Test Driven Development of Scientific Models

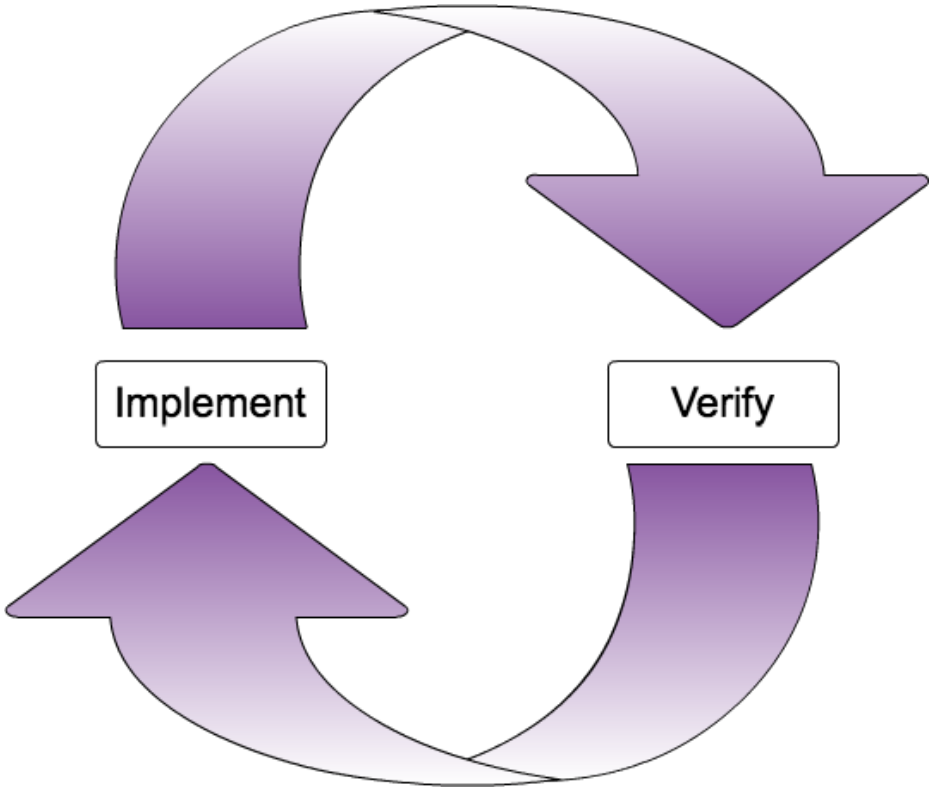
Tom Clune

Software Systems Support Office
Earth Science Division
NASA Goddard Space Flight Center

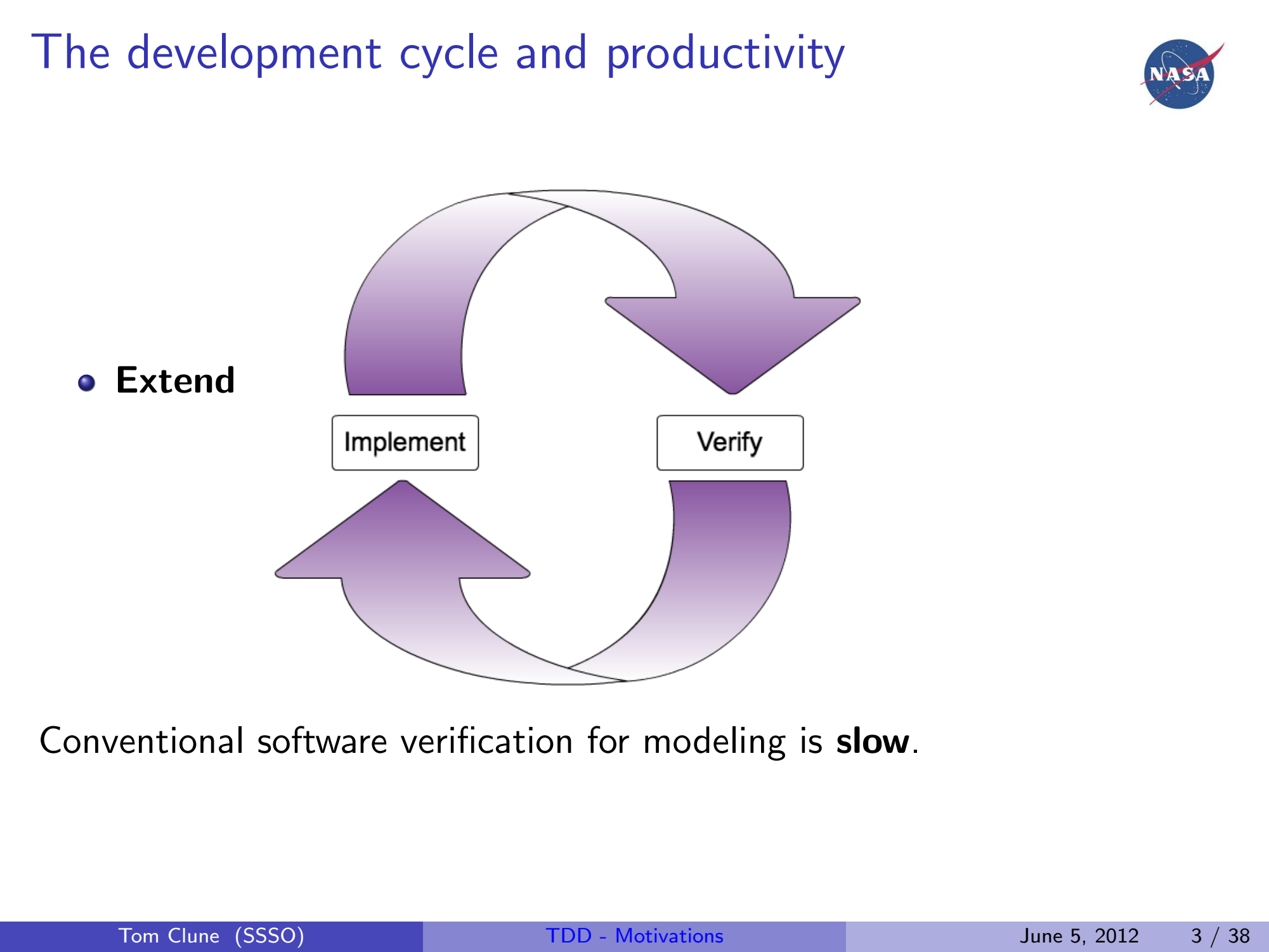
June 5, 2012

Outline

The development cycle and productivity



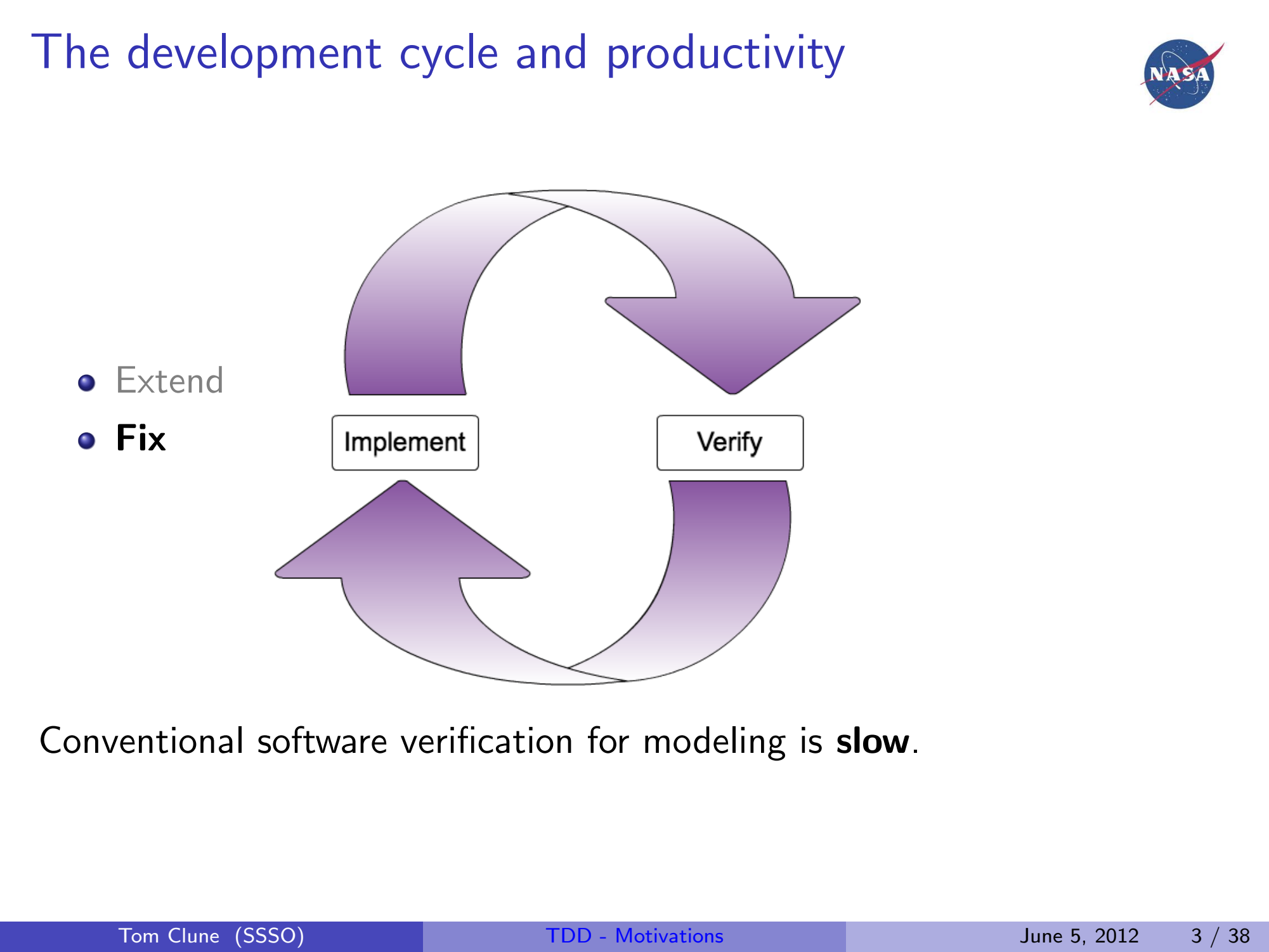
The development cycle and productivity



- **Extend**

Conventional software verification for modeling is **slow**.

