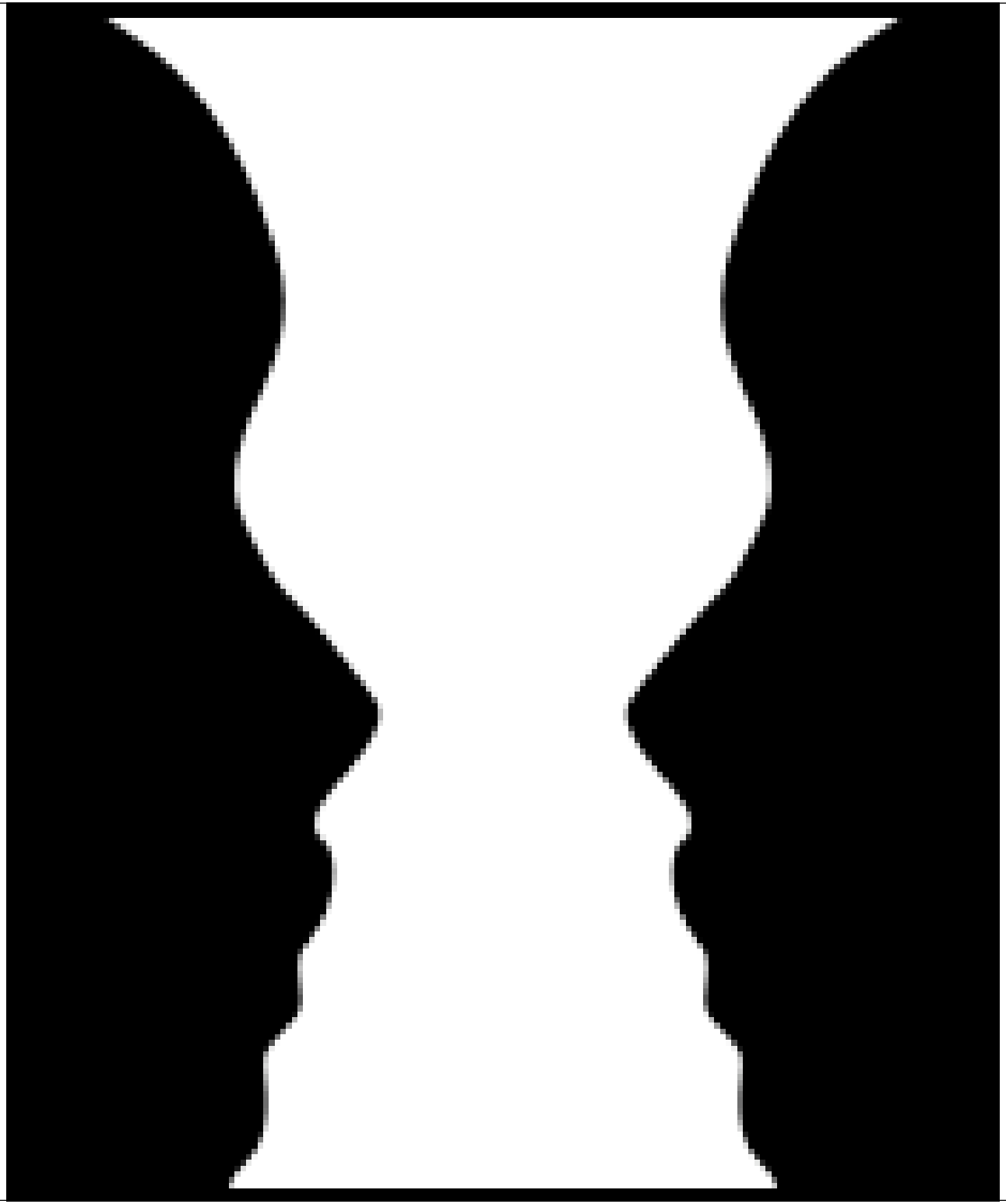
functional

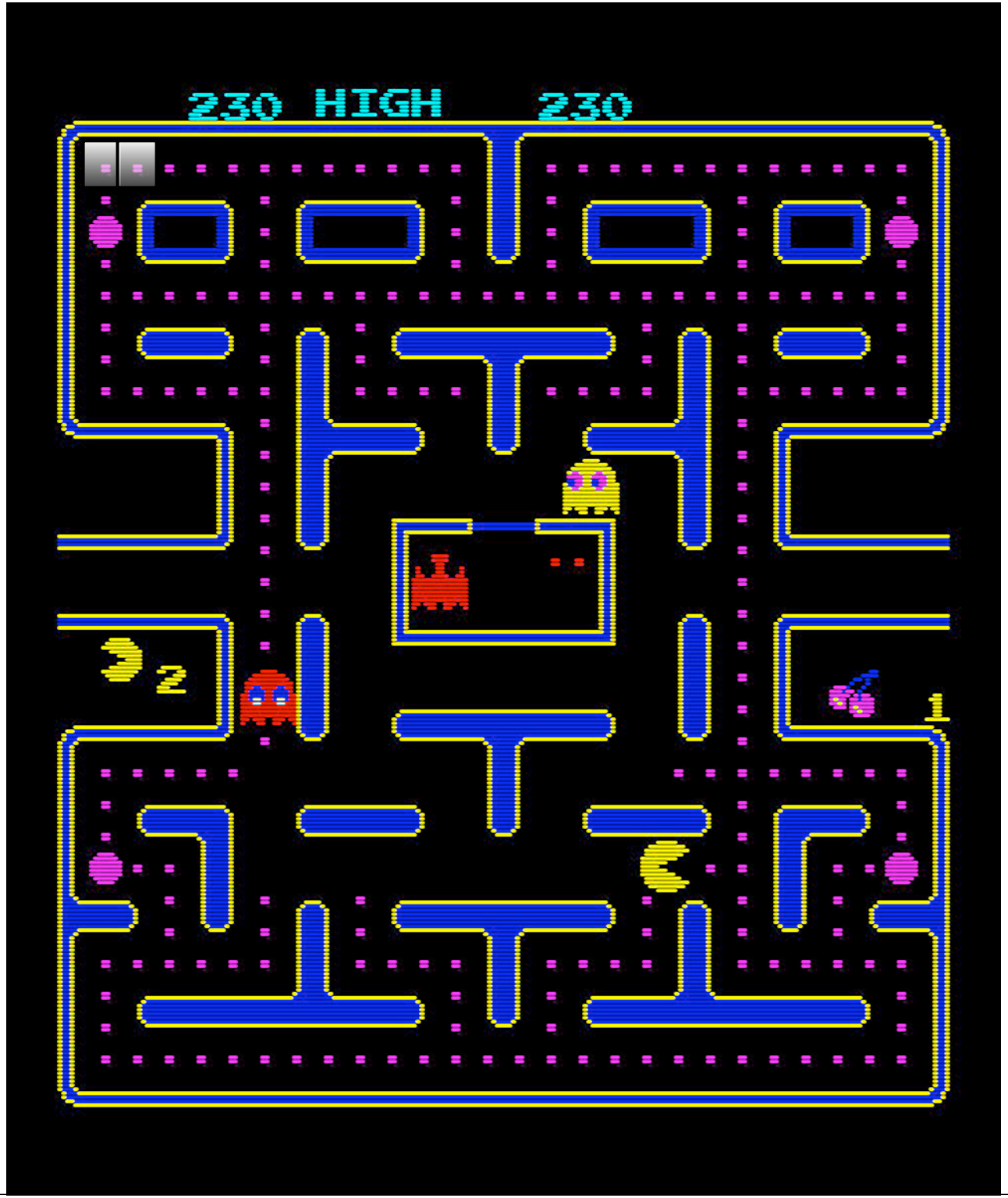
anti-objects

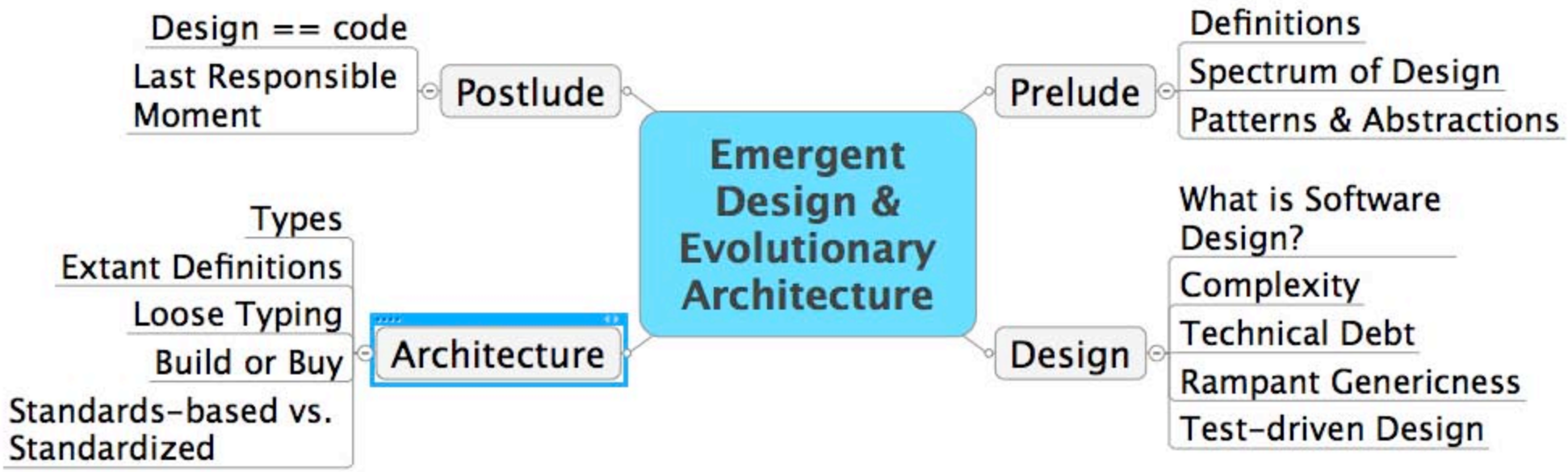
collaborative diffusion

"The metaphor of objects can go too far by making us try to create objects that are too much inspired by the real world."

"...an antiobject is a kind of object that appears to essentially do the opposite of what we generally think the object should be doing."



