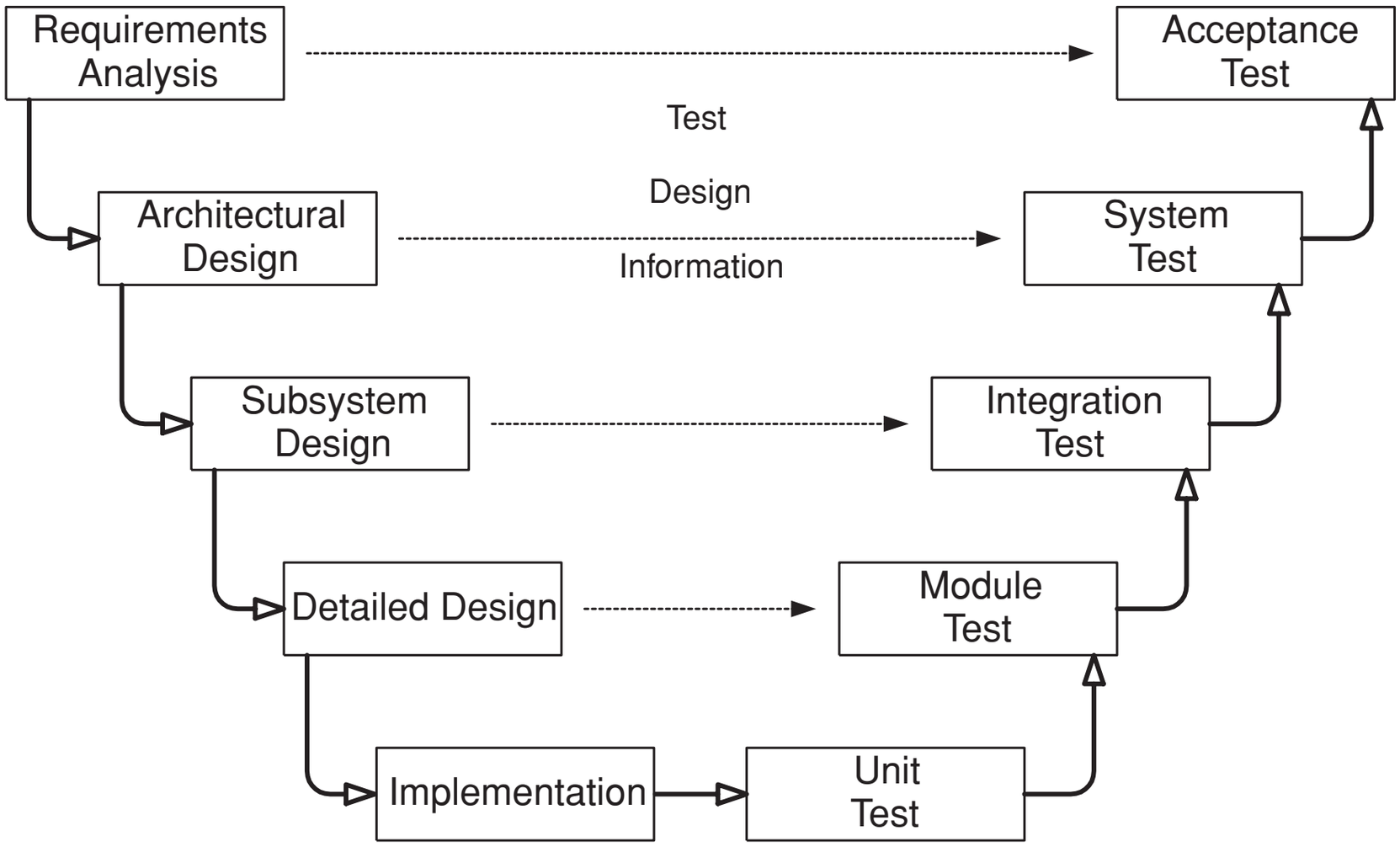
# Software Development Project

Morgan Ericsson
<morgan.ericsson@chalmers.se>

# When / What Do We Test?



# Pass or Fail?

- Test-to-Pass, tests if the program works
  - simple test cases
    - that should work
- Test-to-Fail, tries to break the program
  - evil test cases
  - that might not work

# Initial Example

- Black box testing
  - I had the source, but did not consider it
  - First 60 cases, test to pass
  - Next 20,000 cases, test to fail (approx).
  - Why start with test to pass?

