Chapter 5 Software Reliability and Recovery Techniques

Software is a major component of any system today. Most major advances in the technology world are being accomplished through software.

A system is a collection of interrelated components that work together to achieve some objective.

Software Engineering – a body of engineering and management technologies used to develop quality, cost-effective and on-time software (theories, methods and tools).

$R_{sys} = R_H \times R_S \times R_O$ assuming independence for **H**ardware, **S**oftware and **O**perator (human)

-ilities → maintainability, dependability which includes reliability, security and safety, efficiency and usability.

Software Reliability – measurement and prediction of the probability that software will perform its intended function according to the specifications without error for a given period of time.

Software Recovery – a set of fail-safe design techniques to ensure automatic recovery, reinitialization and restart when an error causes a software malfunction.

Are software errors/failures a deterministic or random event? Due to the large set of input data values, the author treats software reliability as a random event where the random variable is the changing set of inputs.

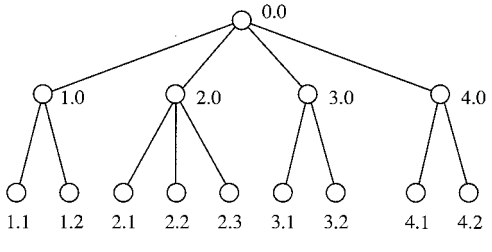
Software Life Cycle – specification, development, validation and evolution.

Main reason for high cost (~ time) of software → changes and maintenance

Primary source of errors/failures in software – incomplete specifications

5.3.2 Requirements – hardware & software, written by customer, bound the problem
5.3.3 Specifications – response to reqt's; the why, what & how of a project
5.3.4 Prototypes – SLOCs, intellectual span of control, reused or legacy code
5.3.4 Design – top down approach, decomposition into smaller elements
system → subsystem → module

Modeled by a hierarchy diagram or H-diagram, which resembles an inverted tree (node/branch)



One feature of top-down decomposition is the delay in dealing with the lower-level elements. This hiding process called ***information hiding*** is sometimes desirable since it allows the designer to proceed without all of the necessary design information which might not be available (other sources, subcontractos)

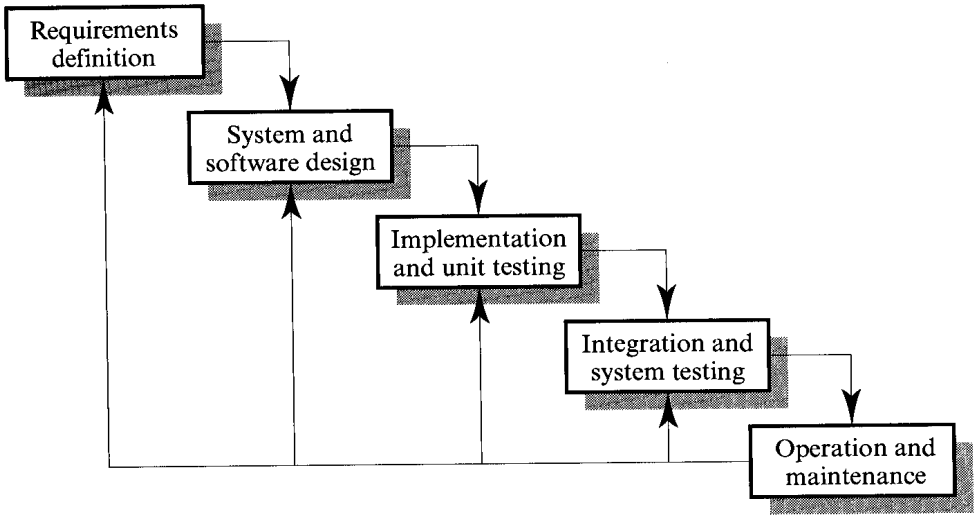
Knowing the estimate for the project lines of code (SLOCs), the number of interfaces can be estimated from an H-diagram which will scope the design and testing tasks.

