Object Design Roots and New Directions

Rebecca Wirfs-Brock

© 2014 Wirfs-Brock Associates

Tektronix Design Branches

Xtreme Programming

Design Patterns

Responsibility-Driven Design

RDD

Recommended Daily Dosage?

Redding Municipal Airport (RDD)?
Radar Detector Detector?

Responsibility Driven Design!

Initial Inspiration…

Smalltalk abstract methods

```
subclassResponsibility
   Self error: ‘My subclass should
      have overridden one of my
      messages.’
```

Object-Oriented Design:
A Responsibility-Driven Approach

Rebecca Wirfs-Brock
rebeccaw@orca.wv.tek.com
(503) 685-2561

P.O. Box 1000, Mail Station 61-028
Tektronix, Inc.
Wilsonville, OR 97070

Brian Wilkerson
(503) 242-0725

921 S.W. Washington, Suite 312
Instantiations, Inc.
Portland, OR 97205

ABSTRACT

Object-oriented programming languages support encapsulation, thereby improving the ability of software to be reused, refined, tested, maintained, and extended. The full benefit of this support can only be realized if encapsulation is maximized during the design process.

We argue that design practices which take a data-driven approach fail to maximize encapsulation because they focus too quickly on the implementation of objects. We propose an alternative object-oriented design method which takes a responsibility-driven approach. We show how such an approach can increase the encapsulation by deferring implementation issues until a later stage.

Introduction

The primary benefit of object-oriented programming is its ability to increase the value of a number of software metrics. These metrics include being able to reuse, refine, test, maintain, and extend the code. Yet the value of these metrics has been decreasing as the size of applications, and hence their complexity, has been increasing.

Object-oriented programming increases the value of these metrics by managing this complexity. The most effective tool available for dealing with complexity is abstraction. Many types of abstraction can be used, but encapsulation is the main form of abstraction by which complexity is managed in object-oriented programming.

Programming in an object-oriented language, however, does not ensure that the complexity of an application will be well encapsulated. Applying good programming techniques can improve encapsulation, but the full benefit of object-oriented programming can be realized only if encapsulation is a recognized goal of the design process.

The approach taken by a designer has a profound impact on the degree to which encapsulation is embodied in a design. We will describe the data-driven approach to design and why it fails to maximize encapsulation. We will then describe an alternative design approach, referred to as responsibility-driven, and explain why it results in designs with a higher degree of encapsulation.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1989 ACM 089791-333-7/89/0010/0071 $1.50

Data-Driven Design

Data-driven design is the result of adapting abstract data type design methods to object-oriented programming. The adaptation is straightforward because classes closely resemble abstract data types.

From a purely pragmatic point of view, objects encapsulate behavior (the implementation of an object's responsibilities) and structure (the other objects known directly by that object). This is similar to the definition of an abstract data type.

Before data-driven design is described, let us briefly review abstract data type design.

Abstract Data Type Design

An abstract data type is the encapsulation of data and the algorithms that operate on that data. Abstract data types are designed by asking the questions:

- What data does this type subsume?

  and

- What algorithms can be applied to this data?

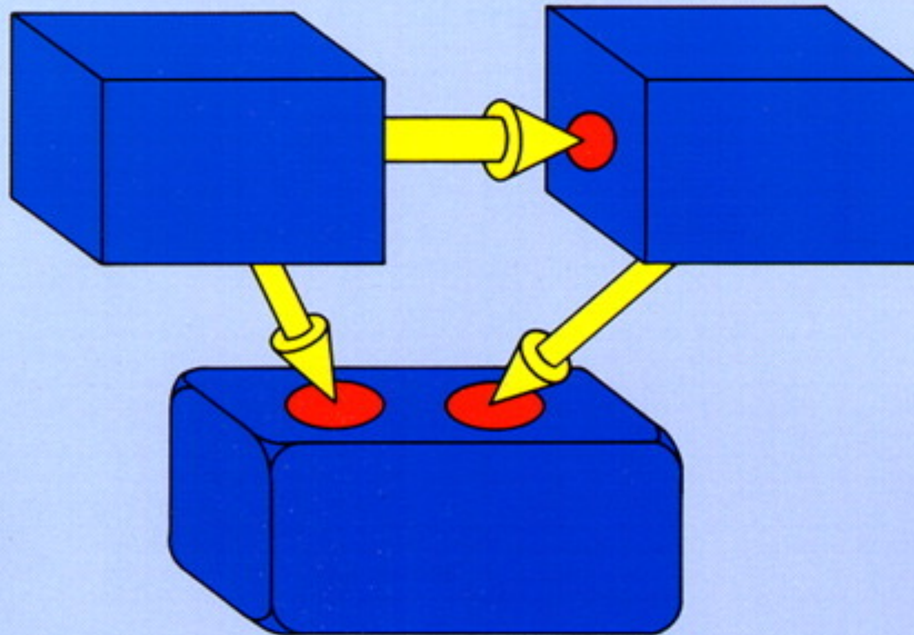
The primary focus of these questions is to determine what data is being represented in the system. This can be done initially by identifying the data required by the program (or perhaps only a portion of it). This data can then be grouped into types using cohesion as a guide. (Cohesion, as applied to a group of data, is a measure of how strongly related the parts of the group are.) Finally, identifying the algorithms associated with those types of data often leads to the discovery of other types that are required.