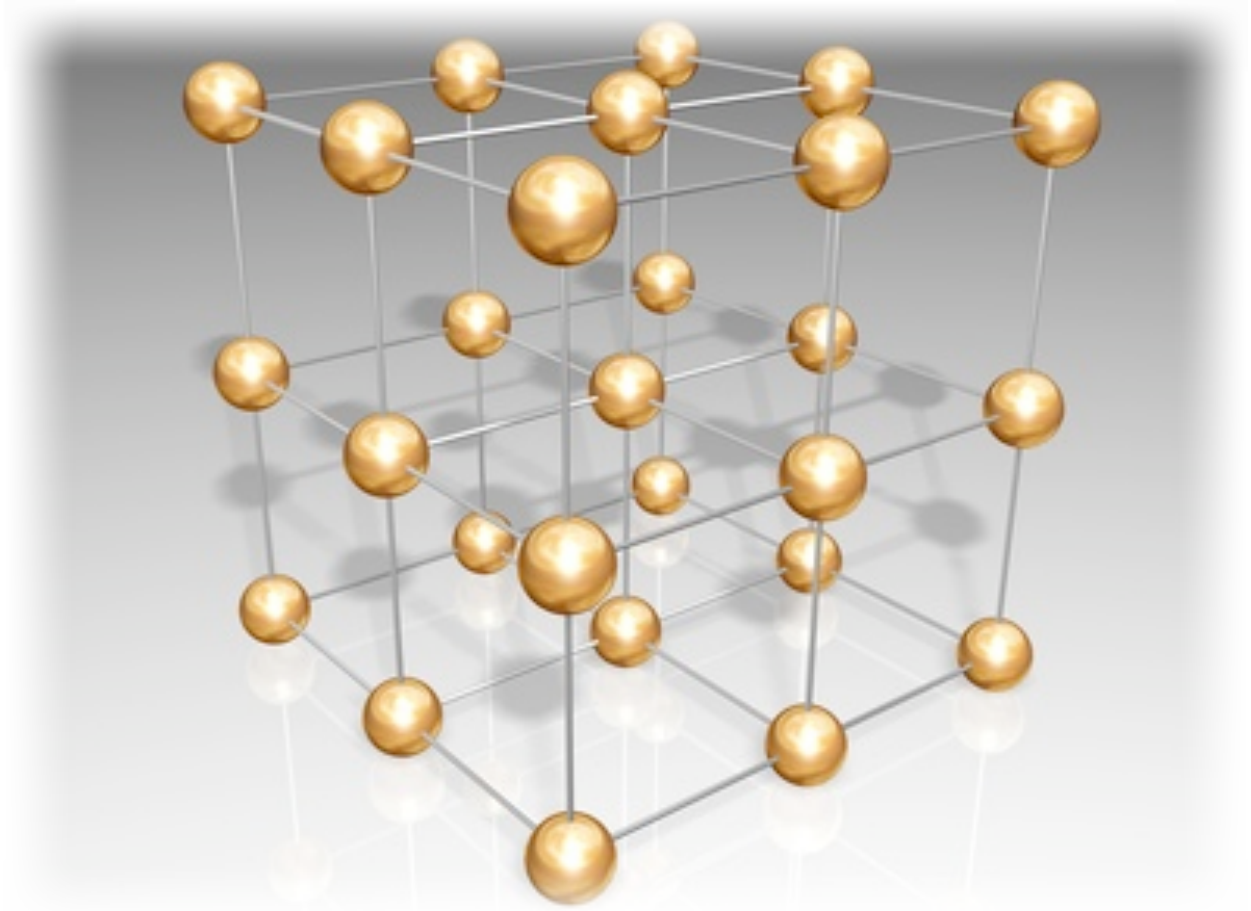

230 HIGH 230
2
1


Emergent Design & Evolutionary Architecture
Postlude
Design == code
Last Responsible Moment
Prelude
Definitions
Spectrum of Design
Patterns & Abstractions
Architecture
Types
Extant Definitions
Loose Typing
Build or Buy
Standards-based vs. Standardized
Design
What is Software Design?
Complexity
Technical Debt
Rampant Genericness
Test-driven Design


architecture

Evolution, n.
[L. evolutio an unrolling: cf. F.
['e]volution evolution

1: a process in which something passes by degrees to a different stage (especially a more advanced or mature stage)

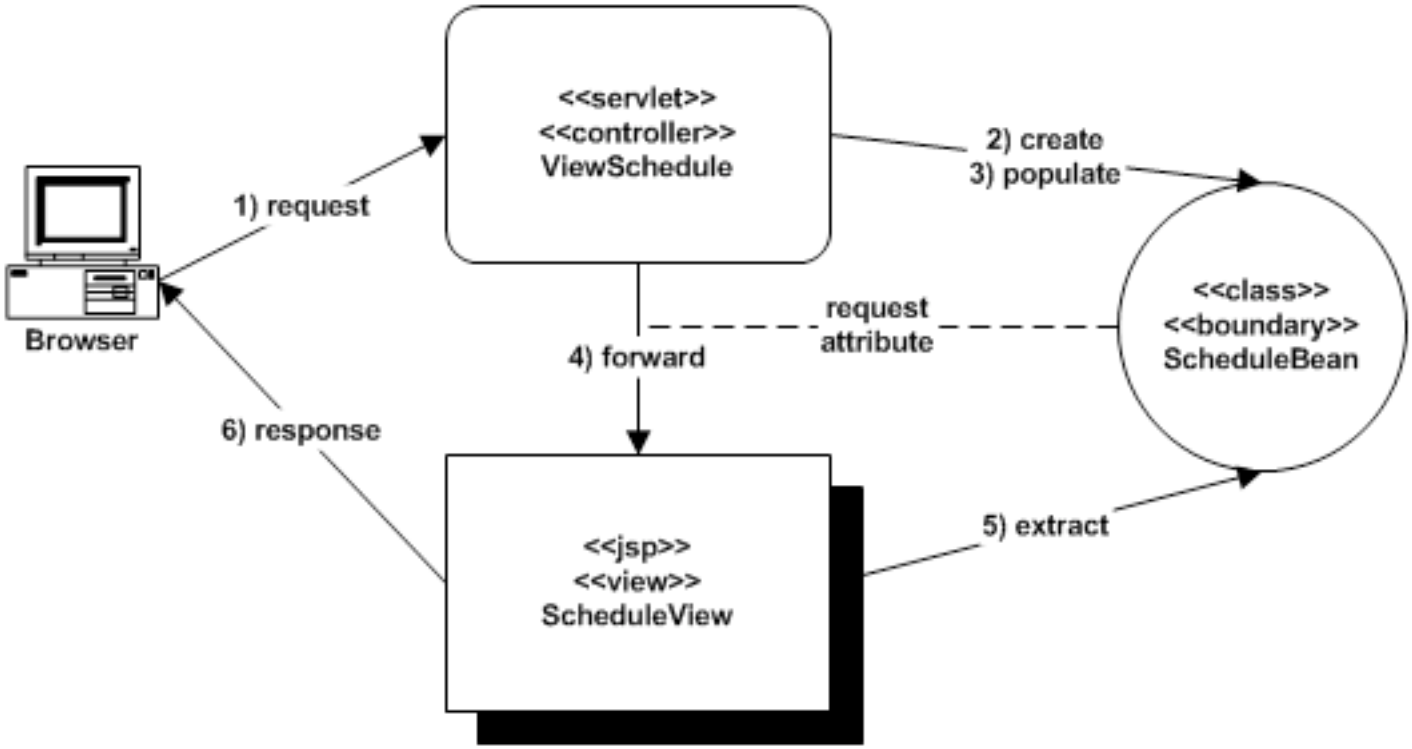


emergent

evolutionary

application architecture

describes the coarse-grained pieces that compose an application



framework level architecture

the unit of reuse in java is the *library*

when was the last time you downloaded a single class?

