Software Quality & Reliability

Letha Etzkorn, Ph.D., P.E.
Computer Science Department
University of Alabama in Huntsville

Dependability and Reliability Definitions

Dependability:

- the ability of a system to deliver the intended level of service to its users—Laprie (1985)
- those system properties that allow us to rely on a system functioning as required—Littlewood and Strigini (2000)
    - Littlewood and Strigini (2000) say that dependability includes reliability, safety, security, and availability among other attributes

Dependability and Reliability Definitions

Reliability:

- the probability of failure-free software operation for a specified period of time in a specified environment—Pan (1999)
- continuity of service—Laprie and Kanoun (1996)

Dependability

How to achieve dependability of software:

- Fault avoidance (process oriented)
  - don’t introduce bugs (faults, defects) into the software in the first place
- Fault tolerance (product oriented)
  - fulfil the system’s function even though faults occur
  - accept that some faults will occur and hide the associated failures

Reliability

How to achieve reliability of software:

- Fault forecasting
  - How to estimate the current and future quantities of faults and their consequences—Laprie and Kanoun (1996)
- Fault removal
  - How to reduce the number of faults and the seriousness of faults—Laprie and Kanoun (1996)

Software vs. Hardware Reliability

Myers (1976)

- Software does not wear out
- Software reliability is due to design errors only, whereas hardware reliability is subject to design errors, manufacturing errors, and errors due to wear and tear

Littlewood and Strigini (2000)

- Software unreliability is always the result of design faults which arise from human intellectual failures
- Hardware unreliability has often resulted from the "perversity of nature"

Software vs. Hardware Reliability

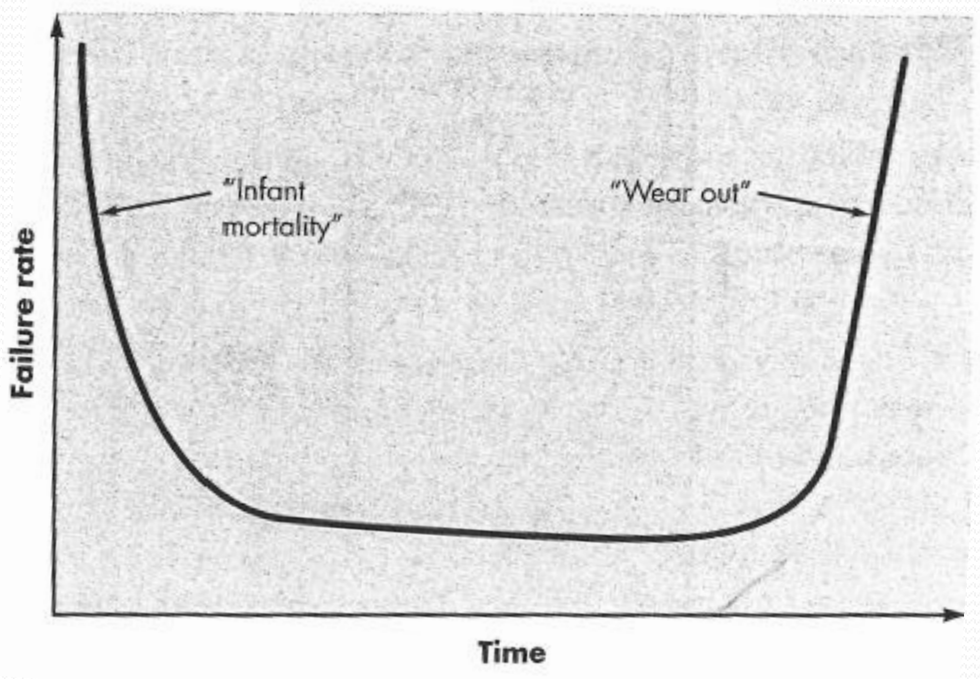


Figure taken from Pressman and Maxim (2015)

Software vs. Hardware Reliability

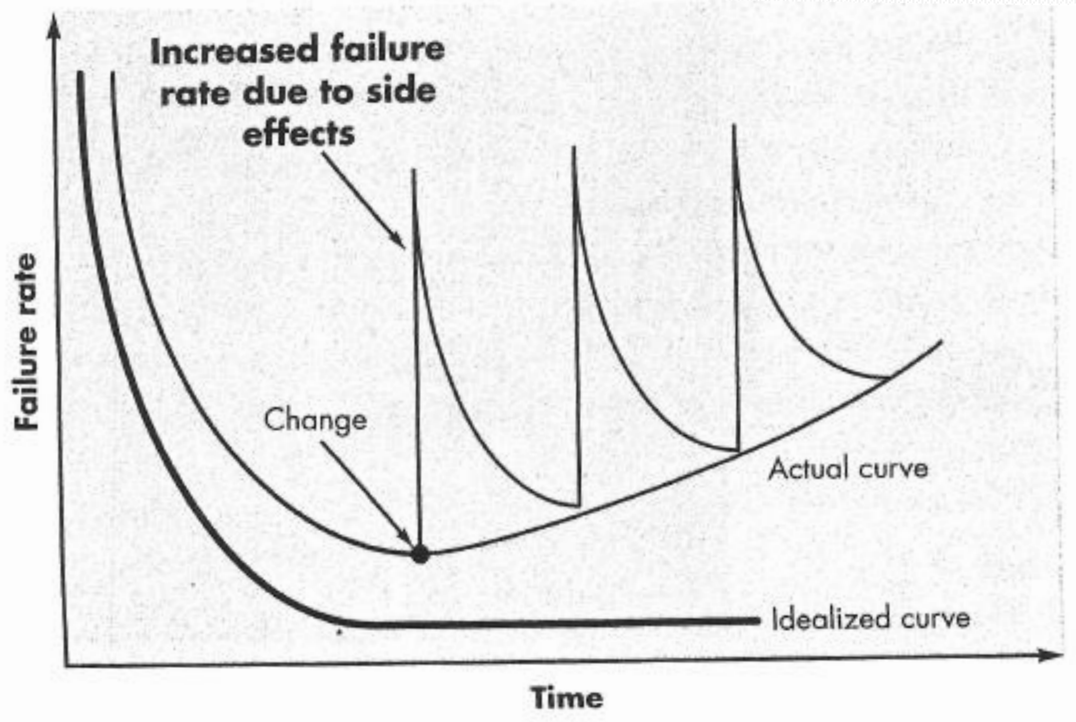


Figure taken from Pressman and Maxim (2015)

Software vs. Hardware Reliability

D.L. Parnas (1985) discusses the reasons for why software is more unreliable than hardware:

"Software systems are discrete state systems that do not have the repetitive structure shown in computer circuitry. There is seldom a reason to construct software as highly repetitive structures. The number of states in software systems is orders of magnitude larger than the number of states in the non-repetitive parts of computers. The mathematical functions that describe the behavior of these systems are not continuous functions and traditional engineering mathematics does not help in their verification. This difference clearly contributes to the relative unreliability of computer systems and the apparent lack of competence of software engineers. It is a fundamental difference that will not disappear with improved technology."

