PROJECT

AGILE SOFTWARE ENGINEERING

M. JEFF WILSON

DATE 2020 FEB 20

CLIENT DEVNEXUS 2020

Who is this guy?

M. JEFF WILSON

Software developer, published author, certified scrum master (CSM) and SAFe® 5 Program Consultant (SPC), and lifelong learner. I have 35 years of working with teams in the telecommunications, insurance, traffic engineering, retail, and other industries to develop and deliver solutions and improve processes.

@mjeffw
geek.macbeer@gmail.com

WHAT IS AGILE SOFTWARE ENGINEERING?

“The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software.”

–IEEE Standard Glossary of Software Engineering Terminology

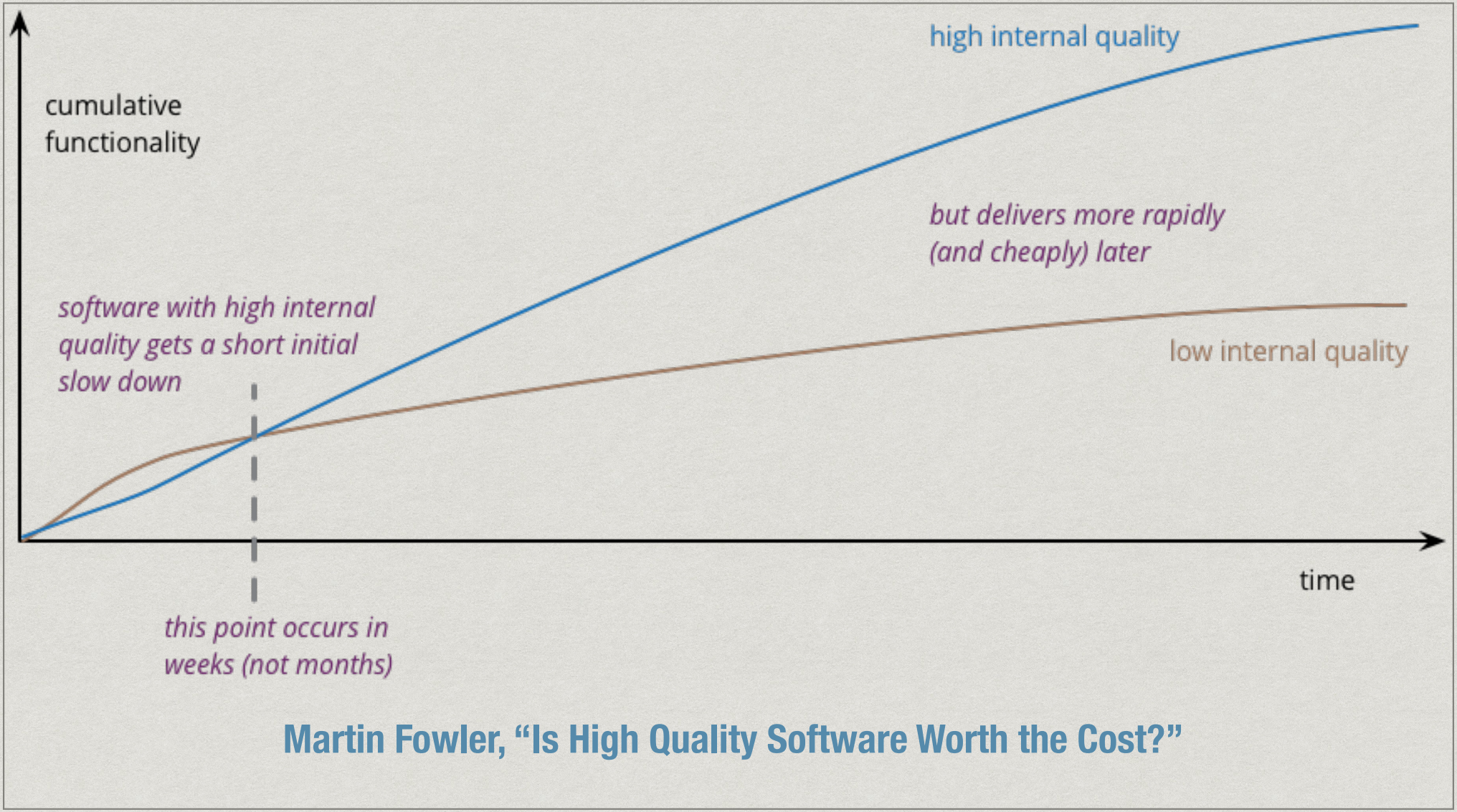
“The thinking and activities we do before and during coding that eliminates more work later.”

–M. Jeff Wilson

Why “agile” software engineering?

“Scrum is like your mother-in-law, it points out ALL your faults.”

— Ken Schwaber, co-creator of Scrum.



Martin Fowler, “Is High Quality Software Worth the Cost?”

“Without requirements or design, programming is the art of adding bugs to an empty text file.”

–Louis Srygley

I'll go up and find out what they want and the rest of you start coding!

We don't understand the user stories for the first 3 or 4 days of the sprint

We can't demo anything until all the layers are addressed

Testers have a different interpretation of the story (requirements) than the developers