The development cycle and productivity

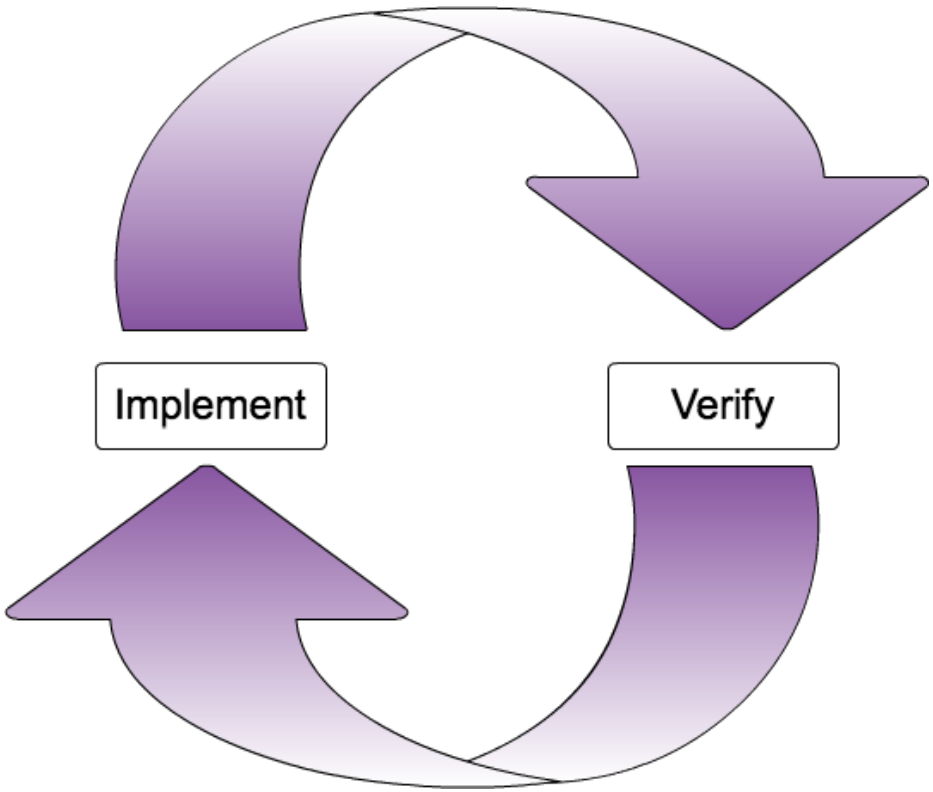


- Extend
- **Fix**

Conventional software verification for modeling is **slow**.

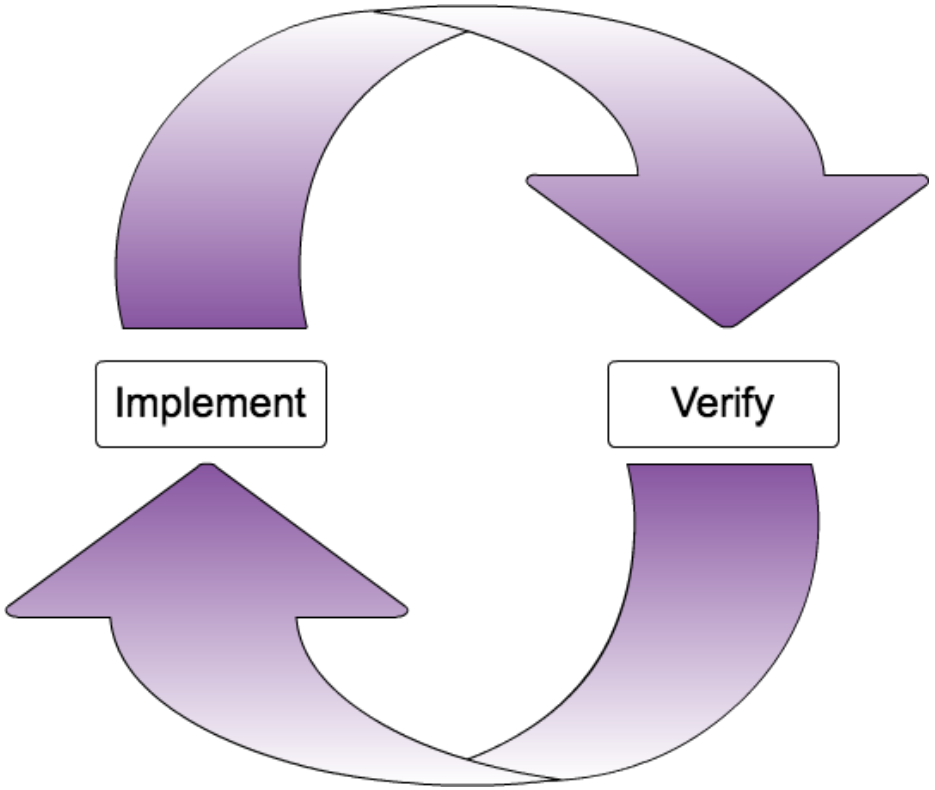
The development cycle and productivity

- Extend
- Fix
- **Port**



Conventional software verification for modeling is **slow**.

The development cycle and productivity

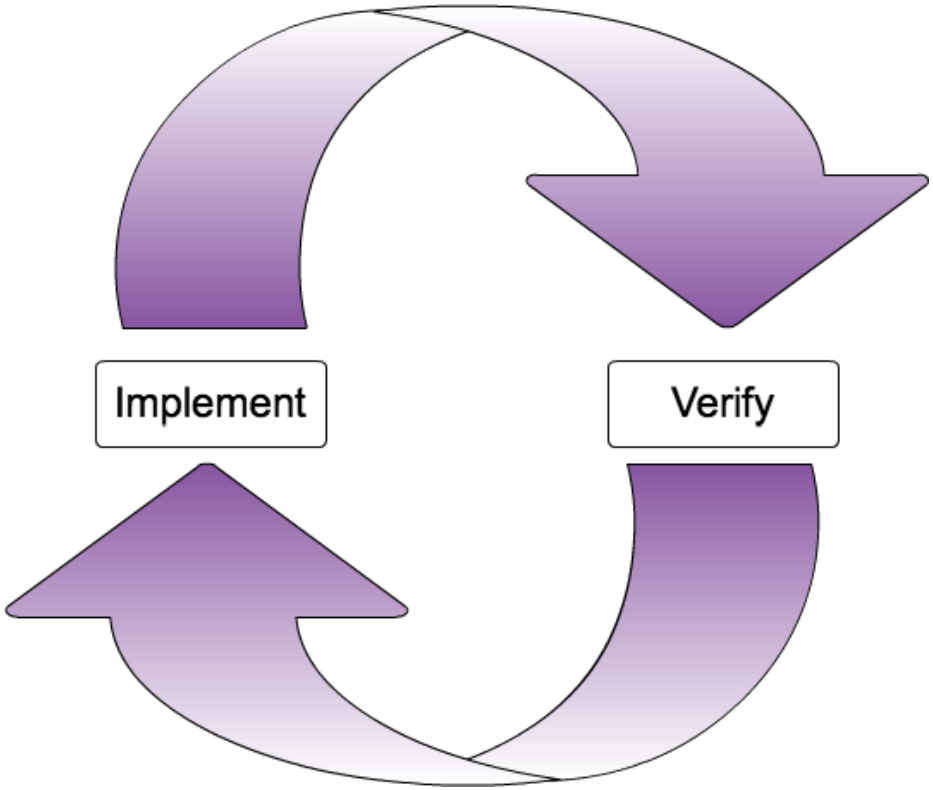


- Extend
- Fix
- Port

- **Compiles?**

Conventional software verification for modeling is **slow**.

The development cycle and productivity

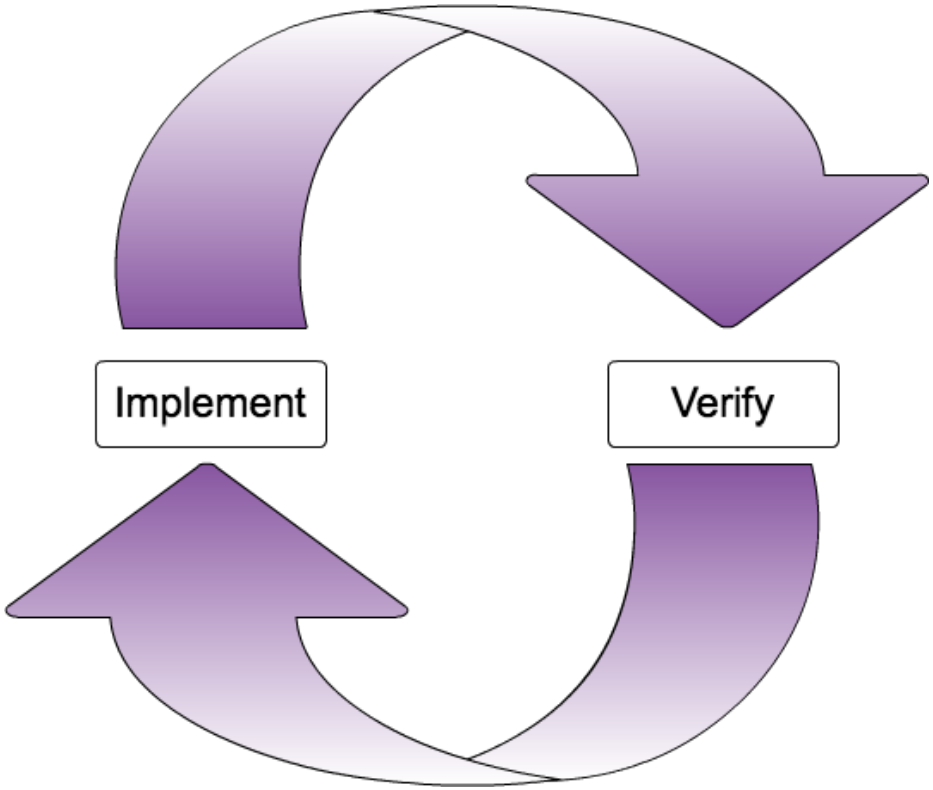


- Extend
- Fix
- Port

- Compiles?
- **Executes?**

Conventional software verification for modeling is **slow**.

The development cycle and productivity

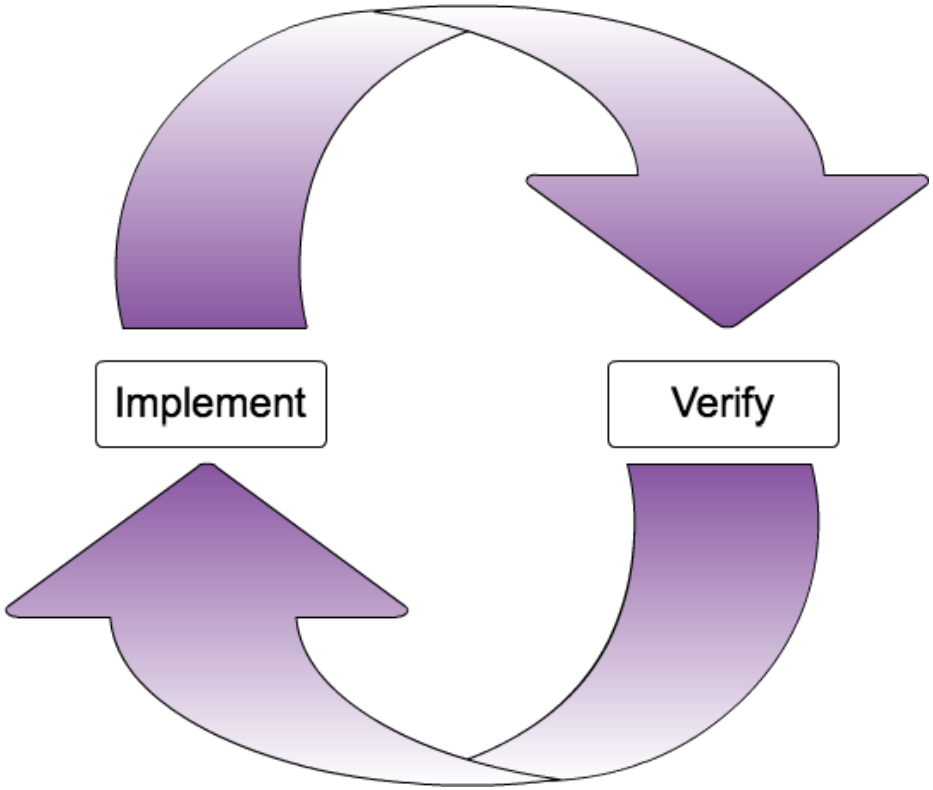


- Extend
- Fix
- Port

- Compiles?
- Executes?
- **Looks ok?**

Conventional software verification for modeling is **slow**.