# A Good Approach?

- 60 + 20,000 randomly selected Java programs
  - so, in a sense, double black box
- Reasonable?
  - quite slow (1 test / second = 5+ hours)
  - all 27 problems were "identical" ("so 1 told me something, 26 did not")

# A Better Approach?

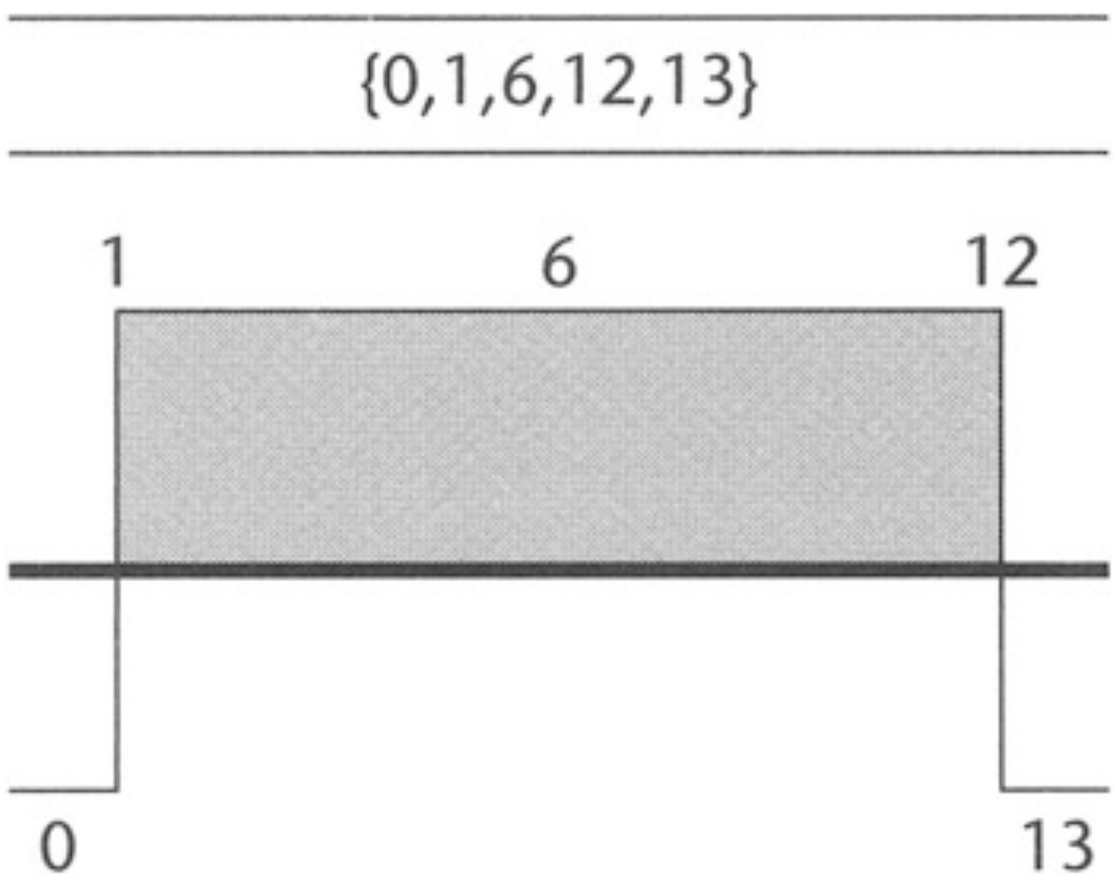
- Consider the program a function
  - with a domain and a range
  - Partition the domain into blocks with similar properties
    - test a block
    - test between blocks (edges)

# Example

| $P_1$: x<y | $P_2$: x=y | $P_3$: x>y |
|---|---|---|
| <(1,2), 2> | <(1,1),1> | <(3,2),3> |
| <(3,9), 9> | <(3,3),3> | <(3,2),3> |
| ... | ... | ... |

