Requirements & Test Driven Development

ATPESC 2019

Jared O'Neal
Mathematics and Computer Science Division
Argonne National Laboratory

Q Center, St. Charles, IL (USA)
July 28 – August 9, 2019

License, citation, and acknowledgments

License and Citation

Acknowledgements

- This work was supported by the U.S. Department of Energy Office of Science, Office of Advanced Scientific Computing Research (ASCR), and by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.
- This work was performed in part at the Argonne National Laboratory, which is managed by UChicago Argonne, LLC for the U.S. Department of Energy under Contract No. DE-AC02-06CH11357
- Jim Willenbring and Reed Milewicz

THE WILD WORLD OF REQUIREMENTS

Informal definition

A complete collection of well-defined, mutually-consistent statements that define what you want to build and why these statements are important.

- What qualifies as “complete” is up to team
- Well-defined & mutually-consistent should not be optional

Requirements

- help understand **what we want** before we address **how to build it**,
- should be **verifiable**, and
- should be documented.

Functional vs. non-functional

Functional Requirements communicate what services should or should not be provided. This can include how they react to

- inputs and
- to corner/edge cases.

Example: A new feature shall be added to the SW such that simulations Z can be configured at runtime to use a lower-order, but more performant solver.

Non-functional Requirements communicate constraints on the services and functionality. These could be related to performance, portability, process, *etc*.

Example: The SW shall be developed as an open source project that is hosted on a Git-based version control host and shall have automated testing integrated in the repository for use with Continuous Integration.

Low-level requirements

- Technically-detailed or result of heavy constraints
- Possibly informed by implementation ideas & constraints
- Overly specific can hinder design, creativity, & freedom

Example: The SW architecture shall be upgraded such that a simulation can be run on nodes with Model X CPUs and Model Y GPUs. The use of GPUs shall be determined by the pre-processor.

High-level requirements

- Broad ideas, concepts, constraints, and abstractions
- Little technical detail
- Can be understood by people from different disciplines
- Not affected as strongly by changes
- Can be difficult for non-experts to turn into implementations

Example: The SW architecture shall be upgraded such that a simulation can be built to run on a node with only CPUs or on a node with accelerators.

Externally-imposed

Functional or Non-functional requirements due to

- Use of third-party libraries
- Working as a team of teams, or
- Including standardization (e.g. xSDK Community Package Policies)

Mathematical example

If a function f is symmetric about $x=0$, then

1. f' is antisymmetric about $x=0$ and
2. f'' is symmetric about $x=0$.

The routines for numerically estimating f' and f'' shall be implemented such that these mathematical properties are also true for the estimations.

Example from Prof. Edward Overman,
Mathematics Department at The Ohio State University

Participants

Requirements should capture viewpoints of different roles related to the development, maintenance, and use of the SW so that we discover more constraints & identify problems early

- Domain experts can define need, limits, & tolerances
- Developers & technical experts understand technical constraints
- Users define interfaces

Example formal design workflow

- Science/Engineering Cases
- Derive Requirements from S/E Cases
  - Requirement elicitation, specification, & validation
  - Determine tests needed to confirm that requirements are satisfied
- Convert Requirements into Design
  - Generate low-level technical specifications
  - Create design that satisfies specifications
- Implement
- Verification – did we satisfy the requirements?
- Validation – do the requirements result in SW that has correct/useful results?

Example incremental & iterative workflow

Incremental, iterative design repeatedly interleaves requirements analysis, design, implementation, & verification.

Workflow for single increment

- Requirement elicitation, specification, & validation
- Identify next necessary, high-priority tasks
- Design, implement, and verify tasks
- Simplify and improve code where possible to avoid degradation (refactor)

User stories
A form of requirement elicitation

As a …, I would like … so that ….

These statements

- express what needs to be done or a constraint on what we can do and
- encapsulate the reasons why the need or constraint should be considered.

User stories should start a discussion that concludes with requirements and possibly tasks to start work.

Elicitation & specification

As a user of the SW, I would like the storage of data to make good use of HPC resources and to leverage pre-existing libraries for reading data so that my simulations run in less time and time to results is reduced.