Software and Hardware Reliability Taken Together

However, one must be cautious treating hardware and software reliability as independent –Bendell and Mellor (1986):

- Faults may result from interactions between hardware and software
- Since the way that hardware reliability (of a repairable system) varies over time is different from the way that software varies over time, combining the two can be difficult mathematically

Why Software Reliability Becomes More Difficult Over Time

Littlewood and Strigini (2000) discuss various reasons why software reliability tends to become more difficult over time:

- The problems being addressed by the software have become more difficult and more novel
  - In the early days of computing, software was used to automate existing successful manual solutions
    - but today problems that were never previously solved are being addressed with software
  - Since software is not subject to typical hardware constraints, it is possible to address problems that were too complex to address using hardware alone

Why Software Reliability Becomes More Difficult Over Time

- Software solutions have necessarily become more complex to address the more complicated problems
- There is a business need for short time periods to address these solutions
- With software, unlike with hardware, it is usually impossible to assume that since the software worked well in one context, it will also perform acceptably in a similar (but different) context

Software Reliability End of Life

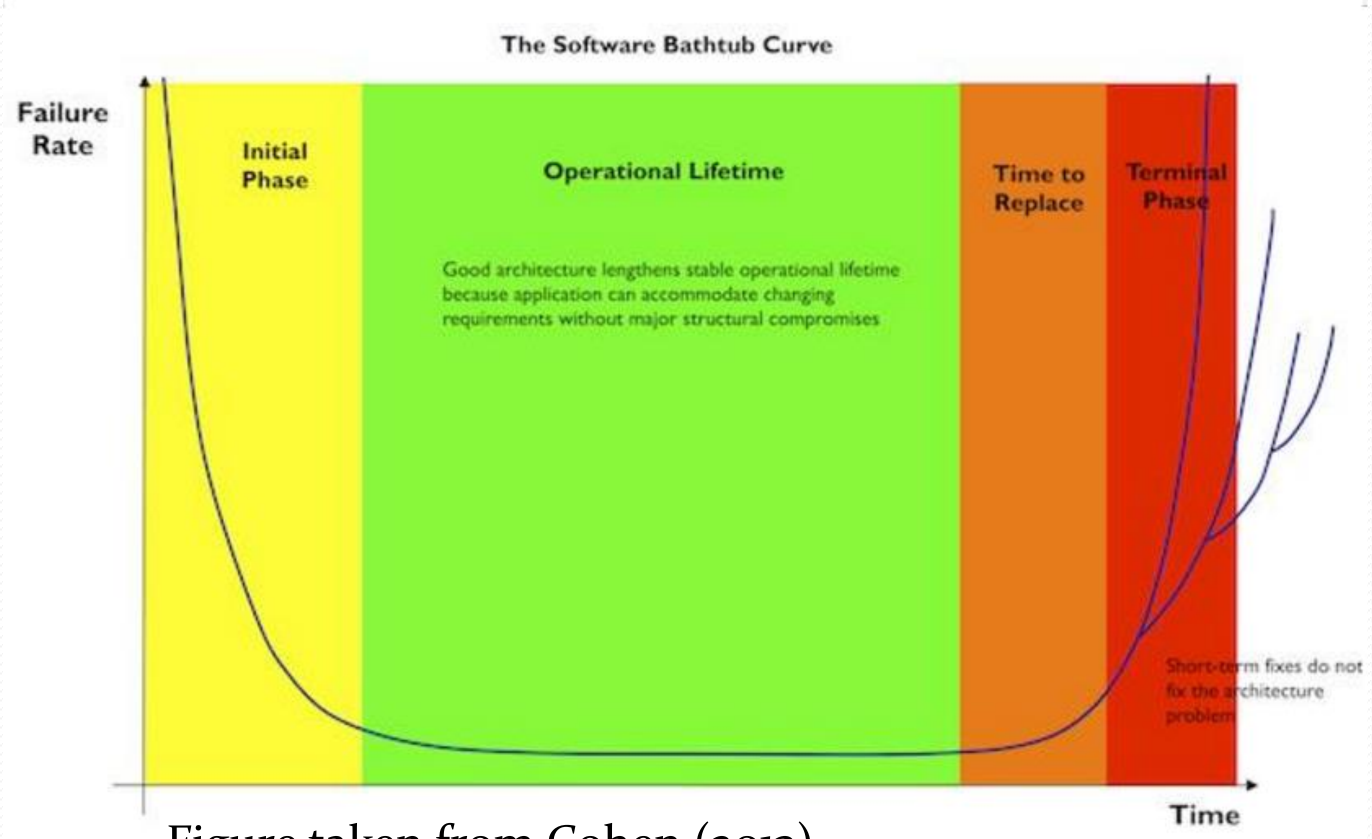


Figure taken from Cohen (2013)

Software Reliability End of Life

Cohen (2015)

- Most software systems follow the bathtub curve
- Some software systems move straight from Initial Phase to Terminal Phase
  - Due to poor architecture
  - Poor understanding of requirements
  - Inadequate understanding of business objectives

Software Reliability End of Life

Cohen (2015)

- Adding new bugs especially occurs when changes are made to add additional features to the code
- Once a software system enters the Terminal Phase, the system owners enter crisis mode
  - This makes it difficult to focus effort and energy on how to strategically replace the software
  - The focus is on tactical fixes to postpone catastrophic failure in the short time
    - However, these typically increase complexity
    - This increase in complexity contributes to an increase in failure rate in the medium term

Software Reliability End of Life

Cohen (2015) makes the case that a replacement phase should be introduced proactively prior to entering the Terminal Phase

- The Operational Lifetime of the software system is limited.
- Everyone knows this but for some reason will not admit it
- He says:

  ”The fact that in most instances this acknowledgement is not accompanied by some obsolescence planning strikes a dissonant chord. It seems that some in the enterprise expect or hope that the software system will have an infinite Operational Lifetime—or at the very least, they hope that they will have moved on long before their successors need to deal with the engineering and commercial challenges of the Terminal Phase.”