Coding – 20% with or without assistance of a compiler (only available to testers), using a structured programming language (C++, Ada)

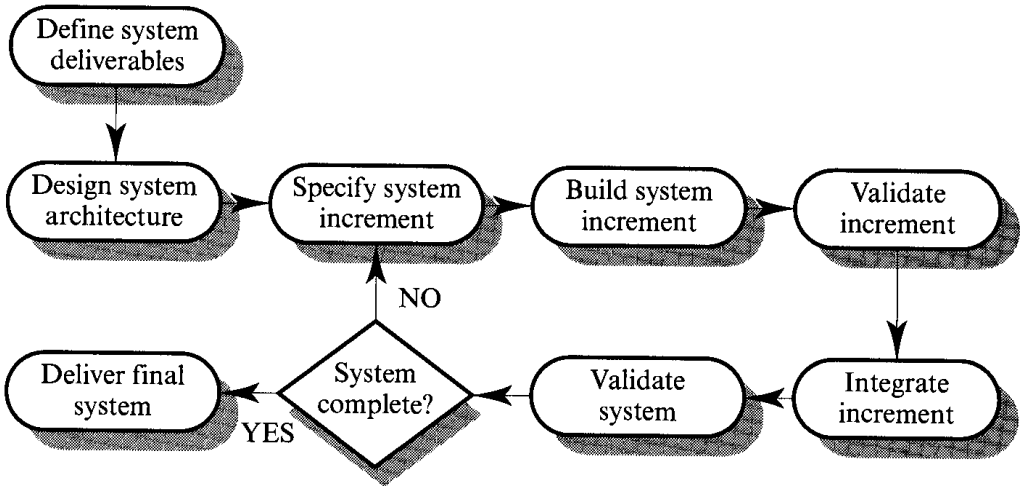
Testing – unit or module
in-house versus outside contractor
white box/black box
acceptance testing
regression testing – retest after corrections & changes, use a selected test set in the retest
alpha (in-house) and beta testing

Development Phases

Waterfall (the classic) Some notable stages: PDR (Preliminary Design Review) and SDR (System Design Review – approval to start implementation)



Rapid Prototyping (incremental development → design/code a little/test a little)



5.4 **Reliability Theory**

f(t) – the pdf (probability density function)

N = # of items placed in test (population)

n(t) = the # of items surviving the test up to time t

f(t) = [n(t) – n(t + dt)] / N dt → reflects the rate of failures based on original population

z(t) = [n(t) – n(t + dt)] / n(t) dt → instantaneous rate of failures based on the number of survivors at the beginning of the interval. (see Appendix B, page 423)

R(t) = probability of no failures from 0 to t given that there was no failures up to t = 0

(probability of failure up to time t is not a constructive way to characterize software)

knowing $R(t) = n(t) / N = 1 - F(t) = 1 - \int f(t)\, dt$ F(t) - cumulative distribution f(t) – prob density

f(t) = – dR(t)/dt z(t) = f(t) / R(t) Hazard Function also known as the failure-rate function (Eq 5.10)

z(t) = – [dR(t)/dt] / R(t) integrating both sides

$\ln\{R(t)\} = -\int z(t)\, dt$ (Eq 5.13a) then exponentiating both sides

see Eq B39 for development

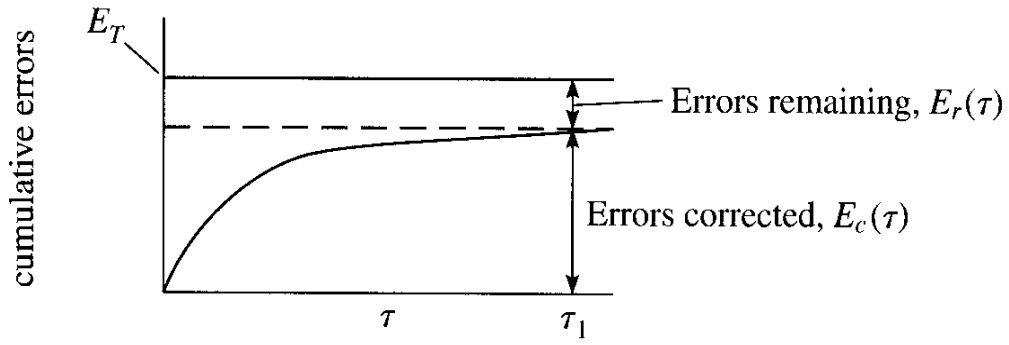
$R(t) = e^{-\int z(t)\, dt}$ function to be used in the s/w reliability model development *(Eq 5.13b)*