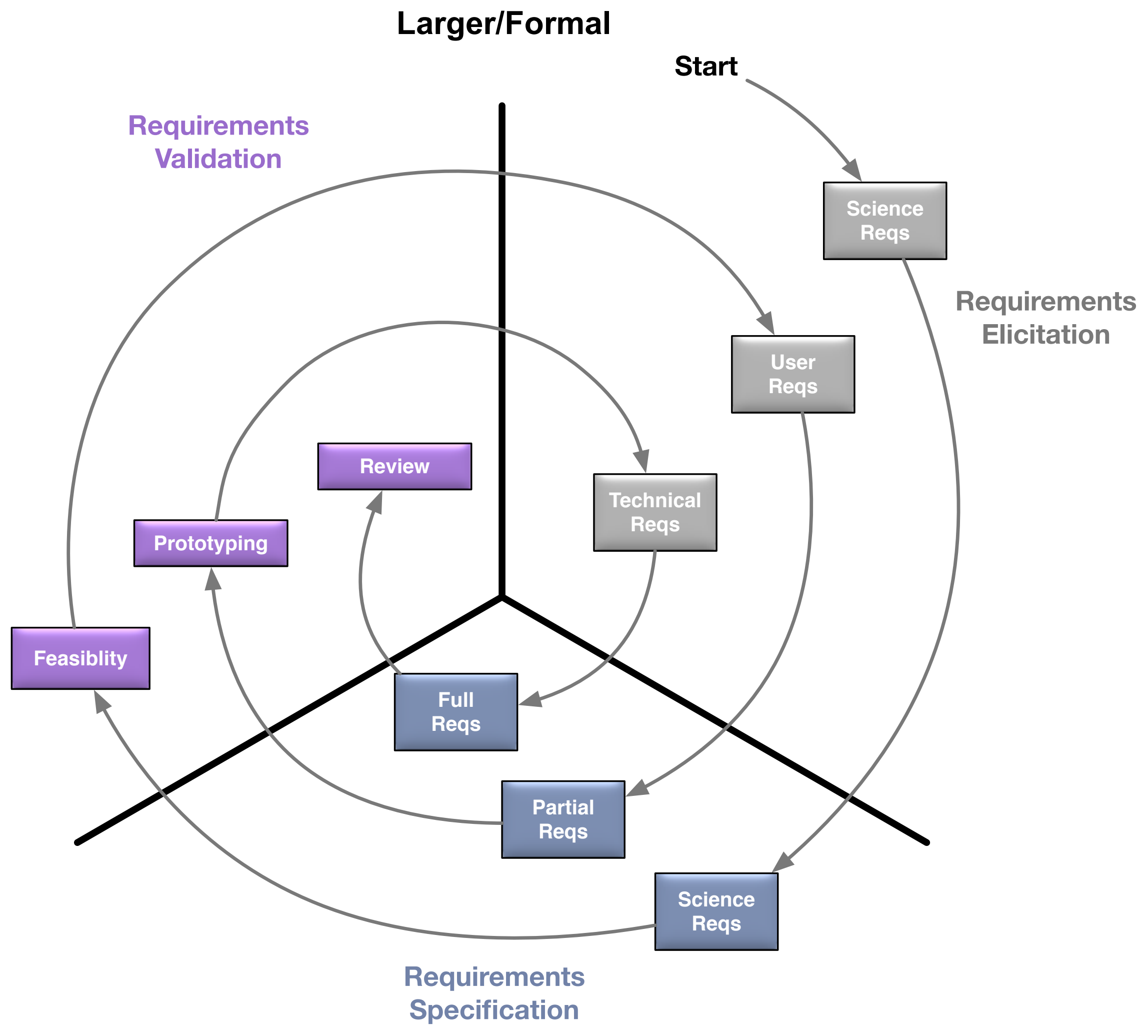
V1: The SW shall record simulation results, configuration values, hardware information, and telemetry via a parallel IO library and using a standard file format.