Testing a function that determines the larger of two integers

# Example

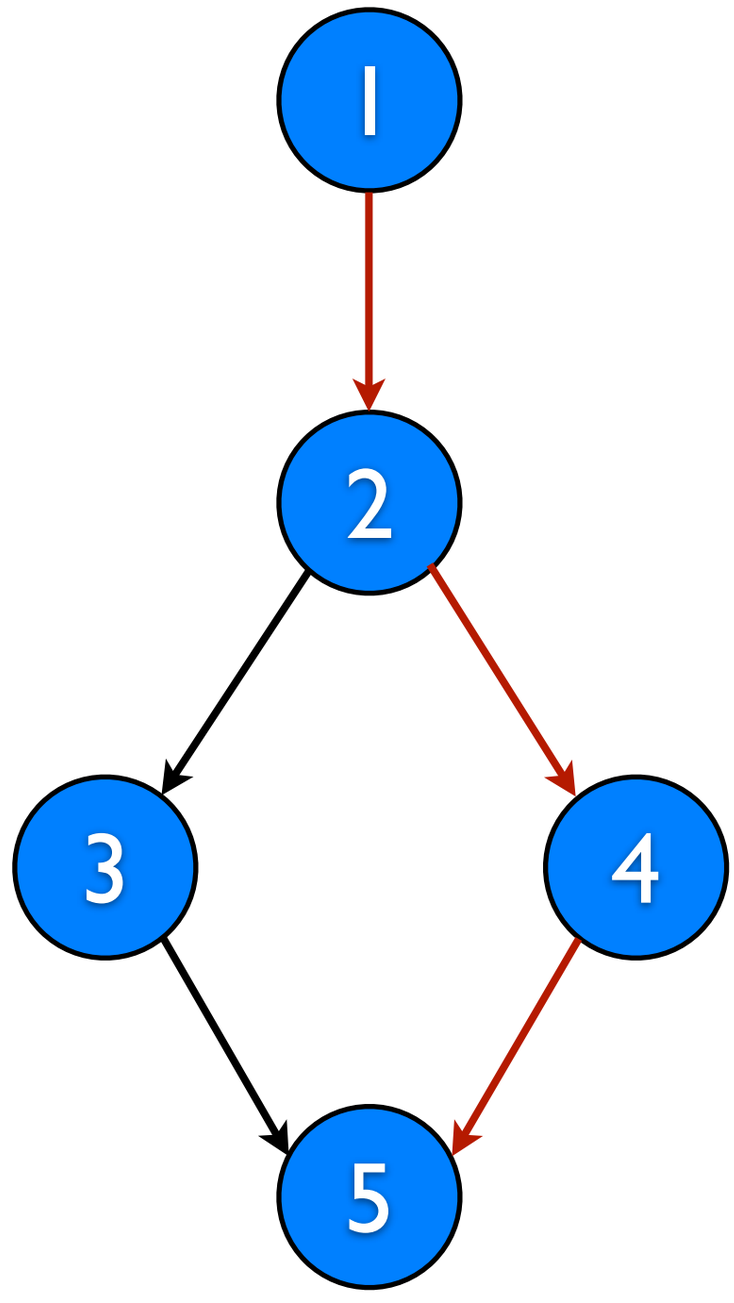


# White Box

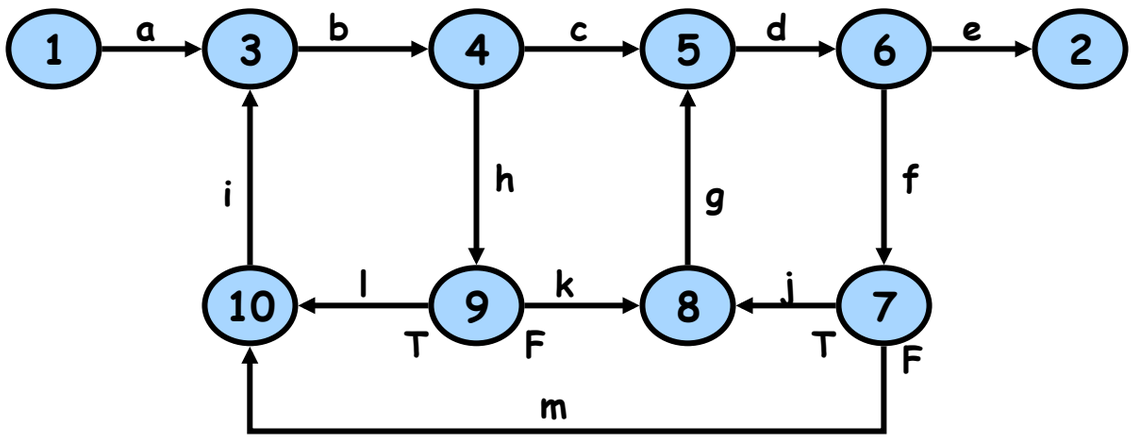
- Find paths through the program ...
- ... and make sure you cover these
- Statement coverage
- Branch coverage