$MTTF = \int_0^\infty R(t)\, dt$ which for a constant-failure rate [$z(t) = \lambda$] then $MTTF = 1 / \lambda$

5.5 **Software Error** (bugs) **Models** *(we'll develop three models)*

Requirements & Specification Phases – design errors, considered code failures when found

Software errors – start failure count normally after the software comes under configuration control (usually at integration/system testing) - although should be earlier.



$E_r(\tau) = E_T - E_c(\tau)$

where
τ is **testing** time

Removal of errors → normally approaches equilibrium (with or without any new errors) BUT the process can diverge IF the generation rate of new errors exceeds the error-removal rate.

Experience tells us that changes normally introduce other errors → dependent on software complexity, percentage of available processing power/memory/input-output; experience of programmers, schedule constraints, financial constraints, etc.

It is normally presumed that you can never eliminate **all** the errors although some applications require that you get as close as possible (loss of life systems).

Collecting enough statistics; environment changes, customer changes, budget changes → all impact trying to develop a stable error removal model. Computer based estimation techniques are highly developed today. Past experience is probably your best indicator.

1. **Constant Error-Removal Rate** – in spite of the fact that experience tells us that the error removal rate decreases as testing progresses. [Outside factors (staff, management & external problems) impact the stability of an error removal rate model and these factors are almost impossible to control.] This model presumes that a total number of errors actually exist so proceeding with the model of a constant error removal rate over the development and testing periods . . .

$\mathbf{E_r(\tau) = E_T - \rho_o \tau}$ where $E_r(\tau)$ is errors remaining, E_T are the total errors and ρ_o is the constant error-correction rate in errors removed per unit of *testing* time (τ) .

2. **Linearly Decreasing Error-Removal Rate**

Based on the observation that the error-removal rate decreases with testing time, τ

$\mathbf{E_r(\tau) = E_T - K \tau (1 - \tau / 2 \tau_0)}$ where $E_r(\tau)$ are errors remaining, E_T are the total errors K is determined by the initial error-removal rate at $\tau = 0$ and τ_0 is the total testing time; that is, when testing stops and thus the error removal rate goes to zero.

$E_T = 130$ errors (estimate at $\tau = 0$) after 8 months of testing we desire to remove 120 errors. The error-removal ***rate*** will be zero at $\tau_0 = 8$ months with $E_r(\tau) = 10$ errors at $\tau = 8$. The model is $E_r(\tau) = E_T - K \tau (1 - \tau / 2\tau_0)$ Solving for K, we obtain K = 30

$E_r(\tau) = 130 - 30 \tau (1 - \tau / 16)$ (Eq 5.24b)

The linearly decreasing straight line which is marked with Δ still somewhat a contrived model