The development cycle and productivity



- Extend
- Fix
- Port

- Compiles?
- Executes?
- Looks ok?
- **Correct?**

Conventional software verification for modeling is **slow**.

Some observations

- Risk grows with magnitude of implementation step
- Magnitude of implementation step grows with cost of verification/validation

Some observations

- Risk grows with magnitude of implementation step
- Magnitude of implementation step grows with cost of verification/validation

Conclusion:
Optimize productivity by reducing cost of verification!

Trusting the Science

Climate modeling has grown to be of extreme socioeconomic importance:

Trusting the Science

Climate modeling has grown to be of extreme socioeconomic importance:

- Adaptation/mitigation strategies easily exceed $100 trillion

Trusting the Science

Climate modeling has grown to be of extreme socioeconomic importance:

- Adaptation/mitigation strategies easily exceed $100 trillion
- Implications are politically sensitive/divisive

Trusting the Science

Climate modeling has grown to be of extreme socioeconomic importance:

- Adaptation/mitigation strategies easily exceed $100 trillion
- Implications are politically sensitive/divisive
- **Scientific integrity is crucial**

Trusting the Science

Climate modeling has grown to be of extreme socioeconomic importance:

- Adaptation/mitigation strategies easily exceed $100 trillion
- Implications are politically sensitive/divisive
- **Scientific integrity is crucial**

Software management and testing have not kept pace

Trusting the Science

Climate modeling has grown to be of extreme socioeconomic importance:

- Adaptation/mitigation strategies easily exceed $100 trillion
- Implications are politically sensitive/divisive
- **Scientific integrity is crucial**

Software management and testing have not kept pace

- Strong *validation* against data, but ...

Trusting the Science

Climate modeling has grown to be of extreme socioeconomic importance:

- Adaptation/mitigation strategies easily exceed $100 trillion
- Implications are politically sensitive/divisive
- **Scientific integrity is crucial**

Software management and testing have not kept pace

- Strong *validation* against data, but ...
- Validation is a blunt tool for isolating issues in coupled systems

Trusting the Science

Climate modeling has grown to be of extreme socioeconomic importance:

- Adaptation/mitigation strategies easily exceed $100 trillion
- Implications are politically sensitive/divisive
- **Scientific integrity is crucial**

Software management and testing have not kept pace

- Strong *validation* against data, but ...
- Validation is a blunt tool for isolating issues in coupled systems
- Validation cannot detect certain types of software defects:

Trusting the Science

Climate modeling has grown to be of extreme socioeconomic importance:

- Adaptation/mitigation strategies easily exceed $100 trillion
- Implications are politically sensitive/divisive
- **Scientific integrity is crucial**

Software management and testing have not kept pace

- Strong *validation* against data, but ...
- Validation is a blunt tool for isolating issues in coupled systems
- Validation cannot detect certain types of software defects:
  - ★ Those that are only exercised in rare/future regimes

Trusting the Science

Climate modeling has grown to be of extreme socioeconomic importance:

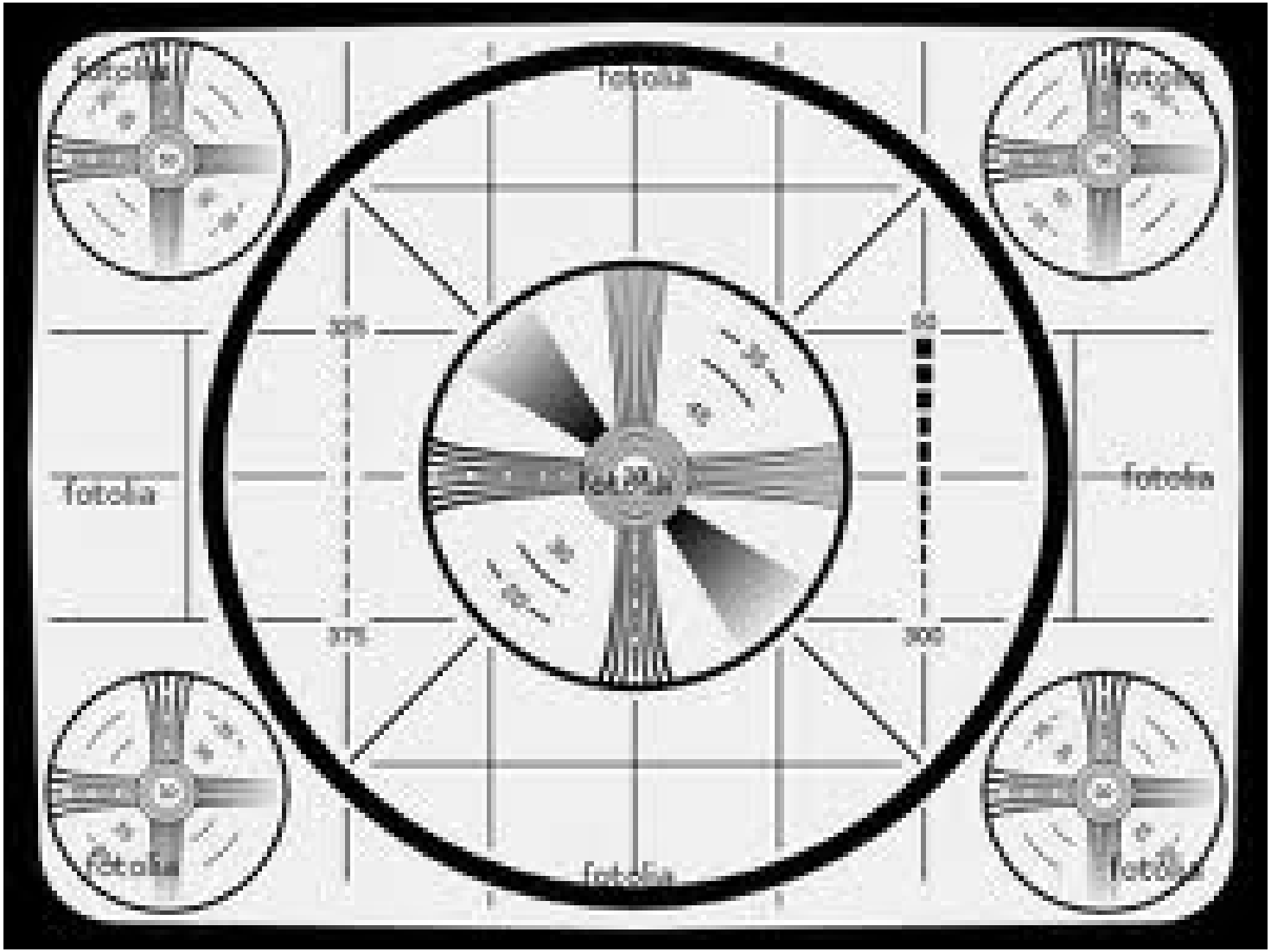
- Adaptation/mitigation strategies easily exceed $100 trillion
- Implications are politically sensitive/divisive
- **Scientific integrity is crucial**

Software management and testing have not kept pace

- Strong *validation* against data, but ...
- Validation is a blunt tool for isolating issues in coupled systems
- Validation cannot detect certain types of software defects:
  - ★ Those that are only exercised in rare/future regimes
  - ★ Those which change results below detection threshold

Outline

Testing


fotolia
fotolia
fotolia
fotolia
fotolia
fotolia
fotolia
fotolia
fotolia

Test Harness - work in safety

Collection of tests that constrain system

Test Harness - work in safety

Collection of tests that constrain system

- **Detects unintended changes**

Test Harness - work in safety

Collection of tests that constrain system

- Detects unintended changes
- **Localizes defects**

Test Harness - work in safety

Collection of tests that constrain system

- Detects unintended changes
- Localizes defects
- **Improves developer confidence**

Test Harness - work in safety

Collection of tests that constrain system

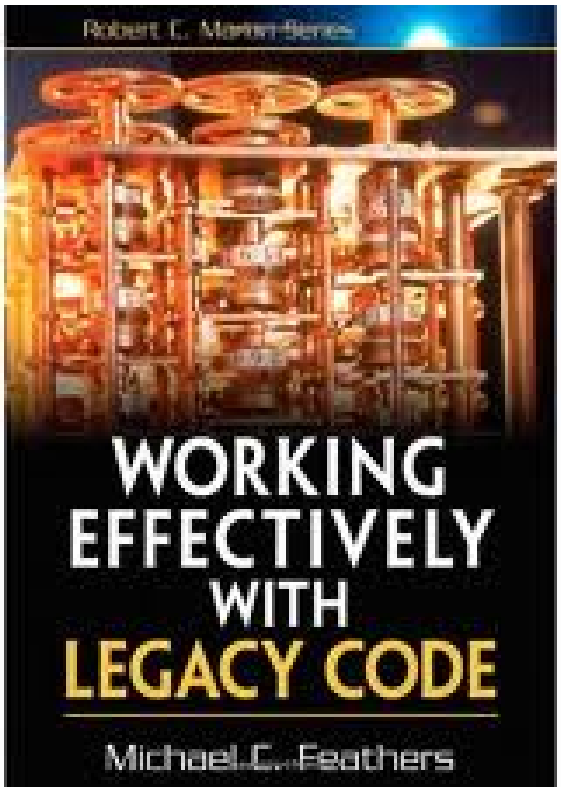
- Detects unintended changes
- Localizes defects
- Improves developer confidence
- **Decreases risk from change**

Do you write legacy code?

Do you write legacy code?

"The main thing that distinguishes legacy code from non-legacy code is tests, or rather a lack of tests."

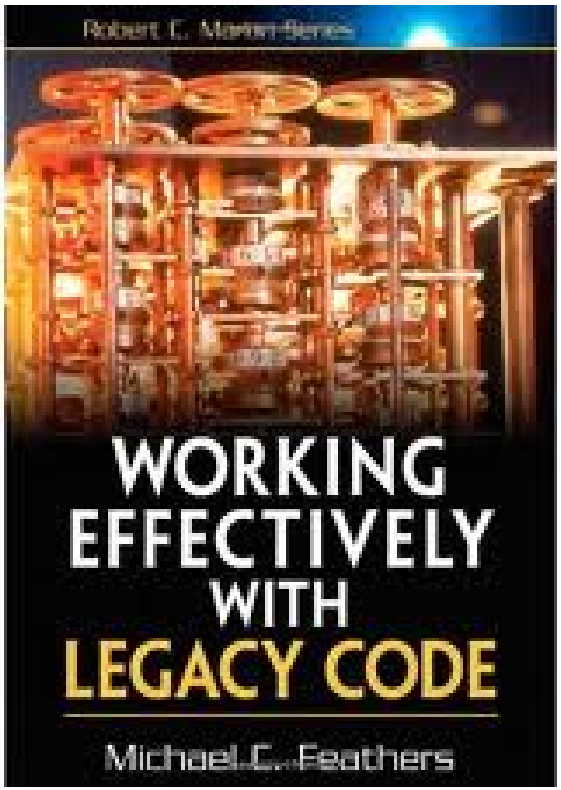
Michael Feathers
Working Effectively with Legacy Code



Do you write legacy code?