# Example

```
   int max(int x, int y)
   {
1:    int max;
2:    if(x>y)
3:       max=x;
      else
4:       max=y
5:    return max;
   }
```



# Example



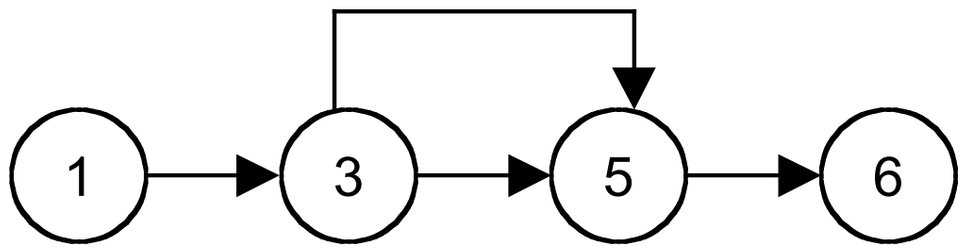

| PATHS | DECISIONS | | | | PROCESS LINKS | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 4 | 6 | 7 | 9 | *a* | *b* | *c* | *d* | *e* | *f* | *g* | *h* | *i* | *j* | *k* | *l* | *m* |
| *abcde* | *T* | *T* | | | * | * | * | * | * | | | | | | | | |
| *abhkgde* | *F* | *T* | | *F* | * | * | | * | * | | * | * | | | * | | |
| *abhlibcde* | *TF* | *T* | | *T* | * | * | * | * | * | | * | * | * | | | * | |
| *abcdfjgde* | *T* | *TF* | *T* | | * | * | * | * | * | * | * | | | * | | | |
| *abcdfmibcde* | *T* | *TF* | *F* | | * | * | * | * | * | * | | * | * | | | | * |

# Sunt förnuft

*"The more we learn about testing, the more we realize that statement and branch coverage are minimum floors below which we dare not fall, rather that ceilings to which we should aspire."* (B. Beizer)

# Example: ABS

```
/* ABS
This program function returns the absolute value of
the integer passed to the function as a parameter.
    INPUT: An integer.
    OUTPUT: The absolute value if the input integer.
*/
1   int ABS(int x)
2   {
3     if(x < 0)
4       x = -x;
5     return x;
6   }
```

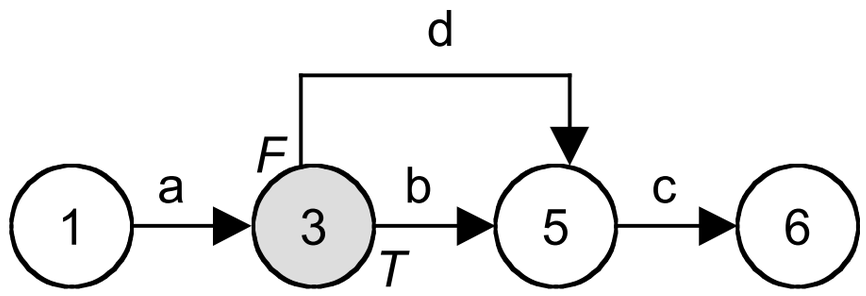


# Example: ABS

- Possible to completely test ABS
- Since there is a maximum integer
- However, not practical since since we often use either 32-bit or 64-bit integers
  - so, best case, 2^32 test cases

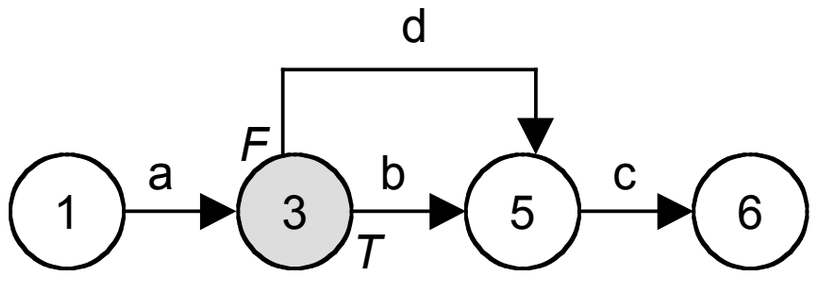
# Statement Coverage

| PATHS | PROCESS LINKS | | | | TEST CASES | |
|---|---|---|---|---|---|---|
| | a | b | c | d | INPUT | OUTPUT |
| abc | √ | √ | √ | | A Negative Integer, x | -x |
| adc | √ | | √ | √ | A Positive Integer, x | x |