How To Achieve Software Reliability

Bendell and Mellor (1986):

- Systems analysis—get the requirements right, develop the right product
- Good management practice
  - Team structure
  - Good well documented software process
  - Design and code inspections
  - Method for fault and failure reporting
  - Change control

How To Achieve Software Reliability

- Good software engineering
  - Good design techniques
  - Pay attention to the human element
- Formal methods (where appropriate and possible)
  - Correctness proofs can be used only on simple algorithms
- Fault tolerant design
- Well designed testing

Reliability Models

Bendell and Mellor (1986) :

- Be quite clear about the distinction between reliability and other measures
- Carry out a representative product trial, keeping adequate records of failure and running time
- Do not rely on one single model but compare the results of several, on the basis of their predictive accuracy

Reliability Models

- Bendell and Mellor (1986) quote Littlewood (from a seminar):

1. Remember that many models perform badly most of the time
2. Some models seem to perform quite well some of the time
3. If you are sold a model as the universal answer to all your problems, be suspicious
4. Put not your trust in optimistic modelers. If the advocate of a model will tell you openly its drawbacks as well as its strengths, cherish him

Reliability Models

Dale and Harris (1982) discussed why software reliability models fail:

"...in many instances software reliability estimates, based on failure data, have proved to be unacceptably accurate. The reasons for this degree of inaccuracy are not yet fully understood but probably include the following:

1. Poor or inappropriate modeling assumptions;
2. Insufficient or poor quality data;
3. Undesirable statistical properties of parameter estimates"

Reliability Models

Harris in Bendell and Mellor (1986) discussed why statistical reliability growth models worked poorly:

> "...the major flaw in this approach is its failure to observe scientific method, particularly the aspect that requires one to show that the premises upon which models are built have some basis in fact which can be shown to be plausible, either by analogy or by empirical or rational justification."

Reliability Models

Characteristics of (traditional) software reliability models:

- Discuss probability of failure over a certain execution exposure
  - Execution time
  - Calendar time
    - sometimes execution time is later converted to calendar time
  - Number of test cases
  - Number of transactions

Reliability Models

Characteristics of (traditional) software reliability models (continued from previous page):

- Failures are characterized by studying numbers and times of previous failure occurrences
- Failures are assumed to be independent of each other
- A failure occurrence is expressed as a random variable
  - Failures are unpredictable because the incidence of bugs (faults) is largely unpredictable
  - Conditions under which a program is executed (for example, input variables) are largely unpredictable

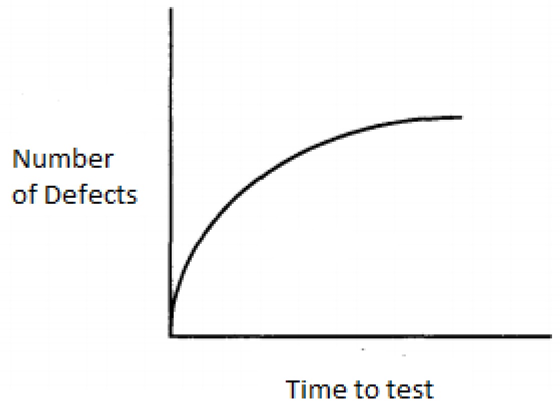
Reliability Models

Dr. Maureen Raley, my former Ph.D. student, spoke to Dr. Littlewood in England in 1997. She says that Dr. Littlewood told her:

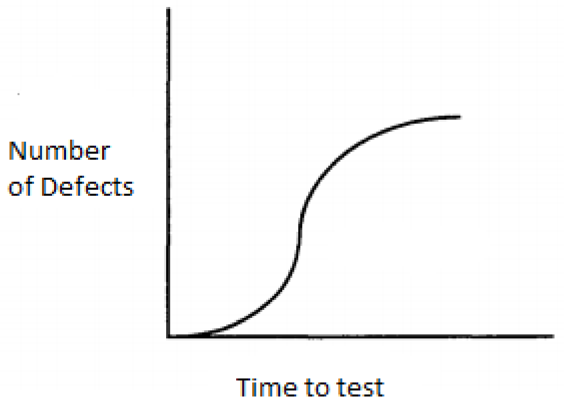
> "Using software reliability models was like looking out the back window to see where you are going."

Reliability Models

Software Reliability Growth Model types:



Concave Model



S-shaped Model

Reliability Models

Software Reliability Growth Model overall assumptions:

- Measuring residual defects (number of defects remaining in the software)
  - Helps know how much more testing is required
  - Helps know whether code is ready to ship
- The number of defects detected per unit time decreases as defects are detected and repaired (assuming some kind of testing is going on)

Reliability Models

Brockhurst and Littlewood (1996) examined the accuracy of 8 different software reliability models, in terms of short term prediction:

- they found 6 of these models were grossly optimistic in their predictions while 2 were grossly pessimistic

Reliability Models

In longer term reliability prediction, examining 2 of these models, Brockhurst and Littlewood found:

- even when a model can do one kind of prediction on a particular data source, different types of predictions don't work even on the same data source
- the accuracy of the models differed on different data sources

Reliability Models

Brockhurst and Littlewood's suggested solution was:

- to evaluate a model's past predictions to actual observations
- over time build up a sequence of prediction/observations comparisons *on a particular set of data*
- then compare multiple models *over the same set of data* and choose which model is more accurate in that way
- then use the differences between predictions and actual observations to recalibrate future predictions over the data set

Reliability Models

Subburaj (2015) discusses two classifications of software reliability models:

- predictive
- estimator

Reliability Models

Subburaj (2015) describes predictive models as models that are designed to predict failure intensity *prior to system test*

- these models are not very accurate because the input data to the models is not very accurate

Reliability Models

According to Subburaj (2015), estimator models use data from the early part of a software project to predict the future

- since the data from the same software project is being used, the estimator models tend to be more accurate than the predictive models (since the predictive models are based on more generic input data)

Reliability Models

Taxonomy of defects from Subbaraj (2015):

Defect Level	Example
Mild	Symptoms of the defects offend us aesthetically. For instance, a misspelled output or a badly aligned printout.
Annoying	Names are truncated
Disturbing	Refuses to handle legitimate transactions. Credit card not accepted.
Serious	Loses track of transactions. Credit made by the customer is lost in the bank records.
Very Serious	Not only the credit is lost, but a debit equal to cheque deposited is made.
Extreme	Such blunders occur frequently.
Intolerable	Long-term irrevocable corruption of database.
Catastrophic	System fails abruptly.
Infectious	System corrupts other systems even though it does not fail by itself.

