Test, Code, Design: Inverting the Waterfall
Agenda

• Design
• Change
• Testing
• Code Reviews
• Refactoring
Who am I?

• John Deters – jadeters@comcast.net
• Son of a programmer
• Programming computers all my life
• I have code written in 1981 still in production today
• Bachelor of Science in Computer Science, 1986
• Joined Dayton’s POS team in 1987
• Master of Science in Engineering Science (Software Engineering), 2010
• Recently moved to Target’s security team
• My son is a programmer
Design
What is design?

definition
1. A plan (with more or less detail) for the structure and functions of an artifact, building or system.
2. A pattern, as an element of a work of art or architecture.
3. The composition of a work of art.
4. Intention or plot.
5. The shape or appearance given to an object, especially one that is intended to make it more attractive.
6. The art of designing
7. To plan and carry out (a picture, work of art, construction, etc.)

https://en.wiktionary.org/wiki/design
Why value design?

Form: predictable, attractive, familiar – usable, maintainable
Function: repeatedly does what we expect – reliable, efficient, secure

These same words are the core attributes we use to define quality.
Software quality

**Application Architecture Standards**
- Multilayer design compliance (UI vs App Domain vs Infrastructure/Data)
- Data access performance
- Coupling Ratios
- Component (or pattern) reuse ratios

**Coding Practices**
- Error/exception handling (all layers UI/Logic/data)
- If applicable - compliance with OO and structured programming practices
- Secure controls (access to system functions, access controls to programs)

**Complexity**
- Transaction
- Algorithms
- Programming practices (eg use of polymorphism, dynamic instantiation)
- Dirty programming (dead code, empty code...)

**Documentation**
- Code readability and structuredness
- Architecture -, program, - and code-level documentation ratios
- Source code file organization

**Portability**: Hardware, OS and Software component and DB dependency levels

**Technical and Functional Volumes**
- # LOC per technology, # of artifacts, files
- Function points - Adherence to specifications (IFPUG, Cosmic references..)

Software design

Valuable design attributes support quality
• Reliable
• Secure
• Efficient
• Maintainable
• Usable
Levels of design

Architecture
The Goal of Object Oriented Design – Modular Code

• Highly cohesive – an object does one thing

• Loosely coupled – an object doesn’t depend on other objects

• Readable, simple, testable, reusable –

  **Maintainable**
Change
Cost of Change

- Programming defect found via Pair Programming
- Programming defect found via Continuous Integration
- Design or programming defect found via Test Driven Development (TDD)
- Requirements or design defect found via Active Stakeholder Participation
- Requirements or design defect found via Model Storming
- Defect found via independent parallel testing
- Requirements defect found via traditional acceptance testing
- Design defect found via traditional system testing
- Defect found via a review or inspection
Design as a verb

- Design is also a verb that describes the act of creating a plan.
- Historic approaches to software construction have followed two approaches:
  - Waterfall: Gather requirements, design the whole product, code to the design, test the product, deploy the product. “Big Design Up Front”
  - Iterative: Gather partial requirements, design an incremental slice of the product, code to the new slice, test the code, add the slice to the product, test the product, deploy the product, repeat. (AKA mini-waterfall.)

- “Agile” (Scrum, Kanban, etc.) is one of the approaches used to formalize iterative development.
  - It doesn’t have to be a mini-waterfall.
Design, code, test

- Understand the requirements
- Model the problem
- Write code to express your model
- Write tests to prove your code is correct
Test, code, design

• Understand the requirement
• Write a test to express one facet of your requirements
• Write code to meet the tests
• Refactor the code base guided by the SOLID design principles and leveraging design patterns until duplication is removed
• Repeat with next facet
• Repeat with next requirement
Agile Model Driven Development

1. Start
2. Create unit test to describe a small coding change to implement missing functionality
3. Execute unit tests
   - Pass → Done
   - Fail → Write just enough code to make the test pass
4. Execute unit tests
   - Pass → Refactor the changed code to eliminate duplication
   - Fail → Write just enough code to make the test pass

Test Driven Development
Why does TDD work?

• By writing the test first, you are testing the requirements first

• By writing the test first, you first focus on the interface to the logic

• By writing a readable test, you create module documentation

• By writing the logic next, the test ensures the requirement is met

• By refactoring at the end, you impart the qualities of good design
  -- without creating a design document!
Testing
“If it's worth building, it's worth testing.

If it's not worth testing, why are you wasting your time working on it?”

- Scott Ambler, IBM
Software construction process

• Source code is the language of the developer. The compiler creates object code.
• Object code can be native instructions (.o, .obj), bytecode (Java) or Intermediate Language (IL in .Net).
• Execution can take place natively (x64) after linking, or in a runtime environment (JRE, .NET Framework) for bytecode.