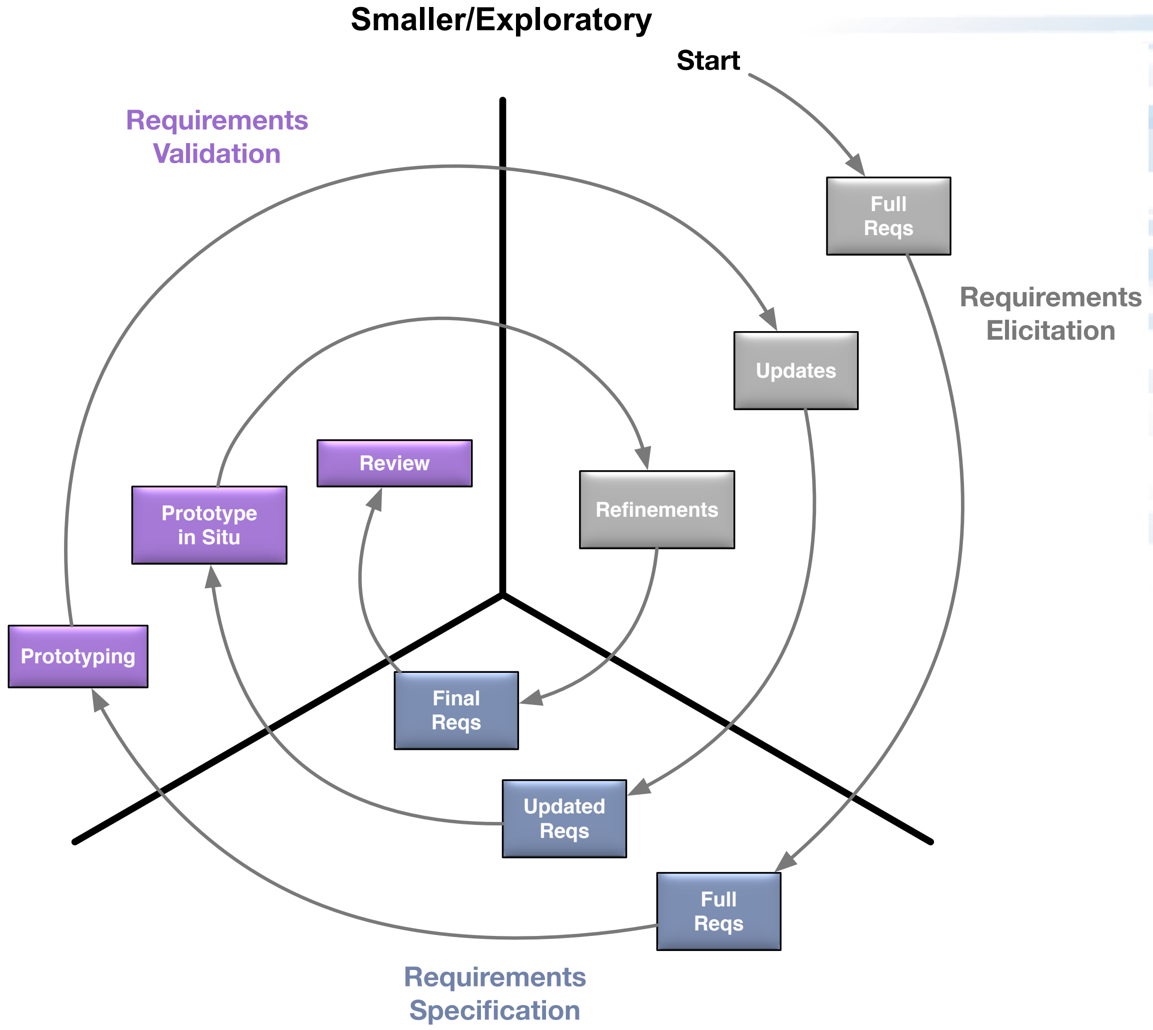
V2: The SW shall record simulation results, configuration values, hardware information, and telemetry via a parallel IO library and using a file format that is included in python, R, MATLAB, and C/C++.

V3: The SW shall record simulation results, configuration values, hardware information, and telemetry via parallel IO library XYZ v1.2.3 or greater.

Iteration & prototyping

Requirements require refining





Documentation
Requirements Management

- Documents should be clear, readable by many, & living
- Documentation maintenance should be easy & simple
- Design-by-contract requirements & motivation can be comments and inline documentation
- Should high-level or system-level requirements
  - Go into dedicated document?
  - Be included in the developer's guide or adapted for user guide?
  - Be a history of static requirements documents?
  - Be encoded in system-level test cases?