JSR 277, the java module system...abandonware

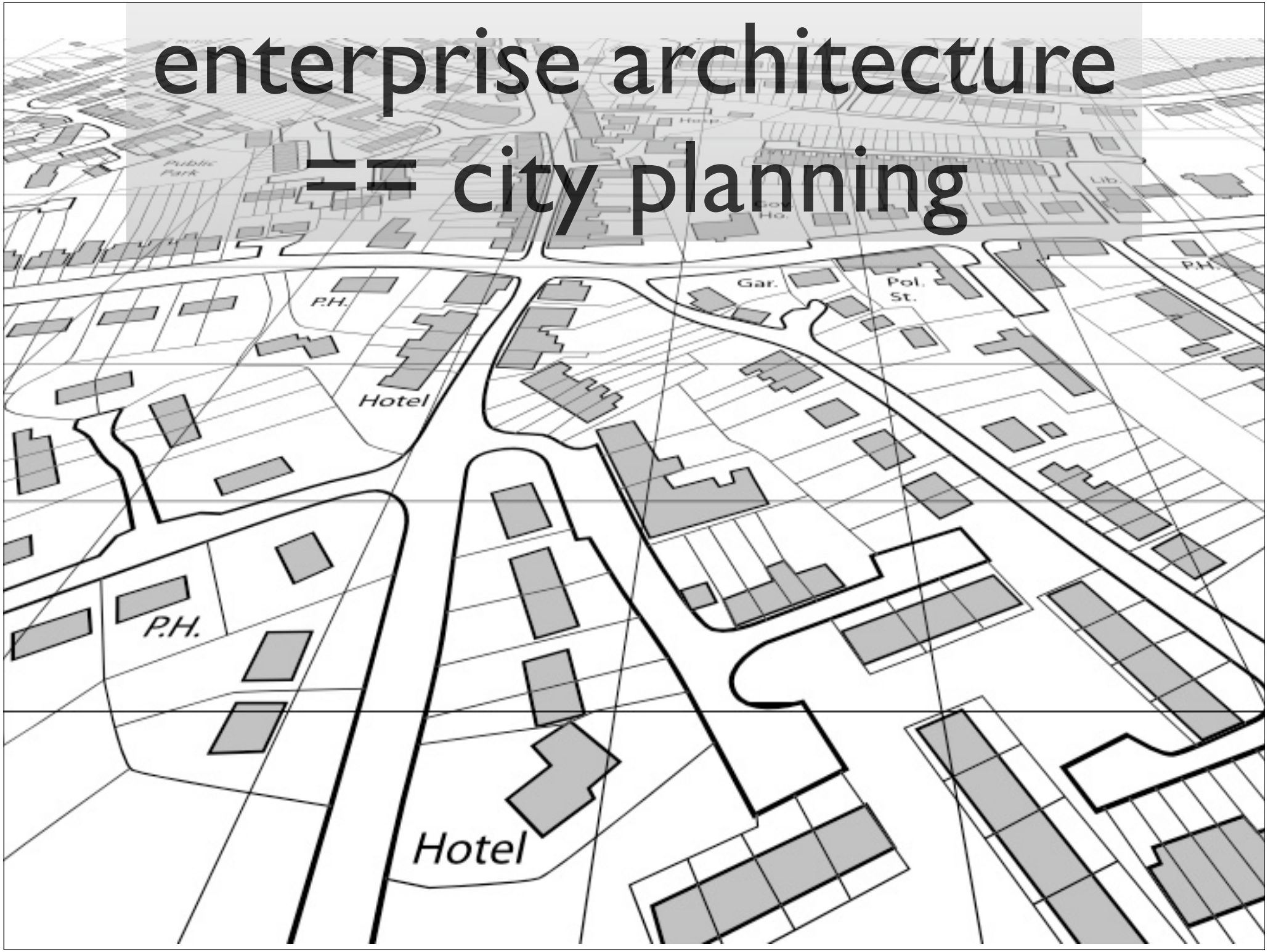
JSR 294 (superpackage)...IN JAVA 7!