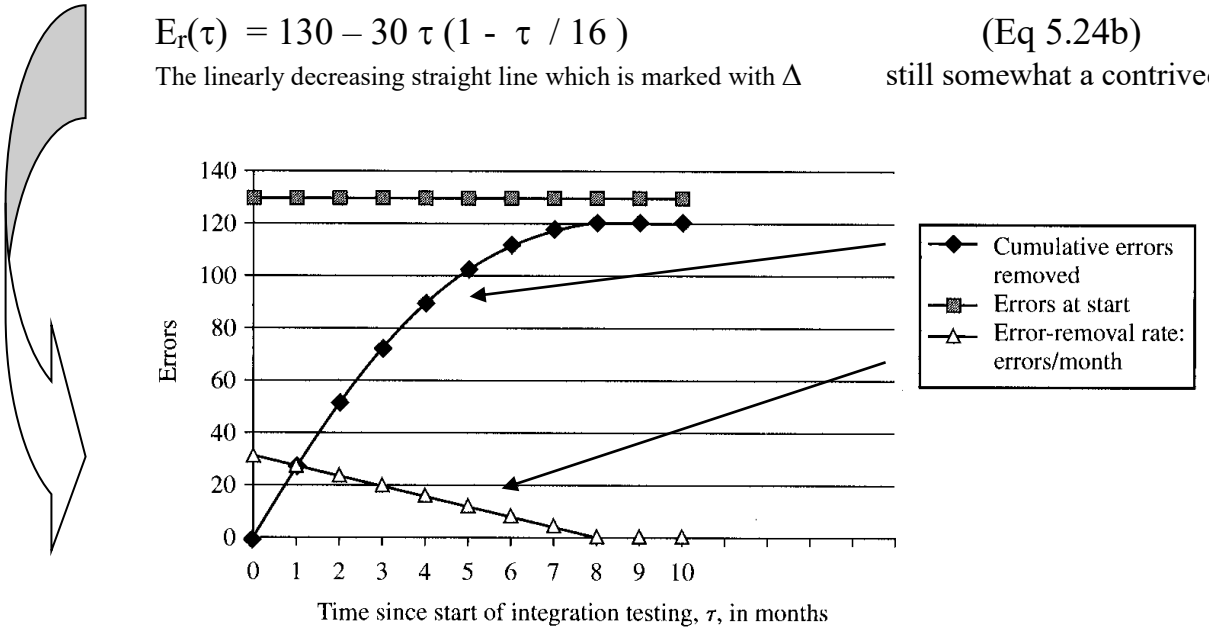


Figure 5.7 Illustration of a linearly decreasing error-removal rate.

3. Exponentially Decreasing Error-Removal Rate (see Figure 5.8 page 236)

$\mathbf{E_r(\tau) = E_T\, e^{-\alpha\tau}}$ where τ is the testing time interval and

α is a proportionality constant $d\,E_d(\tau) / d\tau = \alpha\, E_r(\tau)$ where $E_d(\tau)$ = errors detected

On page 235, similar example to the above cases for exponentially decreasing error-removal rate model.

Note that testing time is any time errors are formally detected/corrected which can be during development, integration, etc. Usually the longer the "testing" period, the better.

5.6 Reliability Models

Modeling attempts to answer: When should we stop testing? How reliable is the software? The answers are always a trade-off of risk, money and schedule.

Errors fixed in the 'field' normally cost ten times as much as those fixed during in-house testing → remember the cost of ownership problems which didn't consider the impacts of customer satisfaction (customer loyalty wrt buggy software).

Normally a good assumption: **Hazard rate(failure rate) $\propto$ remaining # of errors**

$$Z(\tau) = k\, E_r(\tau)$$

It seems reasonable; is supported by experimental data (empirical); and if the rate of errors found is a random process dependent on the input parameters and initial conditions, then the discovery rate is proportional to the number of remaining errors.

Combine $Z(\tau)$ with the 3 software error-removal models →3 software reliability models

$R(t) = e^{-\int Z(\tau)\, dt} = e^{-\int k\, Er(\tau)\, dt}$ where τ is the development/debugging/testing time
(Equation B39 or 5.13b)

Using the three models for error removal → develop three models for software reliability.

For errors remaining $E_r(\tau)$ then for

Constant Error-Removal Rate: $E_r(\tau) = E_T - \rho_o\tau$

Linearly Decreasing Error-Removal Rate: $E_r(\tau) = E_T - K\,\tau\,(1 - \tau / 2\,\tau_0)$

Exponentially Decreasing Error-Removal Rate: $E_r(\tau) = E_T\, e^{-\alpha\tau}$

An example using the Constant Error-Removal Rate model:

$E_r(\tau) = E_T - \rho_o\tau$ where $E_r(\tau)$ are the remaining errors, E_T are the total errors
ρ_o is the ***constant*** error-correction rate in units of errors/test time(τ) .