These differences are not important for unit testing
Unscripted Manual Testing

Experience and Thought

Functional tests

Product Source code -> Compiler -> Product Object code -> Linker -> Product Execution
Scripted Manual Testing

Product Source code → Compiler → Product Object code → Linker → Product Execution

Scripts and Spreadsheets

Functional tests
Automated System Testing

- Product Source code
- Compiler
- Product Object code
- Linker
- Functional tests
- Scripts and Spreadsheets
Unit testing
Automated Unit Testing

Uses a framework to execute tests
Easily repeatable, prevents regression
Writing tests is an investment
Tests must run extremely fast, cheap to repeat
Tests are written by developers
Run by developers so feedback is immediate
Identifies failing code
JUnit, NUnit, xUnit are all tech specific frameworks
Test framework automatically finds and calls unit tests
Test framework can be GUI driven by developer or batch driven by build
Unit tests written by developer in the same language as the production code
Developer writes tests to execute every line of production code
• Including exception handlers
Tools can measure which lines are tested as Code Coverage %
Testing is easy!

Automated Unit Test → Calls → Method under test
Testing is not so easy!

Is the testing system online? Does the database server exist? Is the database server online? Can the testing framework reach it over the network? Does the testing framework have network authority to access it? Has the password expired? Does the testing framework have write permission to the database? Does it have the right schema? Does it have the right rows? Does the right data exist in those rows at the start of each test? How long does the query take to execute? If I run the test twice, will the data still be correct for the second test? Will the database still exist next year when we run this test?

This is now an integration test, not a unit test.
Testing is not so easy!

Is the testing system online? Does the database server exist? Is the database server online? Can the testing framework reach it over the network? Does the testing framework have network authority to access it? Has the password expired? Does the testing framework have write permission to the database? Does it have the right schema? Does it have the right rows? Does the right data exist in those rows at the start of each test? How long does the query take to execute? If I run the test twice, will the data still be correct for the second test? Will the database still exist next year when we run this test?

This is now an integration test, not a unit test.
SOLID object-oriented design principles

- **Single Responsibility Principle** – An object should be responsible for a single concept
- **Open / Closed Principle** – An entity should be open for extension, but closed for modification
- **Liskov Substitution Principle** – Objects of the same type (interface) should be substitutable without altering any of the desirable properties of the program (correctness)
- **Interface Segregation Principle** – Many specific interfaces are better than one general purpose interface
- **Dependency Inversion Principle** – Depend on abstractions, not concretions

A class should have only a single responsibility, and that responsibility should be entirely encapsulated within that class.
Open / Closed Principle

Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.
Liskov Substitution Principle

Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.
Interface Segregation Principle

Many client-specific interfaces are better than one general-purpose interface.

Authorize(int tranNo, Date tranDate, Money amount, bool void, bool return, int origTranNo, Date origTranDate, Account cardNo, int* status);

AuthorizeSale(int tranNo, Date tranDate, Money amount, Account cardNo, int* status);

AuthorizeReturn(int tranNo, Date tranDate, Money amount, int origTranNo, Date origTranDate, int* status);

ProcessVoid(int tranNo, Date tranDate, int origTranNo, Date origTranDate, int* status);
Dependency Inversion Principle

Production system
- Real DB reference

Automated Unit Test
- Fake DB reference

Method
- Real DB connection

Database

Fake DB

Depend upon abstractions. Do not depend upon concretions.
# Code structures

<table>
<thead>
<tr>
<th>Type</th>
<th>Stateless</th>
<th>Stateful</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Construction</strong></td>
<td>Unlikely to exist. Testing limited to validating side effects of construction. Is this code clear in its intent?</td>
<td>Factory objects, data objects. Testing easily limited to proof of existence.</td>
</tr>
<tr>
<td><strong>Execution</strong></td>
<td>Code only returns changed values. Pass in fake objects, verify the results. Easy to test.</td>
<td>Difficult to initialize private members containing state, may be impossible to validate. Mocks will be needed to check changes to state. Consider refactoring to dependency injection</td>
</tr>
<tr>
<td><strong>Mixed</strong></td>
<td>Can’t substitute constructed objects for mocks. Can still test results, but complexity means 100% code coverage may be difficult to achieve.</td>
<td>Almost impossible to adequately unit test. Cannot substitute internally constructed objects for mocks, difficult to initialize or validate private state.</td>
</tr>
</tbody>
</table>
Verifications

Automated Unit Test → call results → Call dependency

If A

Path 1 → Path 2 → return

Calls → Verify

Dependency
Test Doubles

Automated Unit Test  
Pass in double as dependency reference

Call dependency

If A

Path 1
Path 2

return

Actual Dependency

Dummy object
Only meets method signature, null values
Can be verified

call
return A=true
return A=value

call
return A=?
Flavors of Test Doubles

- Automated Unit Test
- Call dependency
- If A
- Path 1
- Path 2
- return
- return A=true
- Return exception: Produces invalid response to test an exception handler
- Responder: Produces valid response to test a specific path
- Temporary stub: Stands in for a component that is not yet available
- Entity chain snipping: A responder that uses a few objects to act as a large network of objects
- return A, B, C, A, B, C, ...
- Hard coded: Produces specific responses
- return A=true
- return A=true
- Configurable: Responses are configured during test setup
  - Configurable: Responses are configured during test setup
    - call
    - return A=test value
    - B, C, A, B, C, ...