implemented by ivy & maven

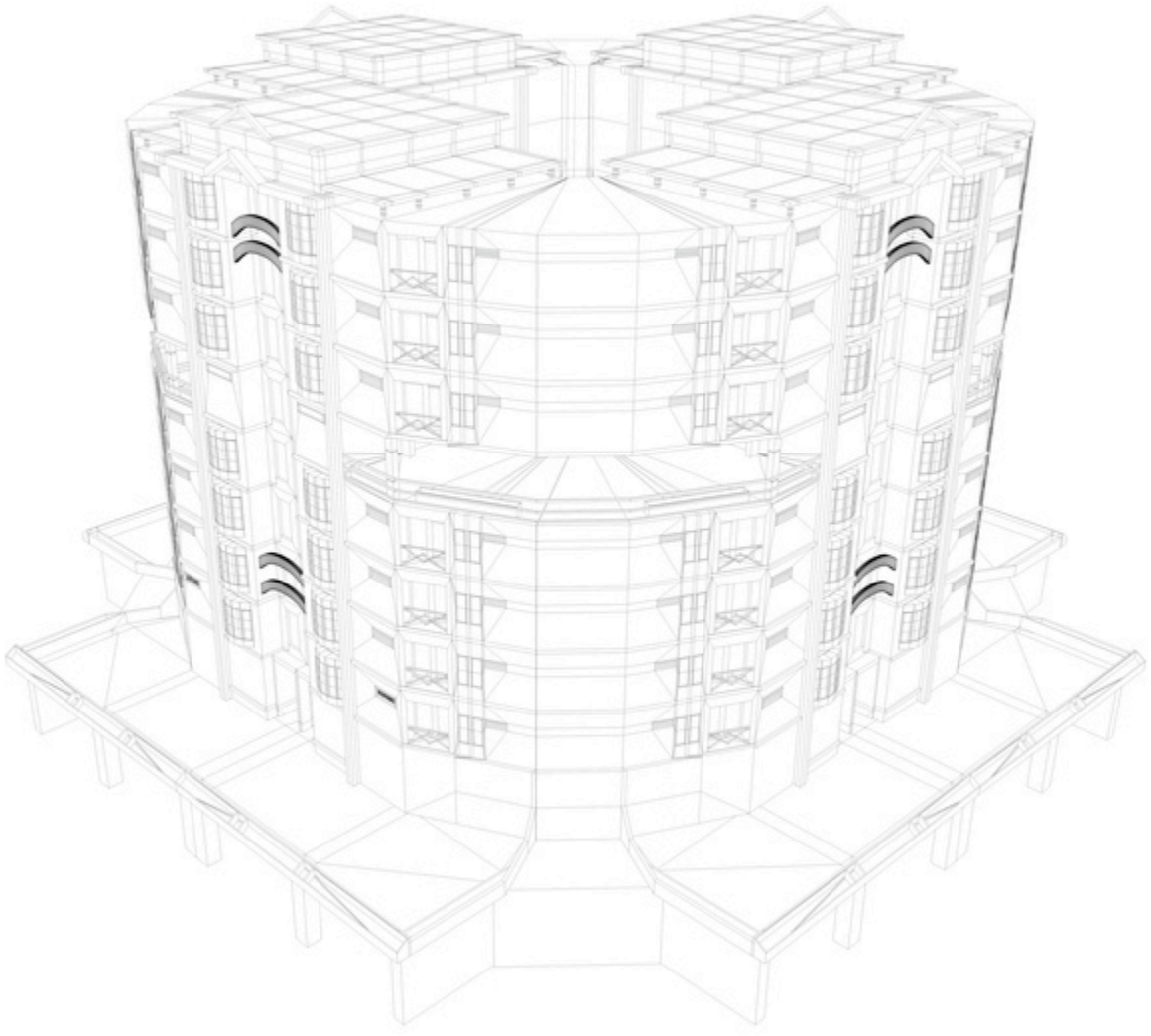
enterprise architecture

concerns itself with how the enterprise as a whole (which usually means the applications running inside a large organization) consumes applications

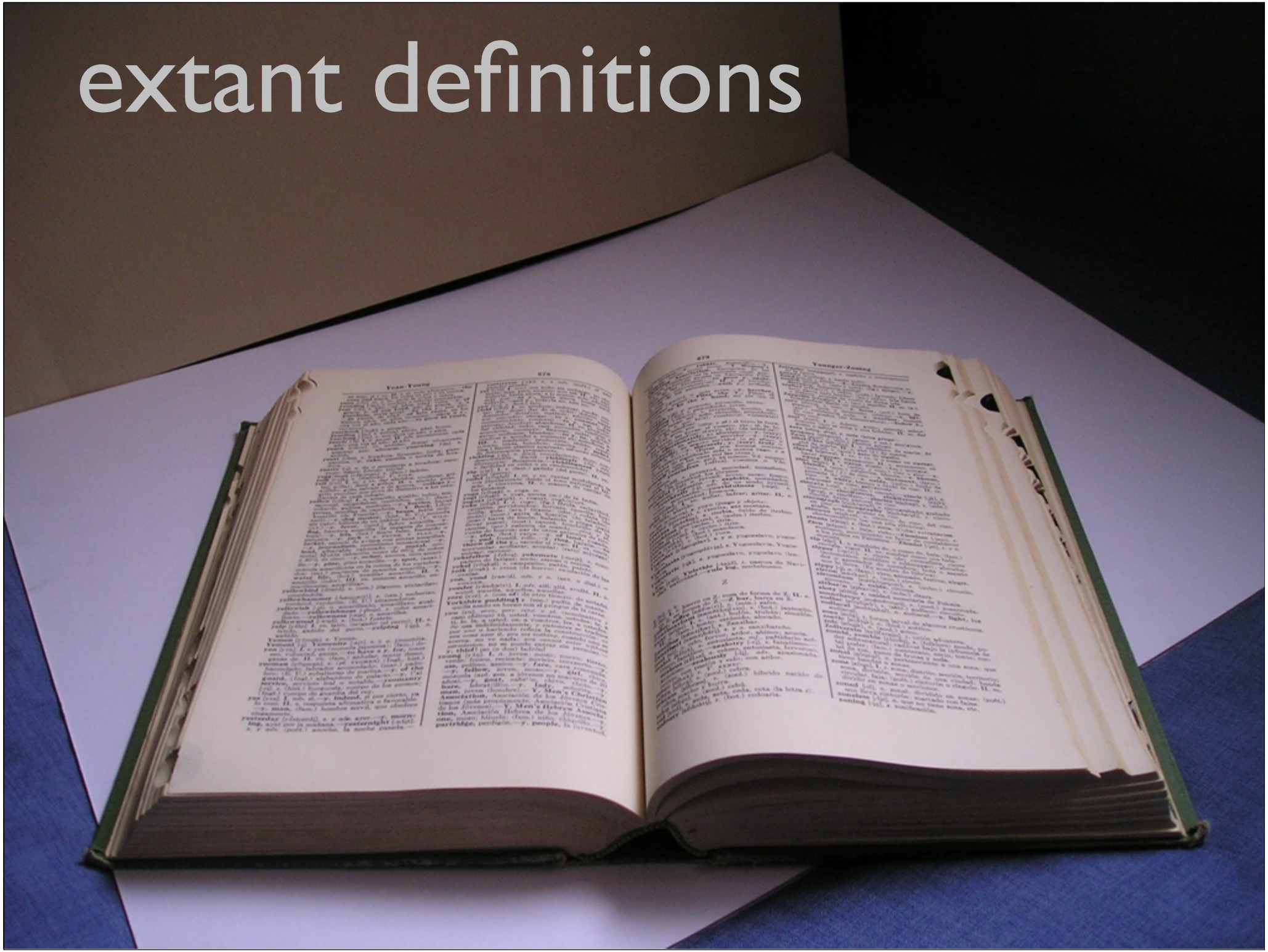
enterprise architecture == city planning



application architecture == building plan



extant definitions



Editor: Martin Fowler ■ ThoughtWorks ■ fowler@acm.org

Who Needs an Architect?

Martin Fowler

Wandering down our corridor a while ago, I saw my colleague Dave Rice in a particularly grumpy mood. My brief question caused a violent statement, "We shouldn't interview anyone who has 'architect' on his resume." At first blush, this was an odd turn of phrase, because we usually introduce Dave as one of our leading architects.

The reason for his title schizophrenia is the fact that, even by our industry's standards, "architect" and "architecture" are terribly overloaded words. For many, the term "software architect" fits perfectly with the smug controlling image at the end of *Matrix Reloaded*. Yet even in firms that have the greatest contempt for that image, there's a vital role for the technical leadership that an architect such as Dave plays.