# Branch Coverage

| PATHS | DECISIONS | TEST CASES | |
|---|---|---|---|
| | | INPUT | OUTPUT |
| abc | T | A Negative Integer, x | -x |
| adc | F | A Positive Integer, x | x |



# Unit Testing

*“If code has no automated test cases to show it works, then the assumption must be that it doesn't”*

# Unit Testing

- Parts of the source code (s.k. units) are tested
- A unit is the smallest testable entity
  - Often procedure / method
- White box
- Often written by the developer (TDD)
- Often automated

# Benefits

- Makes it easier to change the code
  - test checks so everything still works
- Supports integration testing
  - shows that each unit works
  - (but not the same thing)

# Benfits (more later)

- Living documentation
  - Shows how the API/methods should be used
  - Illustrates cases that are important
- Design
  - If (and they should) the test cases are written before implementation, they can guide the implementation

# Form

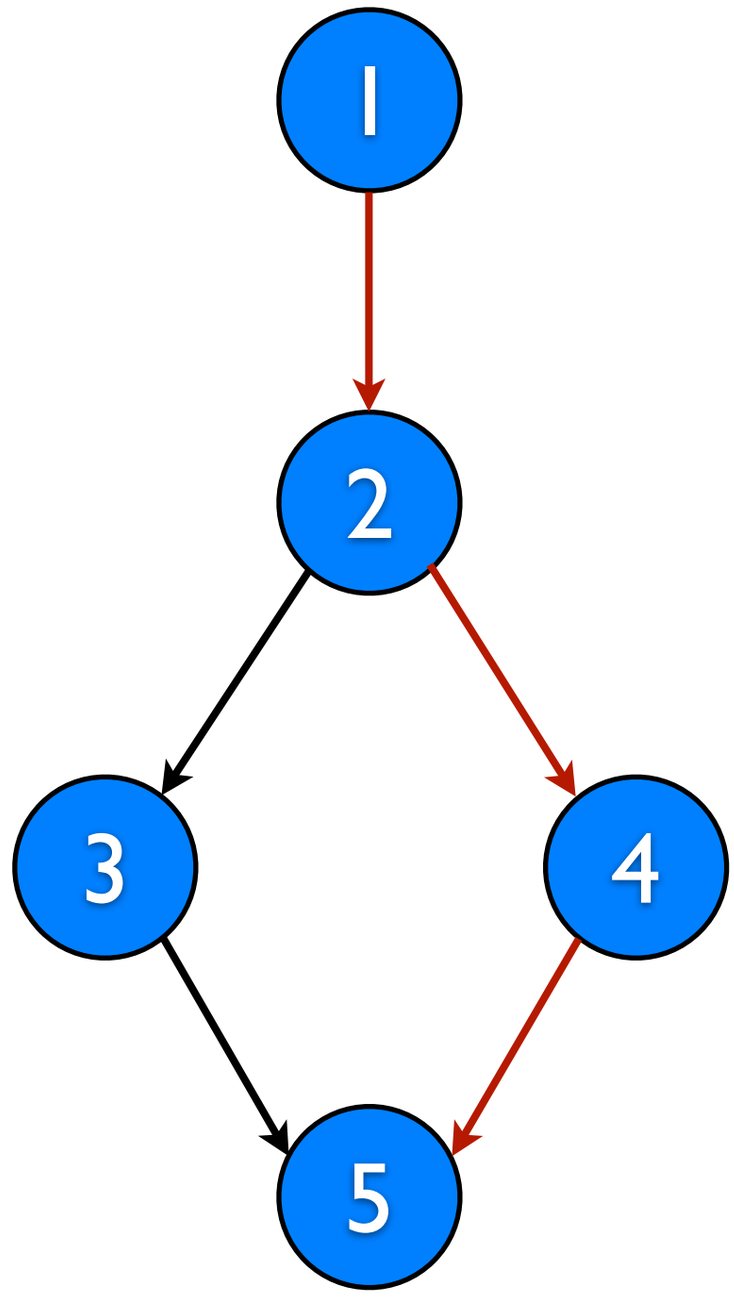
- Unit tests call or executes part of the source code
- The result is validated by asserting facts
  - e.g. assertNotEqual(“apple”,“orange”)