Configurable: Responses are configured during test setup
Code Flaws
Code flaws that make testing hard

**Globals, singletons, and static methods**

*High coupling*

Can’t be substituted for mocks.
Code is brittle.
Code is confusing.
Code flaws that make testing hard

• **Production code includes test flows**

  DWORD tryThing() {
  #ifdef TESTING
  return E_FAIL;
  #else
  return GetLastError();
  #endif
  }

  ...

  if (tryThing() == E_FAIL)
    fix(GetLastError());
Code flaws that make testing hard

• Constructor does real work

Violates the SRP

• C’tors don’t return testable values

• Can’t replace components

class MyData {
private:
    DBQuery* q=null;
public:
    MyData(){};
    MyData(DBConn c){
        q=new DBQuery(c);}
    virtual ~MyData(){
        if (q) delete q;};
    int getData(int v){
        return q->query(v);};
};
Code flaws that make testing hard

- **Digging into collaborators**

  Violates the Law of Demeter

- Code is brittle, confusing
- API hides dependencies
- Hard to find all the dependencies
- High coupling

```cpp
class Tran {
    private:
    vector<Item> items;
    public: Tran();
    double getTotal() {
        double tot;
        for (auto i : items) {
            tot += i->amt() - i->disc->amt();
        }
        return tot;
    }
};
```
Code flaws that make testing hard

- **Class does too much**

  Class does too much
  
  - Violates the SRP
  
  - Hard to debug, test, understand, explain, extend, or name.
  
  - Low cohesion
  
  - High coupling

---

```cpp
class POS {
private:
    vector<Item> items;
    vector<Tender> tenders;
    int registerNumber;
    int storeNumber;
    int cashierNumber;
    DBConn database;
    DBQuery getItem(Item*i);
    double calcTaxes();
    void inventory(barcode);
};
```
Code flaws that make testing hard

- **Mixing construction with execution**
  
  Violates the SRP

- Can’t substitute mock objects

- High coupling

```cpp
class ItemLookUp {
public:
  ItemLookUp() {};
  Item* getItem(ID) {try{
    DBConn c;
    c.connect();
    DBQuery *q;
    q = new DBQuery(c);
    Item*i=q->query(ID);
    if (i->qty()>1) ...
  }catch{Dbexcept &e}... }
};
```
Code flaws that make testing hard

- **Use of inheritance instead of composition**

  Violates the SRP

  - Can’t replace inherited traits with mocks
  - Can’t change inheritance
  - High coupling

```cpp
class Disc : public Item{
public:
    Disc(){};
    DISC_TYPES discType;
    double discAmt;
    double getAmt(){
        return Itm::getAmt() -
               discAmt;}
};
```
Code flaws that make testing hard

- **Use of conditional logic instead of polymorphism**

  Violates the SRP

- High complexity
- Too many tests needed
- Hides intent
- Duplicates logic
- Scatters logic

```cpp
class Tran{
public:
    Tran(){};
    TRA_TYPES tt;
    int getTotal(){
        if (tt==SALE)
            ...
        else if (tt==RETURN)
            ...
        else if (tt==VOID)
            ...}
};
Code flaws that make testing hard

• **Mixing service objects with value objects**

  Violates ISP

• Can’t separate data from logic for testing
• High complexity
• Hard to reuse
Code Reviews
Effective code reviews

• Leverage automated source code analysis tools

• Use automated team review tools built into source control system
  • Use built-in diff functionality
  • Reviews outside of the comment system don’t count

• Invite the whole team to review, but only the team’s members
  • No outside comments by someone who doesn’t have skin in the game
  • Require at least one other person’s “approval” or “approved with comments”
  • Not approved if any “needs work”
Effective code reviews

• Comment only on the code. Be objective, and *never accuse*
  • “This code is highly complex”
  • “This code is a duplicate of function X”
  • “This logic belongs in the controller layer”
  • Never ask “Why did you...?”
  • Never say “This code is bad because...”

• As a team lead, be observant for violations
  • Privately talk to the violator and remind him or her to use caution

• Get management to make it clear to everyone that code reviews will *never* be used for performance evaluations
  • No punishment for bad reviews, or too many “needs work” disapprovals
  • No prizes for good reviews!
  • Any incentives at all, and they become dishonest and stop working
Refactoring
Refactoring – Code smells

- Duplicated Code
- Long Method
- Large Class
- Long Parameter List
- Divergent Change
- Shotgun Surgery

- Feature Envy
- Data Clumps
- Primitive Obsession
- Switch Statements
- Lazy Classes
- Speculative Generality
- Comments
DEMO TIME!