Are requirements for CSEM?
The Bad & Ugly

- Can be challenging and frustrating
- Can be seen as impediment to immediate progress
- Requirements change
  - Due to changing environment
  - Due to improved understanding
- Hard to know when enough is enough

Are requirements for CSEM?

The Good

- Achieve a clear & shared understanding of what needs to be done,
- Arrive at definitions & concepts that are understood by all,
- Bring out in the open ideas that seem obvious to some and usually go unstated,
- Bridge differences between disciplines & levels of expertise,
- Discover constraints/problems early,
- Link requirements with verification,
- Build a team where members feel like an important part of the process, and
- Arrive at idea of SW architecture through structuring/grouping requirements.

Formal vs. exploratory

Plan-based development

- Upfront design efforts
- Clear understanding of needs & goals
- Can be formal, structured, & rigid
- Can be slower due to overhead
- Can produce “useless” outputs
- Could be helpful if
  - Team is large or interdisciplinary
  - Members lack domain expertise or have different levels of experience
  - Team has high turnover
  - SW is large, mission-critical, or long lifetime

Agile development

- Design & understanding continually evolved through implementing
- Produces outputs that are “valuable” and necessary
- Constantly refactor code to simplify and clean
- Delay point of no return
- Could be helpful if
  - Team is small & highly-skilled
  - Requirements are constantly changing
  - Refactoring can be done efficiently

It’s a spectrum

“Software developers should be pragmatic and should choose those methods that are most effective for the type of system being developed, whether or not these are labeled agile or plan-driven.”

– Ian Sommervile [1]

Example

- Design infrastructure of software with plan-based so that design is
  - Mature and stable
  - Flexible and built with “reasonable” speculative generality
- Design localized code (*e.g.* solvers, kernels) with agile so that
  - We write only what is needed
  - We can explore & adjust without “excessive” overheads

Bibliography
Selected Books

1. Ian Sommerville, **Software Engineering**. Pearson, Tenth Edition, 2016.
2. Benjamin S. Blanchard and Wolter J. Fabrycky, **Systems Engineering and Analysis**. Pearson, Fifth Edition, 2011.
3. Andrew Hunt and David Thomas, **The Pragmatic Programmer**. Addison-Wesley, 1999.
4. Steve McConnell, **Code Complete**. Microsoft Press, Second Edition, 2004.
5. Alberto Sillitti and Giancarlo Succi, "Requirements Engineering for Agile Methods" in **Engineering and Managing Software Requirements**. Springer-Verlag, 2005.

Bibliography
Selected Articles

6. Yang Li, Emitza Guzman & Bernd Brügge, "**Effective Requirements Engineering for CSE Projects: A Lightweight Tool**", 2015.
7. Dustin Heaton & Jeffrey C. Carver, "**Claims about the use of software engineering practices in science: A systematic literature review**", 2015.
8. Yang Li, Matteo Harutunian, Nitesh Narayan, Bernd Brügge and Gerrit Buse, "**Requirements Engineering for Scientific Computing: A Model-Based Approach**", 2011.
9. Sarah Thew, Alistair Sutcliffe, Rob Procter, Oscar de Bruijn, John McNaught, Colin C. Venters, & Iain Buchan, "**Requirements Engineering for E-science: Experiences in Epidemiology**", 2009.
10. David Lorge Parnas & Paul C. Clements, "**A Rational Design Process: How and Why to Fake It**". IEEE Transactions on Software Engineering, Vol. SE-12, No. 2, February 1986.
11. **BSSW.io Requirements Engineering Blog Articles** by Reed Milewicz

TEST DRIVEN DEVELOPMENT

Software development aspirations

We would like to

- Write correct code productively,
- Verify the correctness of the code, and
- Write clean, maintainable code with useful documentation.

