SEG 2105 - LECTURE 09

ARCHITECTING AND DESIGNING SOFTWARE

design verb

de·sign | \ di-ˈzīn \

designed; **designing**; **designs**

Definition of *design* (Entry 1 of 2)

transitive verb

1 : to create, fashion, execute, or construct according to plan : DEVISE, CONTRIVE
// *design* a system for tracking inventory

2 **a** : to conceive and plan out in the mind
// he *designed* the perfect crime

b : to have as a purpose : INTEND
// she *designed* to excel in her studies

c : to devise for a specific function or end
// a book *designed* primarily as a college textbook
// a suitcase *designed* to hold a laptop computer

4 **a** : to make a drawing, pattern, or sketch of
// ... a curious woman whose dresses always looked as if they had been *designed* in a rage ...
— Oscar Wilde

b : to draw the plans for
// *design* a building
// *designing* a new bike

Design is a problem-solving process whose objective is to find and describe a way

- To implement the functional requirements
- Within the constraints imposed by the
  - Quality,
  - Platform,
  - Process requirements,
  - Budget, ...
- Adhering to good quality

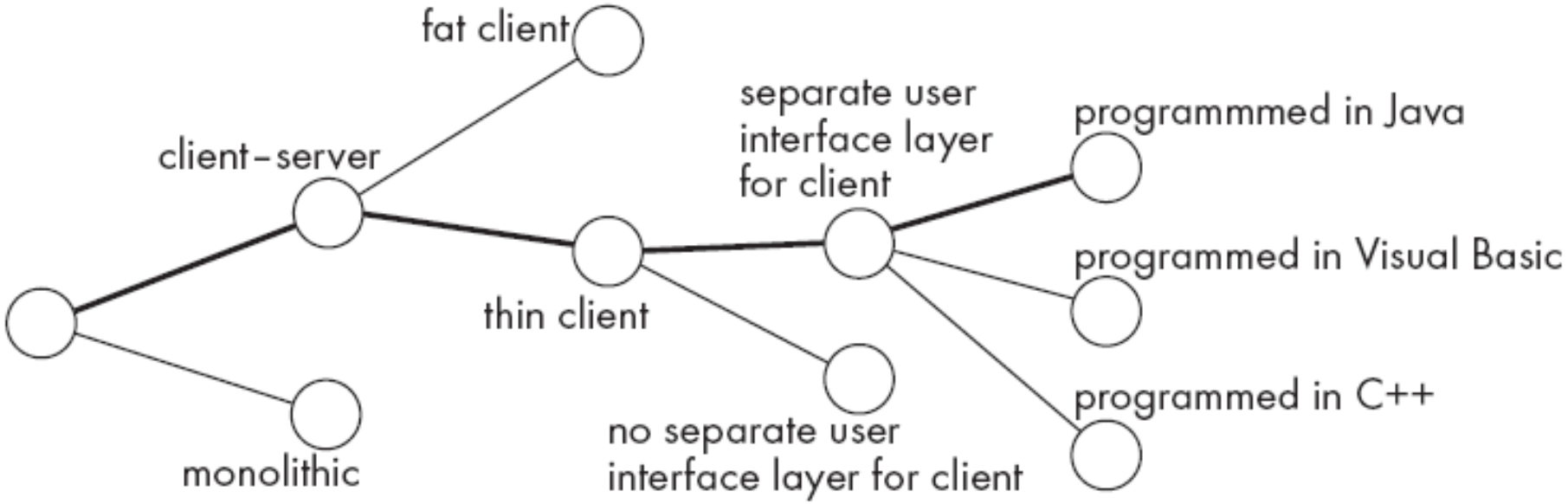
Design as a series of decisions

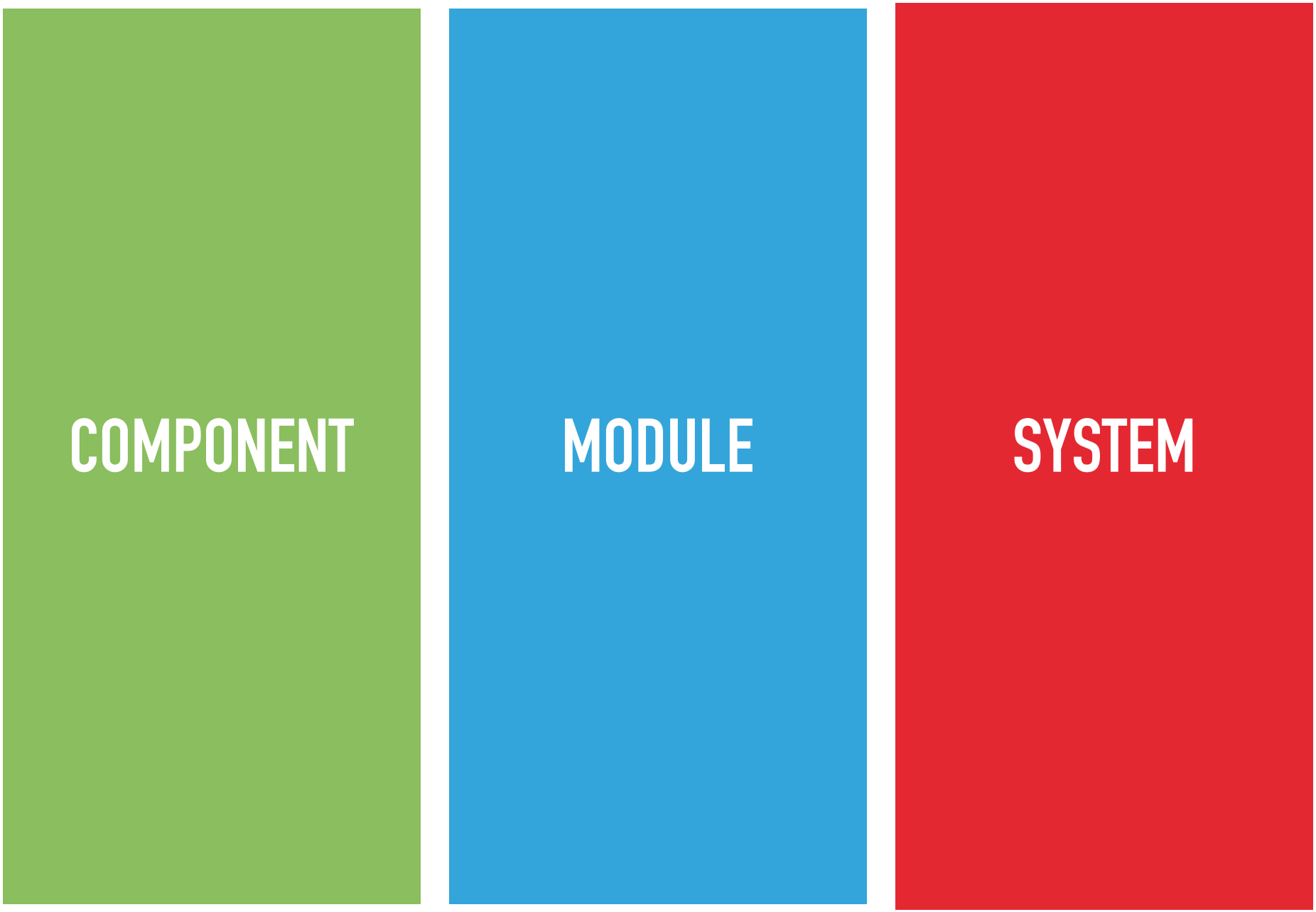
- Sub-problems of overall design problem
- Several alternative solutions / design options
- Decisions to resolve each issue
  - Choosing the most appropriate option
  - Among the available alternatives

Making decisions, using knowledge of

DESIGN SPACE

- The space of possible designs that could be achieved by choosing different sets of alternatives is often called the **design space**




COMPONENT
MODULE
SYSTEM

Component

- Software or Hardware that has a clear role
- Isolated to allow its replacement
  - Much harder in software (e.g. MySql vs Postgres)
  - Much easier in hardware (e.g. Replacing servers)
- Designed to be re-usable
- Some are special-purpose