# Example

```
public void testAbs10() {
    assertEquals(10.0, Operations.myAbs(10.0));
}
public void testAbsn10() {
    assertEquals(10.0, Operations.myAbs(-10.0));
}
public void testSqrt25() {
    assertEquals(Math.round(5.0),
        Math.round(Operations.mySqrt(25.0)));
}
```

# Example

```
   int max(int x, int y)
   {
1:    int max;
2:    if(x>y)
3:       max=x;
      else
4:       max=y
5:    return max;
   }
```



# Acceptance Testing (XP)

- Tests a user story
  - from a "customer" perspective
  - The customer can accept or reject
  - Testing should be at least modeled by the customer
  - AKA functional testing or customer testing

# Documentation

*"Working software over comprehensive documentation."*

# Document What?

1. Requirements - Statements that identify attributes, capabilities, characteristics, or qualities of a system. This is the foundation for what shall be or has been implemented.

2. Architecture/Design - Overview of software. Includes relations to an environment and construction principles to be used in design of software components.

3. Technical - Documentation of code, algorithms, interfaces, and APIs.

4. End user - Manuals for the end-user, system administrators and support staff.

(according to wikipedia...)

# Documentation

*"Working software over comprehensive documentation."*

# Issues

- Software development versus documentation development
- Software developers have the knowledge, technical writers have the skill
- Project-level versus enterprise-level documentation
- Quantity versus quality
- ...

# Why?

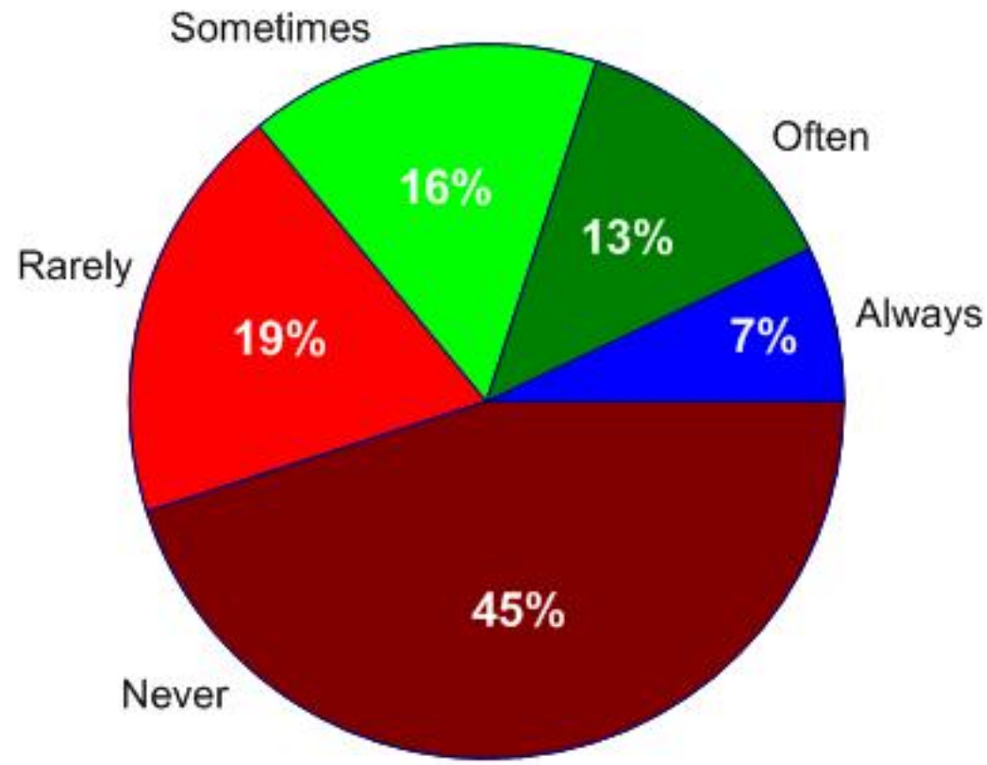
- Your project stakeholders require it
- To define a contract model
- To support communication with an external group
- To support organizational memory
- For audit purposes
- To think something through

# Why? (Bad Reasons)

- The requester wants to be seen to be in control
- The requester mistakenly thinks that documentation has something to do with project success (*)
- The requester wants to justify their existence
- The requester doesn't know any better
- Your process says to create the document
- Someone wants reassurance that everything is okay