Technical debt slows us down

Can only test from the GUI or API

Must have database to test business rules

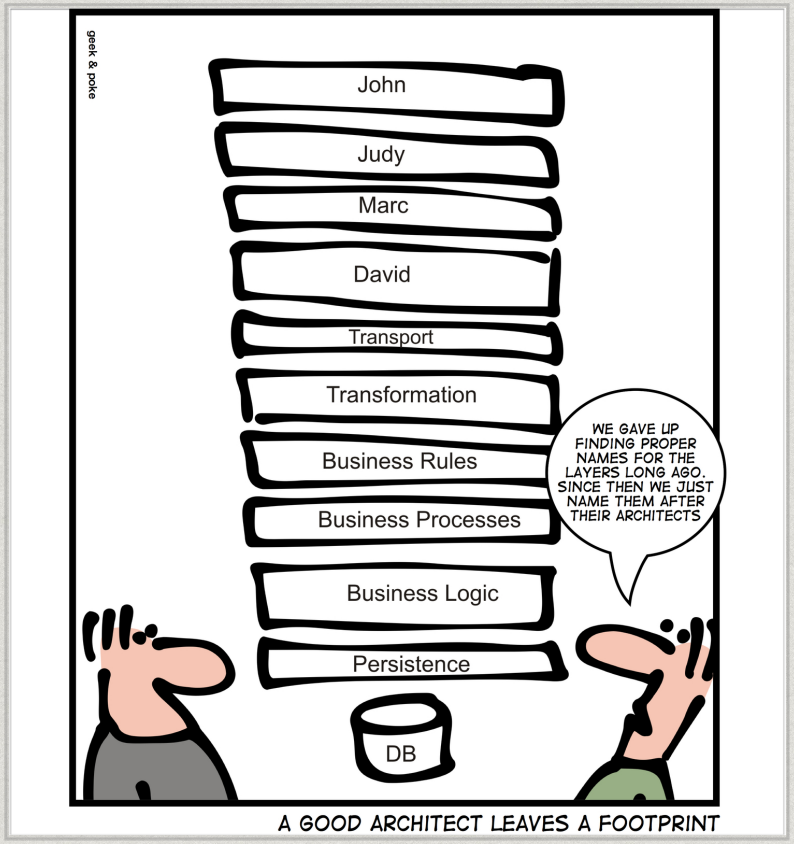
We can't complete coding and testing in same sprint

Changes to existing code takes too long

We can't demo anything until all the layers are addressed

Architecture as Layers

How much code do you need to update to demo new business logic?

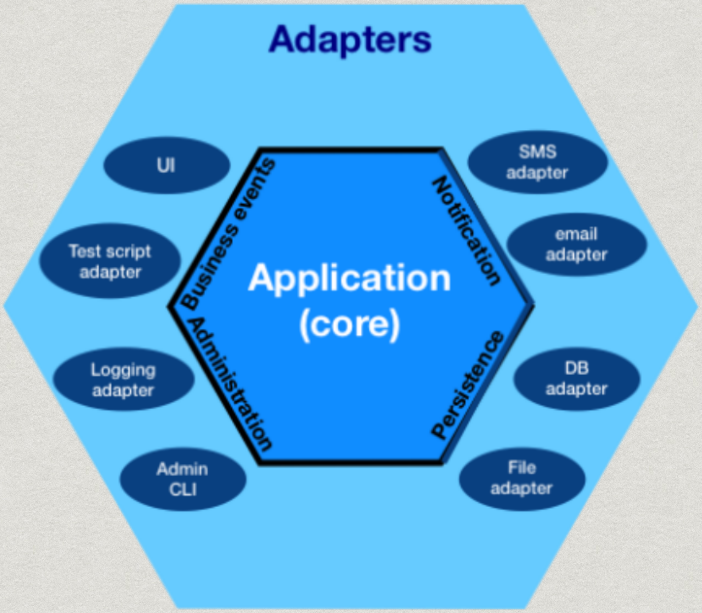


Use an architecture built for testability

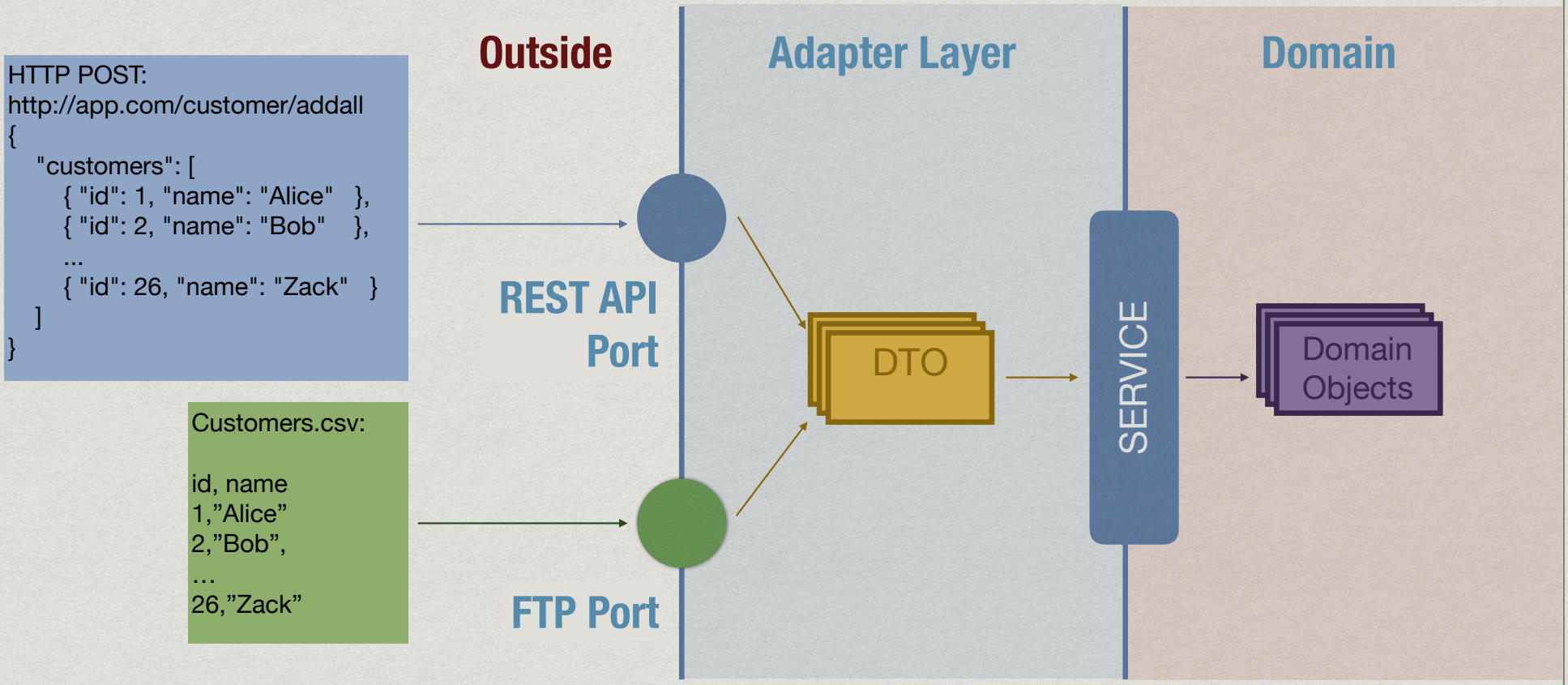
- One example: Hexagonal Architecture, aka Ports-And-Adapters

"Allow an application to be equally driven by users, programs, and automated tests… and to be developed and tested in isolation from its eventual run-time devices and databases."