Definition of Data-Driven Design

In a data-driven design, objects are designed by asking the questions:

1989

Designing Object-Oriented Software



Rebecca Wirfs-Brock
Brian Wilkerson
Lauren Wiener

1990

Designing Object-Oriented Software cover art by Phil Brock

Class-Responsibility-Collaborator Cards

from Ward and Kent

"A Laboratory For Teaching Object-Oriented Thinking,"
Kent Beck, Apple Computer, Inc., Ward Cunningham, Wyatt Software Services, Inc.
OOPSLA 89

Model	
Maintain problem related info	
Broadcast change notification	

View	
Render the model	Model
Transform coordinates	Controller

Controller	
Interpret user input	Model
Distribute control	View

RDD emphasis…

roles

responsibilities

collaborations

informal tools and techniques

concepts and thinking tools

RDD Principles

1. Maximize Abstraction

Hide the distinction between data and behavior. Think of object responsibilities for “knowing”, “doing”, and “deciding”

Focus on what a class should do and how it should be used, first

Then decide on how to implement it

2. Distribute Behavior

Give objects responsibilities to perform operations based on what they know

Make objects smart— have them behave intelligently, not just hold bundles of data…but not too smart

Delegate responsibility

Responsibility-Driven Design Principles

3. Preserve Design Flexibility

Design objects so interior details can be readily changed

Hide implementation details: Do not share visibility of private “helper” classes or attributes

Create well-defined interfaces that are flexible

Implement code so that dependencies between classes are minimized

Understand design variations that need to be supported. Create places where your existing design can be extended