What is architecture?

When I was fretting over the title for *Patterns of Enterprise Application Architecture* (Addison-Wesley, 2002), everyone who reviewed it agreed that "architecture" belonged in the title. Yet we all felt uncomfortable defining the word. Because it was my book, I felt compelled to take a stab at defining it.

My first move was to avoid fuzziness by just letting my cynicism hang right out. In a sense, I define *architecture* as a word we use when we want to talk about design but want to puff it up to make it sound important. (Yes, you can imagine a similar phenomenon for architect.) However, as so often occurs, inside the blighted cynicism is a pinch of truth. Understanding came to me after reading a posting from Ralph Johnson on the Extreme Programming mailing list. It's so good I'll quote it all.

A previous posting said

> The RUP, working off the IEEE definition, defines architecture as "the highest level concept of a system in its environment. The architecture of a software system (at a given point in time) is its organization or structure of significant components interacting through interfaces, those components being composed of successively smaller components and interfaces."

Johnson responded:

> I was a reviewer on the IEEE standard that used that, and I argued uselessly that this was clearly a completely bogus definition. There is no highest level concept of a system. Customers have a different concept than developers. Customers do not care at all about the structure of significant components. So, perhaps an architecture is the highest level concept that developers have of a system in its environment. Let's forget the developers who just understand their little place. Architecture is the highest level concept of the expert developers. What makes a component significant? It is significant because the expert developers say so.
>
> So, a better definition would be "In most successful software projects, the expert developers working on that project have a shared understanding of the

0740-7459/03/$17.00 © 2003 IEEE

"The RUP, working off the IEEE definition, defines architecture as 'the highest level concept of a system in its environment.' The architecture of a software system (at a given point in time) is its organization or structure of significant components interacting through interfaces, those components being composed of successively smaller components and interfaces.'"

post on the XP mail list

technical definition

"A better definition would be: In most successful software projects, the expert developers working on that project have a shared understanding of the system design. This shared understanding is called "architecture." This understanding includes how the system is divided into components and how the components interact through interfaces.'"

Ralph Johnson, rebutting the original post

process definition

Architecture is about the important stuff.

Whatever that is.

Martin Fowler's definition

"the important stuff"

vague but descriptive

many arguments about architecture revolve around misunderstanding what is important

what's important to business analysts differs from important stuff for an enterprise architect

differences can in fact be mutually exclusive

"SOA favors flexibility over performance"

Stuff that's hard to change later.

Martin Fowler, in conversation

There should be as little of that stuff as possible.


architectural
considerations

politics of architecture

build or buy


THIS NOTE IS LEGAL TENDER
FOR ALL DEBTS, PUBLIC AND PRIVATE
Treasurer of the United States
H3

business processes are not commoditizable

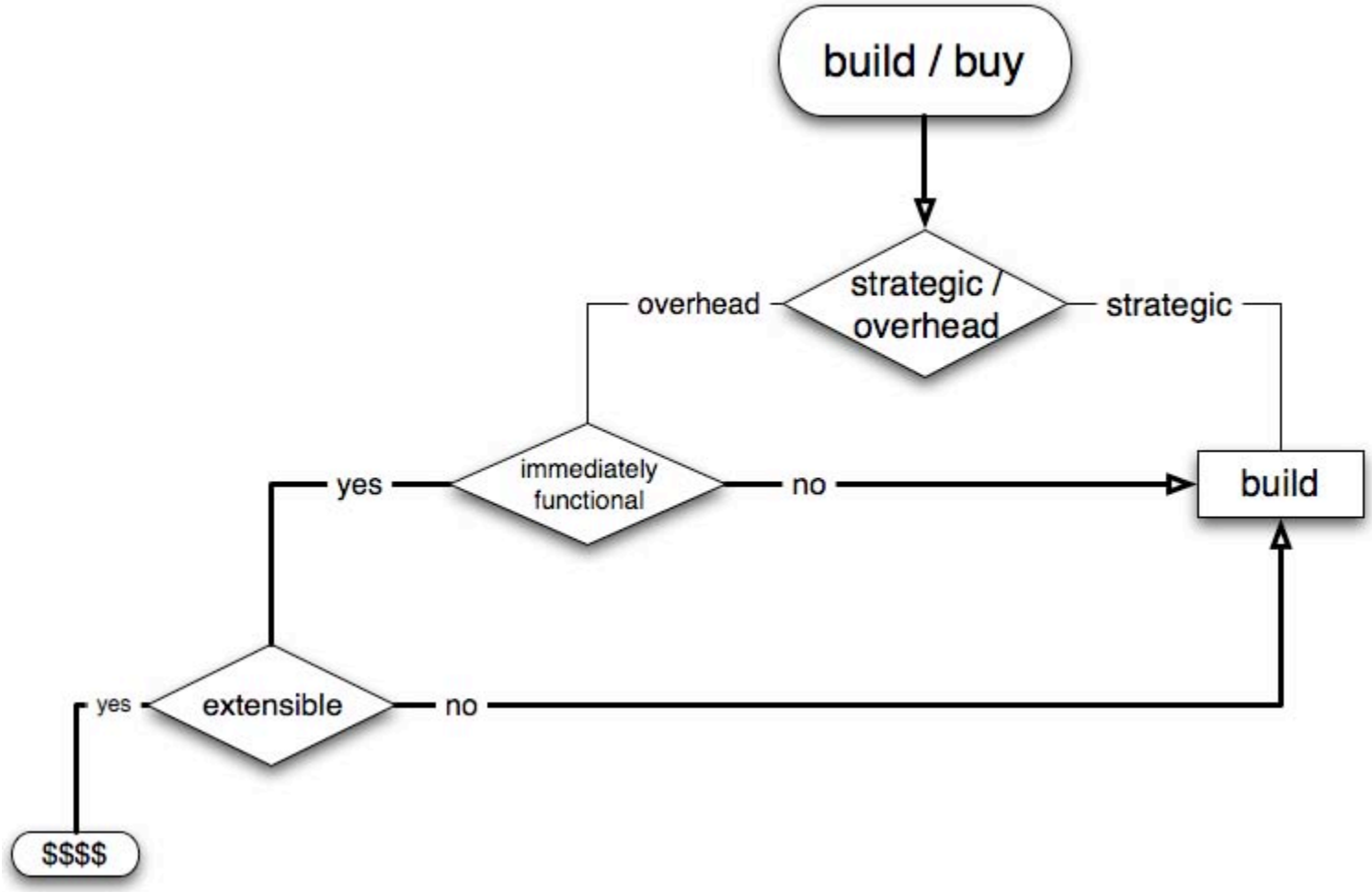
"you can buy this generic business process software..."