Module

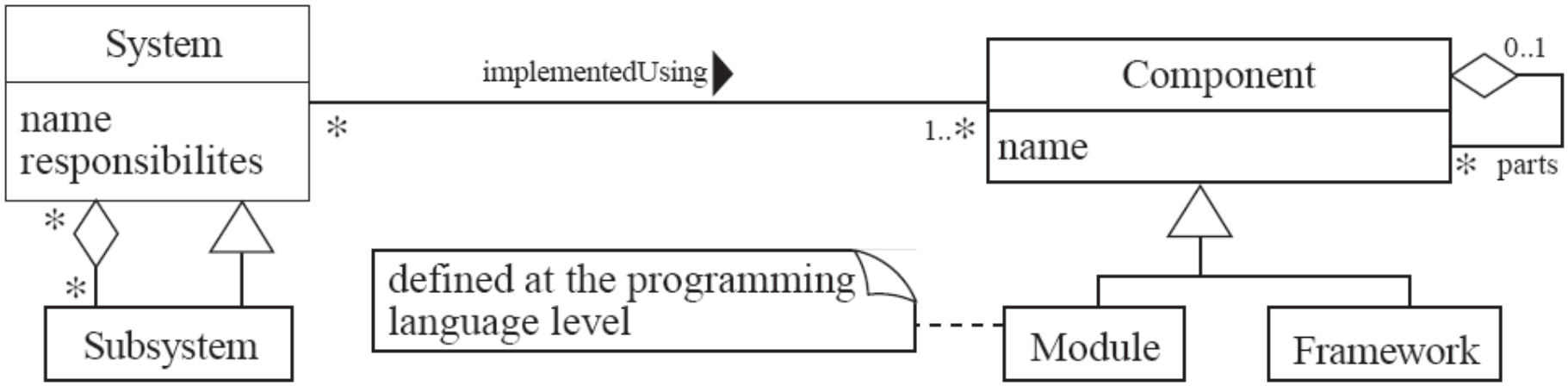
- A component at the programming language level
- In Java,
  - Packages,
  - Classes, and
  - Methods

System

- Logical entity
- Definable responsibilities (or objectives)
- Hardware, software or both
- Can have
  - Specification (and then implemented by a components)
  - Exists even if components are changed (or replaced)
- Requirements analysis identifies the responsibilities of a system

UML DIAGRAM OF SYSTEM PARTS

- Subsystem
  - A system that is part of a larger system,
  - Has an explicit interface



TOP DOWN DESIGN

BOTTON UP DESIGN

BOTH TOP DOWN AND BOTTOM UP DESIGN

Top-down design

- Design very high level structure
- Work down to detailed decisions about low-level constructs
- Arriving at detailed decisions such as:
  - Data formats
  - Algorithms
  - API Interfaces

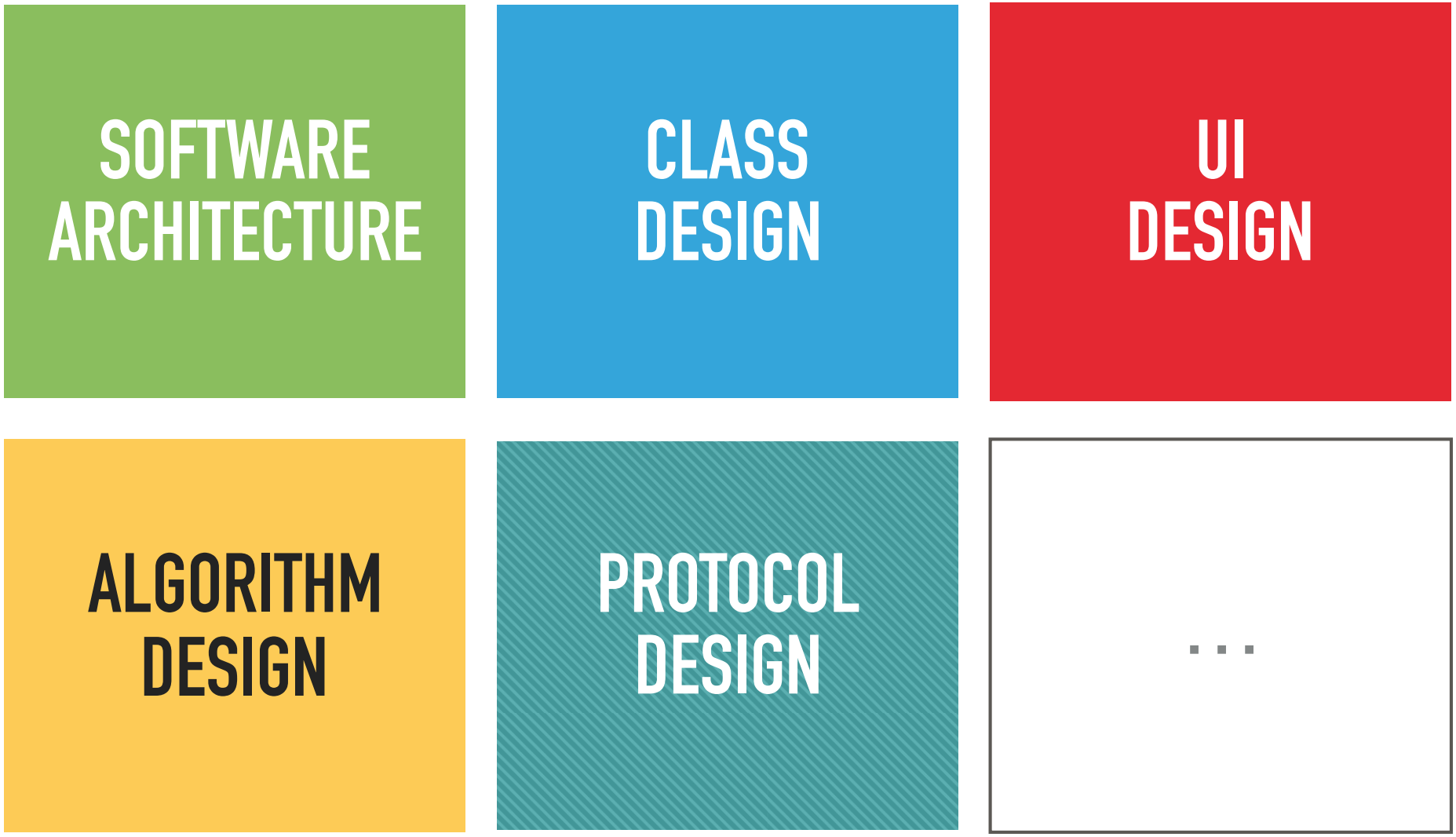
Bottom-Up Design

- Decisions about re-usable low-level utilities
- Decisions on piecing them together
  - Creating higher-level constructs

Mixing Top-Down and Bottom-Up

- A mixture of both normally used
- Top-Down gives a good structure
- Bottom-Up promotes re-usable components

Different aspects of design



Principles Leading to Good Design

INCREASE PROFIT

REDUCE COSTS

INCREASE REVENUE

CONFORM TO REQUIREMENTS

ACCELERATE DEVELOPMENT

INCREASE ...ILITIES

Design Principles

1. Divide and Conquer
2. Increase Cohesion
3. Decrease Coupling
4. Increase Abstractions
5. Increase Re-usability
6. Re-use if possible

Design Principle 1: Divide and conquer

Trying to deal with something big all at once is normally much harder than dealing with a series of smaller things

- Separate people can work on each part.
- An individual software engineer can specialize.
- Each individual component is smaller, and therefore easier to understand.
- Parts can be replaced or changed without having to replace or extensively change other parts.