Starting From Different Points-of-View

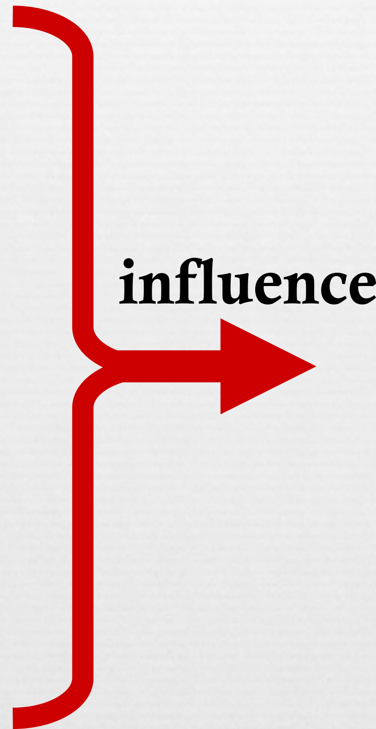
Data-Driven

Responsibility-Driven

Event-Driven

Rule-Based

Ad-Hoc



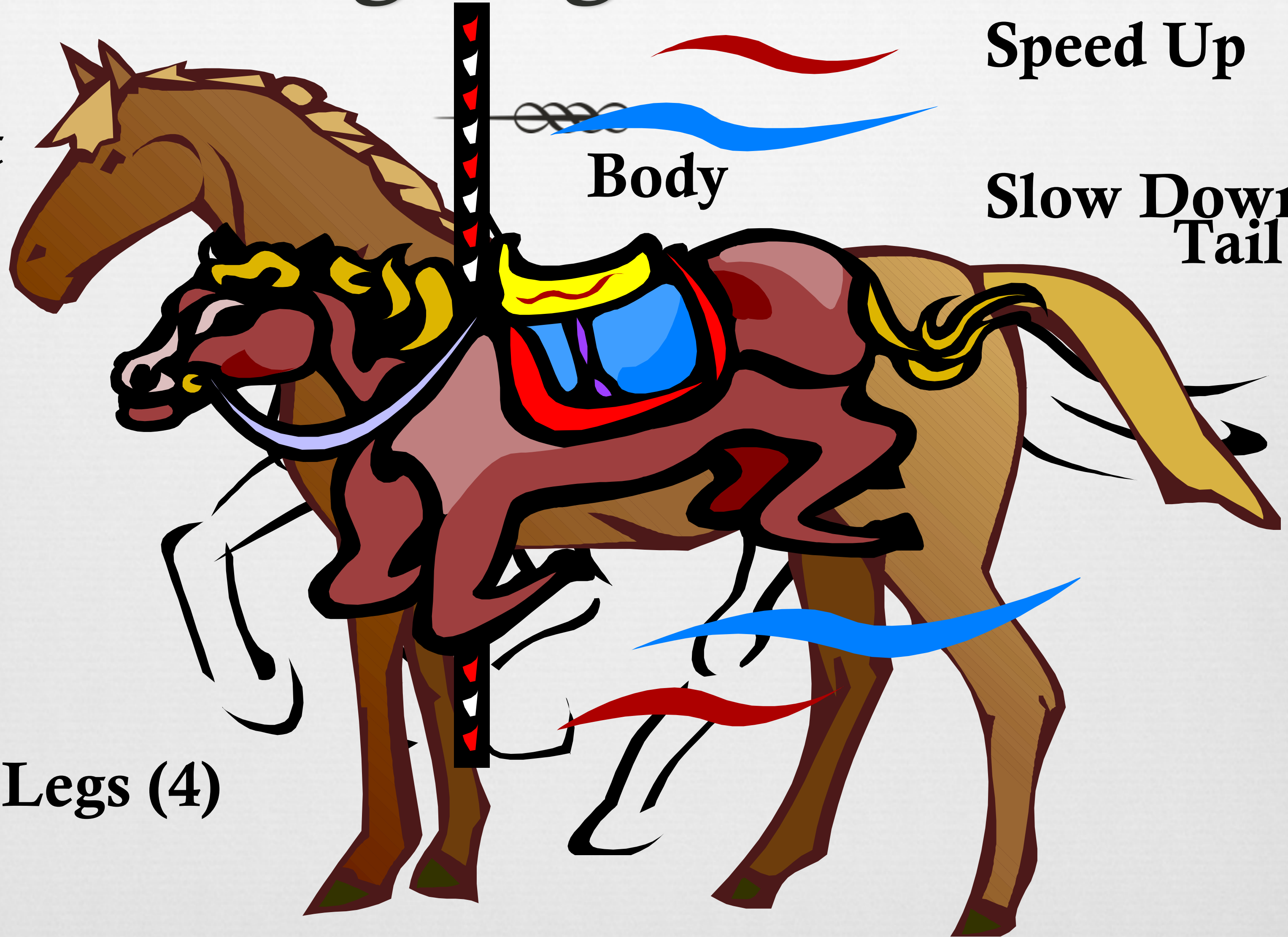
Choice of key abstractions

Distribution of data and behavior

Patterns of collaboration

Object visibilities

Designing a Horse



Designing a Horse Responsibly

A Responsibility Model…

Object Design

Roles, Responsibilities, and Collaborations

Rebecca Wirfs-Brock and Alan McKean

Forewords by Ivar Jacobson and John Vlissides

2002

Role Stereotypes:
A tool for seeing and shaping behaviors

stereotype—A conventional, formulaic, and oversimplified conception, opinion, or image

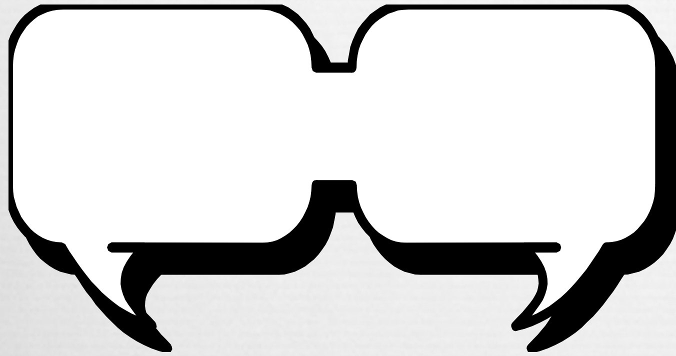
"Characterizing Your Objects", Rebecca Wirfs-Brock, February 1992 Smalltalk Report

Role Stereotypes

Information holder - knows and provides information

Structurer - maintains relationships between objects and information about those relationships

Role Stereotypes



Interfacer - translates information and requests

Service provider - performs work on demand

Role Stereotypes

- mechanically reacts to events

Controller - makes decisions while closely directing others' actions

Using Role Stereotypes

1. Think about objects or components needed
2. Study a design
3. Blend roles to make objects more responsible
   - information holders that compute
   - service providers that maintain information
   - structurers that derive facts
   - interfacers that transform

2.
Pulling up a level…to compare

BOEING

FIGURE 6.3-10 TRANSFER A BATCH

BOEING

FIGURE 7.3-10 TRANSFER A BATCH

"The Object-Oriented Brewery: A Comparison of Two Object-Oriented Methods,"R. Sharble and S. Cohen, Boeing Technical Report BCS-G4059, 1992.

"How Designs Differ", R. Wirfs-Brock, Smalltalk Report, vol. 1, no. 4

…and characterize

Data-Driven Design Approach	Responsibility-Driven Design Approach
centralized control	delegated control
controllers	coordinators
inherited attributes	inherited behavior
many low-level messages	fewer, higher-level messages
lots of simplistic information holders	a few smart objects that blend role stereotypes

2.
Identifying Role Stereotypes in Patterns

A Mediator is a coordinator

A Strategy is a service provider

State objects are too..