"...we just need to tweak it with a few customizations"

myth

radically unique across similar businesses

software can provide strategic business advantage


build / buy
strategic / overhead
overhead
strategic
immediately functional
yes
no
build
extensible
yes
no
$$$$

standards-based vs. standardized

flash-back to java web development before j2ee

standards helped developers tremendously...

...but vendors hate it

the price of commodity software quickly approaches $0

contrast j2ee & sql

ESB: standards-based but not standardized

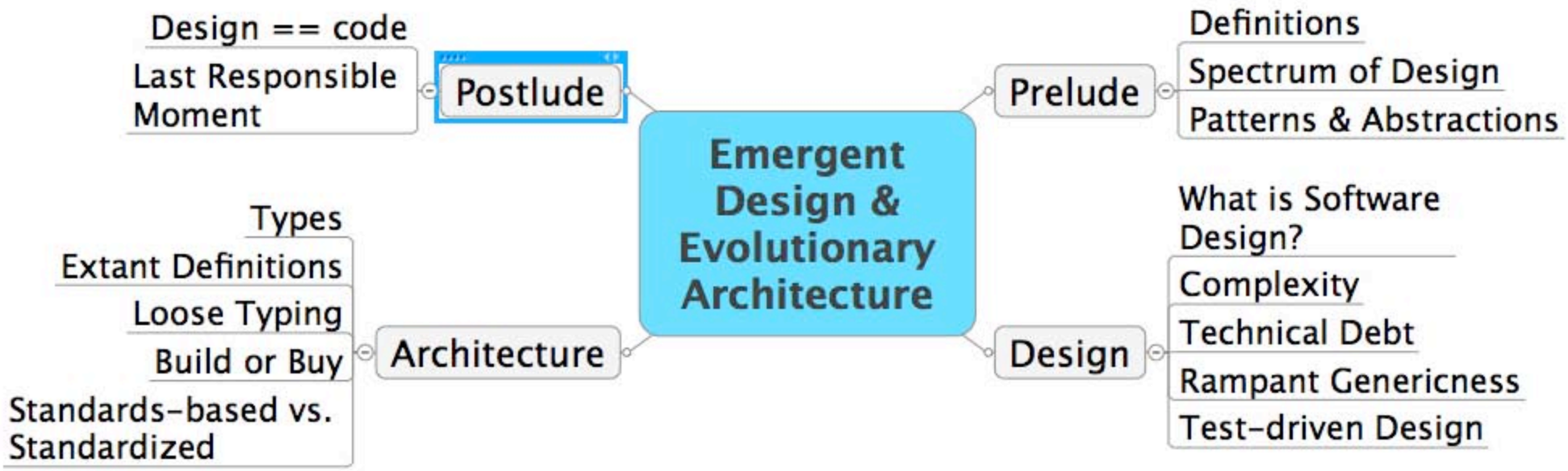
big enterprise package software touts standards-based

held together with highly proprietary glue

even the open source ESBs suffer from this

not likely to change

consider the impact on your overall architecture


Emergent Design & Evolutionary Architecture
Postlude
Design == code
Last Responsible Moment
Prelude
Definitions
Spectrum of Design
Patterns & Abstractions
Architecture
Types
Extant Definitions
Loose Typing
Build or Buy
Standards-based vs. Standardized
Design
What is Software Design?
Complexity
Technical Debt
Rampant Genericness
Test-driven Design


postlude

design is about code

other artifacts aid in creating code

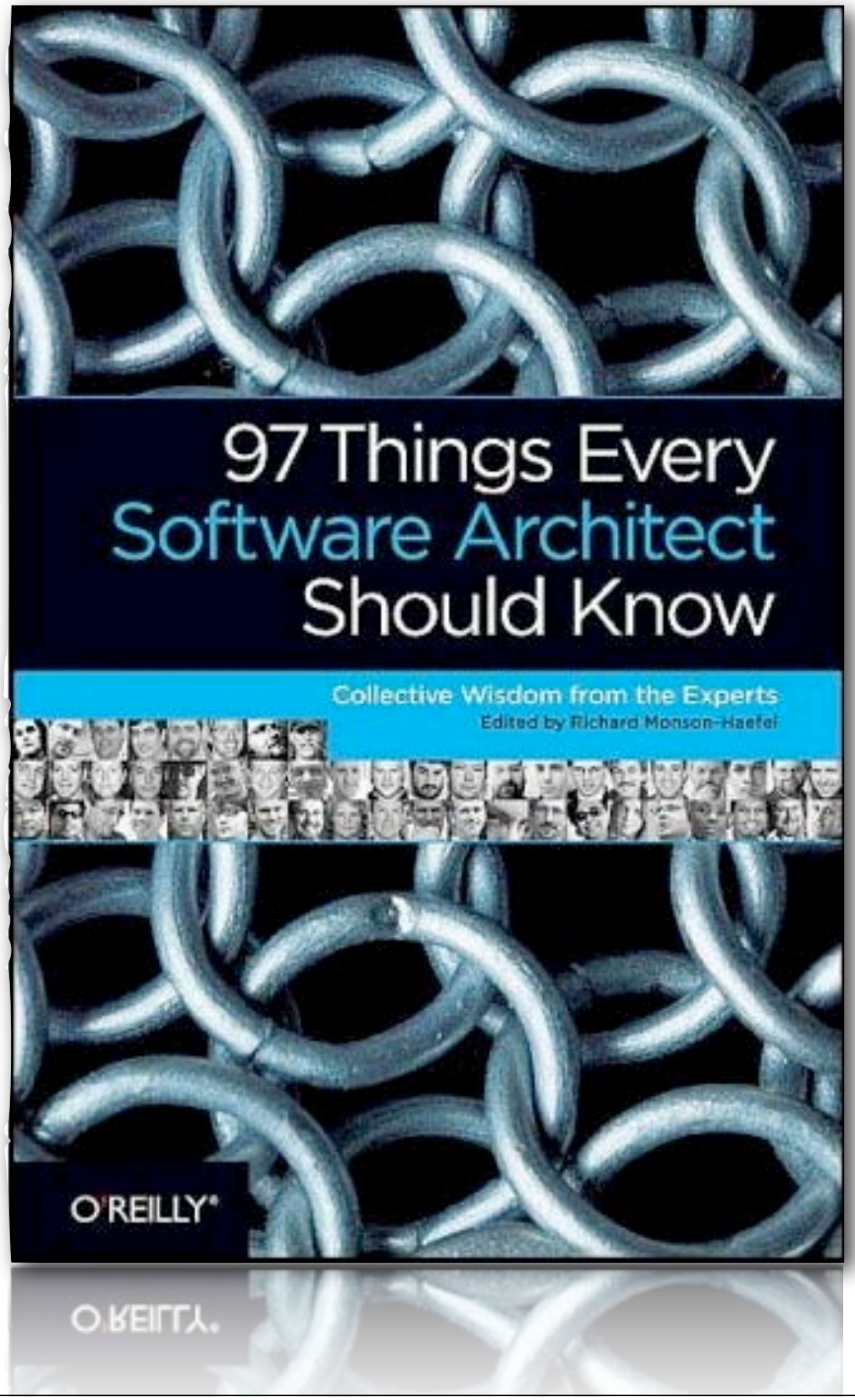
all artifacts besides code are transient