"The main thing that distinguishes legacy code from non-legacy code is tests, or rather a lack of tests."

Michael Feathers
Working Effectively with Legacy Code



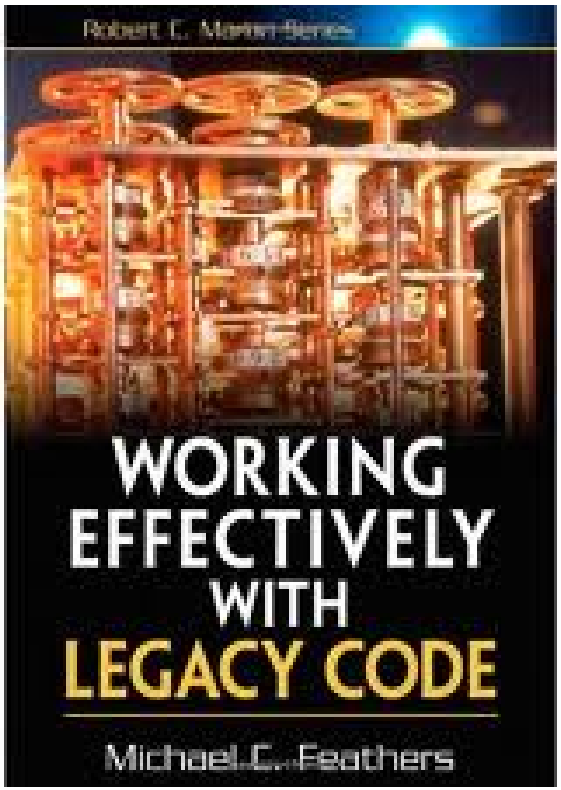
Lack of tests leads to fear of introducing subtle bugs and/or changing things inadvertently.

- Programming on a tightrope

Do you write legacy code?

"The main thing that distinguishes legacy code from non-legacy code is tests, or rather a lack of tests."

Michael Feathers
Working Effectively with Legacy Code



Lack of tests leads to fear of introducing subtle bugs and/or changing things inadvertently.

- Programming on a tightrope

This is also a barrier to involving pure software engineers in the development of our models.

Excuses, excuses ...

- Takes too much time to write tests

Excuses, excuses ...

- Takes too much time to write tests
- Too difficult to maintain tests

Excuses, excuses ...

- Takes too much time to write tests
- Too difficult to maintain tests
- It takes too long to run the tests

Excuses, excuses ...

- Takes too much time to write tests
- Too difficult to maintain tests
- It takes too long to run the tests
- It is not my job

Excuses, excuses ...

- Takes too much time to write tests
- Too difficult to maintain tests
- It takes too long to run the tests
- It is not my job
- “Correct” behavior is unknown

Excuses, excuses ...

- Takes too much time to write tests
- Too difficult to maintain tests
- It takes too long to run the tests
- It is not my job
- "Correct" behavior is unknown

http://java.dzone.com/articles/unit-test-excuses

- James Sugrue

Excuses, excuses ...

- Takes too much time to write tests
- Too difficult to maintain tests
- It takes too long to run the tests
- It is not my job
- "Correct" behavior is unknown

  http://java.dzone.com/articles/unit-test-excuses

  - James Sugrue

- **Numeric/scientific code cannot be tested, because ...**

Just what is a test anyway?

Tests can exist in many forms

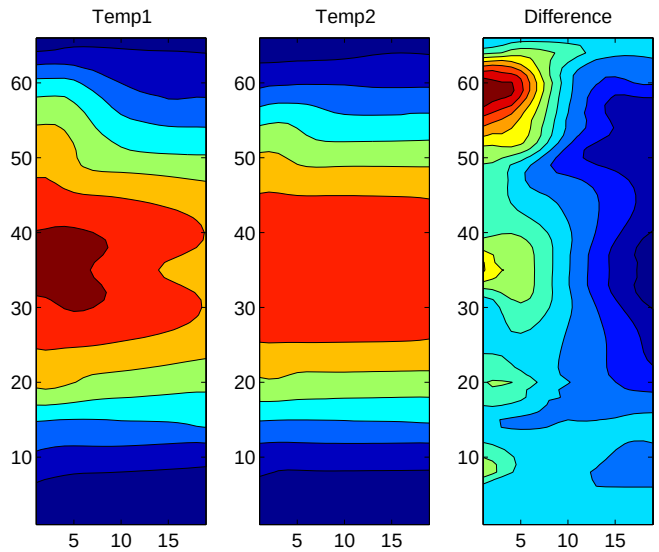
- Conditional termination:

```
IF (PA(I,J)+PTOP.GT.1200.) &
    call stop_model('ADVECM: Pressure diagnostic error',11)
```

- Diagnostic print statement

```
print*, 'loss of mass = ', deltaMass
```

- Visualization of output



Analogy with Scientific Method?

Scientists ought to like TDD:

Analogy with Scientific Method?

Scientists ought to like TDD:

Objective reality $\longrightarrow$ Requirements

Analogy with Scientific Method?

Scientists ought to like TDD:

Objective reality	$\longrightarrow$	Requirements
Constraints: theory and data	$\longrightarrow$	Constraints: existing tests

Analogy with Scientific Method?

Scientists ought to like TDD:

Objective reality	$\longrightarrow$	Requirements
Constraints: theory and data	$\longrightarrow$	Constraints: existing tests
Formulate hypothesis	$\longrightarrow$	Select a feature

Analogy with Scientific Method?

Scientists ought to like TDD:

Objective reality	$\longrightarrow$	Requirements
Constraints: theory and data	$\longrightarrow$	Constraints: existing tests
Formulate hypothesis	$\longrightarrow$	Select a feature
Design experiment	$\longrightarrow$	Write a test

Analogy with Scientific Method?

Scientists ought to like TDD:

Objective reality	$\longrightarrow$	Requirements
Constraints: theory and data	$\longrightarrow$	Constraints: existing tests
Formulate hypothesis	$\longrightarrow$	Select a feature
Design experiment	$\longrightarrow$	Write a test
Run experiment	$\longrightarrow$	Run tests

Analogy with Scientific Method?

Scientists ought to like TDD:

Objective reality	$\longrightarrow$	Requirements
Constraints: theory and data	$\longrightarrow$	Constraints: existing tests
Formulate hypothesis	$\longrightarrow$	Select a feature
Design experiment	$\longrightarrow$	Write a test
Run experiment	$\longrightarrow$	Run tests
Refine hypothesis	$\longrightarrow$	Refine implementation

http://agile2003.agilealliance.org/files/P6Paper.pdf

Properties of good tests

Properties of good tests

- Isolating
  - Test failure indicates location in source code

Properties of good tests

- Isolating
  - Test failure indicates location in source code
- Orthogonal
  - Each defect results in failure of small number of tests

Properties of good tests

- Isolating
  - Test failure indicates location in source code
- Orthogonal
  - Each defect results in failure of small number of tests
- Complete
  - Each bit of functionality covered by at least one test

Properties of good tests

- Isolating
  - Test failure indicates location in source code
- Orthogonal
  - Each defect results in failure of small number of tests
- Complete
  - Each bit of functionality covered by at least one test
- Independent
  - No side effects
  - Test order does not matter
  - Corollary: cannot terminate execution

Properties of good tests

- Isolating
  - Test failure indicates location in source code
- Orthogonal
  - Each defect results in failure of small number of tests
- Complete
  - Each bit of functionality covered by at least one test
- Independent
  - No side effects
  - Test order does not matter
  - Corollary: cannot terminate execution
- Frugal
  - Run quickly
  - Small memory, etc.

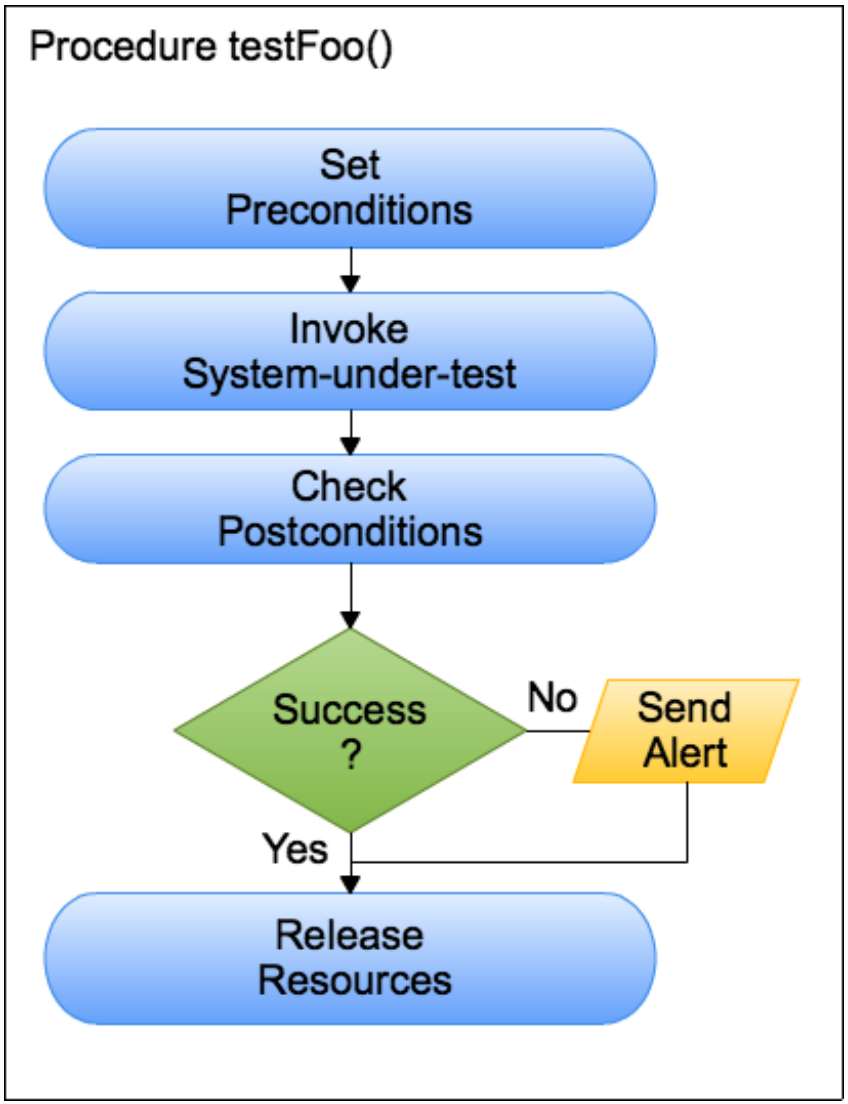
Properties of good tests

- Isolating
  - Test failure indicates location in source code
- Orthogonal
  - Each defect results in failure of small number of tests
- Complete
  - Each bit of functionality covered by at least one test
- Independent
  - No side effects
  - Test order does not matter
  - Corollary: cannot terminate execution
- Frugal
  - Run quickly
  - Small memory, etc.
- Automated and repeatable