$Z(\tau) = k\, E_r(\tau) = k\,(E_T - \rho_o\tau)$ from the previous s/w error model discussions in Section 5.5

Eq 5.30a $R(t) = e^{-\int k(E_T - \rho_o \tau)\, dt} = e^{-k(E_T - \rho_o \tau)t}$ $\int a\, dt = a \int dt = a\, t$

Eq 5.30b $MTTF = \int_0^{\infty} R(t)\, dt = 1 / [k(E_T - \rho_o \tau)]$ from $R(t) = e^{-\lambda t}$ for constant λ

$\int e^{-a x}\, dx = e^{-a x} / -a$ along with these limits of integration

$e^0 = 1$ and $e^{\infty} = 1$ then $MTTF = 1 / a$

Summary:
Use the $E_r(\tau)$ models with the definition of R(t) to obtain the S/W Reliability Models

Constant Error-Removal Rate: $R(t) = e^{-k[E_T - \rho_o \tau]t}$ Section 5.6.2

Linearly Decreasing Error-Removal Rate: $R(t) = e^{-k[E_T - K\tau(1 - \tau / 2\tau_0)]t}$ Section 5.6.3

Exponentially Decreasing Error-Removal Rate: $R(t) = e^{-k[E_T e^{-\alpha\tau}]t}$ Section 5.6.4

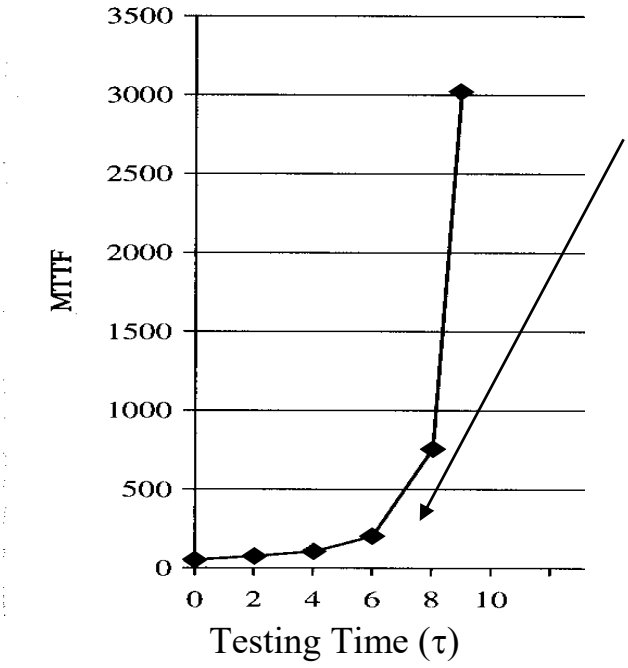
MTTF is the integral of R(t) which for the three error models is simply the reciprocal of the exponent given in the three above equations (*see development Eq. 5.30b above*).

for linearly decreasing for exponential

$MTTF = 1/k[E_T - K\tau(1 - \tau / 2\tau_0)]$ or $MTTF = 1/k[E_T e^{-\alpha\tau}]$

So how do you estimate the parameters used in the error-removal models? They will essentially be estimated from the early testing, previous experience with similar software and/or simulation data obtained from the pre-release software in a modeled operational environment.

Examining the constant error-removal model results (from the MTTF function above):