An Adapter is an interfacer

3. Blending Stereotypes
The Whole Value Pattern

Classes that represent meaningful domain quantities

- Examples: currency, calendar periods, temperature, color, weight, brightness.
  - Color (50% red, 30% green, 10% blue)
  - Temperature (75 degrees Fahrenheit)
  - Currency (100 U.S. Dollars)

Hold information *and* perform comparisons and translations

Streamlining Collaborations

- trust region—an area where trusted collaborations occur

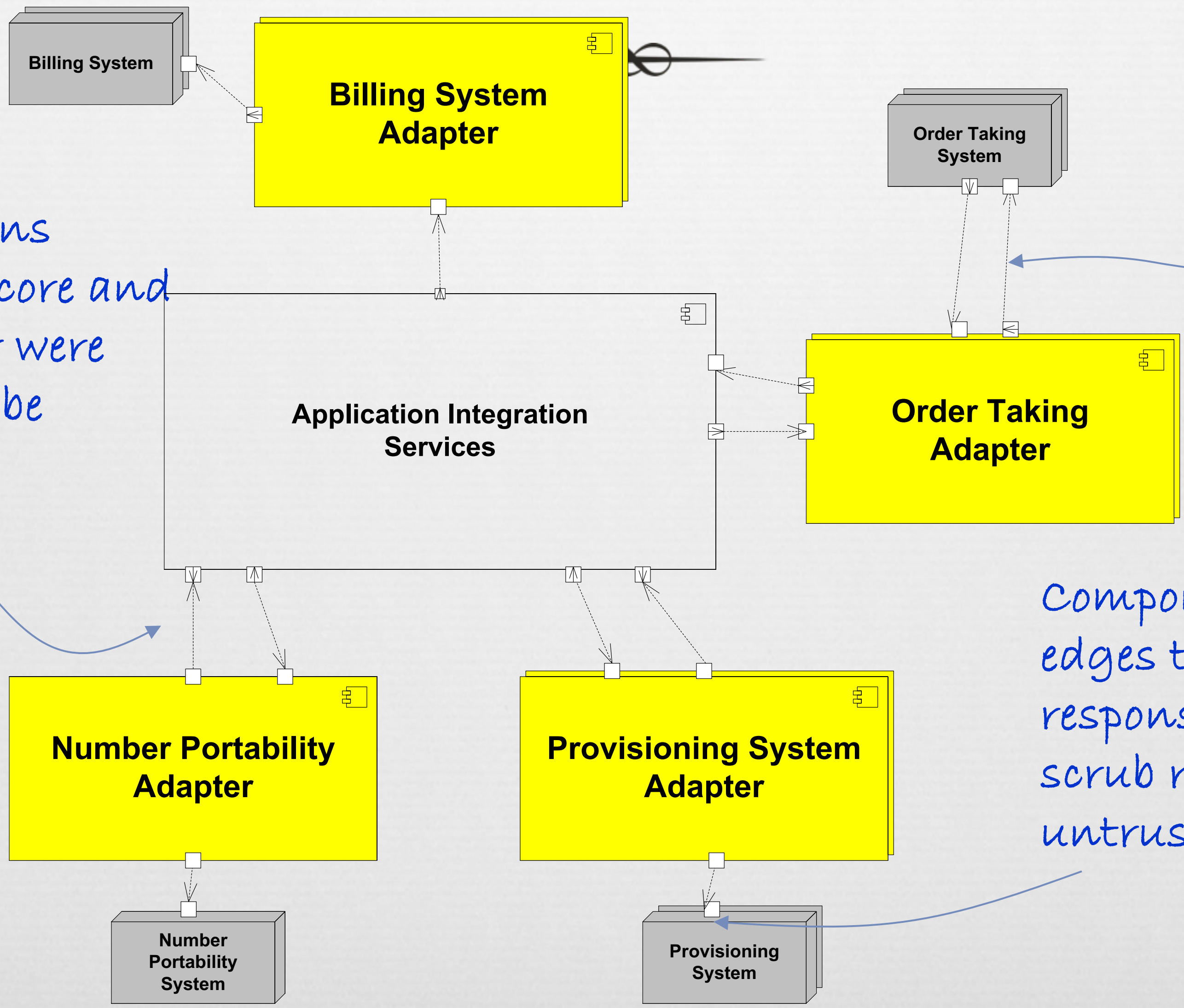
Collaborate

To work together, especially in a joint intellectual effort

Collaborate

To cooperate treasonably, as with an enemy occupation force

Trust In A Telco Integration Application



Collaborations between the core and any adapter were designed to be trusted

Components at the edges take on extra responsibility to scrub requests from untrusted sources

Influential Early Object Design Approaches

Shlaer-Mellor

Booch method

Object Modeling Technique

Objectory

→ Rational Unified Process

OORAM – Trygve Reenskaug → BabyUML

The Driven Meme

RDD ← started it!!!!

DDD

FDD

Agile Practices & Approaches

AMDD

TDD

BDD

ATDD

xx-Driven Design

Responsibility-Driven Design – Rebecca Wirfs-Brock, Brian Wilkerson, Lauren Weiner, Alan McKean

Data-Driven Design

Domain-Driven Design – Eric Evans

Test-Driven Design – Kent Beck

x-Driven Development

Test-Driven Development – Kent Beck

Behavior Driven Development – Dan North

Contract-Driven Development AKA Design by Contract™ – Bertrand Meyer

Agile Model-Driven Development – Scott Ambler

Feature Driven Development – Jeff De Luca and Peter Coad

Model-Driven Development™ – OMG

Model-Driven Engineering

Robert Martin's S.O.L.I.D. Principles

Single Responsibility Principle (SRP): A class should have only one reason to change.