-Ports & Adapters, Alistair Cockburn.



Ports and Adapters

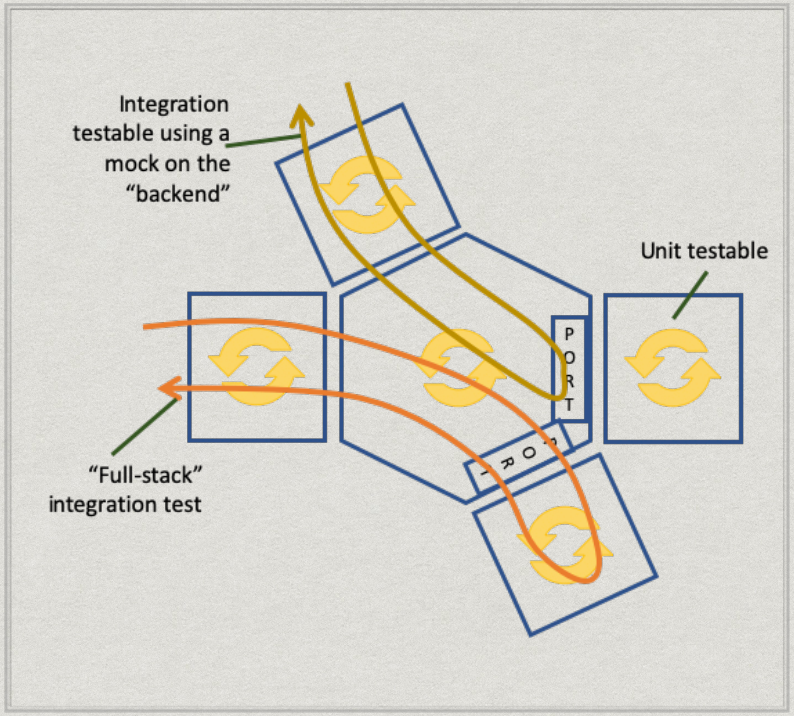


Testability, Delivered

Hexagonal architecture enables a large number of test scenarios. E.g.,

- Unit test of just the domain
- Unit test of a port/adapter
- Integration test of any port/adapter combination with others mocked
- Full-stack integration

Testability of the components allows for the implementation to be done in any order

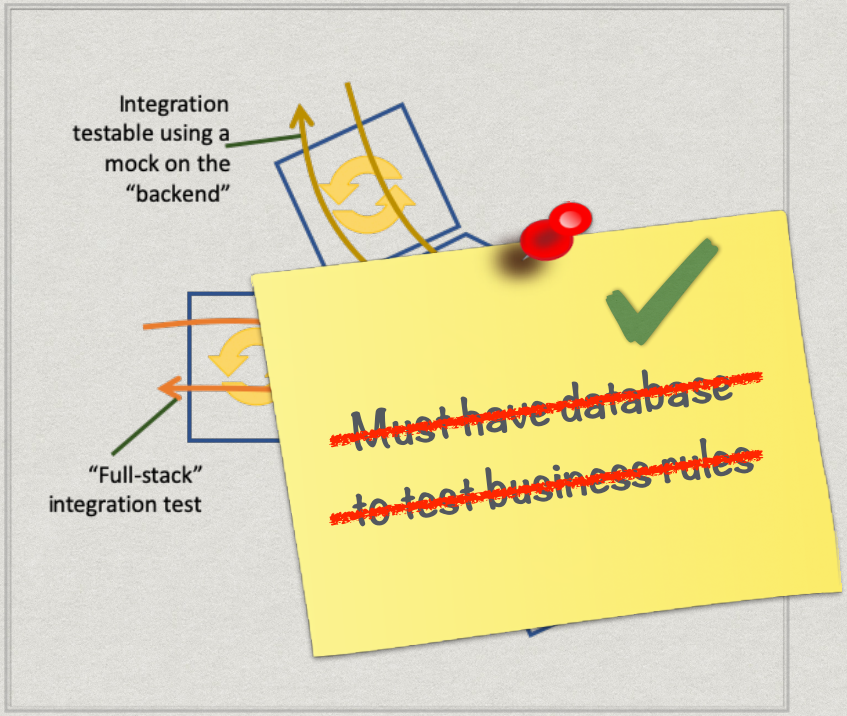


Testability, Delivered

Hexagonal architecture enables a large number of test scenarios. E.g.,

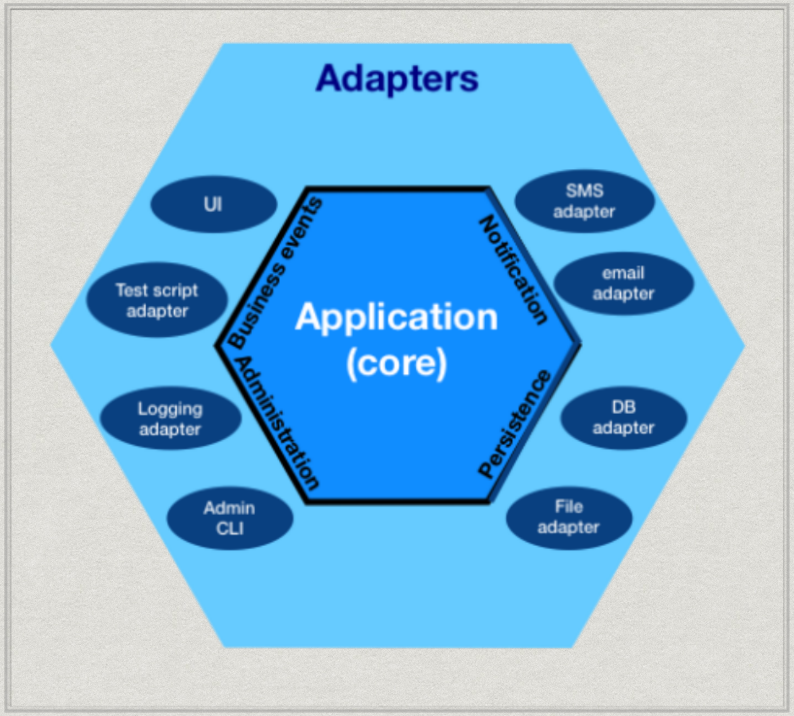
- Unit test of just the domain
- Unit test of a port/adapter
- Integration test of any port/adapter combination with others mocked
- Full-stack integration

Testability of the components allows for the implementation to be done in any order



Hexagonal architecture references

- Reference: https://fideloper.com/hexagonal-architecture
- Hexagons with Spring: https://dzone.com/articles/hexagonal-architecture-it-works
- Spring-Boot: https://www.slideshare.net/alimenkou/hexagonal-architecture-with-spring-boot-116495140



Testers have a different interpretation of the story (requirements) than the developers

Requirements and Tests

- You cannot release code until all requirements are met.
- You cannot release code if there is a failing test case.
- Therefore, every test is a requirement.