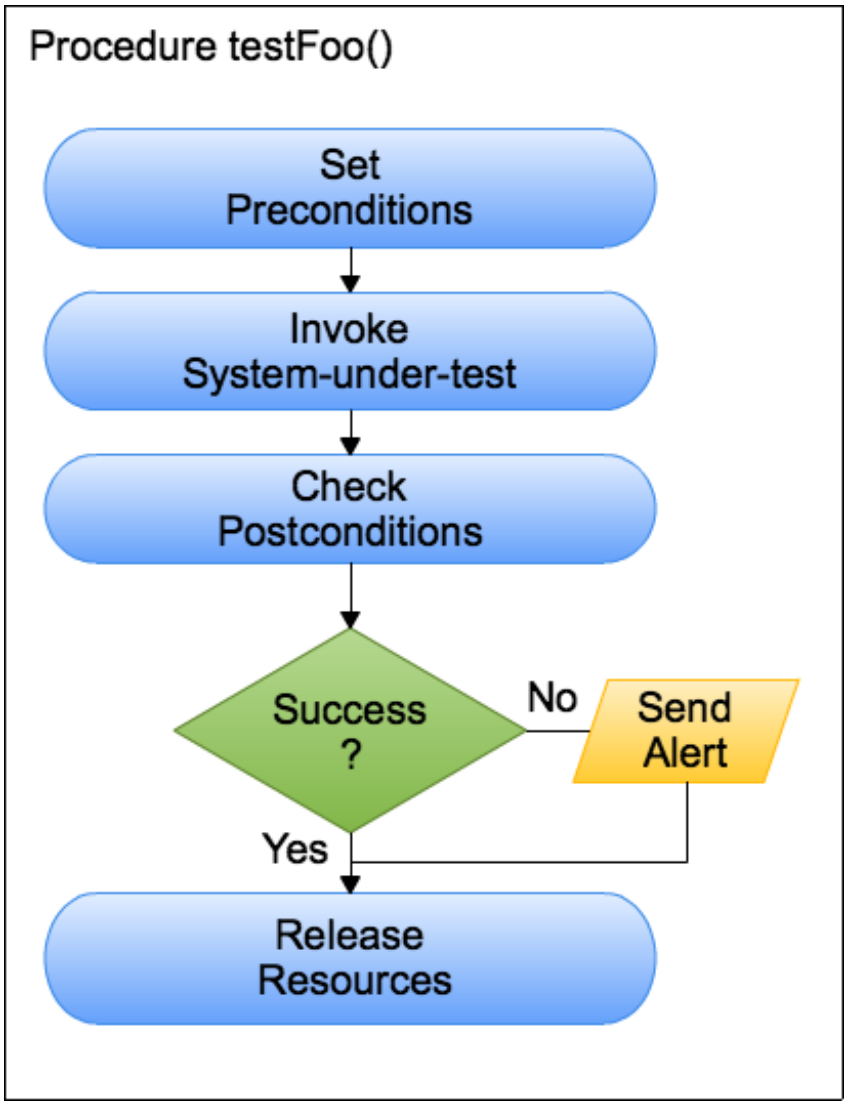
Properties of good tests

- Isolating
  - Test failure indicates location in source code
- Orthogonal
  - Each defect results in failure of small number of tests
- Complete
  - Each bit of functionality covered by at least one test
- Independent
  - No side effects
  - Test order does not matter
  - Corollary: cannot terminate execution
- Frugal
  - Run quickly
  - Small memory, etc.
- Automated and repeatable
- Clear intent

Anatomy of a Software Test Procedure

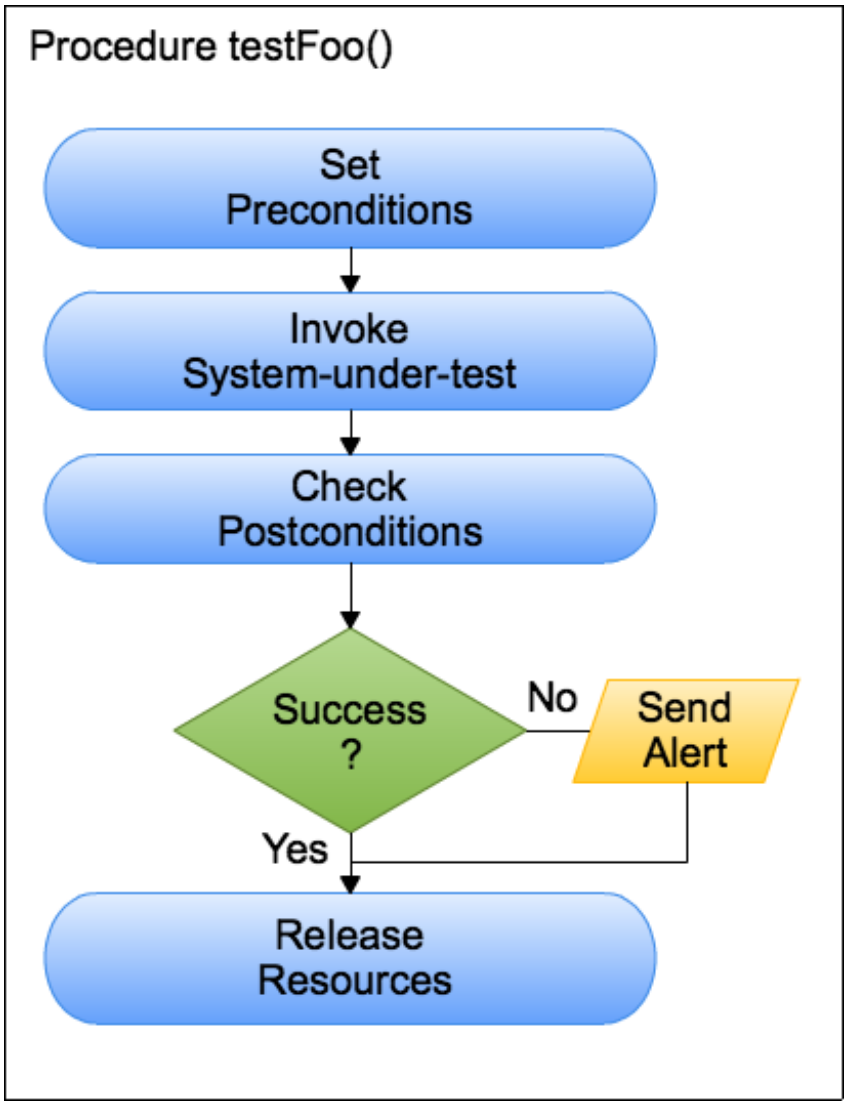


Anatomy of a Software Test Procedure



testTrajectory() ! $s = \frac{1}{2}at^2$

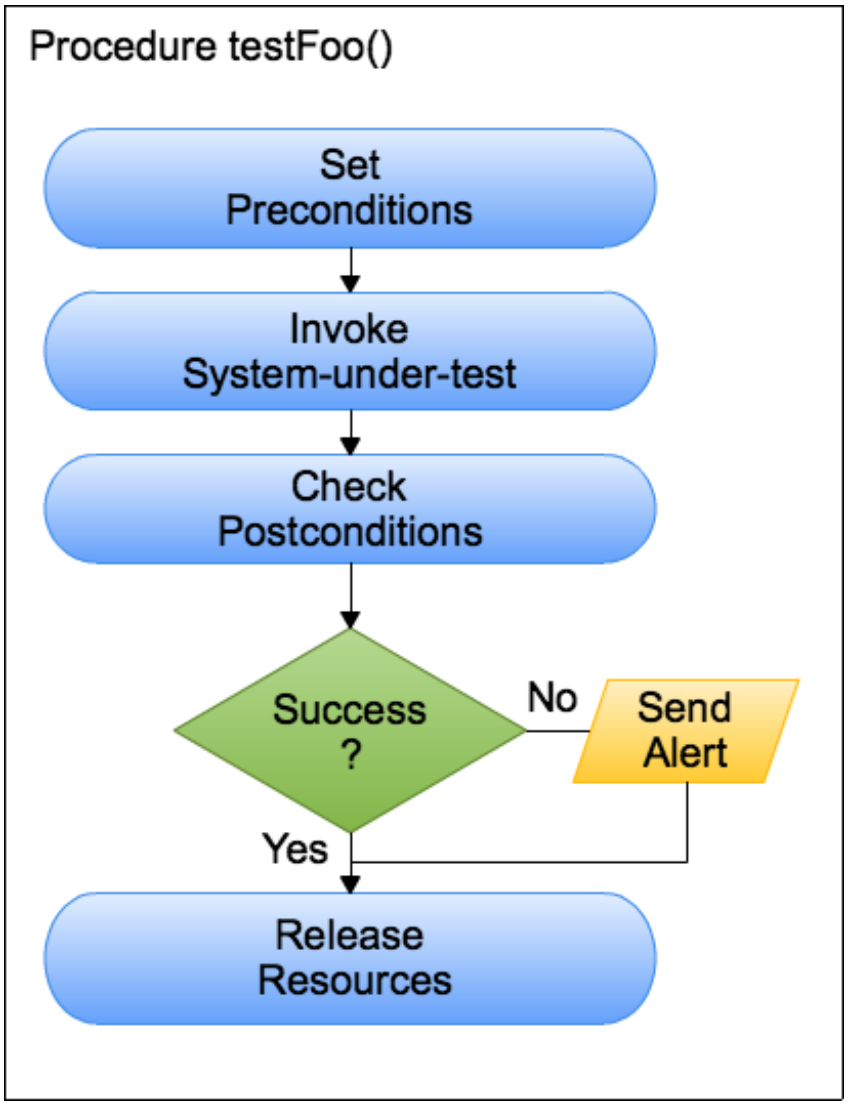
Anatomy of a Software Test Procedure



testTrajectory() ! $s = \frac{1}{2}at^2$

a = 2.; t = 3.