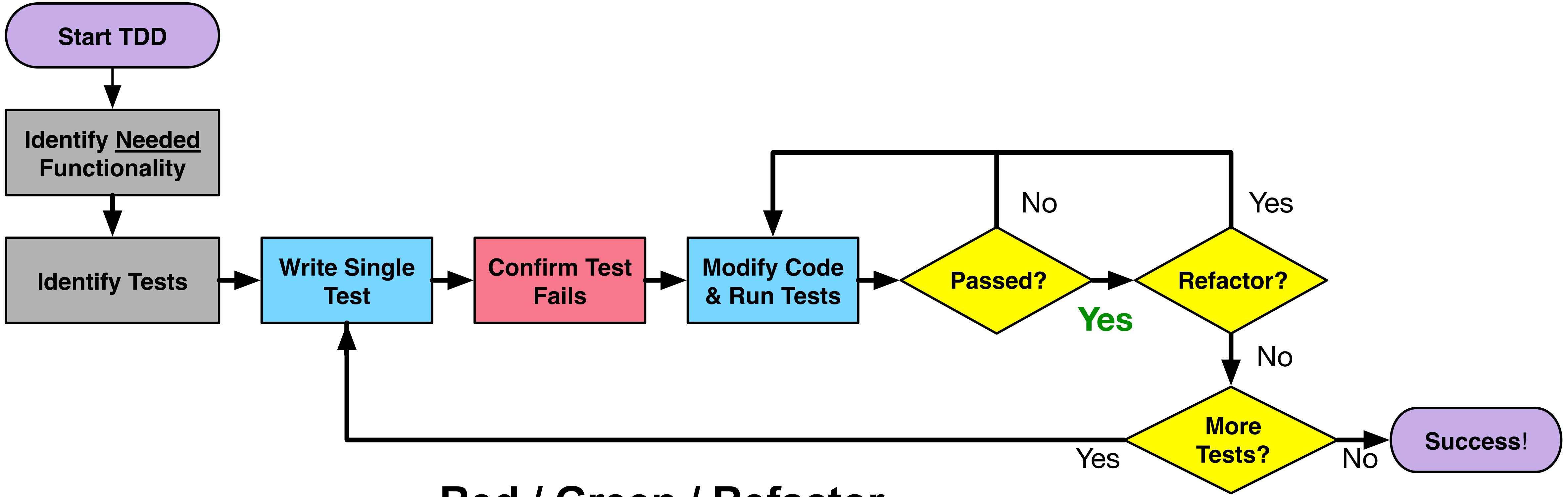
The reality of development can be

- Prototyped code,
- Little or out-of-date documentation, and
- No or subpar testing.

Test Driven Development helps us achieve our aspirations

Test driven development (TDD)

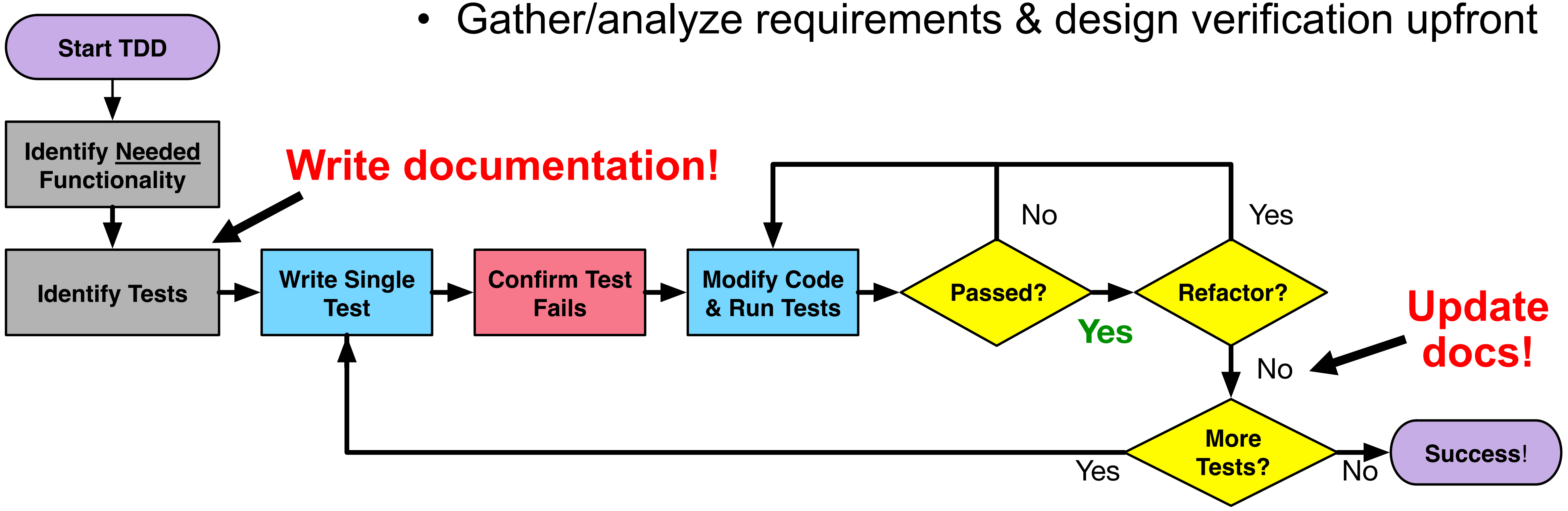
- Introduced as an XP and agile SW development method
- Simultaneously develop code & create **automated** verification tests
- Develop to pass tests in quickest/easiest way and then refactor