Requirements should be complete before coding starts, right?

Behavior Driven Development (BDD)

Three Amigos practice of BDD:

- A collaboration between the roles of business, developer and QA
- Capture examples using real data and business domain terminology to clarify the requirement
- Describe the desired behavior from the perspective of users of the system
- Convert examples into acceptance criteria before the story is committed in sprint planning

BDD References

- https://school.cucumber.io
- What is BDD? https://youtu.be/VS6EEUVZGLE
- Get Started with BDD: https://tinyurl.com/ydfzw9y9
- An Intro to BDD with Cucumber for Java
  https://youtu.be/MCaXumfckmQ

Card, Conversation, Confirmation

- Card: a brief description of the desired behavior
- Conversation: taking place at different time and places during a project between the various people concerned by a given feature of a software product: customers, users, developers, testers; this conversation is largely verbal but most often supplemented by documentation;
- Confirmation: How to verify that the objectives the conversation revolved around have been reached (the more formal the better)

- Ron Jeffries

Card, Conversation, Confirmation

- Card: a brief description of the desired behavior
- Conversation: taking place at different time and places during a project between the various people concerned by a given feature of a software product: customers, users, developers, testers; this conversation is largely verbal but most often supplemented by documentation;
- Confirmation: How to verify that the objectives the conversation revolved around have been reached (the more formal the better)

Three Amigos

Gherkin Acceptance Tests

- Ron Jeffries

We don't understand the user stories for the first 3 or 4 days of the sprint

“If you cannot come up with three solutions to a problem, you do not understand the problem.”

–Gerald Weinberg, The Rule of Three

Behavior Driven Development (BDD)

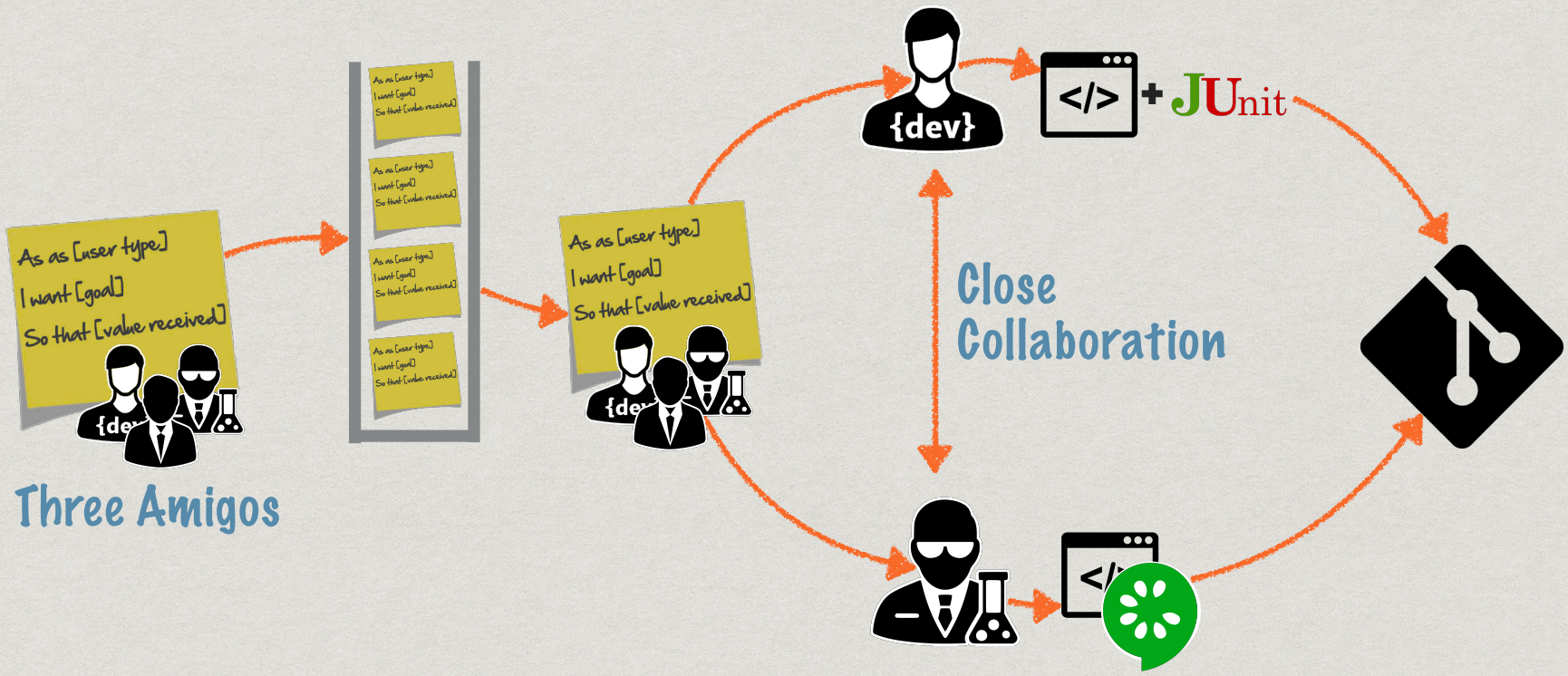
Three Amigos practice of BDD:

- A collaboration between the roles of business, developer and QA
- **Capture examples using real data and business domain terminology to clarify the requirement**
- Describe the desired behavior from the perspective of users of the system
- Complete before the story is committed to in sprint planning

We can't complete coding and testing in same sprint

- Test automation is a software development activity.
- You are creating a separate application to test the production application.
- Test automation framework and code must be well-thought out, have the appropriate design and frameworks to support maintainability, readability, reuse, etc. It is at least as important as the production code and should not be delegated to junior developers.