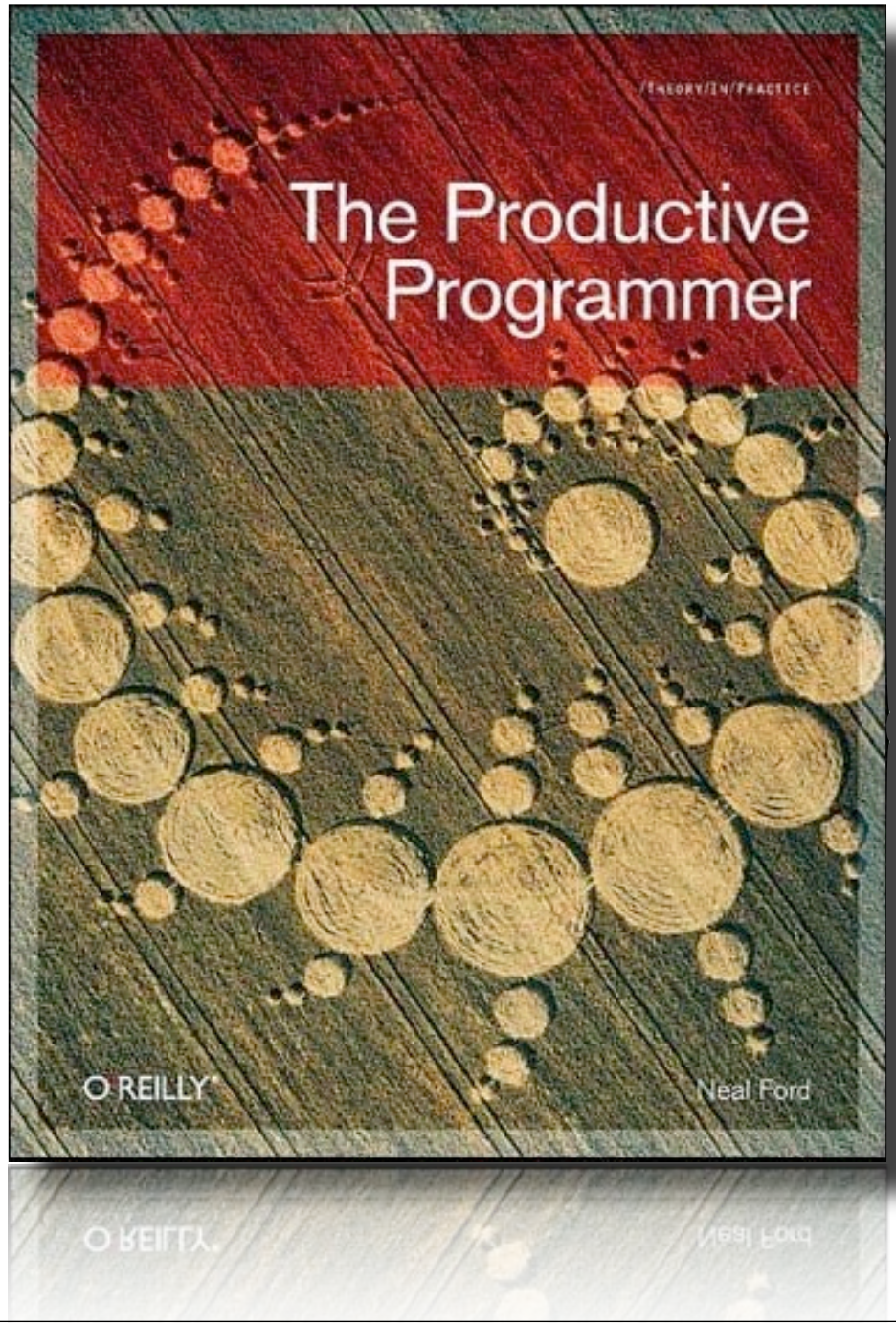
code hygiene matters

fix broken windows

pay back technical debt

last responsible moment


97 Things Every
Software Architect
Should Know
Collective Wisdom from the Experts
Edited by Richard Monson-Haefel
O'REILLY


THEORY/IN/PRACTICE
The Productive Programmer
O'REILLY
Neal Ford

?'s

please fill out the session evaluations

samples at `github.com/nealford`

NEAL FORD software architect / meme wrangler

ThoughtWorks

nford@thoughtworks.com
3003 Summit Boulevard, Atlanta, GA 30319
www.nealford.com
www.thoughtworks.com
memeagora.blogspot.com

resources

Emergent Design & Evolutionary Architecture series:

http://tinyurl.com/nf-ead