Reliability Models

Ullah, Morisio, and Vetro (2015) provided a methodology to select the best reliability model to predict residual defects in Open Source Softwre:

Step 1. Collect defect data

Step 2. Extract defects

Step 3. Apply models to data

Step 4. Test against fit and prediction thresholds

Step 5. Test against prediction stability threshold

Step 6. Select the best model

Step 7. Compute Residual Defects

Reliability Models

Ullah, Morisio, and Vetro (2015)

Step 1. Collect defect data

- Since looking at open source data, collect defect data from online repositories

Reliability Models

Ullah, Morisio, and Vetro (2015)

Step 2. Extract defects

1. Want only bugs that are closed or resolved, so filter out:
   - Unresolved bugs
   - Enhancements
   - Feature requests
   - Tasks
   - Patches
2. Group data for entire release interval T into cumulative defects by week

Reliability Models

Ullah, Morisio, and Vetro (2015)

Step 3. Apply models to data

- Select data from a time period ending at ¾ of T to use in model fitting
- Use nonlinear regression to fit model to this data
  - If the model can describe the data, then:
    - Evaluate the model's goodness of fit based on R^2

Reliability Models

Ullah, Morisio, and Vetro (2015)

Step 4. Test against fit and prediction thresholds

- Compare goodness of fit of fitted models to a (subjectively chosen) R^2 threshold
  - They used 0.95 based on previous work by Stringfellow and Andrews (2002)
- Check fitted models' predictions against actual number of defects found
  - Reject any models that predict fewer defects than the actual number of defects

Reliability Models

Ullah, Morisio, and Vetro (2015)

Step 5. Test against prediction stability threshold

- A model's prediction is stable if the prediction for week j is +/- 10% of the prediction for week j-1
  - 10% threshold chosen based on work by Zeephongsekul, Xia, and Kumar (1994)
- Check model stability for the defect data from time period 3/4T to T
  - i=1
  - Repeat until week ending in T
    - Check stability of cumulative defects from 3/4T+i weeks
    - i=i+1

Reliability Models

Ullah, Morisio, and Vetro (2015)

Step 6. Select the best model

Choose the model that has:

- Passed all threshold tests
- Shows the highest number of predicted defects
  - This results in worst case cost estimates
- If models widely disagree, consider using additional assessment techniques before choosing a model

Reliability Models

Ullah, Morisio, and Vetro (2015)

Step 7. Compute Residual Defects

Choose whether or not to use an open source software package based on the chosen model

Reliability Models

Ullah, Morisio, and Vetro (2015) applied their technique to the following open source projects:

- Apache
- Gnome
- C++ Standard library
- JUDDI
- HTTP Server
- XML Beans
- Enterprise Social Messaging Environment (ESME)

Reliability Models

Ullah, Morisio, and Vetro (2015) Examined the following models:

Model	Type	Date	Mean Value Function
Musa-Okumoto	Concave	1984	$m(t) = a\ln(1+bt),\ a>0,\ b>0$
Inflection S-shaped	S-shaped	1984	$m(t)=a\frac{1-\exp[-bt]}{1+\psi(r)\exp[-bt]},\ \psi(r)=\frac{1-r}{r},\ a>0,\ b>0,\ r>0$
Goel-Okumoto	Concave	1979	$m(t)=a(1-\exp[-bt]),\ a>0,\ b>0$
Delayed S-shaped	S-shaped	1984	$m(t)=a(1-(1+bt)\exp[-bt]),\ a>0,\ b>0$
Generalized Goel	Concave	1985	$m(t)=a(1-\exp[-bt^c]),\ a>0,\ b>0,\ c>0$
Gompertz	S-shaped	Original (1832)	$m(t)=ak^{b^t},\ a>0,\ 0<b<1,\ 0<k<1$
Logistic	S-shaped	1991	$m(t)=\frac{a}{1+k\exp[-bt]},\ a>0,\ b>0,\ k>0$
Yamada Exponential	Concave	1986	$m(t)=a(1-\exp(-r(1-\exp(-bt)),\ a>0,\ b>0,\ r>0$

Reliability Models

They measured which models would have done the best over the data:

Prediction relative error =
(predicted defects-actual defects)/
predicted defects

Where:

- predicted defects is number of defects a model predicted at 2/3 of the time interval
- actual defects is defects at the end of the time interval

The model with the lowest prediction relative error is the best model for that release

Reliability Models

Partial Ullah, Morisio, and Vetro (2015) results:

Project	Release	Using Prediction Relative Error	Using Ullah et al. method
Gnome	V2.0	Delayed S-shaped	Delayed S-shaped
	V2.2	Goel-Okumoto, Yamada Exponential	Goel-Okumoto
	V2.4	Inflection S-shaped, Gompertz	Inflection S-shaped
Apache	2.0.35	Delayed S-shaped, Logistic	Gompertz
	2.0.36	Delayed S-shaped, Logistic, Gompertz, Generalized Goel	Generalized Goel
	2.0.39	Inflection S-shaped, Goel-Okumoto, Generalized Goel	Goel-Okumoto
C++ Std Lib	4.1.3	Musa-Okumoto	Inflection S-shaped
	4.2.3	Musa-Okumoto	Gompertz
	5.0.0	Inflection S-shaped, Yamada Exponential, Generalized Goel	Yamada exponential

Reliability Models

Ullah, Morisio, and Vetro (2015) overall results:

- For 17 of 21 releases, their method chose the same model as was chosen by prediction relative error
- For the other 4 releases, their method chose the second best model according to prediction relative error
  - Their method threw out the best models in these 4 cases because of a negative prediction relative error

Agile Software Development Processes

Agile Manifesto, from the Agile Alliance (2016):

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- **Individuals and interactions** over processes and tools
- **Working software** over comprehensive documentation
- **Customer collaboration** over contract negotiation
- **Responding to change** over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

© 2001, the Agile Manifesto authors
This declaration may be freely copied in any form, but only in its entirety through this notice.