BDD Process Overview

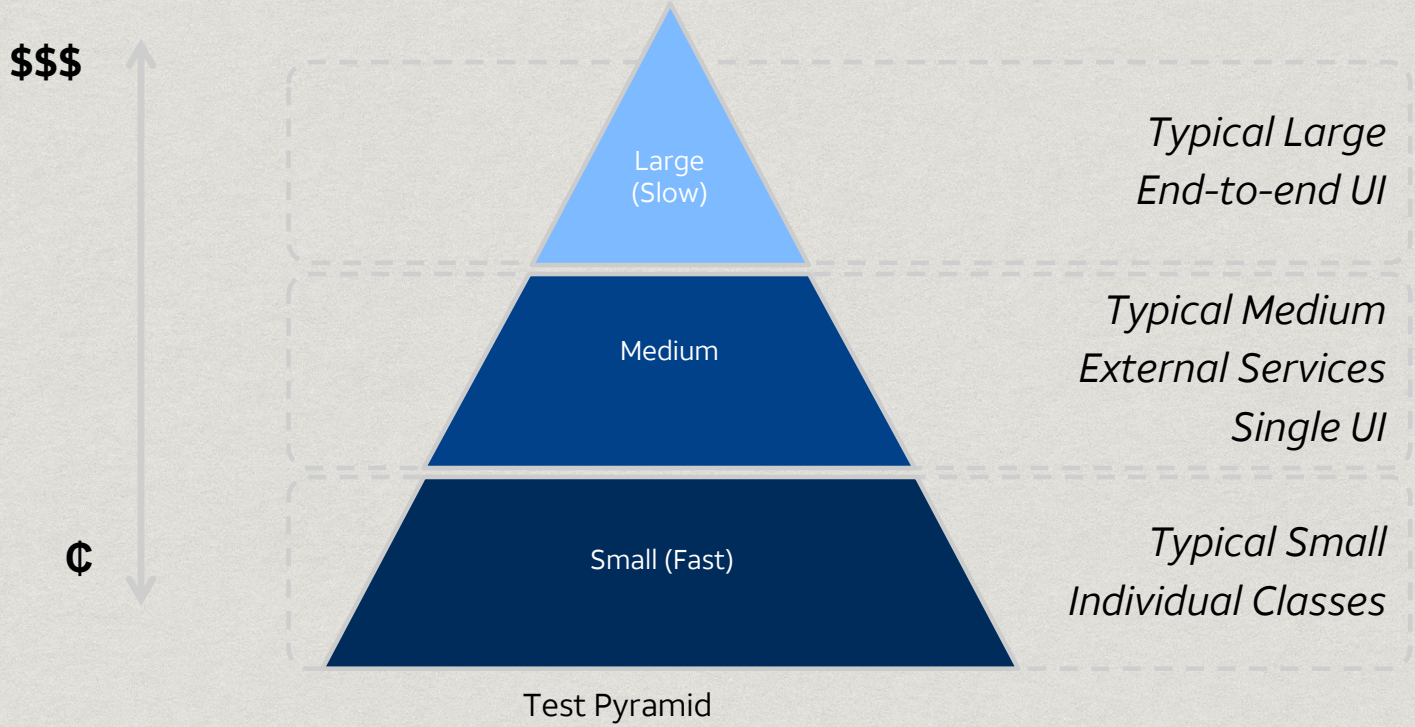


Can only test from the GUI (or API)

Agile testing strategy

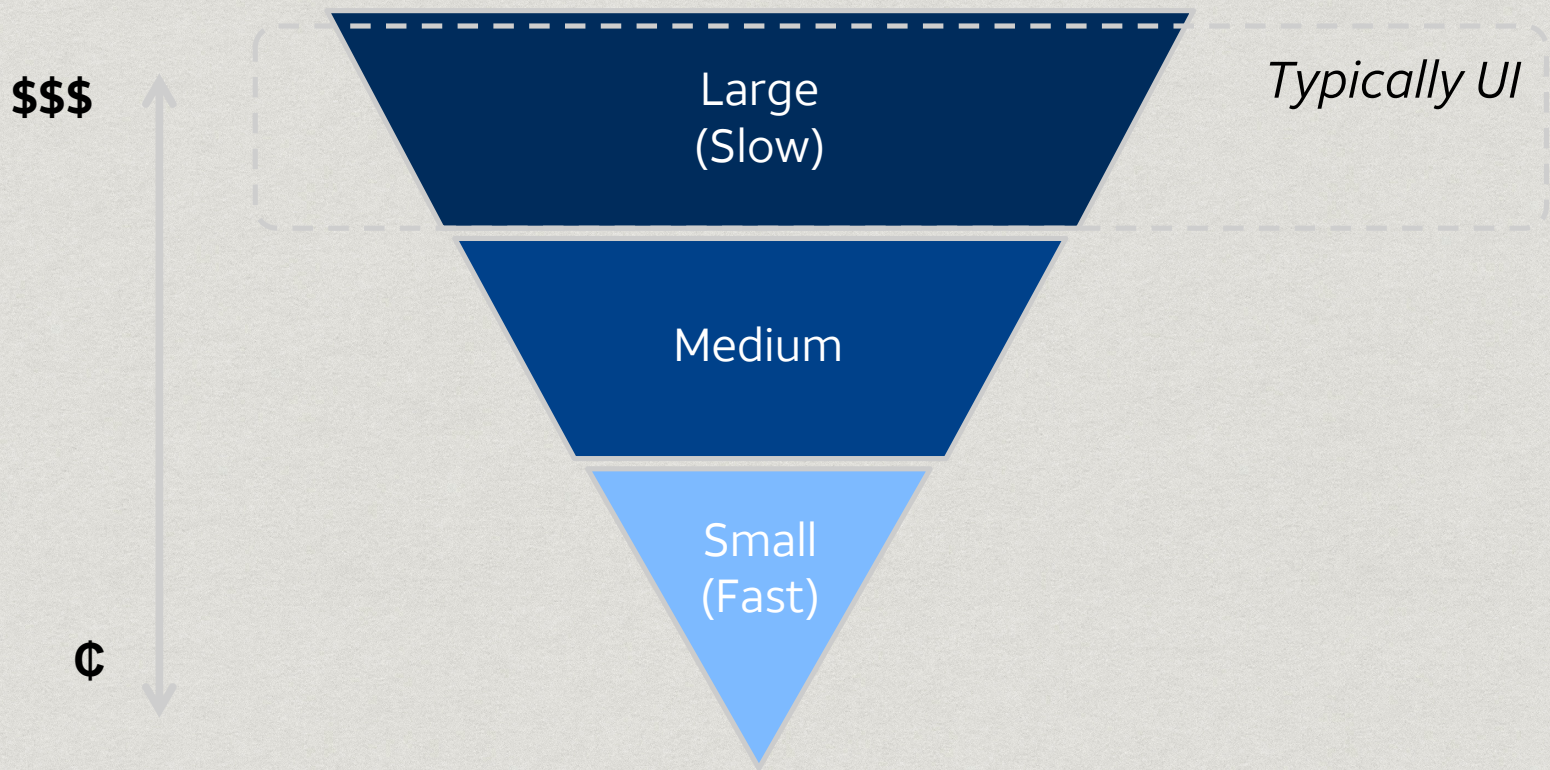
- Move tests to lowest possible level in the system
  - Business rule variations should be tested directly in the component where implemented
- Test UI and APIs separately from the rest of the system
  - Does the UI/API generate correct messages for inputs?
  - Does it render the response correctly?
  - Test a single ‘end-to-end’ variation using the UI/API to show integration is complete and correct