Ways of dividing a software system

- A distributed system is divided up into clients and servers
- A system is divided up into subsystems
- A subsystem can be divided up into one or more packages
- A package is divided up into classes
- A class is divided up into methods

Design Principles

1. DIVIDE AND CONQUER

2. INCREASE COHESION

3. DECREASE COUPLING

4. INCREASE ABSTRACTIONS

5. INCREASE RE-USABILITY

6. RE-USE IF POSSIBLE

Design Principle 2: Increase cohesion where possible

A subsystem or module has high cohesion if it keeps together things that are related to each other, and keeps out other things

- This makes the system as a whole easier to understand and change
- Type of cohesion:
  - Functional,
  - Layer,
  - Communicational
  - Sequential,
  - Procedural,
  - Temporal,
  - Utility

Functional cohesion

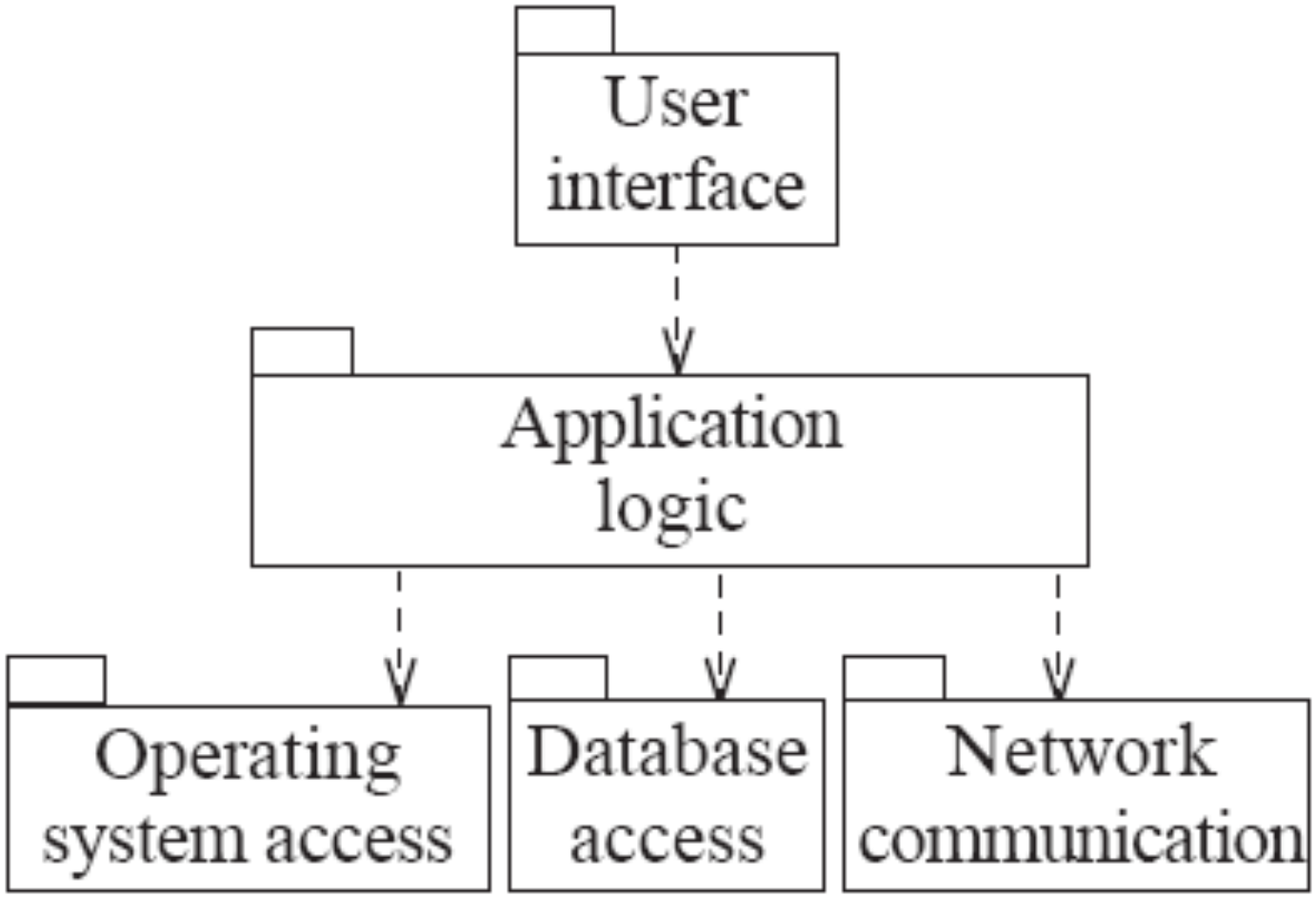
This is achieved when all the code that computes a particular result is kept together - and everything else is kept out

- i.e. when a module only performs a single computation, and returns a result, without having side-effects.
- Benefits to the system:
  - Easier to understand
  - More reusable
  - Easier to replace
- Modules that update a database, create a new file or interact with the user are not functionally cohesive

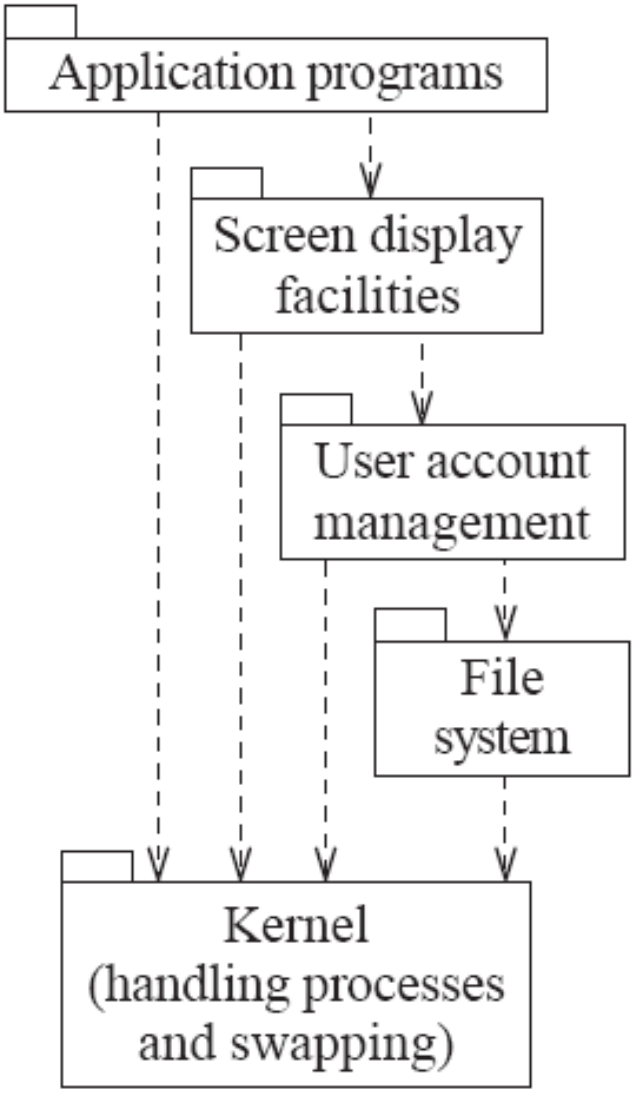
Layer cohesion

All the facilities for providing or accessing a set of related services are kept together, and everything else is kept out