Open-Closed Principle (OCP): Extending a class shouldn't require modifying that class.

Liskov Substitution Principle (LSP): Subclasses should be substitutable for their superclasses.

Interface Segregation Principle (ISP): Users of a class should not be forced to depend on interfaces they do not need.

Dependency-Inversion Principle (DIP): Abstractions should not depend on details. Details should depend on abstractions.

Agile Software Development: Principles, Patterns, and Practices (Prentice Hall, 2003)

Major Differences

Design rhythms

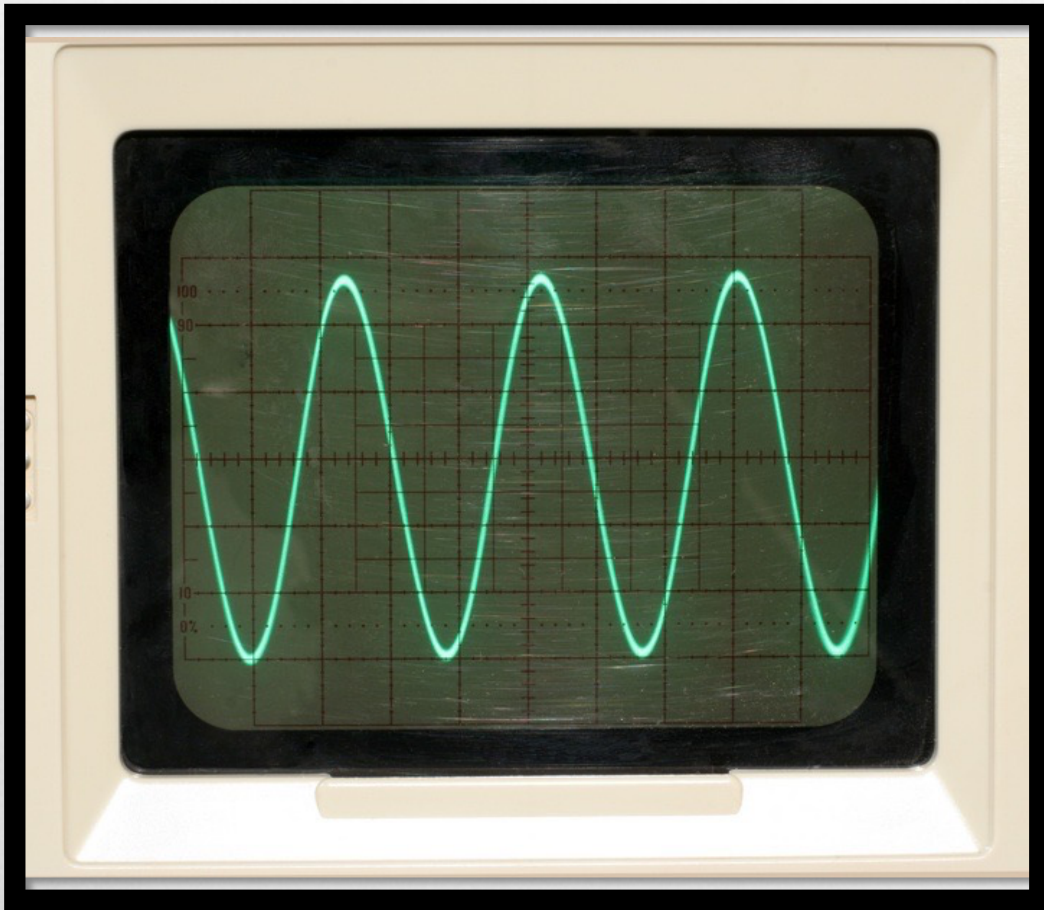
Focus

Artifacts

Properties of “good” software

Ownership

Emphasis



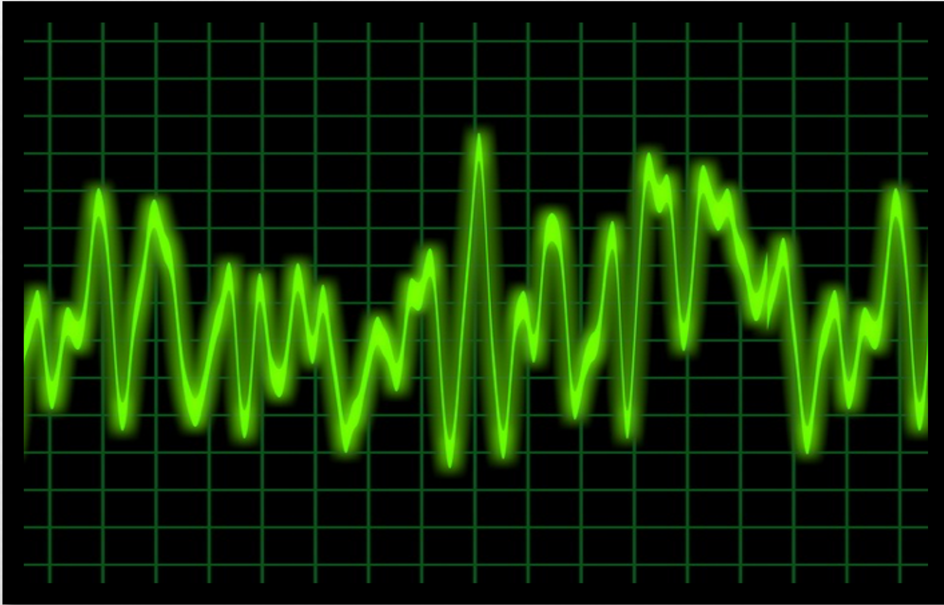
FDD Design Rhythm

Feature by Feature:

- Domain walk through
- Design
- Design Inspection
- Code
- Code Inspection

Design in the first week, code in the second

TDD Design Rhythm

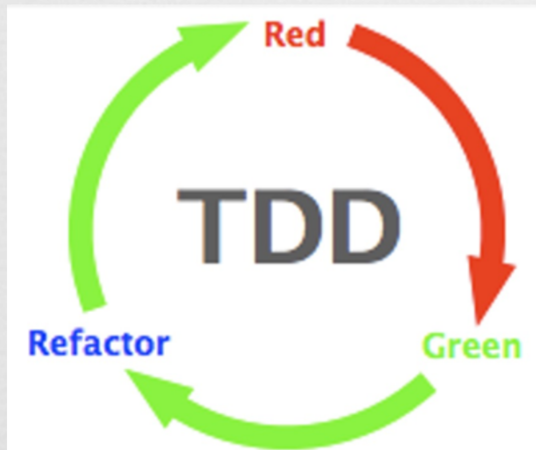


Story-by-story:

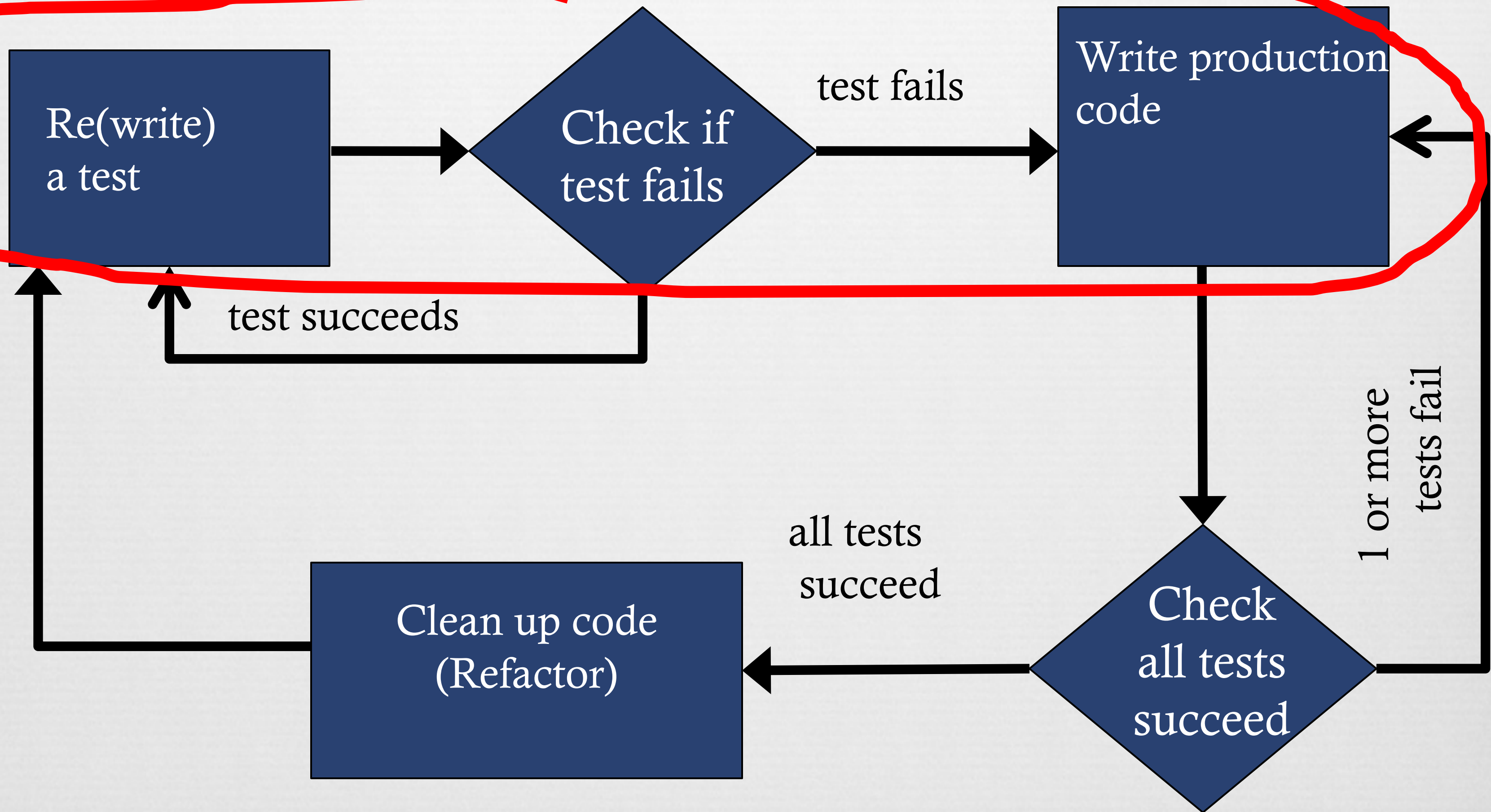
- Write the simplest test
- Run the test and make it fail
- Write the simplest code that will pass the test