Anatomy of a Software Test Procedure

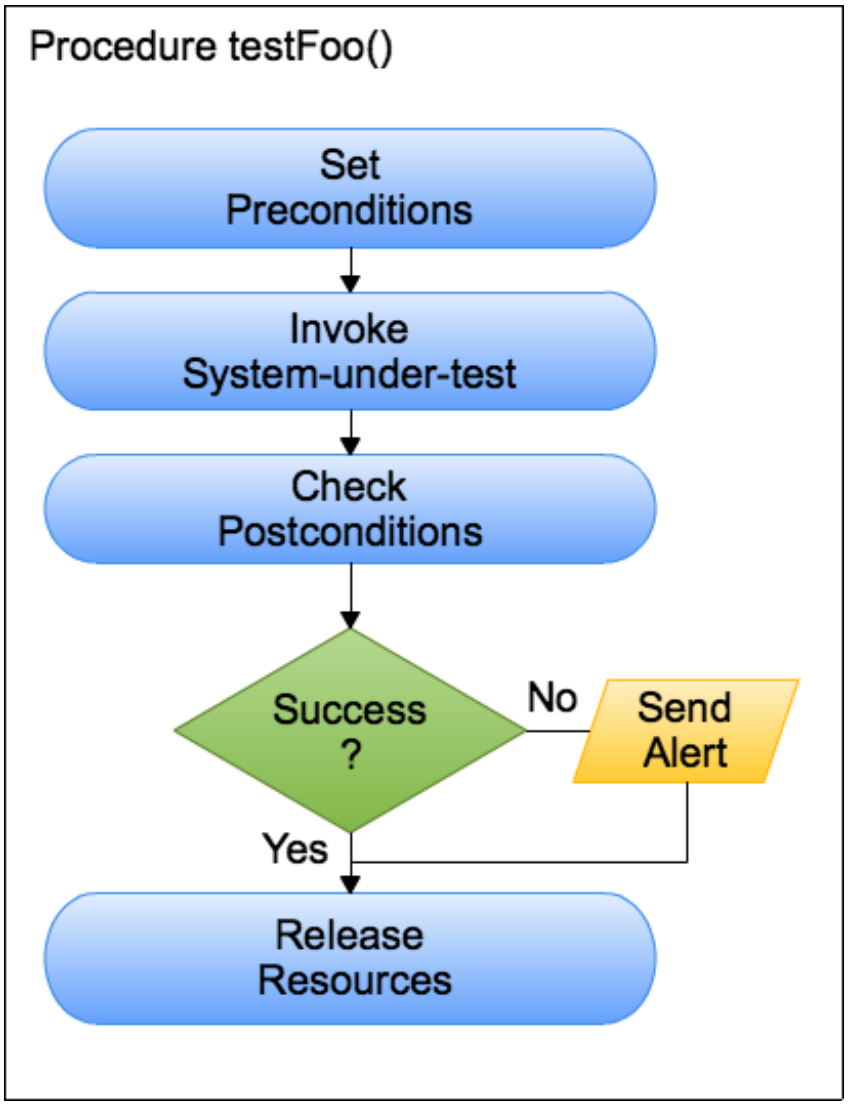


testTrajectory() ! $s = \frac{1}{2}at^2$

a = 2.; t = 3.

s = trajectory(a, t)

Anatomy of a Software Test Procedure



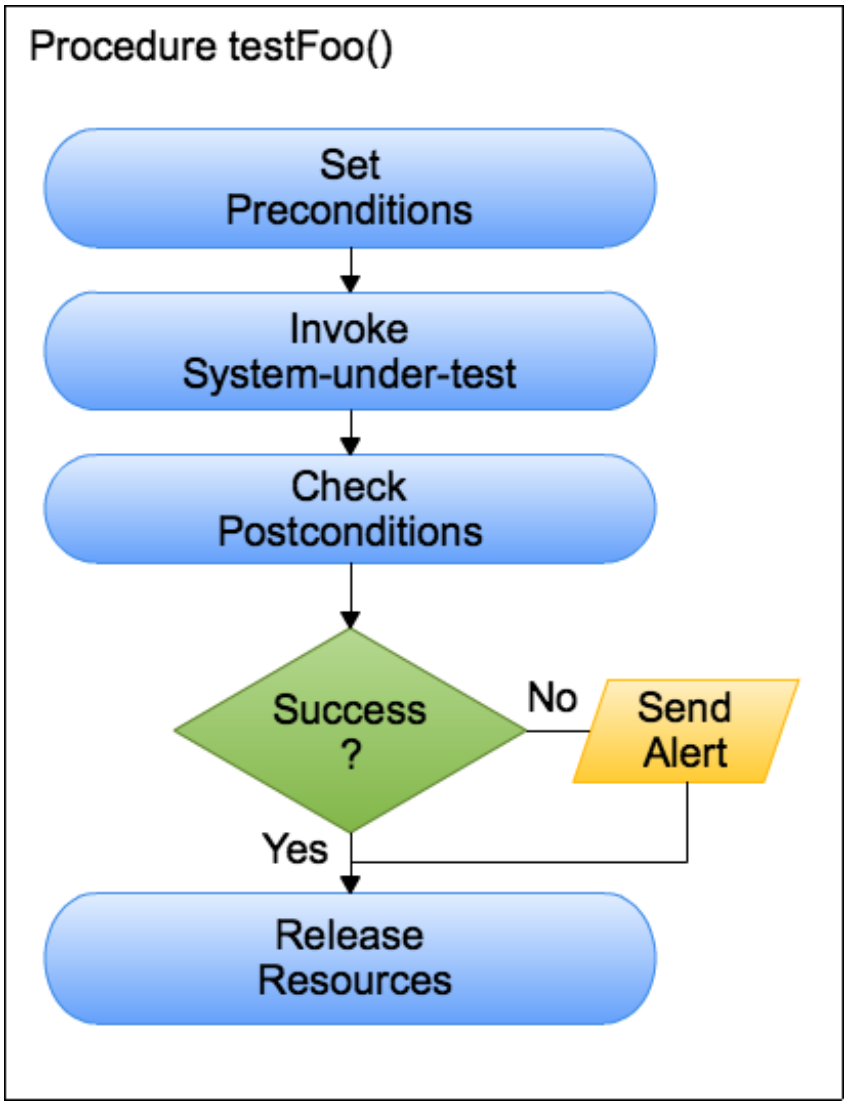
testTrajectory() ! $s = \frac{1}{2}at^2$

a = 2.; t = 3.

s = trajectory(a, t)

call **assertEqual** (9., s)

Anatomy of a Software Test Procedure



testTrajectory() ! $s = \frac{1}{2}at^2$

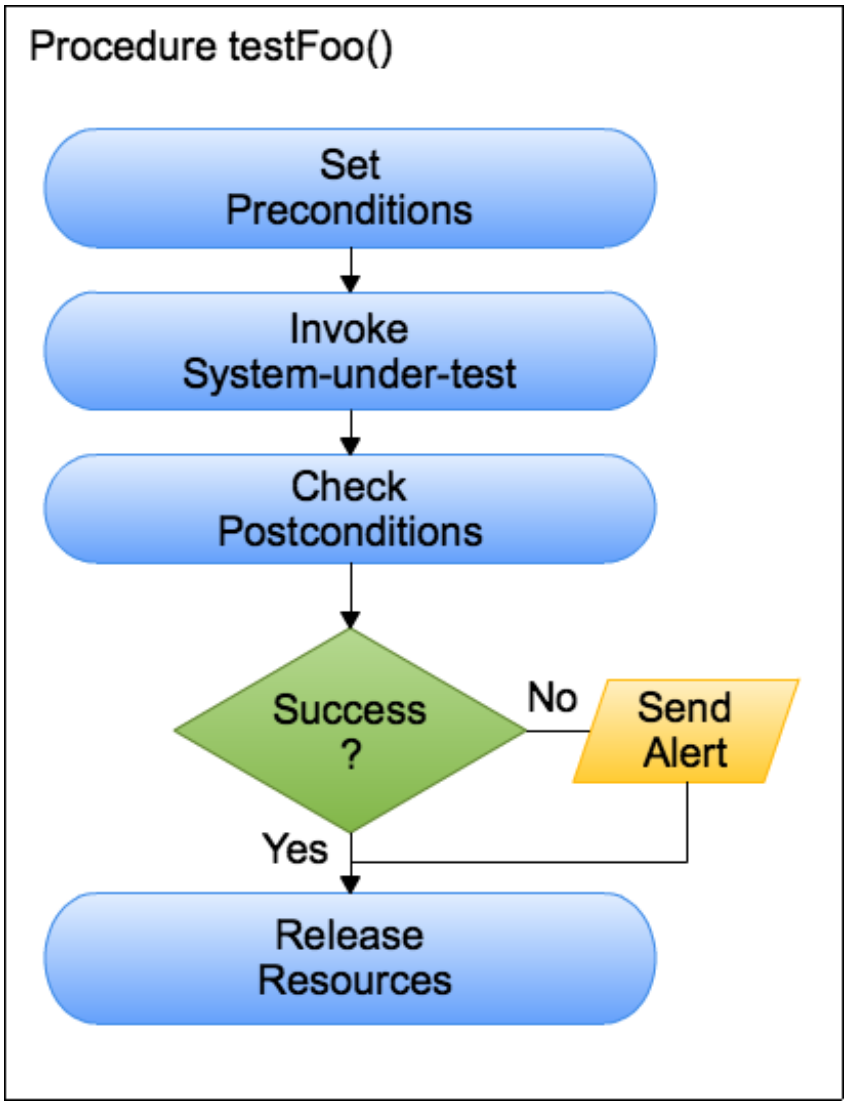
a = 2.; t = 3.

s = trajectory(a, t)

call **assertEqual** (9., s)

! no op

Anatomy of a Software Test Procedure



testTrajectory() ! $s = \frac{1}{2}at^2$

call **assertEqual** (9., trajectory (2.,3.))

Outline

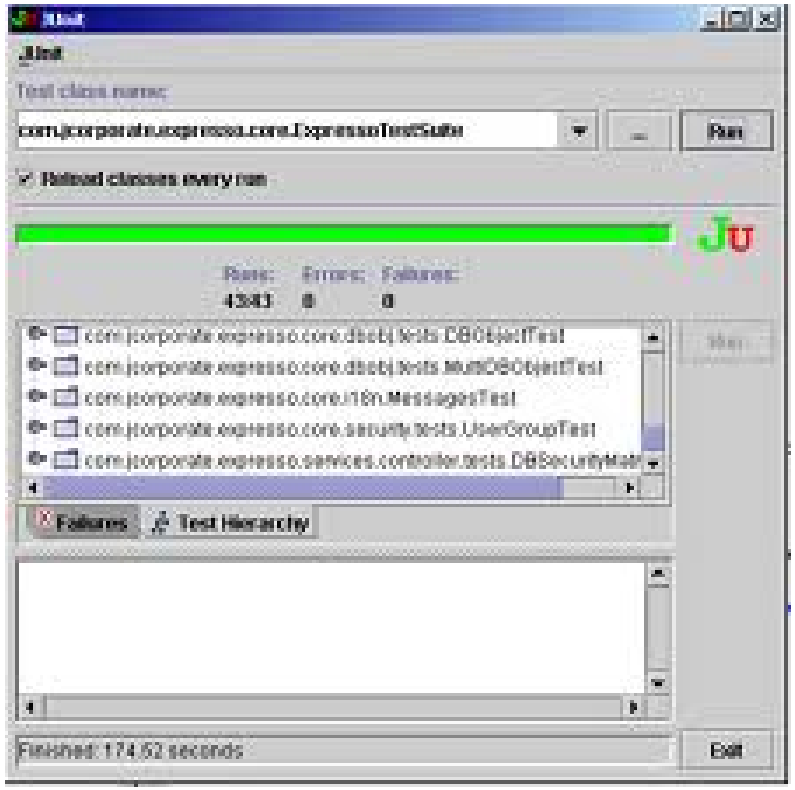
Testing Frameworks

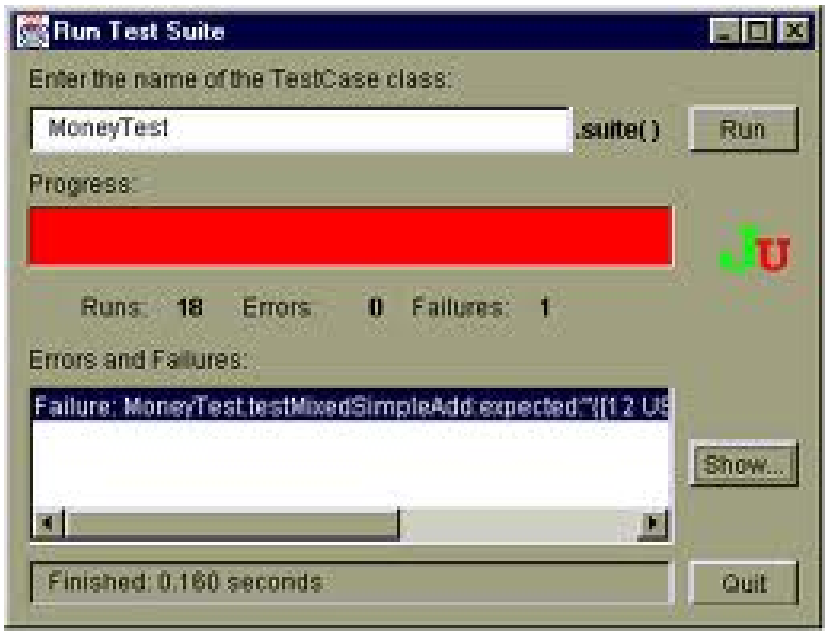
- Provide infrastructure to radically simplify:
  - Creating test routines (Test cases)
  - Running collections of tests (Test suites)
  - Summarizing results
- Key feature is collection of assert methods
  - Used to express expected results

```
call assertEqual(120, factorial(5))
```