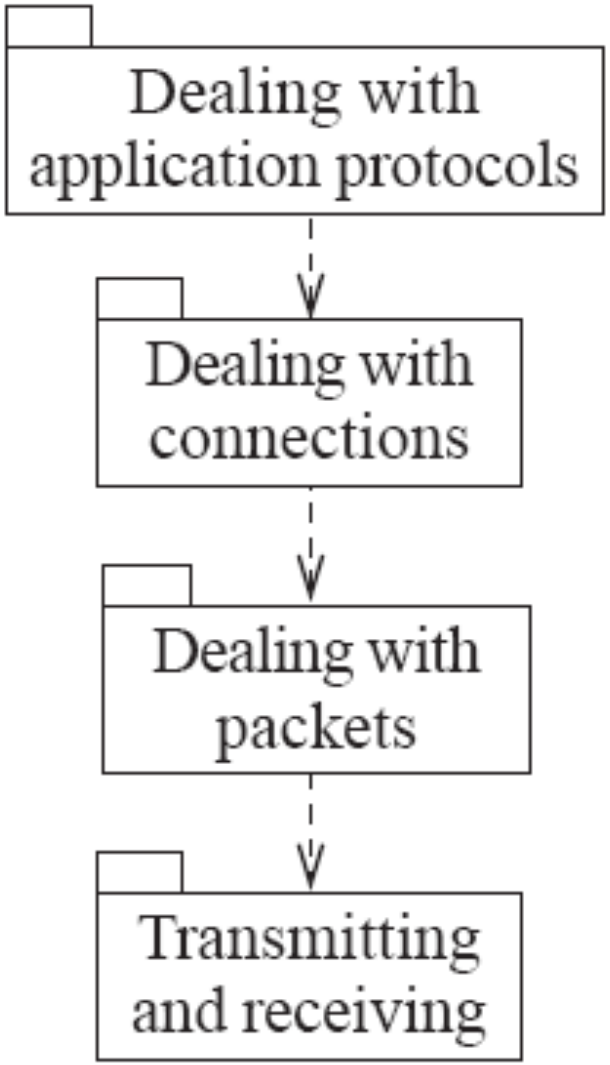
- The layers should form a hierarchy
  - Higher layers can access services of lower layers,
  - Lower layers do not access higher layers
- The set of procedures through which a layer provides its services is the application programming interface (API)
- You can replace a layer without having any impact on the other layers
  - You just replicate the API



(a) Typical layers in an application program



(b) Typical layers in an operating system



(c) Simplified view of layers in a communication system

Communicational cohesion

All the modules that access or manipulate certain data are kept together (e.g. in the same class) - and everything else is kept out

- A class would have good communicational cohesion
  - if all the system's facilities for storing and manipulating its data are contained in this class.
  - if the class does not do anything other than manage its data.
- Main advantage: When you need to make changes to the data, you find all the code in one place

Sequential cohesion

Procedures, in which one procedure provides input to the next, are kept together – and everything else is kept out

- You should achieve sequential cohesion, only once you have already achieved the preceding types of cohesion.

Procedural cohesion

Procedures that are used one after another are kept together

- Even if one does not necessarily provide input to the next.
- Weaker than sequential cohesion.

Temporal Cohesion

Operations that are performed during the same phase of the execution of the program are kept together, and everything else is kept out

- For example, placing together the code used during system start-up or initialization.
- Weaker than procedural cohesion.

Utility cohesion

When related utilities which cannot be logically placed in other cohesive units are kept together

- A utility is a procedure or class that has wide applicability to many different subsystems and is designed to be reusable.
- For example, the **java.lang.Math** class.

Design Principles

1. Divide and Conquer
2. Increase Cohesion
3. **Decrease Coupling**
4. Increase Abstractions
5. Increase Re-usability
6. Re-use if possible

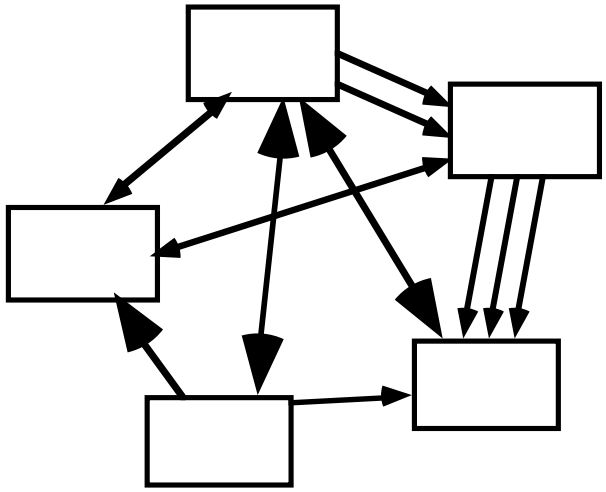
Design Principle 3: Reduce coupling where possible

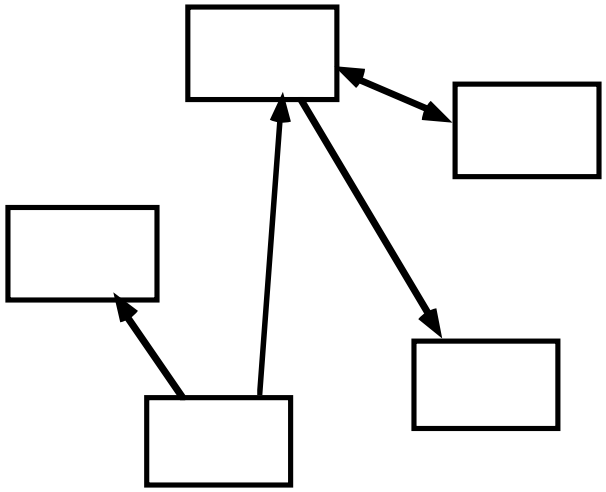
Coupling occurs when there are interdependencies between one module and another

- When interdependencies exist, changes in one place will require changes somewhere else.
- A network of interdependencies makes it hard to see at a glance how some component works.

Type of coupling

- Content,
- Common,
- Control,
- Stamp,
- Data,
- Routine Call,
- Type use,
- Inclusion/Import,
- External





Content coupling

Occurs when one component surreptitiously modifies data that is internal to another component

- To reduce content coupling you should therefore encapsulate all instance variables
  - declare them private
  - and provide get and set methods
- A worse form of content coupling occurs when you directly modify an instance variable of an instance variable

```
public class Line
{
  private Point start, end;
  ...
  public Point getStart() { return start; }
  public Point getEnd()  { return end; }
}

public class Arch
{
  private Line baseline;
  ...
  void slant(int newY)
  {
    Point theEnd = baseline.getEnd();
    theEnd.setLocation(theEnd.getX(),newY);
  }
}
```

Example of content coupling

Common coupling

Using a global variable

- All components using the global are now coupled
- A weaker form is when a variable can be accessed by a subset of the system's classes
  - e.g. a Java package
- Acceptable for global variables of system-wide default values
- The Singleton pattern encapsulates global access to an object, but it is still a global.

Control coupling

Calling a procedure with a '**flag**' or '**command**' to control its behaviour

- To make a change you have to change both the
  - Calling method (that send the *command*) and
  - Called method (that interprets the *command*)
- Polymorphic operations a good way to avoid control coupling
- One way to reduce control coupling
  - Using a look-up table
  - Commands are mapped to a the callable method

```
public routineX(String command)
{
   if (command.equals("drawCircle")
   {
      drawCircle();
   }
   else
   {
      drawRectangle();
   }
}
```

More of a flag to instruct which path to take

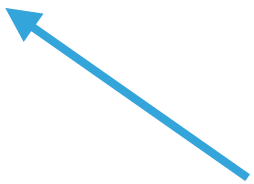
Caller / callee now coupled on those commands.

Stamp coupling

An application class is declared as the type of a method argument

- Since one class now uses the other,
  - changing either becomes harder
  - Reusing one class requires reusing the other
- Two ways to reduce stamp coupling,
  - Use an interface as the argument type
  - Passing simple variables (beware of content coupling)

```
public class Emailer
{
  public void sendEmail(Employee e, String text)
  {...}
  ...
}
```



Send email only works with an Employee (Stamp coupling)

```
public class Emailer
{
  public void sendEmail(String name, String email, String text)
  {...}
  ...
}
```

Using simple data avoids the need to couple to an Employee

```
public interface Addressee
{
  public abstract String getName();
  public abstract String getEmail();
}

public class Employee implements Addressee {...}

public class Emailer
{
  public void sendEmail(Addressee e, String text)
  {...}
  ...
}
```

Now you can continue to pass in Employee without it being coupled to the Emailer

Or use an interface to encapsulate the required fields

Data coupling

The types of method arguments are primitive or simple library classes

- The more arguments, the higher the coupling
- Using that method means passing all those arguments
- Avoid it by limiting unnecessary arguments
- There is a trade-off (increasing one often decreases the other) between
  - Data Coupling and
  - Stamp coupling

Routine call coupling

One routine (or method) calls another

- The routines are coupled as they depend on each other's behaviour
- Routine call coupling is always present in any system.
  - But should still be considered as coupling
  - Is sequences of methods are use repeatedly then reduce call coupling by encapsulating into a single method