Agile test strategies naturally create a pyramid of tests



Test Pyramid

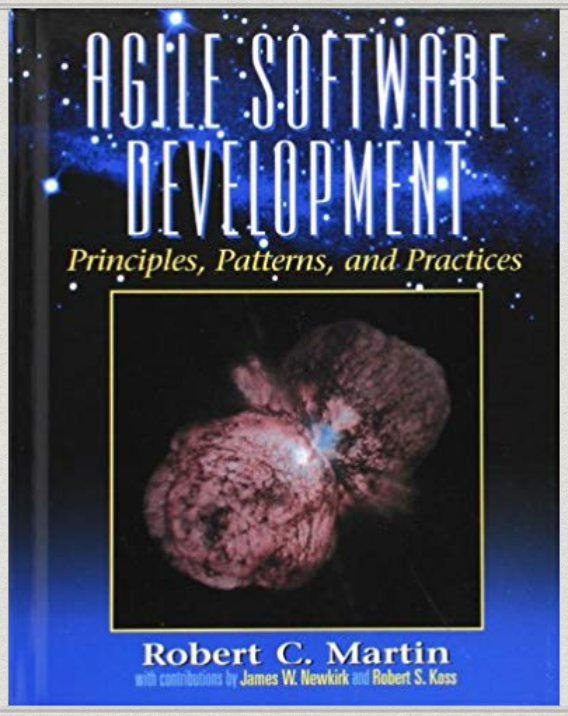
The inverted pyramid is a test strategy anti-pattern



Changes to existing code takes too long

Design to interfaces, not implementations

- Interface Separation Principle (SOL**I**D)
- Dependency Inversion Principle (SOLI**D**)
- Liskov Substitution Principle (SO**L**ID)



Interface Separation Principle

Favor multiple client-specific over one, large general-purpose interface. Only provide what each client needs.

```
// Violates ISP
interface ISmartDevice {
  void print();
  void fax();
  void scan();
}

class AllInOnePrinter implements ISmartDevice {
  public void print() { /* print code*/}
  public void fax() { /* fax code */ }
  public void scan() { /* scan code*/ }
}

class EconomyPrinter implements ISmartDevice {
  public void print() {
    //Yes I can print.
  }
  public void fax() { throw new NotSupported();}
  public void scan(){ throw new NotSupported();}
}
```

```
// Good example
interface IPrinter{
  void print();
}
interface IFax{
  void fax();
}
interface IScanner{
  void scan();
}

class EconomyPrinter implements IPrinter {
  public void print() { /* Printing code. */ }
}

class AllInOnePrinter implements IPrinter, IFax,
IScanner {
  public void print() { /* Printing code */ }
  public void fax() { /* Beep booop biiiiip */ }
  public void scan() { /* Scanning code */ }
}
```

Dependency Inversion Principle

High-level modules should not depend upon low-level modules. Both should depend upon abstractions.

```java
// Violates DIP
public class CustomerService {
  private final JdbcCustomerAdapter db;

  public updateAddress(int id, Address address) {
    Customer customer = db.findById(id).
      orElse(NullCustomer);
    customer.setAddress(address);
    db.update(customer);
  }
}

public class JdbcCustomerAdapter {
  Optional<Customer> findById(int id) {
    // SELECT * FROM CUSTOMER where ...
  }

  void update(Customer customer) {
    // UPDATE CUSTOMER WHERE...
  }
}
```

```java
// Good example
public class CustomerService {
  private final CustomerDao dao;

  public updateAddress(int id, Address address) {
    Customer customer = dao.findById(id).
      orElse(NullCustomer);
    customer.setAddress(address);
    dao.update(customer);
  }
}

interface CustomerDao {
  Optional<Customer> findById(int id);
  void update(Customer customer);
}

public class JdbcCustomerDao implements CustomerDao {
  Optional<Customer> findById(int id) { /* SELECT */ }
  void update(Customer customer) { /* UPDATE */ }
  }
}
```

Liskov Substitution Principle

Clients of an interface have no dependency on the implementation and may use any implementation without knowing.

```
// Violates LSK
interface IRectangle {
  void setHeight(int height);
  void setWidth(int width);
  int getArea();
}

class Rectangle implements IRectangle {
  private int h;
  private int w;

  void setHeight(int height) { h = height; }
  void setWidth(int width) { w = width; }
  int getArea() { return h * w; }
}

class Square implements IRectangle {
  private int side;

  void setHeight(int h) { side = h; }
  void setWidth(int w) { side = w; }
  int getArea() { return side * side; }
}

// unit test
IRectangle r1 = new Rectangle();
r1.setHeight(10);
r1.setWidth(12);
assertEquals(120, r1.getArea());


IRectangle r2 = new Square();
r2.setHeight(10);
r2.setWidth(12);
assertEquals(120, r2.getArea()); // FAILS!
```

Liskov Substitution Principle

- The Liskov Substituton Principle is also violated if client code must check the type of the implementor of the interface

```
List<IRectangle> rects;
IRectangle r = rects.get(4);
if (r instanceof Square) // VIOLATION!
...
```

Separation of Concerns

- Single Responsibility Principle (**S**OLID):
  - Every class should have one and only one reason to change
- Open-Closed Principle (S**O**LID):
  - A class should be open for extension, but closed for modification.