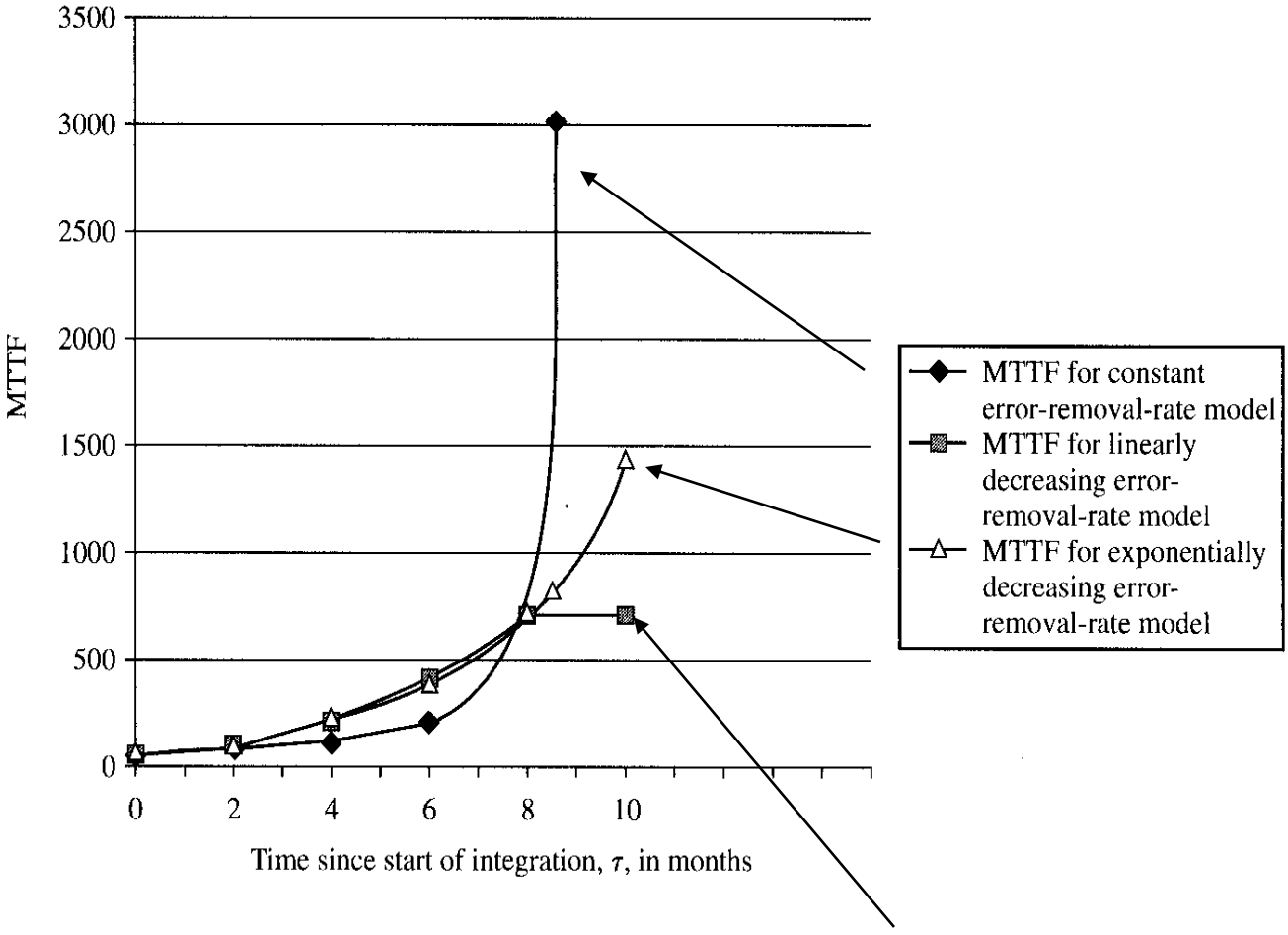


You are striving to understand the magnitude of the error problem, obtain a decent MTTF and spend enough time testing to get past the knee in the MTTF curve where the increase in reliability will be significantly improved with very little additional testing effort (having a small E_r and not wasting time/money eliminating the remaining ***few*** errors.) Constant-error model falls apart as remaining errors → 0 but the general result is the key (and who says you'll be lucky enough to get near 0 errors remaining anyway).

Reliability Model for the Linearly Decreasing Error-Removal Rate – model shows an intermediate testing time improvement in reliability because of the initially higher error-removal rate but there is a point (τ_0) where more testing fails to remove more errors, which is counterintuitive.

Reliability Model for Exponentially Decreasing Error-Removal Rate – model represents the hypothesis that in addition to the rate of error detection being proportional to the number of errors present (low hanging fruit), the model also fits the belief that the finding errors after reasonable testing is very difficult (later errors are subtle and deeply embedded).