Type use coupling

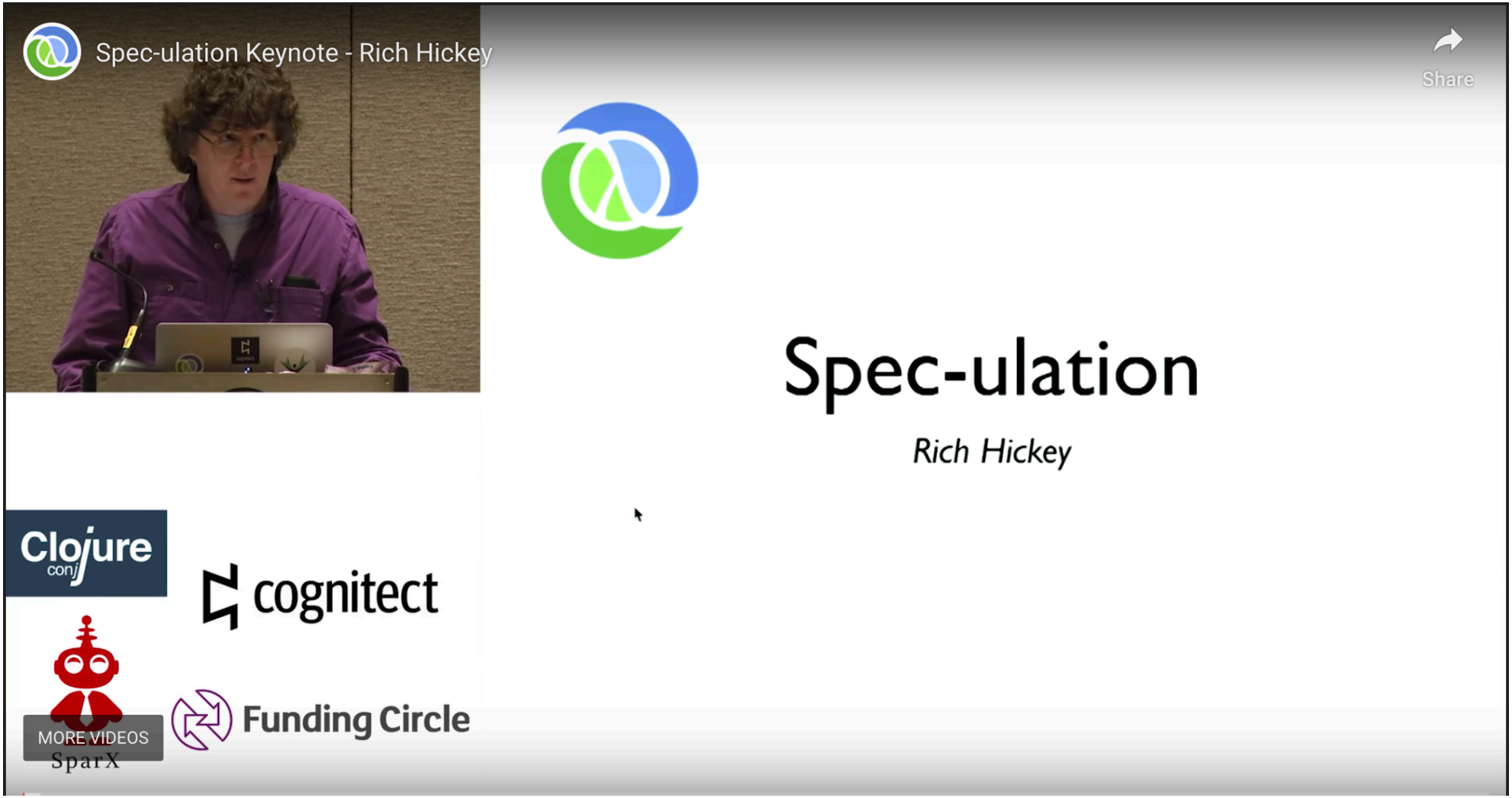
A module uses a data type defined in another module

- Any time a class declares a variable of another class' type
- If the type definition changes, then the declaring classes might be affected
- Always declare the type of a variable to be the most general possible class or interface that contains the required operations

Inclusion or import coupling

One component imports a package (as in Java) or includes another (as in C++).

- The including/importing component is now
  - exposed to everything in the included or imported component.
- If the included/imported component changes
  - The including/importing component might have to change
  - Can cause conflict forcing a change (e.g. duplicate names)



https://youtu.be/oyLBGkS5ICk

External coupling

A module has a dependency on external things like the operating system, shared libraries or hardware

- Reduce the number of places where such dependencies exist (high cohesion).
- The Façade design pattern helps

Design Principles

1. DIVIDE AND CONQUER
2. INCREASE COHESION
3. DECREASE COUPLING
4. INCREASE ABSTRACTIONS
5. INCREASE RE-USABILITY
6. RE-USE IF POSSIBLE

Design Principle 4: Increase abstraction

Ensure that your designs allow you to hide or defer consideration of details, thus reducing complexity

- A good abstraction is said to provide information hiding
- Abstractions allow you to understand the essence of a subsystem without having to know unnecessary details

NOVEMBER 11, 2002 *by* JOEL SPOLSKY

The Law of Leaky Abstractions

TOP 10, ROCK STAR DEVELOPER, NEWS

There's a key piece of magic in the engineering of the Internet which you rely on every single day. It happens in the TCP protocol, one of the fundamental building blocks of the Internet.

TCP is a way to transmit data that is *reliable*. By this I mean: if you send a message over a network using TCP, it will arrive, and it won't be garbled or corrupted.

We use TCP for many things like fetching web pages and sending email. The reliability of TCP is why every email arrives in letter-perfect condition. Even if it's just some dumb spam.

https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/

Examples of abstractions:

- Classes
- UML associations
- Interfaces
- State machines
- Domain specific languages (DSLs)