Single Responsibility Principle

Every class should have one and only one reason to change

```
interface IEmployeeDao {
  Employee findById(Long id);
  void add(Employee employee);
  void email(Employee employee, String content);
}


class EmployeeStore implements IEmployeeDao {
  public Employee findById(Long id) {
    // retrieve Employee
  }

  public void add(Employee employee) {
    // add Employee to database

    // send Employee welcome email
    notify(employee.emailAddress, welcomeText);
  }

  public void email(EmailAddress email, String content) {
    // code to send email
  }
}
```

```
interface IEmployeeDao {
  Employee findById(Long id);
  void add(Employee employee);
  void notifyOnAdd(AddNotification employee);
}

interface IEmployeeNotification {
  void add(Employee employee);
}


class EmployeeStore implements IEmployeeDao {
  public Employee findById(Long id) { }

  public void add(Employee employee) {
    // add Employee to database

    // send Employee welcome email
    notify(employee.emailAddress, welcomeText);
  }

  public void notifyOnAdd(Employee employee) {
    addNotification.add(new AddNotification(employee);
  }
}
```

Open-Closed Principle

A class should be open for extension, but closed for modification.

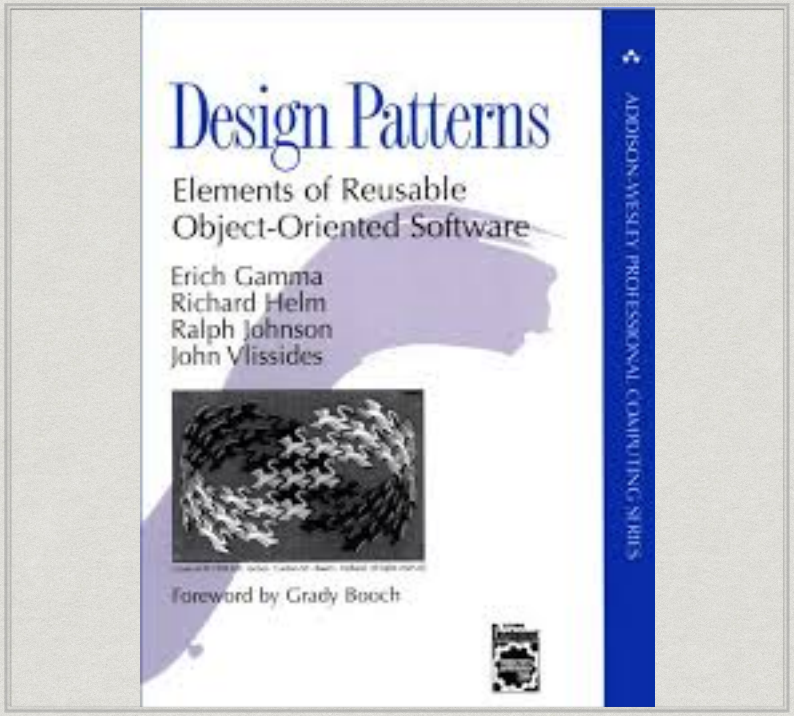
```java
interface Shape {}

class Circle implements Shape {
   private int radius;
   public double getRadius() { return radius; }
}

class Rectangle implements Shape {
   private int width;
   private int height;

   public int getWidth() { return width; }
   public int getHeight() { return height; }
}


  public double calculate(Shape shape) {
      if (shape instanceof Circle) {
        return ((Circle) shape).getRadius() * Math.PI * Math.PI;
      }
      else if (shape instanceof Rectangle) {
        return ((Rectangle) shape).getWidth() * r.getHeight();
      }
      throw new OperationNotImplemented();
   }
}
```

```java
interface Shape {
   double calculateArea();
}

class Circle implements Shape {
   private int radius;

   public double getRadius() { return radius;  }
   public double calculateArea()    {
      return radius * Math.PI * Math.PI;
   }
}

class Rectangle implements Shape {
   private int width;
   private int height;
   public int getWidth() { return width; }

   public int getHeight() { return height; }
   public double calculateArea() {
      return width * height;
   }
}


class AreaCalculator {
   public double calculate(Shape shape) {
      return shape.calculateArea();
   }
}
```

Design Patterns

Improve quality by adding flexibility to a system

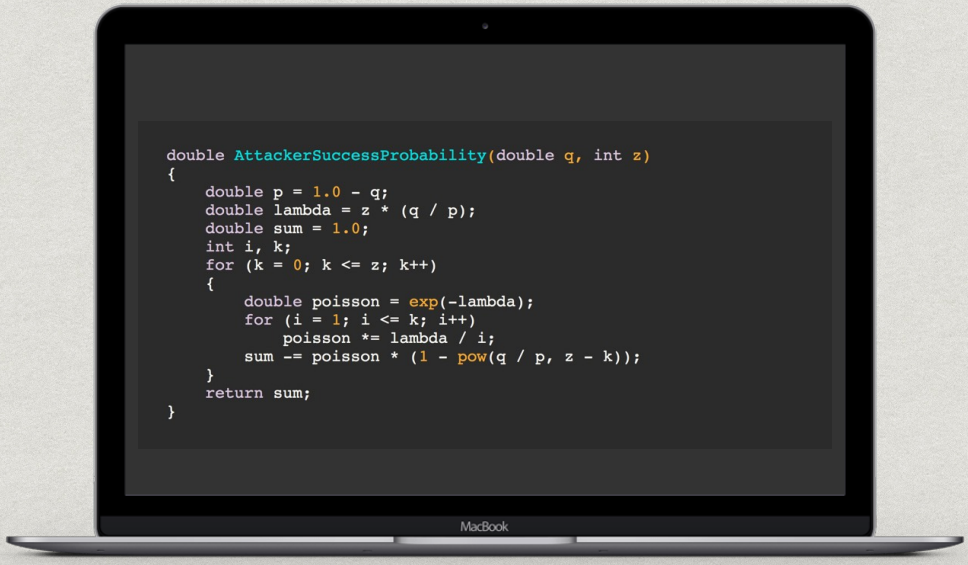
- Makes extension easier to add new implementations
- Simplifies testing to easily inject test doubles
- The names of patterns can become design domain terms
- Improves readability and maintenance