Agile Software Development Processes

Agile Principles

1 Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

2 Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

3 Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

4 Business people and developers must work together daily throughout the project.

5 Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

6 The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

Agile Software Development Processes

Agile Principles

7 Working software is the primary measure of progress.

8 Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

9 Continuous attention to technical excellence and good design enhances agility.

10 Simplicity--the art of maximizing the amount of work not done--is essential.

11 The best architectures, requirements, and designs emerge from self-organizing teams.

12 At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Reliability with Agile Software Processes

Far (2007) says:

"In [the] software industry there is a common assumption that deployment of software reliability engineering (SRE) contributes to huge overhead in development and its practice does not match agile software development which puts emphasis on traveling light and generating a minimum amount of project artifacts."

Reliability with Agile Software Processes

Far (2007) specifies the Software Reliability Engineering process as consisting of the following steps:

1. Define necessary reliability
2. Develop an operational profile
3. Prepare for testing
4. Execute tests
5. Apply test data to guide decisions related to whether or not the developed software is ready for release

Reliability with Agile Software Processes

Far (2007) then discusses how these software reliability engineering process steps can be achieved during agile software development:

- Same as in non-agile process, define necessary reliability in terms of failure intensity. This includes:
  - Define failure intensity objectives:
    - Find acquired hardware and software failure intensities
    - Determine developed product failure intensity objectives

Reliability with Agile Software Processes

Far (2007) further discusses balance between fault prevention, fault removal, and fault tolerance:

- In non-agile development:
  - fault prevention is accomplished through requirements engineering, stakeholder involvement, and following standards
  - Fault removal is accomplished through code reviews and testing after code has been written
- In agile development:
  - In test driven development, unit tests and acceptance tests are written prior to writing the code. So testing is a fault prevention technique
    - With continuous integration combined with test driven development, regression testing is performed automatically
  - When pair programming is used, code reviews become a fault prevention technique
  - Since the process is test driven, the system will continuously be in a stable, testable, verifiable state
    - So the failure intensity value will continually be held below the failure intensity objective

Reliability with Agile Software Processes

Far (2007) discusses the use of an operational profile in agile software development:

- determining usage data based on user story is often not prescribed by agile software development processes
- however, the general idea of operational development fits well into agile development, since it goes along with the idea of develop the most critical functionality first
- Far suggests developing an operational profile as recommended by Musa

Operational Profiles

Musa (1993) developed operational profiles in five steps:

1. Find customer profile
   - Person or group acquiring the system
2. Find user profile
   - Person or group using the system
3. Find system-mode profile
   - Group operations into execution related activities called system modes. Determine their occurrence probablities.
4. Find functional profile
   - Create a function list for each system mode. Construct a work flow profile showing the overall process being implemented. This will provide a quantitative view of how and how often different functions are used.
5. Find operational profile itself
   - Develop the operational architecture, the way the user will perform operations to accomplish functions
   - Divide execution into runs—end to end user activities
   - Identify input space—the set of input states that can occur during operation (based on number of input variables) and occurrence probabilities
   - Partition input space into operations—sample across the different input states/runs

Reliability with Agile Software Processes

Boerman et al. (2015) examine a model to report software status to stakeholders outside the agile development activity.

This is based on the Goal Question Metric approach and resulted in the table on the following slide.

Some definitions:

- Enhancement rate—percentage of project size of previous sprint that has gone into acceptance or production in current sprint
- Scope prognosis—combines project size with enhancement rate to give project owner an idea of the level of functional completeness, and compared to desired project end date
- Estimation shift—looks at changes between sprints in person hours to complete a story point

Reliability with Agile Software Processes

Goal	Question	Metric
1. Achieve the functional compliance of the software system from the viewpoint of the project owner/sponsor	1.a. To what extent are the functional requirements being implemented?	Enhancement Rate
		Scope Prognosis
		Project Size Remaining
	1.b. How much scope churn is there?	Changed product backlog items (PBIs)
		Added PBIs
		Rejected PBIs
		Project Size
2. Fulfill the expected schedule of the SD activity from the viewpoint of the project owner/sponsor?	2.a. When is the delivery of the software system expected?	Enhancement Rate
		Time Prognosis
		Project Size remaining
3. Optimize value for money of the SD activity from the viewpoint of the project owner/sponsor	3.a. What is the quality of the development process?	Enhancement Rate
		Estimation Shift
		Priority Shift
		PBIs at risk
	3.b. What is the quality of the product?	Software Quality –quality measurement based on ISO/IEC 25010—requires effort beyond typical agile artifacts
	3.c. What is the financial status of the project?	Expenses Prognosis—requires effort beyond typical agile artifacts
4. Minimize the risk of wasting SD effort (from the viewpoint of the project owner/sponsor)	4.a. What is the amount of effort at risk?	Effort at Risk

Reliability with Agile Software Processes

Kiwan, Morgan,a nd Benedicenti (2013) suggest the following metrics to examine critical factors in an Extreme Programming (XP) software process, these are shown in the table on the following slide.

Reliability with Agile Software Processes

Metric Description	Purpose
Lines of code/hour	Measures actual production rate
Number of users stories per release	Allows comparison of work rates at different time periods
Percentage of practicing pair programmers	Improves software quality
Number of post release defects	Measures software quality
Percentage of customer involvement based on face to face meetings vs. remote meetings, time zones of customers	Affects software development cycle

Some Recent Reliability Models

Fiondella and Gokhale (2011) present a model with a bathtub shaped fault detection rate (fault detection rate varies over time according to the bathtub curve, increases toward the end of testing).

Over the Ohba failure count data set, using predictive mean square error and Akaike Information Criteria, it outperforms the following models:

- Inflexion S-shaped model (same as earlier slide)
  - Mostly constant fault detection rate—some increase in later stage testing based on one parameter
- Goel-Okumoto model (same as earlier slide)
  - Constant fault detection rate
  - Overestimates faults because assumes in later phases that testing based on techniques used, and ignores that code comprehension in later stages accounts for a decrease in the remaining number of faults
- Burn-in model—$m(t) = \alpha\ (1\text{-}\exp(-\lambda\gamma t^{\gamma})$
  - simplifies to Goel-Okumoto when $\lambda=\beta$ and $\gamma=1$)
  - Varying fault detection rate—increasing over time
  - Overestimates faults because assumes in later phases that testing based on techniques used, and ignores that code comprehension in later stages accounts for a decrease in the remaining number of faults

Some Recent Reliability Models

Predictive Mean Square Error

- the sum of the squared differences between the cumulative number of faults observed and the cumulative number of faults predicted for the last k observations

Akaike Information Criteria

- Two times the number of parameters of the model minus the log-likelihood

Some Recent Reliability Models

Park et al. (2012) proposed a software reliability model for embedded systems that takes hardware-related software failures into account.