- Generally specific to programming language (xUnit)
  - Java (JUnit)
  - Pnython (pyUnit)
  - C++ (cxxUnit, cppUnit)
  - Fortran (FRUIT, FUNIT, **pFUnit**)

GUI - JUnit in Eclipse





Outline

(Somewhat) New Paradigm: TDD

Old paradigm:

- Tests written by separate team (black box testing)
- Tests written *after* implementation

(Somewhat) New Paradigm: TDD

Old paradigm:

- Tests written by separate team (black box testing)
- Tests written *after* implementation

Consequences:

- Testing schedule compressed for release
- Defects detected late in development ($$)

(Somewhat) New Paradigm: TDD

Old paradigm:

- Tests written by separate team (black box testing)
- Tests written *after* implementation

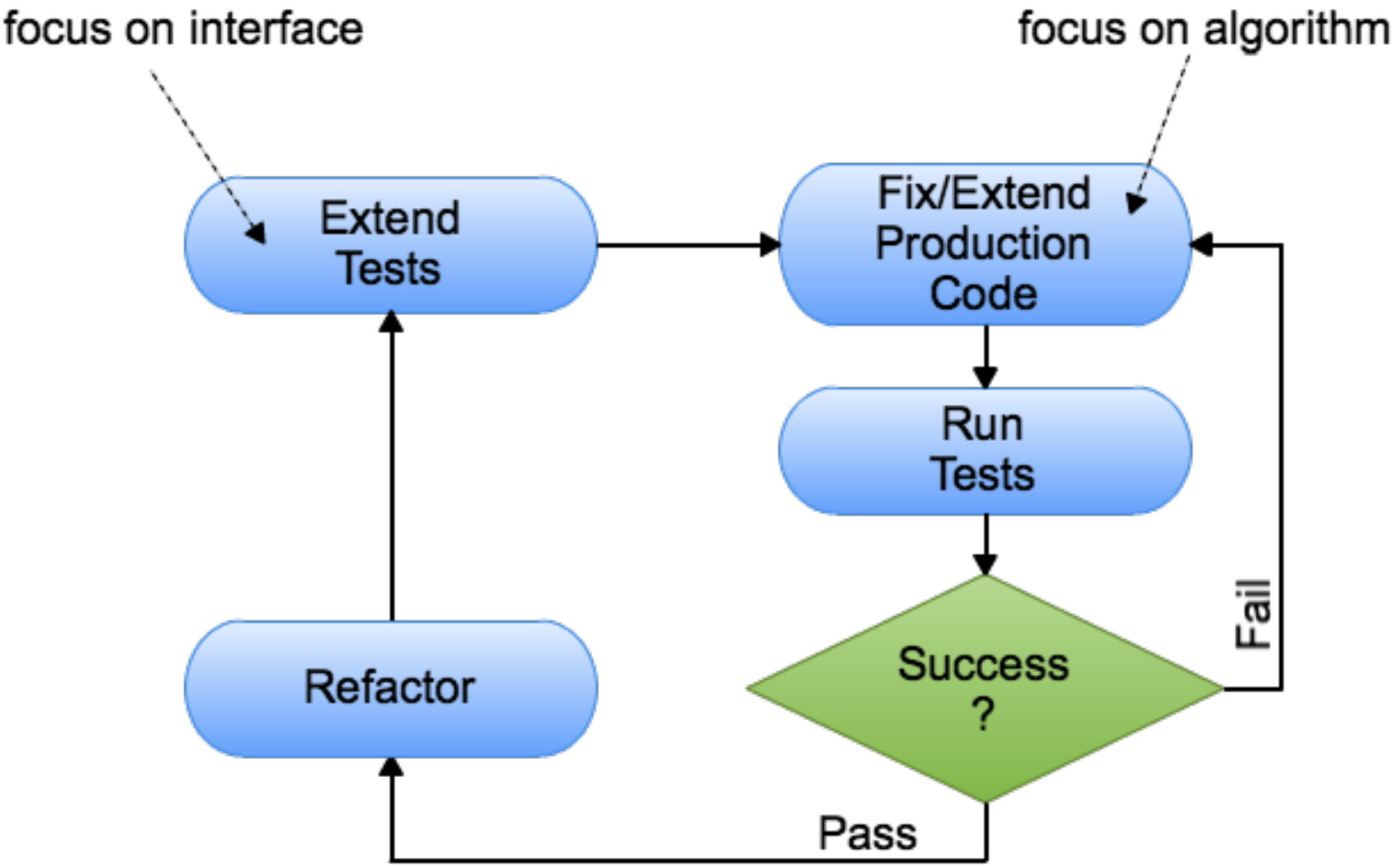
Consequences:

- Testing schedule compressed for release
- Defects detected late in development ($$)

New paradigm

- Developers write the tests (white box testing)
- Tests written before production code
- Enabled by emergence of strong unit testing frameworks

The TDD cycle



Benefits of TDD

Benefits of TDD

- High reliability

Benefits of TDD

- High reliability
- Excellent test coverage

Benefits of TDD

- High reliability
- Excellent test coverage
- Always “ready-to-ship”

Benefits of TDD

- High reliability
- Excellent test coverage
- Always “ready-to-ship”
- Tests act as *maintainable* documentation
  - Test shows real use case scenario
  - Test is maintained through TDD process

Benefits of TDD

- High reliability
- Excellent test coverage
- Always “ready-to-ship”
- Tests act as *maintainable* documentation
  - Test shows real use case scenario
  - Test is maintained through TDD process
- Less time spent debugging

Benefits of TDD

- High reliability
- Excellent test coverage
- Always “ready-to-ship”
- Tests act as *maintainable* documentation
  - Test shows real use case scenario
  - Test is maintained through TDD process
- Less time spent debugging
- Reduced stress / improved confidence

Benefits of TDD

- High reliability
- Excellent test coverage
- Always “ready-to-ship”
- Tests act as *maintainable* documentation
  - Test shows real use case scenario
  - Test is maintained through TDD process
- Less time spent debugging
- Reduced stress / improved confidence
- Productivity

Benefits of TDD

- High reliability
- Excellent test coverage
- Always "ready-to-ship"
- Tests act as *maintainable* documentation
  - Test shows real use case scenario
  - Test is maintained through TDD process
- Less time spent debugging
- Reduced stress / improved confidence
- Productivity
- Predictable schedule

Benefits of TDD

- High reliability
- Excellent test coverage
- Always "ready-to-ship"
- Tests act as *maintainable* documentation
  - Test shows real use case scenario
  - Test is maintained through TDD process
- Less time spent debugging
- Reduced stress / improved confidence
- Productivity
- Predictable schedule
- Porting

Benefits of TDD

- High reliability
- Excellent test coverage
- Always "ready-to-ship"
- Tests act as *maintainable* documentation
  - Test shows real use case scenario
  - Test is maintained through TDD process
- Less time spent debugging
- Reduced stress / improved confidence
- Productivity
- Predictable schedule
- Porting
- **Quality implementation?**

Outline

Unique challenges of numerical software

Unique challenges of numerical software

- Difficult to estimate error
  - Roundoff
  - Truncation

Unique challenges of numerical software

- Difficult to estimate error
  - Roundoff
  - Truncation
- Insufficient analytic cases

Unique challenges of numerical software

- Difficult to estimate error
  - Roundoff
  - Truncation
- Insufficient analytic cases
- Irreducible complexity
  - Test would require the same redundant logic

Unique challenges of numerical software

- Difficult to estimate error
  - Roundoff
  - Truncation
- Insufficient analytic cases
- Irreducible complexity
  - Test would require the same redundant logic
  - Appeals to vanity?

Unique challenges of numerical software

- Difficult to estimate error
  - Roundoff
  - Truncation
- Insufficient analytic cases
- Irreducible complexity
  - Test would require the same redundant logic
  - Appeals to vanity?
- Stability/Nonlinearity
  - Problems that occur only after long integrations
  - More generally - emergent properties of coupled systems

Unique challenges of numerical software

- Difficult to estimate error
  - Roundoff
  - Truncation
- Insufficient analytic cases
- Irreducible complexity
  - Test would require the same redundant logic
  - Appeals to vanity?
- Stability/Nonlinearity
  - Problems that occur only after long integrations
  - More generally - emergent properties of coupled systems

General mitigation strategy:

Unique challenges of numerical software

- Difficult to estimate error
  - Roundoff
  - Truncation
- Insufficient analytic cases
- Irreducible complexity
  - Test would require the same redundant logic
  - Appeals to vanity?
- Stability/Nonlinearity
  - Problems that occur only after long integrations
  - More generally - emergent properties of coupled systems

General mitigation strategy:

- Fine-grained implementation (each routine does just one thing)
- Test layers in isolation

Numerical Tolerance

For testing numerical results, a good estimate for the tolerance is necessary:

Numerical Tolerance

For testing numerical results, a good estimate for the tolerance is necessary:

- If too *low*, then test fails for uninteresnting reasons.

Numerical Tolerance

For testing numerical results, a good estimate for the tolerance is necessary:

- If too *low*, then test fails for uninteresnting reasons.
- If too *high*, then the test has no teeth.

Numerical Tolerance

For testing numerical results, a good estimate for the tolerance is necessary:

- If too *low*, then test fails for uninteresnting reasons.
- If too *high*, then the test has no teeth.

Unfortunately ...

- Error estimates are seldom available for complex algorithms

Numerical Tolerance

For testing numerical results, a good estimate for the tolerance is necessary:

- If too *low*, then test fails for uninteresnting reasons.
- If too *high*, then the test has no teeth.

Unfortunately ...

- Error estimates are seldom available for complex algorithms
- Best case - usually asymtotic form with unknown leading coefficient!