Technical debt slows us down

Code Qualities

- Cohesion
- Coupling
- Naming
- Readability
- Testability
- Assertive
- Abstract
- DRY



Focus on maintainability

- Readability and naming
- Extensibility — high cohesion, low coupling
- Clear design and implementation
- Well-defined interfaces
- Ease of build and reuse
- Sufficient documentation

Readability & Naming

Coding standards and automated checking

Use business domain names to improve understanding

- Create an ADT for every domain name

Comment the *why*, not the how

- Class comments should include the list of responsibilities of that class

Use named constants instead of magic numbers (no constant literal should appear in executable code)

Spell check your code

Extensibility

Favor delegation over inheritance (tight coupling)

- Avoid premature hierarchies; create them when they emerge
- Sharing code is not the main reason for inheritance

Decoupled systems are easier to extend

- Code to interfaces, not implementations

DRY - Don’t Repeat Yourself

- “Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.”

Develop your sense of smell

- Code Smell - “**Code smells**, or bad smells in code, refer to symptoms in code that may indicate deeper problems. They're a diagnostic tool used when considering refactoring.”

The smell of bugs

Comments

- There's a fine line between comments that illuminate and comments that obscure. Are the comments necessary? Do they explain "why" and not "what"?

Long Method

- All other things being equal, a shorter method is easier to read, easier to understand, and easier to troubleshoot.

Long Parameter List

- The more parameters a method has, the more complex it is. Limit the number of parameters you need in a given method, or use an object to combine the parameters.

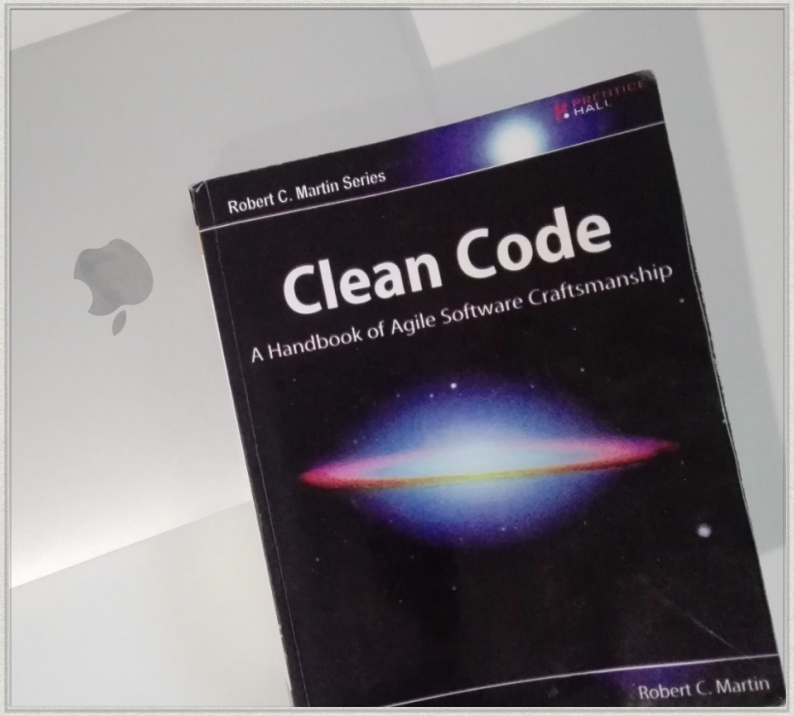
Duplicated code

- Duplicated code is the bane of software development. Stamp out duplication whenever possible. You should always be on the lookout for more subtle cases of near-duplication, too. Don't Repeat Yourself!

Suggested Reading

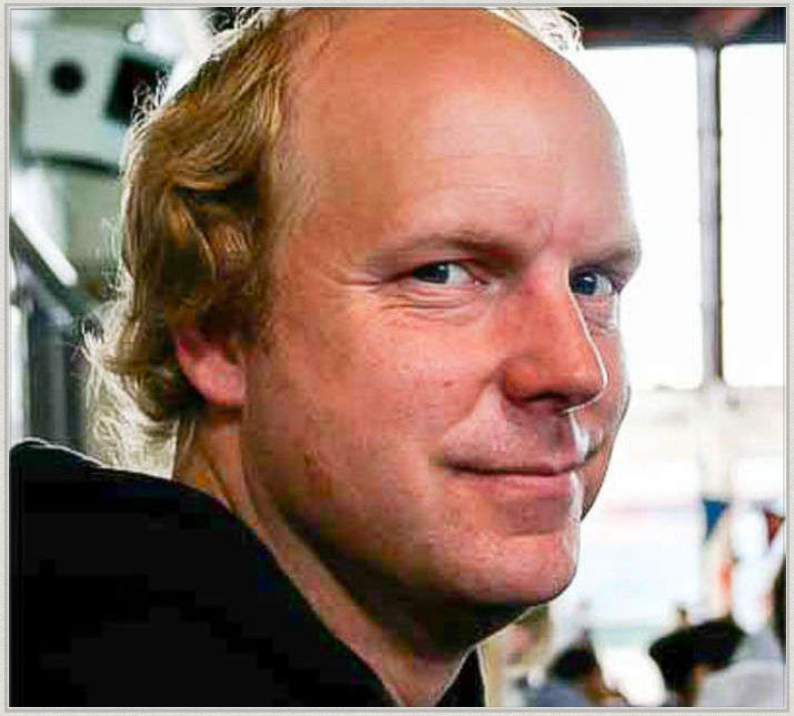
“Even bad code can function. But if code isn’t clean, it can bring a development organization to its knees. Every year, countless hours and significant resources are lost because of poorly written code.”

— *Clean Code, Robert Martin*



“Make it work,
make it right,
make it fast.”

— *Kent Beck*



“High-performing IT teams had 50 percent higher market capitalization growth over three years than those with low-performing IT organizations.”

-Puppet Labs, 2014 State of DevOps Report

“If you are releasing software into production once every six months, you aren’t very good at it. If you can release software into production every 11.6 seconds like Amazon, you are bloody good at it.”

– Dave Farley

Thank you, DevNexus!

M. JEFF WILSON

Software developer, published author, certified scrum master (CSM) and SAFe® 5 Program Consultant (SPC), and lifelong learner.

@mjeffw
geek.macbeer@gmail.com