Red / Green / Refactor

Augmented TDD

- Express goals in human language **first**
- Gather/analyze requirements & design verification upfront



Augmented TDD example – Fibonacci sequence

The sequence is defined recursively as

$$F_n = F_{n-1} + F_{n-2}$$

```
/*
 * This routine shall calculate the n^{th} number in the famous Fibonacci
 * sequence, which is defined recursively by F_n = F_{n-1} + F_{n-2}.
 *
 * Parameters
 *    n - the index of the desired number in the sequence
 * Returns
 *    the number F_n
 */
int fibonacci(int n) {
    return -1000;
}
```

First design decision

The sequence is defined recursively as

$$F_n = F_{n-1} + F_{n-2}$$

Two definitions

1. $F_0 = 0$ and $F_1 = 1$
2. $F_1 = 1$ and $F_2 = 1$

The difference seems trivial, but we need to make a decision.

Study design decision

Starting Interface

```
// Poor first attempt
int fibonacci(int n) {
    return -1000;
}
```

Interface 1

```
// F_0=0 / F_1=1
unsigned int fibonacci(unsigned int n) {
    // No need to error check n
    return 0;
}
```

Interface 2

```
// F_1=1 / F_2=1
unsigned int fibonacci(unsigned int n) {
    // TODO: Throw error if n=0.
    return 0;
}
```

Design by contract & first test

```
/*
 * This routine shall calculate the n^{th}
 * number in the famous Fibonacci sequence,
 * which is defined recursively by
 *       F_n = F_{n-1} + F_{n-2}
 * with F_0 = 0 and F_1 = 1.
 *
 * Parameters
 *    n - the index of the desired number
 *        in the sequence
 * Returns
 *    the number F_n
 */
unsigned int fibonacci(unsigned int n) {
    // No need to error check n
    return 0;
}
```

```
int main(int argc, char* argv[]) {
    unsigned int nPassed = 0;
    unsigned int nTests  = 0;

    ++nTests;
    if (fibonacci(0) == 0)   ++nPassed;
    ++nTests;
    if (fibonacci(1) == 1)   ++nPassed;

    printf("\nPassed %d of %d tests\n\n",
           nPassed, nTests);
    if (nPassed < nTests) {
        return 1;
    }
    return 0;
}
```

We wanted failure $\longrightarrow$ ●

```
Passed 1 of 2 tests
```

Code to pass the test

```
int main(int argc, char* argv[]) {
    unsigned int nPassed = 0;
    unsigned int nTests  = 0;

    ++nTests;
    if (fibonacci(0) == 0)   ++nPassed;
    ++nTests;
    if (fibonacci(1) == 1)   ++nPassed;

    printf("\nPassed %d of %d tests\n\n",
           nPassed, nTests);
    if (nPassed < nTests) {
        return 1;
    }
    return 0;
}
```

Fails for n > 1

```
unsigned int fibonacci(unsigned int n) {
    if (n < 2) {
        return n;
    }
    return 0;
}
```

Red to green! → ●

```
Passed 2 of 2 tests
```

Be destructive

Fails for n > 1

```
unsigned int fibonacci(unsigned int n) {
    if (n < 2) {
        return n;
    }
    return 0;
}
```

```
++nTests;
if (fibonacci(0) == 0)    ++nPassed;
++nTests;
if (fibonacci(1) == 1)    ++nPassed;
++nTests;
if (fibonacci(2) == 1)    ++nPassed;
++nTests;
if (fibonacci(3) == 2)    ++nPassed;
++nTests;
if (fibonacci(6) == 8)    ++nPassed;
```

● `Passed 2 of 5 tests`

Contract satisfied!