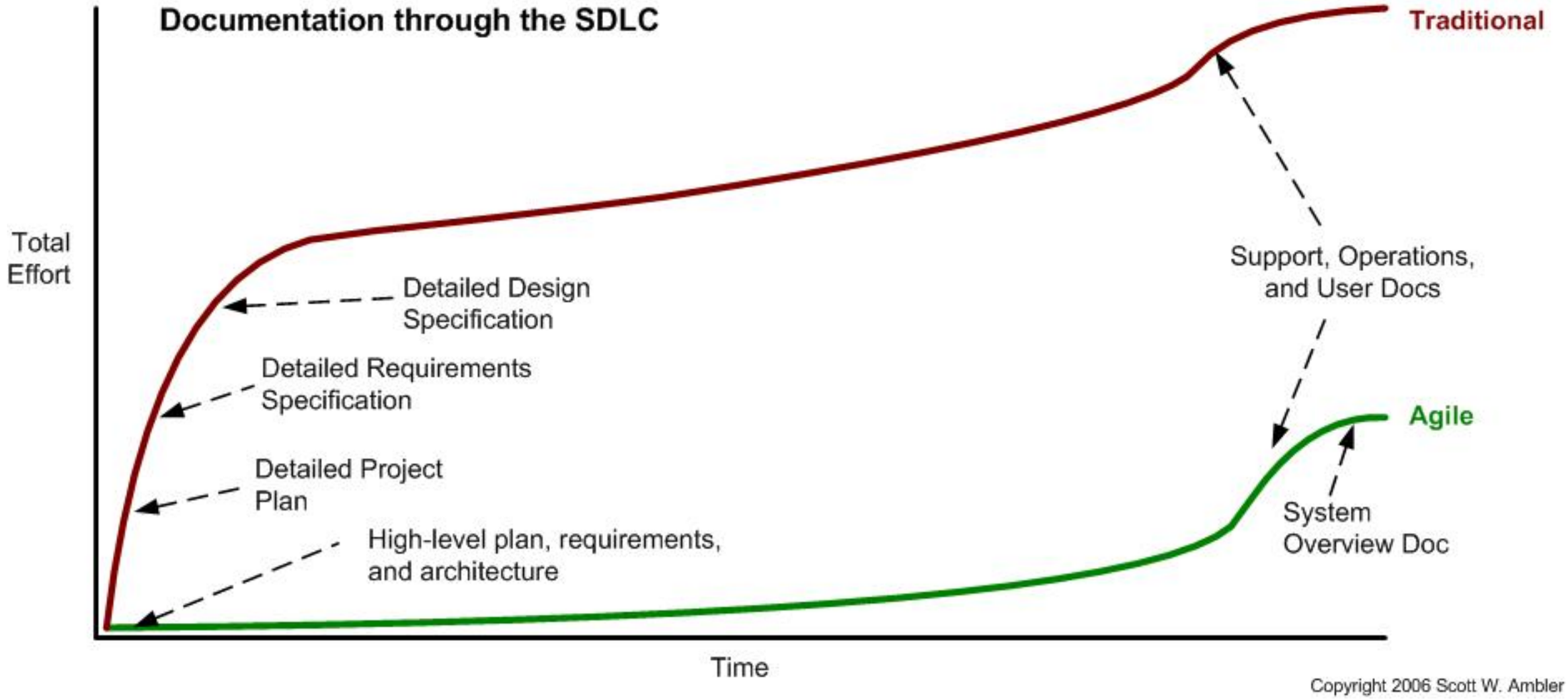
# Really? (*)

Average percentage of delivered functionality actually used when a serial approach to requirements elicitation and documentation is taken on a "successful" information technology project.




Source: Chaos Report v3, Standish Group.

# Really? (*)


Documentation through the SDLC
Traditional
Total Effort
Detailed Design Specification
Detailed Requirements Specification
Detailed Project Plan
High-level plan, requirements, and architecture
Support, Operations, and User Docs
Agile
System Overview Doc
Time
Copyright 2006 Scott W. Ambler