They defined hardware related software failures as those caused by faulty hardware or hardware configuration changes.

They developed two new models:

- Random model—assumes that hardware related software failures occur randomly at a constant rate
- Weibull-based model—assumes that hardware related software failures occur based on a Weibull distribution

Some Recent Reliability Models

Over historical failure data of a combat system developed by a contractor (at CMMI level 5) with the Korea Ministry of National Defense,

using Mean Magnitude of Relative Error and Mean Square Error,

Park et al. (2012) compared their models to:

- Goel-Okumoto model
- Delayed S-shape Model

Their Weibull-based model was better than either Goel-Okumoto or Delayed S-shape

Their Random model was better than Delayed S-shape but slightly worse than Goel-Okumoto

Software Quality and Software Defect Prediction

Predicting defects (bugs) based on analysis of source code is usually considered to be more software quality related than software reliability related

Much work in this area has involved various kinds of complexity metrics. Two well known examples traditionally used in functionally oriented software are:

McCabe's cyclomatic complexity metric, McCabe (1976):

- A count of the number of decision points in code

Halstead's Software Science, Halstead (1977):

- Measures program vocabulary, length, volume, difficulty and effort based on number of distinct operators, number of distinct operands, total number of operators, and total number of operands in a program

Software Quality and Software Defect Prediction

- Object-oriented software consists of classes and objects
  - Classes are user-defined data types
  - An object is an instance of a class
  - A class/object contains data and functions
    - Data is called attributes or member variables
    - Functions are called methods, operations, or member functions
  - Inheritance occurs when one class is an extension of an existing class (known as a parent-child relationship)
  - Methods from a base class can be overridden in child classes
  - Methods can be overloaded—methods with the same names could do somewhat different (though similar) tasks…these methods are differentiated by which parameters are passed to the method

Software Quality and Software Defect Prediction

To measure object-oriented software, several suites of metrics have been defined, these include:

- Chidamber and Kemerer
- Lorenz and Kidd
- Abreu
- Bansiya and Davis
- Etzkorn and Delugach
- Li
- Stein, Etzkorn, Gholston, Farrington, Utley, Cox, and Fortune
- Poshyvanyk and Marcus

Software Quality and Software Defect Prediction

The Chidamber and Kemerer Metrics Suite (1991, 1994) consists of the following metrics:

- Depth of Inheritance Tree (DIT)
  - Distance from the Base Class to the bottommost children in an inheritance tree
- Number of Children (NOC)
  - Number of classes that inherit from the current class
- Response for a Class (RFC)
  - All local methods plus all methods called by local methods
- Lack of Cohesion of Methods (LCOM)
  - The cohesion of a class is characterized by how closely the local methods are related to the local instance variables
- Weighted Methods per Class (WMC)
  - Sum of the complexities of all local methods
- Coupling Between Objects
  - A count of the non-inheritance-related couples with other classes.

Software Quality and Software Defect Prediction

Lorenz and Kidd suite:

Method metrics:

- Number of message sends
- Lines of Code, Number of statements

Class metrics:

- Number of public instance methods
- Number of instance methods
- Number of instance variables
- DIT
- Number of abstract classes
- Avg number of parameters per method
- Avg number of comment lines/method

Etc.

Software Quality and Software Defect Prediction

Brito e Abreu (MOOD metrics)(1995):

- Method Hiding Factor 1995 and 1996
  - – Proportion of methods that are hidden (private or protected)
- Attribute Hiding Factor 1995 and 1996
  - Proportion of attributes that are hidden (private or protected)
- Method Inheritance Factor
  - Proportion of Methods that are inherited
- Attribute Inheritance Factor
  - Proportion of Attributes that are inherited
- Polymorphism Factor 1994 and 1995
  - Proportion of all possible polymorphic situations that are actually polymorphic
- Density of Client/Server relationships between classes
- Etc.

The MOOD metrics specify both per-system metrics as well as per-class metrics

Software Quality and Software Defect Prediction

Bansiya and Davis (QMOOD) suite:

- Total number of ancestors
- Proportion of methods accessible from other classes
- Proportion of data which is inaccessible from other classes
- Number of classes directly related to current class
- Number of classes related to current class by attribute definitions alone
- Cohesion between methods of a class, based on parameter type definitions
- Extent of the part-whole relationship
- Number of new or inherited polymorphic methods
- Proportion of methods available to the class that are inherited
- Etc.

The QMOOD metrics include both class level and system level metrics

Software Quality and Software Defect Prediction

Li and Henry suite (1993) (1993):

- Message Passing Coupling
- Data Abstraction Coupling
  - Number of abstract data types employed
- Number of Methods
- Number of semicolons
- Number of attributes+number of operations

Software Quality and Software Defect Prediction

Semantic Metrics Suites:
Etzkorn and Delugach (2001);
Stein, Etzkorn, Gholston, Farrington, Utley, Cox, and Fortune (2004, 2009):

- Semantic metrics note that syntactic metrics are indirect and often arguable
- Instead, semantic metrics employ a program understanding engine to automatically understand the software, then measure the knowledge-base
- Includes metrics for cohesion, complexity, coupling, among others

- Example metrics:
  - LORM, LORM1, LORM2, LORM3
    - Look at the overlap of conceptual graphs representing the class to indicate cohesion
  - CDC (Class Domain Complexity)
    - Examines the complexity of the conceptual graphs that represent the class
  - SCDE (Semantic Class Definition Entropy)
    - Examines the frequency with which domain related concepts occur in the class

Software Quality and Software Defect Prediction

Poshyvanyk and Marcus Conceptual Coupling (2009) and Conceptual Cohesion Metrics (2009)

These are based on semantic information from source code, calculated with information retrieval techniques (latent semantic analysis)

Example metrics:

Conceptual Similarity between methods of a class—the cosine between the vectors corresponding to methods as they appear within the semantic space constructed by LSI

Conceptual similarity between two classes—the average of the similarity methods between all unordered pairs of methods of two classes

Software Quality and Software Defect Prediction

Olague, Etzkorn, Gholston, and Quattlebaum (2007) examined defect prediction of object-oriented classes developed using highly iterative/agile methods:

- metrics suites examined:
  - Chidamber and Kemerer (CK) metrics
  - Brito e Abreu MOOD metrics
  - Bansiya QMOOD metrics
- Analyzed Rhino, a Java implementation of JavaScript
- Used multivariate binary logistic regression
  - does not assume linearity of relationship between independent and dependent variables
  - Does not assume variance around the regression line is the same for all values of the predictor value
- Conclusions:
  - MOOD metrics were not good defect predictors
  - The CK metrics suites produced the best 3 models for defect prediction, followed by one QMOOD model
  - CK and QMOOD metrics were highly correlated

Software Quality and Software Defect Prediction

Menzies, et al. (2013) examined whether defect prediction and effort estimation is best done on a local basis or a global basis:

- Used data from the PROMISE data set::
  - For defects: 7 software packages
  - For effort: two systems, one with 499 projects, one with 156 projects
- Metrics examined were subsets of Chidamber and Kemerer, QMOOD, MOOD, variations of McCabe metrics
- Used clustering algorithms/rule learners
- Conclusions:
  - Lessons learned from clustering (combining small parts of different data sources) were superior to global generalizations (over all data) or local lessons learned (from particular projects)
  - Clusters from other sources nearest to the test data give the best lessons learned

Software Quality and Software Defect Prediction

Shatnawi (2013) examined defect prediction for open source systems using the Chidamber and Kemerer metrics:

- Used data from the PROMISE repository over 3 or 4 releases of 4 different open source projects (ant, camel, jedit, xerces)
- Examined Chidamber and Kemerer metrics only
- Used 7 machine learning classifiers: bayesian networks, support vector machines, neural networks, k nearest neighbors, decision trees: (C4.5)(CART) and random forest
- Examined binary variables (fault/no fault) and 4 category dependent variables (none, low, medium, high):
  - None—class does not have faults in current release or previous release
  - Low—some faults were fixed for a class in the previous release but no faults are fixed in the current release
  - Medium—no faults in the previous release, but there are faults in the current release
  - High—Faults in both previous and current release
- Conclusions:
  - All classifiers did a better job of defect prediction for later releases
  - Classifier results for ant and jedit were good, for camel and xerces were not good

Automated Bug Localization

Automated bug localization refers to techniques that automatically map bug reports to the sections in code where the bugs reside.

Many modern techniques are based on information retrieval such as Latent Semantic Indexing (LSI) or Latent Dirichlet Allocation (LDA).

- The comments and identifiers in class definitions in the code represent a text document
- The bug report represents another text document

LDA is a probabilistic topic model.

- It models each topic as a probability distribution over the set of terms that make up the vocabulary of the document collection
- Each document is a finite mixture over the set of topics

Similarity measures between documents are computed as the conditional probability of the query given the document

- A document is relative to a query if it has a high probability of generating the words in the query

Automated Bug Localization

Lukins, Kraft, and Etzkorn (2010) compared LDA to LSI to map bug reports to code classes:

- Compared LSI to LDA in Mozilla and Eclipse over 8 bugs analyzed in a previous study
  - For LSI, only 3 out of 8 bugs resulted in the first relevant method ranked in the top 10 results returned
  - For LDA, all eight bugs resulted in the first relevant method ranked in the top 10 results returned
- Performed a study to analyze how LDA worked on large software systems with many bugs
  - Analyzed 322 bugs across 25 versions of two software systems: Rhino and Eclipse
  - Study was much larger than prior LSI-only studies
  - Study showed LDA can scale to work with large systems

Automated Bug Localization

Sisman, Akbar, and Kak (2016) examined automatic bug localization that employed structural information in addition to the bag of words concept used by LSI and LDA:

- Used Markov Random Field-based retrieval
  - This can be used to take into account proximity of terms and ordering relationships between terms
- Examined 4000 bugs in AspectJ, Eclipse v3.1, Chrome v. 4.0
  - Used BUGLinks (Eclipse, Chrome) and iBugs (AspectJ) datasets
- Conclusions:
  - Markov Random Field-based modeling is far superior to bag-of-words approaches

Recommendations for a Metrics Program

- It's important to have a baseline measurement
  - Even if there aren't good thresholds for some metrics, you can at least examine changes.
  - Over time, you should be able to collect the mean values of metrics …anything which varies considerably from the mean should be examined
- There's no point in collecting data unless you use it!
- As much metrics collection as possible should be automated
- DON'T use metrics on software as a performance metric for programmers!
  - Grady, R., and Caswell, D., *Software Metrics, Establishing a Company-Wide Program,* Prentice-Hall, 1987.

References

Agile Alliance. 2016. https://www.agilealliance.org/agile101

Bendell, A. and Mellor, P. (Eds.) 1986. *Software Reliability State of the Art Report 14:2.* Pergmon Infotech.

Boerman, M.P., Lubsen, Z., Tamburri, D., and Visser, J. 2015. "Measuring and Monitoring Agile Software Development Status," *Proceedings of the 6th International Workshop on Emerging Trends in Software Metrics*, pp. 54-62.

Abreu, F., Goulao, M., and Esteves, R. 1995 "Toward the Design Quality Evaluation of OO Software Systems," *ICSC*.

Chidamber , S.R., and Kemerer, C.F., 1991. "Towards a Metrics Suite for Object-Oriented Design," *OOPSLA*, 1991.

Chidamber, S.R., and Kemerer, C.F., 1994. "A Metrics Suite for Object-Oriented Design, *IEEE Trans. On SW Eng.,* 1994.

Cohen, J. 2014. The Software Bathtub Curve—Understanding the Software Systems Lifecycle. https://www.linkedin.com/pulse/20140723115956-15133887-the-software-bathtub-curve-understanding-the-software-systems-lifecycle

Dale, C.J., and Harris, L.N. 1982. Approach to Software Reliability Prediction. *Proceedings of the Annual Reliability and Maintainability Symposium, pp. 167-175.*

Etzkorn, L., and Delugach, H. 2001. Towards a Semantic Metrics Suite for Object-Oriented Design, *Proceedings of the 34th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS).*

Far, B.. 2007. "Software Reliability Engineering for Agile Software Development," *Proceedings of the IEEE Canadian Conference on Electrical and Computer Engineering,* pp.694-607.

Fiondella, L., and Gokhale, S.2011. "Software Reliability Model with Bathtub-shaped Fault Detection Rate," *Proceedings of the IEEE Reliability and Maintainability Symposium*.