Repeat until a “story” is tested and implemented

Design between the keystrokes

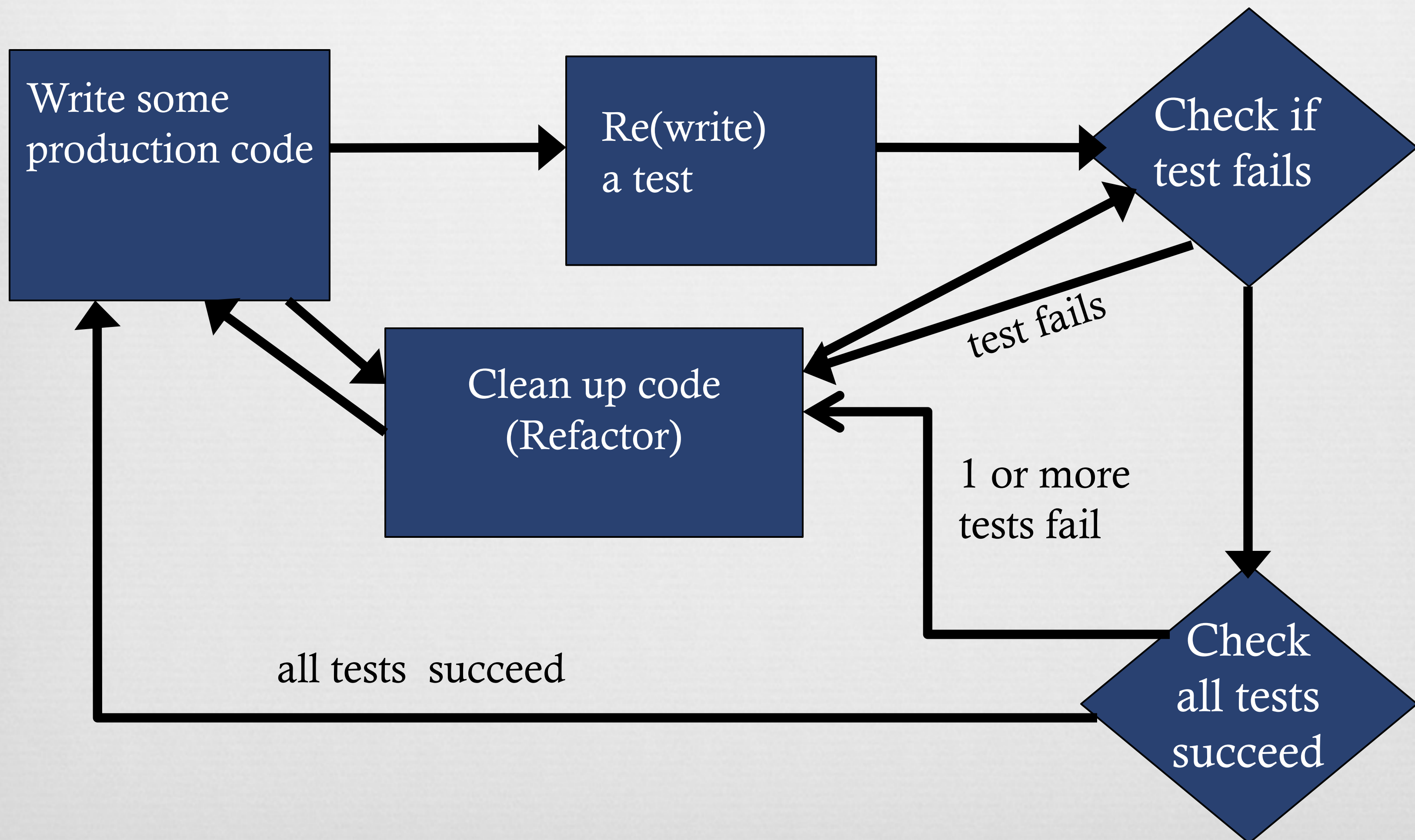


Test-First Development

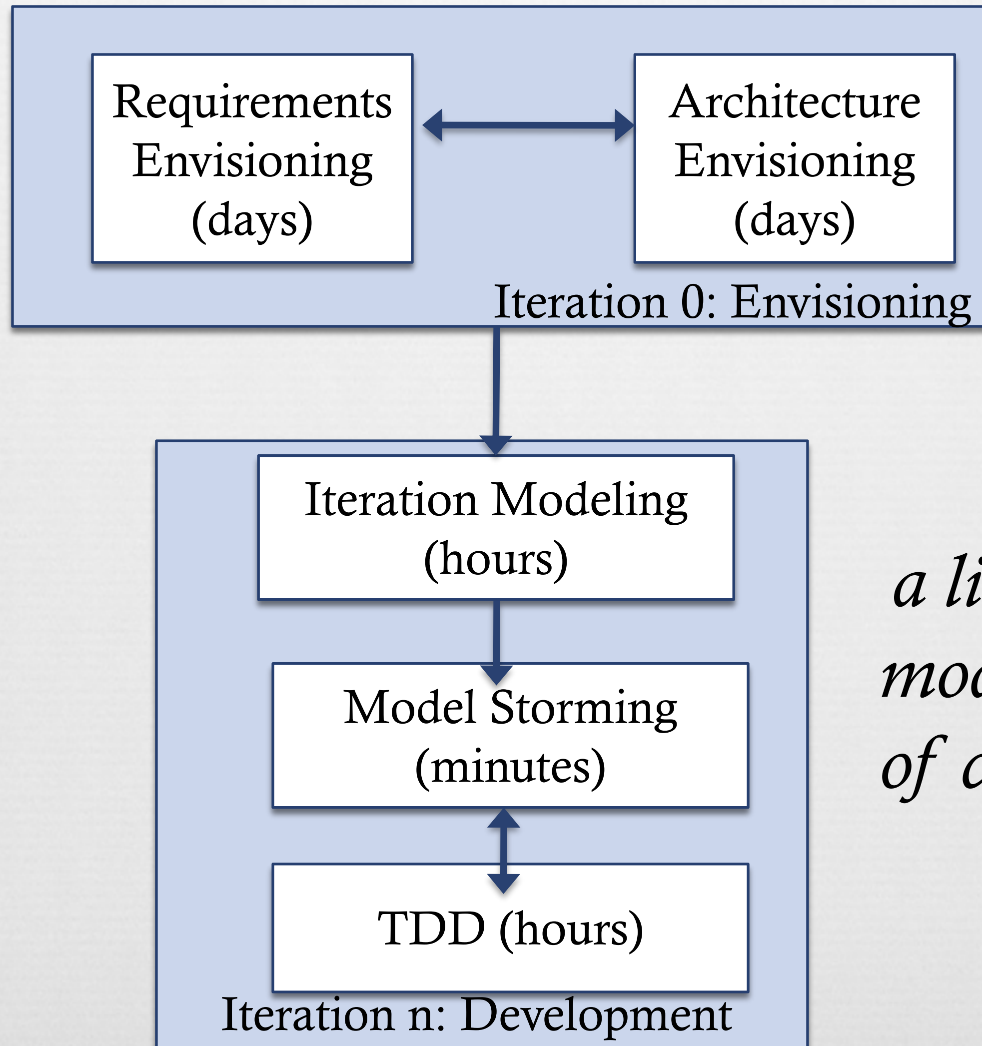


Test-Frequent Development

Tests don't always get written first.
Tests written & must pass before checking in production code.



Agile Model-Driven Development



a little bit of modeling then a lot of coding

Behavior-Driven Development

Specifications of desired behavior

Acceptance scenario structure:

Given some initial context **when a**n event occurs **then** ensure some outcomes.

BDD Example

WindowControl should close windows

```
public class WindowControlBehavior {

    @Test
    public void shouldCloseWindows() {

        // Given
        WindowControl control = new WindowControl("My AFrame");
        AFrame frame = new AFrame();

        // When
        control.closeWindow();

        // Then
        ensureThat(!frame.isShowing());
    }
}
```

"a rephrasing of existing good practice…not a radical departure"

Domain-Driven Design

focus on domain model

Entity object—distinguished by *who* it is.
Has lifecycle, can change form.

"You are who you are and you are unique."

Value object—Needn't be unique. Typically describes some characteristic.

"I don't care which blue crayon I use, just that I have one."

Eric Evans, Domain-Driven Design, Addison-Wesley, 2004

Domain-Driven Design

focus on “strategic” design

A context map

Model in Context

Translation
Map

Model in Context

preserving, protecting,
and strengthening
domain concepts

Software Design Values

Expressive

Understood

Coherent

Suited for use

Testable

Predictable

Changeable

Software Design Values

Habitable Software

places where designers feel comfortable growing, extending their designs and living with them for a period of time

Sustainable Design

Stewardship

- Follow through
- Ongoing attention
- Not ignoring the little things that can undermine your ability to grow, change and adapt your software

Firefox
rebecca@wirfs-brock.com
twitter: @rebeccawb