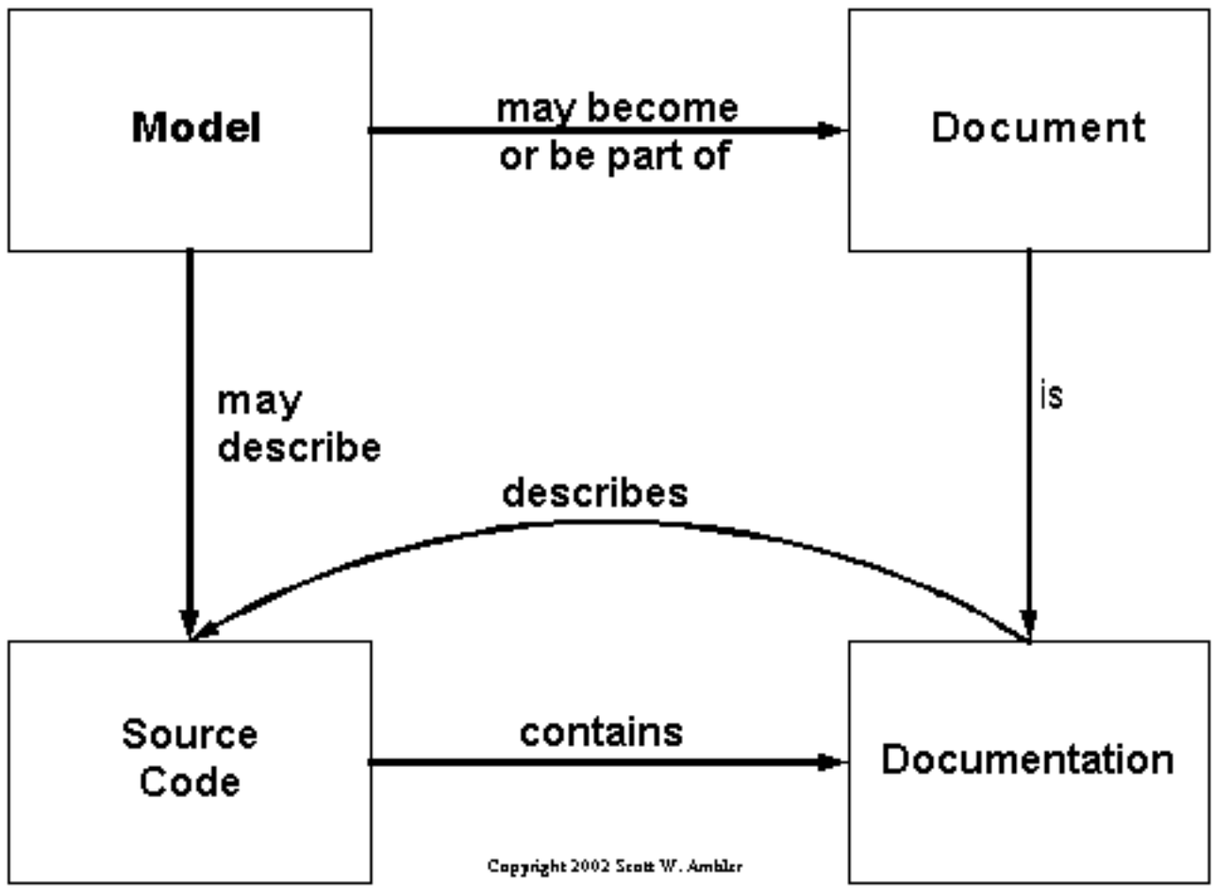
# But...

- You should understand the total cost of ownership (TCO) for a document, and someone must explicitly choose to make that investment.
- The fundamental issue is communication, not documentation.

# So...

*"Agilists write documentation if that's the best way to achieve the relevant goals, but there often proves to be better ways to achieve those goals than writing static documentation."*

# Relationships



# So...

- Prefer executable work products such as customer tests and developer tests over static work products such as plain old documentation (POD)
- With high quality source code and a test suite to back it up you need a lot less system documentation.

# So...

- Models are not necessarily documents, and documents are not necessarily models
- Developers rarely trust the documentation, particularly detailed documentation because it's usually out of sync with the code

# When/What

- Ask whether you NEED the documentation, not whether you want it
- Document stable things, not speculative things
- Create documentation only when you need it at the appropriate point in the life cycle

# When/What

- Design decisions
- Vision Statement
- Operations documentation
- Project overview
- Requirements document
- Support documentation
- System Documentation
- User documentation

# CRUFT

- Badness = 100% - CRUFT
  - C = The percentage of the document that is currently “correct".
  - R = The chance that the document will be read by the intended audience.
  - U = The percentage of the document that is actually understood by the intended audience.
  - F = The chance that the material contained in document will be followed.
  - T = The chance that the document will be trusted.

# When/What

- Vision Statement (yes)
- Project overview (yes)