References

Gall, C. (Stein), Lukins, S., Etzkorn, L., Gholston, S., Farrington, P., Utley, D., Fortune, J., and Virani, S.2008. "Semantic Software Metrics Computed from Natural Language Design Specifications," IET Software (formerly IEE Proceedings Software), Vol. 2, No. 1, pp. 17-26

Goel, A.L. "Software Reliability Models: Assumptions, Limitations, and Applicability," *IEEE Transactions on Software Engineering*. Vol SE-11, No. 12, pp.1411-1423.

Goel, A.L. and Okumoto, K. 1979. "A Time Dependent Error Detection Model for Software Reliability and Other Performance Measures," *IEEE Transactions on Reliability, pp. 206-211.*

Halstead, M. 1977. Elements of Software Science Elsevier.

Kececioglu, D.B. 1991. "Reliability Growth," *Reliability Engineering Handbook Vol 2*, Prentice-Hall.

References

Kiwan, H., Morgan, Y.L, Benedicenti, L. 2013. Two Mathematical Modeling Approaches for Extreme Programming. *Proceedings of the 26th IEEE Canadian Conference on Electrical and Computer Engineering*, .

Laprie, J.C. 1985. Dependable computing and fault tolerance: Concepts and terminology. In: Proceedings of 15th International Symposium on Fault-Tolerant Computing (FTSC-15), IEEE Computer Society, pp. 2–11

Laprie, J., Kanoun, K. 1996. Software Reliability and System Reliability. Ed. M. Lyu, *Handbook of Software Reliability Engineering*, McGraw-Hill. https://secweb.cs.odu.edu/~zeil/cs795SR/Papers/TextBook/

Li, W., 1998. "Another Metric Suite for Object-Oriented Programming," *Journal of Systems and Software*.

Li, W., and Henry, S., 1993. "Object-Oriented Metrics that Predict Maintainability," *Journal of Systems and Software.*

Littlewood, B., and Strigini, L. 2000. Software Reliability and Dependability: A Roadmap. *Proceedings of the International Conference on Software Engineering (ICSE)—Future of SE Track*, pp. 175-188.

Marcus, A., Poshyvanyk, D., Ferenc, R. 2008. Using the Conceptual Cohesion of Classes for Fault Prediction in Object-Oriented Systems. IEEE Trans. Software Eng. Vol. 34, No. 2, pp. 287-300

McCabe, T.J. 1976. A Complexity Measure. *IEEE Trnasactions on Software Engineering*, vol.2, no. 4.,pp .308-320.

Menzies, T., Butcher, A., Cok, D., Marcus, A., Layman, L., Shull, F. Turhan, B., and Zimmerman, T. 2013. "Local versus Global Lessons for Defect Prediction and Effort Estimation," *IEEE Transactions on Softwre Engineering*, Vol. 39, No. 6, pp. 822-834.

Musa, J.D., 1993. "Operational Profiles in Software-Reliability Engineering," *IEEE Software*. Vol. 10, no 2, pp. 14-32.

References

Olague, H.M., Etzkorn, L.H., Gholston, S., and Quattlebaum, S. 2007. "Empirical Validation of Three Software Metrics Suites to Predict Fault-Proneness of Object-Oriented Classes Developed using Highly Iterative or Agile Software Development Processes," *IEEE Transactions on Software Engineering*, Vol. 33, No. 6.

Pan, Jiantao. 1999. Software Reliability, Dependable Embedded Systems, Carnegie Mellon. https://users.ece.cmu.edu/~koopman/des_s99/sw_reliability/

Park, J., Kim, H., Shin, J., Baik, J. 2012. "An Embedded Software Reliability Model with Consideration of Hardware related Software Failures," *Proceedings of the 6th IEEE Conference on Software Security and Reliability*, pp. 207-214.

Parnas, D.L. 1985. Software aspects of strategic defense systems. *Communications of the ACM*, Vol. 28, Issue 12, pp. 1326-1335.

Poshyvanyk, D. Marcus, A., Ferenc, R., Gyimóthy, T. 2009.
Using information retrieval based coupling measures for impact analysis. *Empirical Software Engineering* Vol. 14, No. 1, pp. 5-32

Pressman, R.S. and Maxim, B.R. 2015. *Software Engineering: A Practitioner's Approach*. McGraw-Hill.

Quyoum, A., Mehraj, U.D.D., Quadri, S.M.K. 2010. Improving Software Reliability using Software Engineering Approach. *International Journal of Computer Applications*, Vol 10, No 5,pp. 41-47.

References

Shatnawi, R. (2013) "Empirical Study of Fault Prediction for Open-Source Systems using the Chidamber and Kemerer Metrics," IET Software, pp. 113-120.

Sisman, B., Akbar, S.A., and Kak, A.C. (2016). "Exploiting Spatial Code Proximity and Order for Improved Source Code Retrieval for Bug Localization," *Journal of Software Evolution and Process*, early-published online.

Stein C., Etzkorn,L., Gholston, S., Farrington, P., Utley, D., Cox, G., and Fortune, J. 2009. "Semantic Metrics: Metrics Based on Semantic Aspects of Software," Applied Artificial Intelligence, Vol. 23, Issue 1, pp.44-77.

Stringfellow, C., and Andrews, A.A. 2002. "An Empirical Method for Selecting Software Reliability Growth Models," *Empirical Software Engineering*, vol. 7, no. 4, pp. 319-343.

Subburaj, R. 2015. *Software Reliability Engineering*. McGraw-Hill.

Ullah, N., Morisio, M., and Vetro, A. 2015. "Selecting the Best Reliability Model to Predict Residual Defects in Open Source Software," *IEEE Computer*, Vol. 48, No. 6, pp. 50-58.

Wood, A. 1996. Software Reliability Growth Models. TANDEM. Technical Report 96.1

References

Yamada, S., Ohba, M., Osaki, S. 1984. "s-Shaped Software Reliability Growth Models and Their Applications," *IEEE Transactions on Reliability*, Vol. R-33.

Yamada, S., Ohtera, H., and Narihisa, H. 1986. "S-Shaped Reliability Growth Modeling for Software Error Detection," *IEEE Transactions on Reliability*, Vol. R-32, pp. 475-484.

Zeephongsekul, P., Xia,G., and Kumar, S. 1994. "Software Reliability Growth Model: Primary Failures Generate Secondary Faults under Imperfect Debugging," *IEEE Transactions On Reliability, Vol. 42, no. 3, pp. 408-413.*