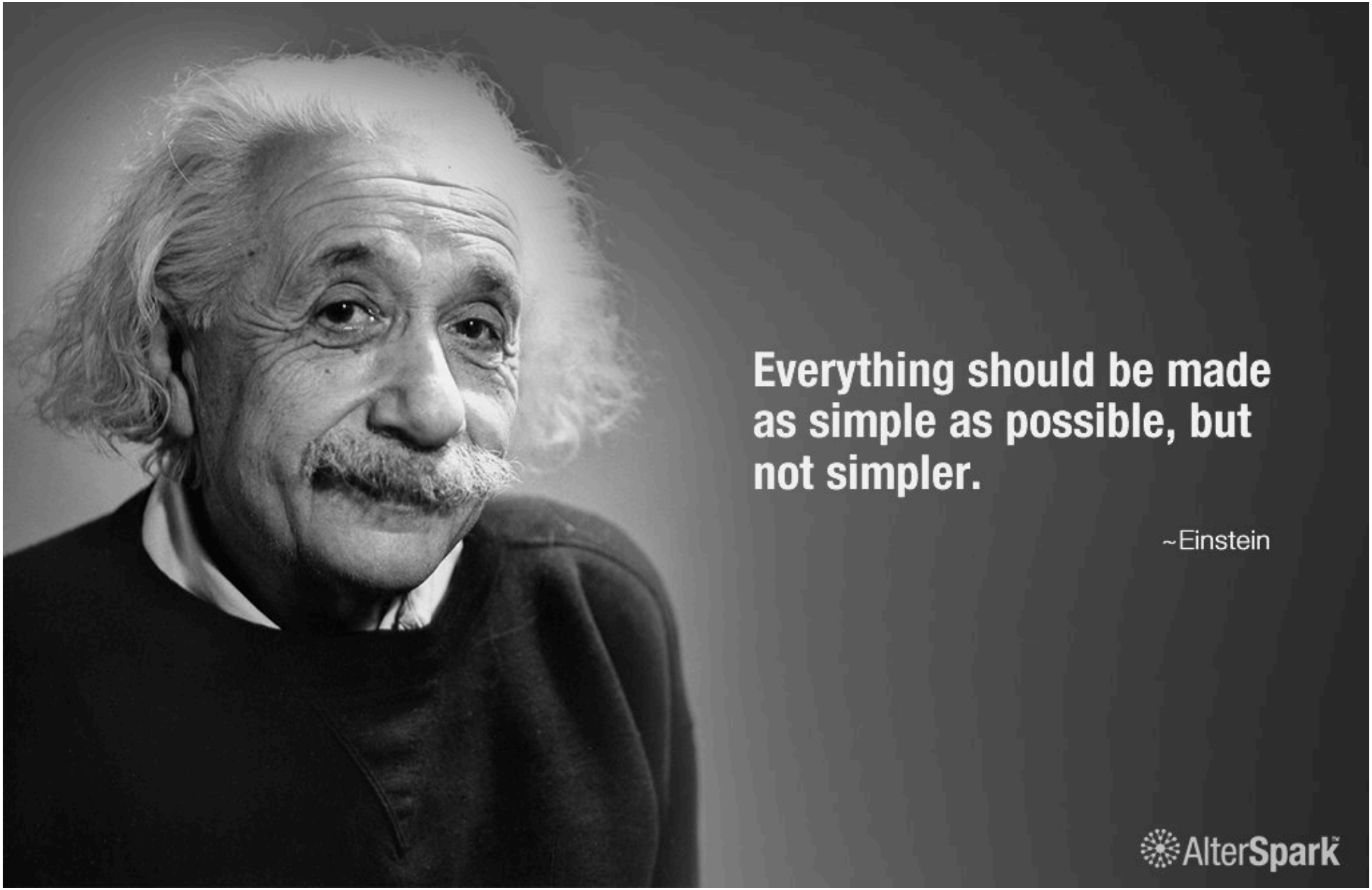
Design Principles

1. DIVIDE AND CONQUER
2. INCREASE COHESION
3. DECREASE COUPLING
4. INCREASE ABSTRACTIONS
5. INCREASE RE-USABILITY
6. RE-USE IF POSSIBLE

Design Principle 5: Increase reusability where possible

Design the various aspects of your system so that they can be used again in other contexts

- Generalize your design
- Follow the preceding three design principles
- Design your system with hooks
- Simplify your design


Everything should be made as simple as possible, but not simpler.
~Einstein
AlterSpark

Design Principles

1. DIVIDE AND CONQUER
2. INCREASE COHESION
3. DECREASE COUPLING
4. INCREASE ABSTRACTIONS
5. INCREASE RE-USABILITY
6. RE-USE IF POSSIBLE

Design Principle 6: Reuse where possible

Design with reuse is complementary to design for reusability

- Actively reusing designs or code allows you to take advantage of the investment you or others have made in reusable components
- Cloning should not be seen as a form of reuse

How one programmer broke the internet by deleting a tiny piece of code

By Keith Collins • March 27, 2016

leftpad.js package.json

```
1  module.exports = leftpad;
2  function leftpad (str, len, ch) {
3    str = String(str);
4    var i = -1;
5    if (!ch && ch !== 0) ch = ' ';
6    len = len - str.length;
7    while (++i < len) {
8      str = ch + str;
9    }
10   return str;
11 }
```

https://qz.com/646467/how-one-programmer-broke-the-internet-by-deleting-a-tiny-piece-of-code/

Design Principles

7. FLEXIBILITY	8. ANTICIPATE OBSOLESCENCE	9. PORTABILITY
10. TESTABILITY	11. DEFENSIVELY	. . .

Design Principle 7: Design for flexibility

Actively anticipate changes that a design may have to undergo in the future, and prepare for them

- Reduce coupling and increase cohesion
- Create abstractions
- Do not hard-code anything (magic numbers)
- Leave all options open
  - Do not restrict the options of people who have to modify the system later
- Use reusable code and make code reusable

Design Principles

7. FLEXIBILITY	**8. ANTICIPATE OBSOLESCENCE**	9. PORTABILITY
10. TESTABILITY	11. DEFENSIVELY	. . .

Design Principle 8: Anticipate obsolescence

Plan for changes in the technology or environment so the software will continue to run or can be easily changed

- Avoid early releases of technology
- Avoid software libraries specific to one environment
- Avoid undocumented features
- Avoid software / hardware from companies unlikely to provide long-term support
- Use standard languages and technologies that are supported by multiple vendors

Google Open Source Blog

The latest news from Google on open source releases, major projects, events, and student outreach programs.

Bidding farewell to Google Code

Thursday, March 12, 2015

When we started the Google Code project hosting service in 2006, t
limited. We were worried about reliability and stagnation, so we too
source community another option to choose from. Since then, we've
project hosting services such as GitHub and Bitbucket bloom. Many
Google Code to those other systems. To meet developers where the
nearly a thousand of our own open source projects from Google Co

GitHub, Inc.

A Short History of Git

As with many great things in life, Git began with a bit of creative destruction and fiery controversy.

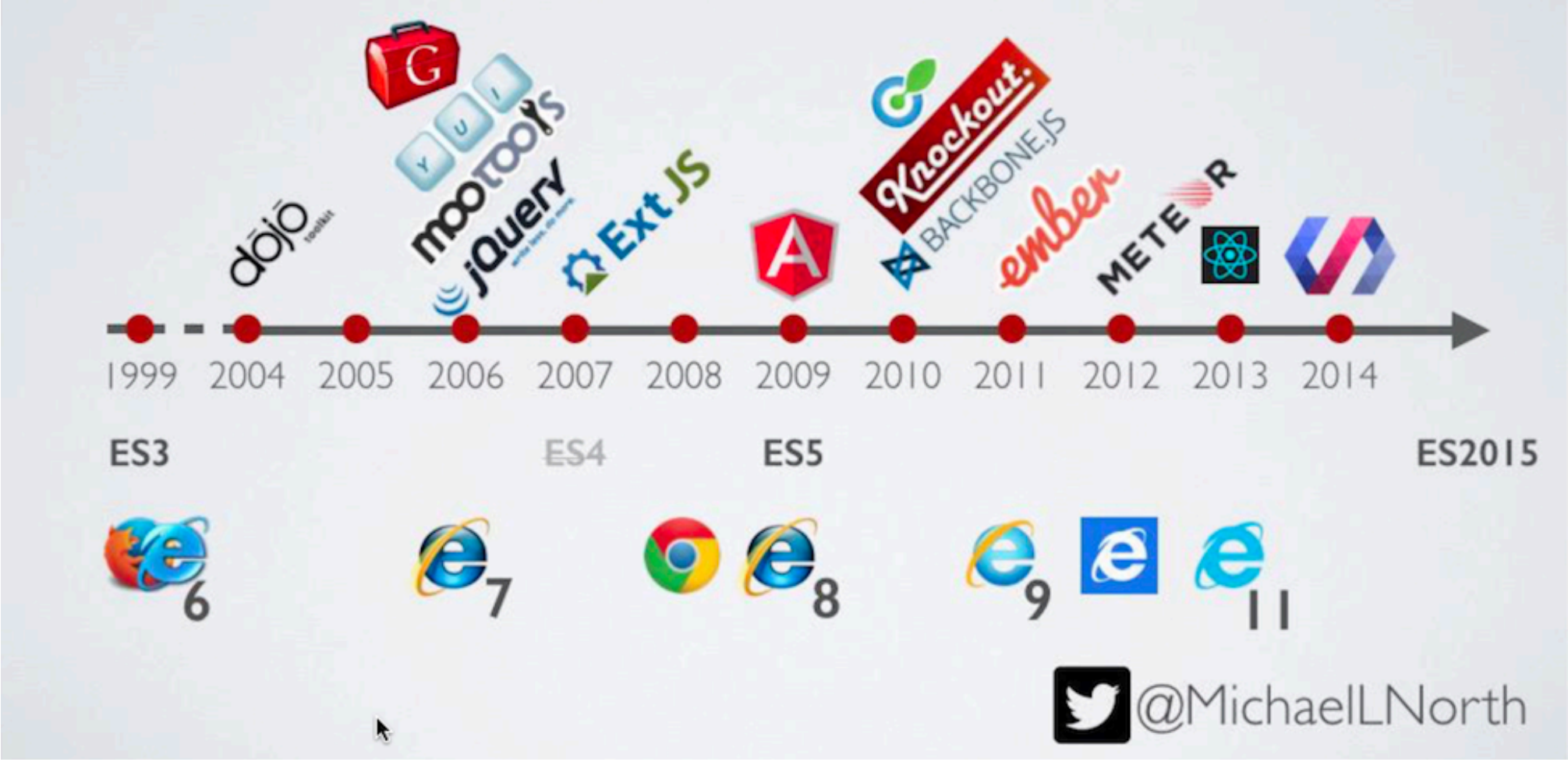
The Linux kernel is an open source software project of fairly large scope. For most of the lifetime of the Linux kernel maintenance (1991–2002), changes to the software were passed around as patches and archived files. In 2002, the Linux kernel project began using a proprietary DVCS called BitKeeper.