Simplest, easiest to write

```
unsigned int fibonacci(unsigned int n) {
    if (n < 2) {
        return n;
    }
    return fibonacci(n-1) + fibonacci(n-2);
}
```

Sufficient verification tests

```
++nTests;
if (fibonacci(0) == 0)    ++nPassed;
++nTests;
if (fibonacci(1) == 1)    ++nPassed;
++nTests;
if (fibonacci(2) == 1)    ++nPassed;
++nTests;
if (fibonacci(3) == 2)    ++nPassed;
++nTests;
if (fibonacci(6) == 8)    ++nPassed;
```

●
```
Passed 5 of 5 tests
```

Refactor

Change implementation, **not** contract

- No need to update documentation
- No need to update test suite

```
unsigned int fibonacci(unsigned int n) {
    if (n < 2) {
        return n;
    }
    return fibonacci(n-1) + fibonacci(n-2);
}
```

```
unsigned int fibonacci(unsigned int n) {
    if (n < 2) {
        return n;
    }
    // Start with i=2
    unsigned int  F_i_m1 = 1;
    unsigned int  F_i    = 1;
    unsigned int  tmp    = 0;
    for (unsigned int i=3; i<=n; ++i) {
        tmp     = F_i;
        F_i    += F_i_m1;
        F_i_m1  = tmp;
    }
    return F_i;
}
```

● Passed 5 of 5 tests

Test driven development

Incremental development by writing/modifying code & getting immediate feedback

- Coevolves the functionality of the code with verification of that functionality
- Simultaneous requirements gathering/analysis & verification planning
- All code is testable & code that you need
- Produces a proven test suite for future regression testing
- Minimizes tedium of writing tests
- Proactively **prevents** bug creation
- Adopt to move plan-based more toward agile

Note that

- Occasionally breaking a test or the code temporarily to see how a test fails can be helpful
- Many IDEs integrate TDD testing into the interface

EXASCALE COMPUTING PROJECT

Bibliography

1. Kent Beck, Test Driven Development: By Example. Addison-Wesley, 2002.
2. Steve Freeman & Nat Pryce, Growing Object-Oriented Software, Guided by Tests. Addison-Wesley, 2009.
3. Ian Sommerville, Software Engineering. Pearson, Tenth Edition, 2016.
4. Aziz Nanthaamornphong & Jeffrey C. Carver, "Test-Driven Development in scientific software: a survey". *Software Quality Journal*, Vol 25, Issue 2, 2017, pp. 343-372.
5. Aziz Nanthaamornphong, Jeffrey C. Carver, Karla Morris, *et. al.*, "Building CLiiME via Test-Driven Development: A Case Study", May/June 2014.

Agenda

Time	Module	Topic	Speaker
9:30am-10:15am	01	Objectives, Motivation, & Overview	Katherine Riley, ANL
10:15am-10:45am		*Break*	
10:45am-11:30am	02	Requirements & Test-Driven Development	Jared O'Neal, ANL
11:30am-12:30pm	03	Software Design & Testing	Anshu Dubey, ANL
12:30pm-1:30pm		*Lunch*	
1:30pm-2:15pm	04	Licensing	James Willenbring, SNL
2:15pm-3:15pm	05	Agile Methodologies & Useful GitHub Tools	James Willenbring, SNL
3:15pm-3:45pm		*Break*	
3:45pm-4:15pm	06	Git Workflows	Jared O'Neal, ANL
4:15pm-4:55pm	07	Code Coverage & Continuous Integration	Jared O'Neal, ANL
4:55pm-5:30pm	08	Software Refactoring & Documentation	Anshu Dubey, ANL

Why documentation?

“Those who read the software documentation want to understand the programs, not to relive their discovery.”
– Parnas & Clements [10]