Numerical tolerance (cont'd)

Numerical tolerance (cont'd)

Sources of roundoff

Numerical tolerance (cont'd)

Sources of roundoff

1. Ordinary arithmetic - machine epsilon (not a concern)

Numerical tolerance (cont'd)

Sources of roundoff

1. Ordinary arithmetic - machine epsilon (not a concern)
2. Nonlinearity - esp. small denominators

Numerical tolerance (cont'd)

Sources of roundoff

1. Ordinary arithmetic - machine epsilon (not a concern)
2. Nonlinearity - esp. small denominators
3. *Composition and iteration*

Numerical tolerance (cont'd)

Sources of roundoff

1. Ordinary arithmetic - machine epsilon (not a concern)
2. Nonlinearity - esp. small denominators
3. *Composition and iteration*

Mitigation

Numerical tolerance (cont'd)

Sources of roundoff

1. Ordinary arithmetic - machine epsilon (not a concern)
2. Nonlinearity - esp. small denominators
3. *Composition and iteration*

Mitigation

- **Tailored synthetic inputs**:
  eliminate/minimize roundoff from nonlinearity

Numerical tolerance (cont'd)

Sources of roundoff

1. Ordinary arithmetic - machine epsilon (not a concern)
2. Nonlinearity - esp. small denominators
3. *Composition and iteration*

Mitigation

- **Tailored synthetic inputs**:
  eliminate/minimize roundoff from nonlinearity
- **Test layers in isolation**:
  circumvent growth from composition

Numerical tolerance (cont'd)

Sources of roundoff

1. Ordinary arithmetic - machine epsilon (not a concern)
2. Nonlinearity - esp. small denominators
3. *Composition and iteration*

Mitigation

- **Tailored synthetic inputs**:
  eliminate/minimize roundoff from nonlinearity
- **Test layers in isolation**:
  circumvent growth from composition
- **Put iteration logic in separate layer**:
  circumvent growth from iteration

Numerical tolerance (cont'd)

Sources of roundoff

1. Ordinary arithmetic - machine epsilon (not a concern)
2. Nonlinearity - esp. small denominators
3. *Composition and iteration*

Mitigation

- **Tailored synthetic inputs**:
  eliminate/minimize roundoff from nonlinearity
- **Test layers in isolation**:
  circumvent growth from composition
- **Put iteration logic in separate layer**:
  circumvent growth from iteration

Conclusion: Decomposition and synthetic inputs yield testing tolerances that are of the same order as machine epsilon.

Test layers in isolation

Example: Procedure that does too much

```
...
a = <complex expression>
b = <complex expression>
c = <complex expression>
return a + sqrt(b/c)
```

Test layers in isolation

Example: Procedure that does too much

```
...
a = <complex expression>
b = <complex expression>
c = <complex expression>
return a + sqrt(b/c)
```

Same capability, but split into two decoupled levels

```
...
a = f1(...)
b = f2(...)
c = f3(...)
return g(a, b, c)
```

Test layers in isolation

Example: Procedure that does too much

```
...
a = <complex expression>
b = <complex expression>
c = <complex expression>
return a + sqrt(b/c)
```

Same capability, but split into two decoupled levels

```
...
a = f1(...)
b = f2(...)
c = f3(...)
return g(a, b, c)
```

Higher level test ensures proper coupling, but not fully expanded arithmetic.

Test layers in isolation (cont'd)

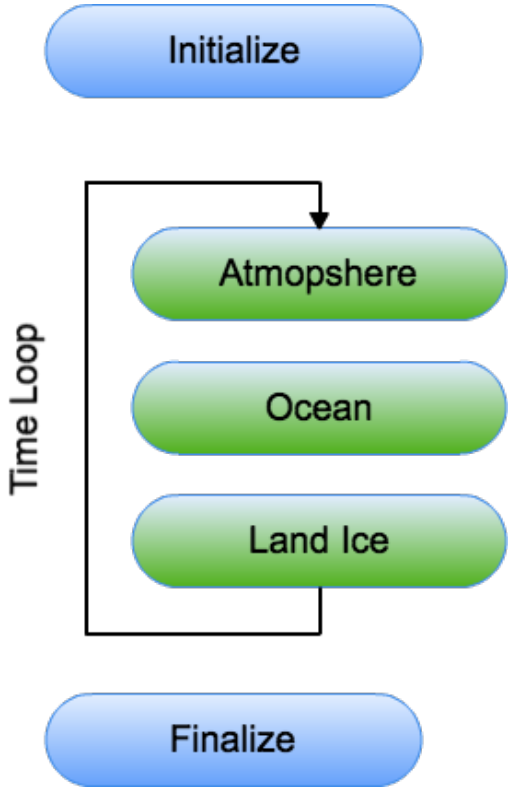
Consider the main loop of a climate model:

Do test

- Proper # of iterations
- Pieces called in correct order
- Passing of data between components

Do NOT test

- Calculations inside components



Much easier to do in practice with *objects* than with procedures.

TDD and lack of analytic results

- Complex algorithms often yield few if any analytic solutions

TDD and lack of analytic results

- Complex algorithms often yield few if any analytic solutions
- And yet we attempt software implementations. How can this be?

TDD and lack of analytic results

- Complex algorithms often yield few if any analytic solutions
- And yet we attempt software implementations. How can this be?
- Difficulty generally arises from composition and iteration

TDD and lack of analytic results

- Complex algorithms often yield few if any analytic solutions
- And yet we attempt software implementations. How can this be?
- Difficulty generally arises from composition and iteration
- Mitigation:
  - Test algorithmic steps in isolation
  - Tailor synthetic inputs to yield "obvious" results for each step
  - Use integration tests to verify that steps are composed correctly

TDD and lack of analytic results

- Complex algorithms often yield few if any analytic solutions
- And yet we attempt software implementations. How can this be?
- Difficulty generally arises from composition and iteration
- Mitigation:
  - Test algorithmic steps in isolation
  - Tailor synthetic inputs to yield "obvious" results for each step
  - Use integration tests to verify that steps are composed correctly
- *But still use high level analytic solutions as tests whenever possible*

Consider Newton's three-body problem - no analytic solution

TDD and lack of analytic results

- Complex algorithms often yield few if any analytic solutions
- And yet we attempt software implementations. How can this be?
- Difficulty generally arises from composition and iteration
- Mitigation:
  - Test algorithmic steps in isolation
  - Tailor synthetic inputs to yield "obvious" results for each step
  - Use integration tests to verify that steps are composed correctly
- *But still use high level analytic solutions as tests whenever possible*

Consider Newton's three-body problem - no analytic solution

- Test generation of pairwise forces
- Test time integration (e.g., RK4)
- Use special cases that have solutions as additional tests

Irreducible complexity

"Aren't my tests as complex as the implementation?"
"Aren't my tests doing redundant calculations (tautological)?"

Irreducible complexity

"Aren't my tests as complex as the implementation?"
"Aren't my tests doing redundant calculations (tautological)?"

- Short answer: **No**

Irreducible complexity

"Aren't my tests as complex as the implementation?"
"Aren't my tests doing redundant calculations (tautological)?"

- Short answer: **No**
- Long answer: Well, they shouldn't be ...

Irreducible complexity

"Aren't my tests as complex as the implementation?"
"Aren't my tests doing redundant calculations (tautological)?"

- Short answer: **No**
- Long answer: Well, they shouldn't be ...
  - Unit tests use tailored inputs - implementation handles generic case

Irreducible complexity

"Aren't my tests as complex as the implementation?"
"Aren't my tests doing redundant calculations (tautological)?"

- Short answer: **No**
- Long answer: Well, they shouldn't be ...
  - Unit tests use tailored inputs - implementation handles generic case
  - Model layers are tested in *isolation*

Irreducible complexity

"Aren't my tests as complex as the implementation?"
"Aren't my tests doing redundant calculations (tautological)?"

- Short answer: **No**
- Long answer: Well, they shouldn't be ...
  - Unit tests use tailored inputs - implementation handles generic case
  - Model layers are tested in *isolation*
  - Tests are *decoupled* - low complexity

Irreducible complexity

"Aren't my tests as complex as the implementation?"
"Aren't my tests doing redundant calculations (tautological)?"

- Short answer: **No**
- Long answer: Well, they shouldn't be ...
  - Unit tests use tailored inputs - implementation handles generic case
  - Model layers are tested in *isolation*
  - Tests are *decoupled* - low complexity
  - Actual model *couples* layers - huge complexity

Long integration and emergent properties

- TDD generally does not directly address such issues

Long integration and emergent properties

- TDD generally does not directly address such issues
- If long integration gets incorrect results, one of the following holds:

Long integration and emergent properties

- TDD generally does not directly address such issues
- If long integration gets incorrect results, one of the following holds:
  1. Individual steps have defects - add tests

Long integration and emergent properties

- TDD generally does not directly address such issues
- If long integration gets incorrect results, one of the following holds:
  1. Individual steps have defects - add tests
  2. Integration has a defect - add tests

Long integration and emergent properties

- TDD generally does not directly address such issues
- If long integration gets incorrect results, one of the following holds:
  1. Individual steps have defects - add tests
  2. Integration has a defect - add tests
  3. Component steps lack necessary accuracy - need tests and improved algorithm

Long integration and emergent properties

- TDD generally does not directly address such issues
- If long integration gets incorrect results, one of the following holds:
  1. Individual steps have defects - add tests
  2. Integration has a defect - add tests
  3. Component steps lack necessary accuracy - need tests and improved algorithm
  4. Insufficient physical fidelity - genuine science challenge

Long integration and emergent properties

- TDD generally does not directly address such issues
- If long integration gets incorrect results, one of the following holds:
  1. Individual steps have defects - add tests
  2. Integration has a defect - add tests
  3. Component steps lack necessary accuracy - need tests and improved algorithm
  4. Insufficient physical fidelity - genuine science challenge

- At the very least, TDD can reduce the frequency at which long integrations are needed/performed

TDD and performance

- TDD emphasizes small fine-grained implementations
- Such implementations are often sub-optimal in terms of performance
- Optimized implementations typically fuse multiple operations

TDD and performance

- TDD emphasizes small fine-grained implementations
- Such implementations are often sub-optimal in terms of performance
- Optimized implementations typically fuse multiple operations
- Solution: bootstrapping
  - Use initial TDD solution as unit test for optimized implementation
  - Maintain *both* implementations

TDD and the legacy burden

- TDD was created for developing *new* code, and does not directly speak to maintaining legacy code.
- Adding new functionality
  - Avoid *wedging* new loging directly into existing large procedure
  - Use TDD to develop separate facility for new computation
  - Just *call* the new procedure from the large legacy procedure
- Refactoring
  - Use unit tests to constrain existing behavior
  - Very difficult for large procedures
  - Try to find small pieces to pull out into new procedures

References

- pFUnit: http://sourceforge.net/projects/pfunit/
- Tutorial materials
  - https://modelingguru.nasa.gov/docs/DOC-1982
  - https://modelingguru.nasa.gov/docs/DOC-1983
  - https://modelingguru.nasa.gov/docs/DOC-1984
- TDD Blog
  https://modelingguru.nasa.gov/blogs/modelingwithtdd
- *Test-Driven Development: By Example* - Kent Beck
- MIler and Padberg,"About the Return on Investment of Test-Driven Development," http://www.ipd.uka.de/mitarbeiter/muellerm/publications/edser03.pdf
- *Refactoring: Improving the Design of Existing Code* - Martin Fowler
- JUnit http://junit.sourceforge.net/