In 2005, the relationship between the community that developed the Linux kernel and the commercial company that developed BitKeeper broke down, and the tool's free-of-charge status was revoked. This prompted the Linux development community (and in particular Linus Torvalds, the creator of Linux) to develop their own tool based on some of the lessons they learned while using BitKeeper. Some of the goals of the new system were as follows:

- Speed
- Simple design
- Strong support for non-linear development (thousands of parallel branches)
- Fully distributed
- Able to handle large projects like the Linux kernel efficiently (speed and data size)

Since its birth in 2005, Git has evolved and matured to be easy to use and yet retain these initial qualities. It's amazingly fast, it's very efficient with large projects, and it has an incredible branching system for non-linear development (See Git Branching).

https://git-scm.com/book/en/v2/Getting-Started-A-Short-History-of-Git



https://www.freecodecamp.org/news/overcoming-javascript-framework-fatigue-741dac9370ee/

Design Principles

7. FLEXIBILITY	8. ANTICIPATE OBSOLESCENCE	9. PORTABILITY
10. TESTABILITY	11. DEFENSIVELY	. . .

Design Principle 9: Design for Portability

Have the software run on as many platforms as possible

- Avoid the use of facilities that are specific to one particular environment
- E.g. a library only available in Microsoft Windows

React Native vs Native: What to choose for App Development

By Dileep Gupta

December 28, 2018 5 min read
Last update on: September 17, 2019

https://appinventiv.com/blog/react-native-vs-native-apps/

Design Principles

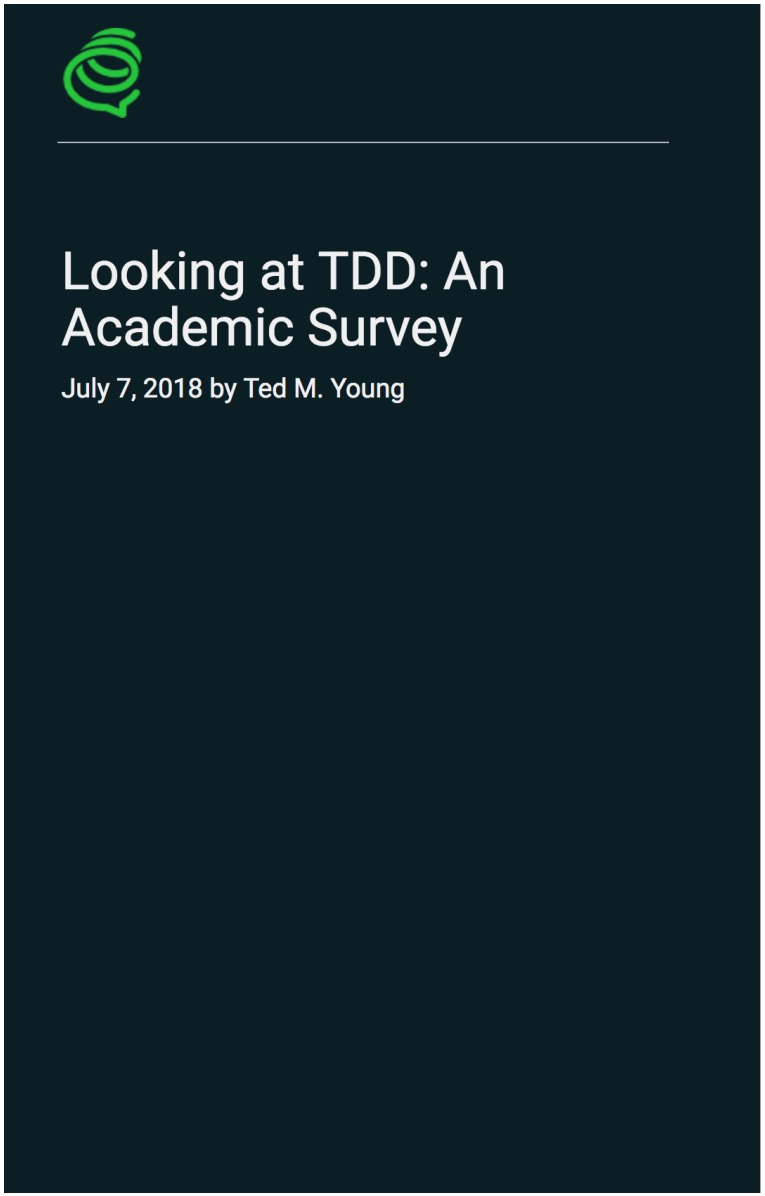
7. FLEXIBILITY	8. ANTICIPATE OBSOLESCENCE	9. PORTABILITY
10. TESTABILITY	11. DEFENSIVELY	. . .

Design Principle 10: Design for Testability

Take steps to make testing easier

- Design a program to automatically test the software
- Ensure that all the functionality of the code can by driven by an external program, bypassing a graphical user interface

- Testing frameworks like jUnit (generically called xUnit) are available in almost all langauges



And So It Begins

The tweetstorm started because I read a post on TDD that was clearly biased against TDD, and I tweeted:

> **Ted M. Young (dev guacamole)**
> @jitterted
>
> Some days I'm really frustrated by people holding opinions in the face of experiments and data showing the opposite. It's one thing to say that the experiment isn't valid (and then please say why you think so), it's another to misread it entirely.
>
> 5:23 PM - Jul 5, 2018
>
> 4 See Ted M. Young (dev guacamole)'s other Tweets

The post I was referencing is titled "Lean Testing or Why Unit Tests Are Worse Than You Think" (by Eugen Kiss). Just from the title, I knew it was trouble (the "Lean Testing" doesn't help), but I try to approach things with an open mind, even if I disagree, because I've often been wrong before.

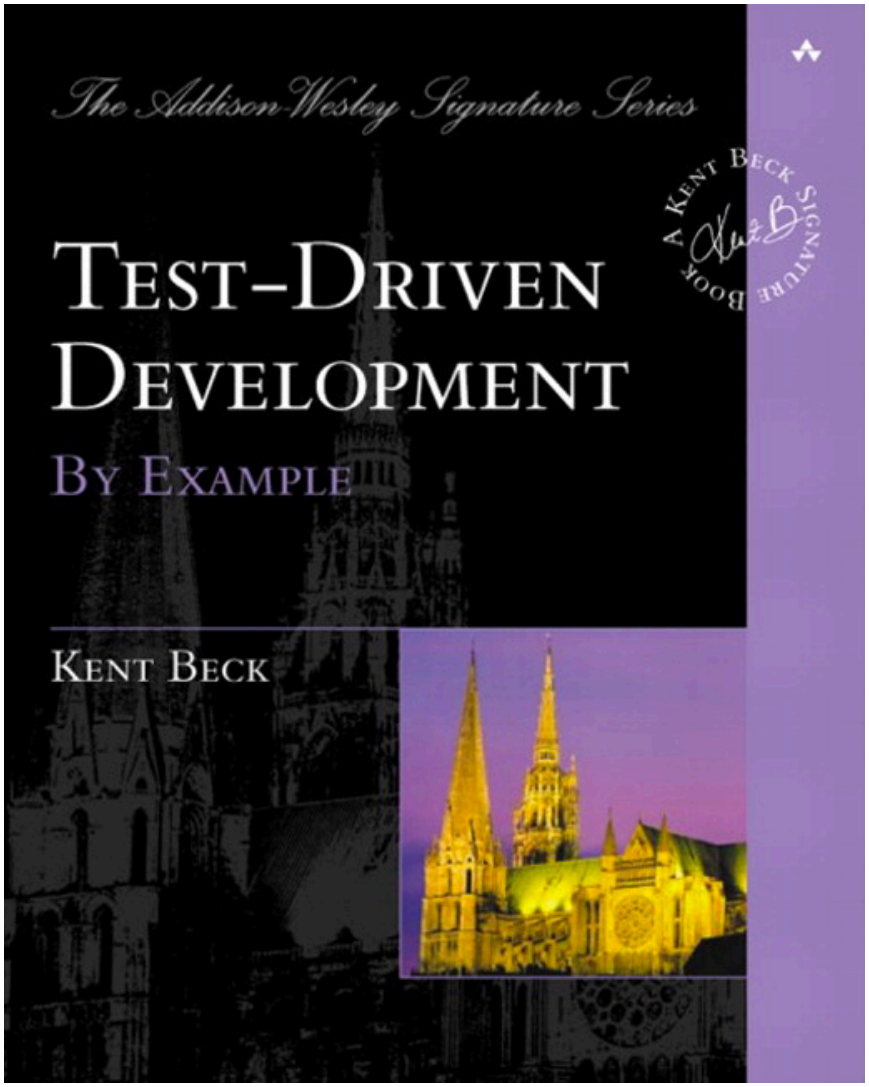
As I read the post, I ran into a reference to some academic papers, and thought that the author was going to cite data from those papers to support their point, but was not just dismayed, but a bit angry. First the Microsoft Research article (not a paper) was mentioned in the context of code-coverage. I generally agree with the idea that 100% code coverage is not worth the cost. However, Kiss's "Additional Notes" section dives into TDD:

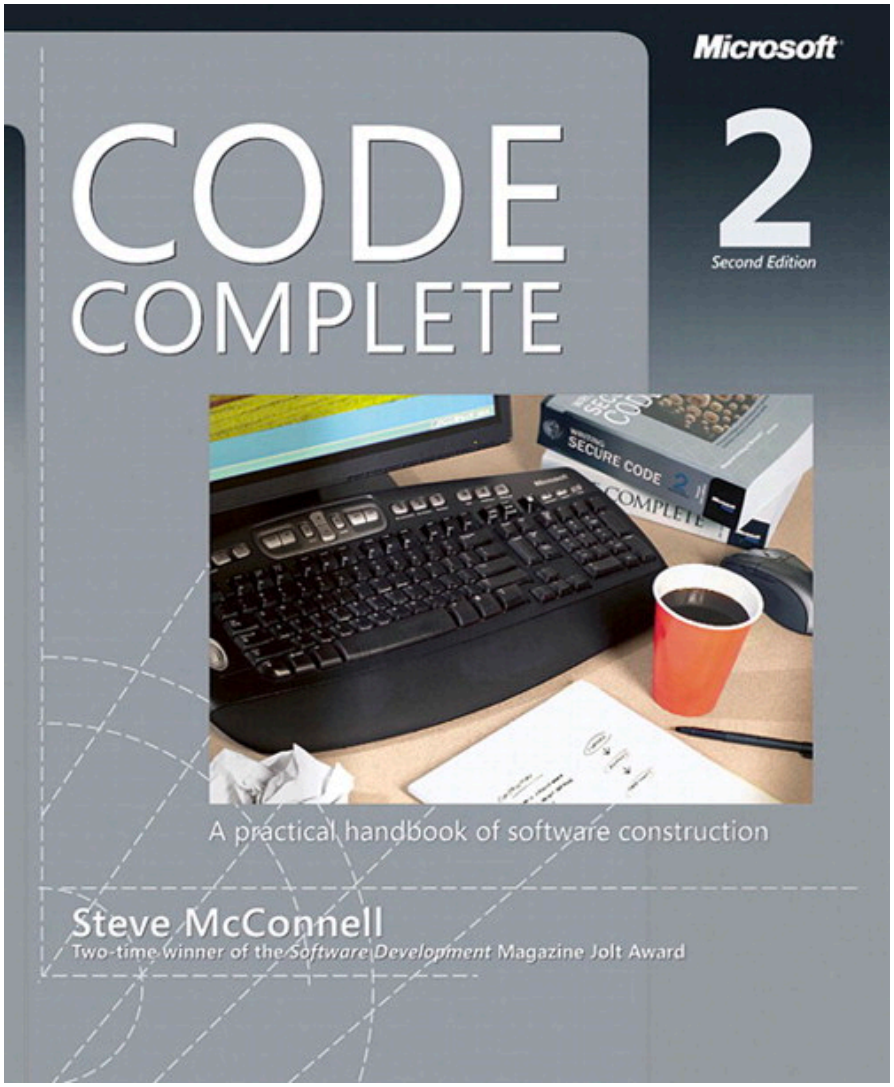
> There is research that didn't find Test Driven Development (TDD) improving coupling and cohesion metrics

https://www.tedmyoung.com/looking-at-tdd-an-academic-survey/

Testing *is a* ***separate skill*** *and that's why you are frustrated.*

https://medium.com/planet-arkency/testing-is-a-separate-skill-and-thats-why-you-are-frustrated-7239b1500a6c


The Addison-Wesley Signature Series
A Kent Beck Signature Book
Test-Driven Development
By Example
Kent Beck


Microsoft
CODE COMPLETE
2
Second Edition
A practical handbook of software construction
Steve McConnell
Two-time winner of the Software Development Magazine Jolt Award

Design Principles

7. FLEXIBILITY	8. ANTICIPATE OBSOLESCENCE	9. PORTABILITY
10. TESTABILITY	11. DEFENSIVELY	. . .

Design Principle 11: Design defensively

Never trust how others will try to use a component you are designing

- Handle all cases where other code might attempt to use your component inappropriately
- Check that all of the inputs to your component are valid: the preconditions
  - Unfortunately, over-zealous defensive design can result in unnecessarily repetitive checking

Generous on input, strict on output

May 27, 2009 • Ted Wise

I develop by some general principles. One of the strongest is “generous on input, strict on output”. This means that the code should be tolerant of variations in it’s input but should be strictly correct when outputting data. This ranges from the small to the large.

As an example, when you accept a boolean as a string, ignore the case and allow it to be expressed as “1, 0, T, F, Y, N, TRUE, FALSE, YES, NO, etc.”. If it’s not anything you’re prepared to see, assume it’s false. But when outputting a boolean as a string, stick to one single convention and one single case religiously.

http://tedwise.com/2009/05/27/generous-on-input-strict-on-output/

AGGRESSIVE TYPE COERCION

JavaScript practices an extremely aggressive type coercion doctrine where comparing apples with bananas always makes sense, especially when they are actually oranges. Everything works somehow. Or not, depending on how you look at it and what time of the day it is. The language also seems to have a deep sensual relationship with strings, trying to convert in and out of them. If you look at the lengthy definition of the **Abstract Equality Comparison Algorithm**, you might realize why sometimes the thing doesn't just quite do what you want:

> *11.9.3 The Abstract Equality Comparison Algorithm*
>
> *The comparison x == y, where x and y are values, produces true or false. Such a comparison is performed as follows:*

https://whydoesitsuck.com/why-does-javascript-suck/

20

Yes, I believe you should throw an exception, to help your fellow programmers notice the error at compile-time, by also adding `throws ArgumentOutOfBoundsException` in your method declaration.

That is, unless you use imaginary numbers in your project, where -1 is a perfectly valid argument for your square root method. :)

share improve this answer

answered May 13 '14 at 8:30

mavrosxristoforos

869 ● 8 ● 7

https://softwareengineering.stackexchange.com/questions/238896/should-you-throw-an-exception-if-a-methods-input-values-are-out-of-range