The graph for all three MTTF models (Figure 5.17 on page 249)



Outcome of stopping testing at $\tau_0 = 8$ months for the linear decreasing error-removal rate model (E_r stays the same and error-removal rate is 0)

Data for any of the three models in the first two months pretty much yields the same results. The key difference is between $8 < \tau < 10$ months testing (the key period for these examples) but a more detailed model would also provide better results.

Months	Constant	Linear	Exp
0	130	130	130
1	115	101.9	94.3
2	100	77.5	68.5
3	85	56.9	49.7
4	70	40.0	36.1
5	55	26.9	26.2
6	40	17.5	19.0
7	25	11.9	13.8
8	10	10.0	10.0
9	-5	11.9	7.3
10	-20	17.5	5.3
11	-35	26.9	3.8
12	-50	40.0	2.8

ERRORS REMAINING

Models only valid up to 8 months of testing

Textbook Examples Pages 230 thru 235

Keeping track of the data over a long period of time for the same organization will improve the predictions from the initial data and may actually show that the constant-error model works better for 'your' organization because of the bias within the organization for collecting data (What is an error? Do I need to count that one? The model just seems to work for our company and the software that is developed.)

Estimating the Model Constants (k, E_T, ρ_o, etc)

The bottom line is "*always have a process in place to collect the s/w project data* "

Parameter Estimation - curve fitting to experimental data or statistical parameter estimation

When there is little detailed data at the beginning of a project, the predictions will have a wide range of uncertainty but as we've seen, this is useful in any removal-rate model.

If we get into trouble toward the end of the project, the reliability models can predict quantitatively how much effort will be required to remedy the problem (management will always ask how bad is it? How much time do you need to fix the problem?)

If the error-removal rate is getting small, the model might show the product can be released early (time incentive contract or respond to competition)

Handbook Estimation – reliability model constant estimation based on past experience. Another reason good software organizations have extensive past project data. Information and data collection is critical in today's environment – the information age.

Moment Estimates for Constant, Linearly Decreasing and Exponentially Decreasing: Using the Error-Removal-Rate Data – take a look at the error-removal rate versus τ and see how it fits all three models. Pick a model. The first moment of the data or the mean will provide the first moment of the model, the second moment of the data is equated to the second moment of the model, etc. Then compute the # of moments needed to solve for the # of parameters to be estimated.

Operational data provides the best estimates but if you don't have operational data (very likely) then you could obtain the data from a simulated operational mode → run the software that is available for reliability testing in a simulated operational environment until it fails. This is not testing since testing/debugging does not accrue time to failure.

Obtain as many parameters (model constants) as possible from the error-removal data and then use the test data to estimate the remaining.

As the project progresses and more error-removal data and operational time is accumulated, develop and maintain several models to make comparisons and ascertain which model is providing the best estimates (reliability and MTTF).

Moment estimates don't provide a clear direction when several data sets are available (for the difference methods used between two data sets for example).

Least-Squares Estimates use all of the data as does Maximum-Likelihood Estimates (MLE) giving us three model parameter estimation techniques.

These more complex estimation techniques are readily available in computer based mathematical packages but they provide little insight into the methods and the usefulness of the data available (garbage in → garbage out). Initial data analysis should be done with calculators, graph paper, i.e., paper and pencil (to gain some insight)

5.7 Software Reliability Models

It is easy to argue against using S/W reliability models (see text). Not enough time (money) to do it right but always enough time (money) to fix it.

Development of s/w reliability models requires an effort but if done correctly, it will provide significant insight into software projects. Data collection is the key ingredient.

Software has become far to complex not to require use of numerous analytical tools. Software Cost Estimation Models provide insight into reliability considerations (COCOMO model, which uses Raleigh manpower distributions, takes into account numerous factors that not only effect s/w cost but s/w reliability – personal attributes like quality & continuity, schedule & budget constraints, software language, requirements volatility, hardware configuration, testing, interface complexity, size of project, etc.)

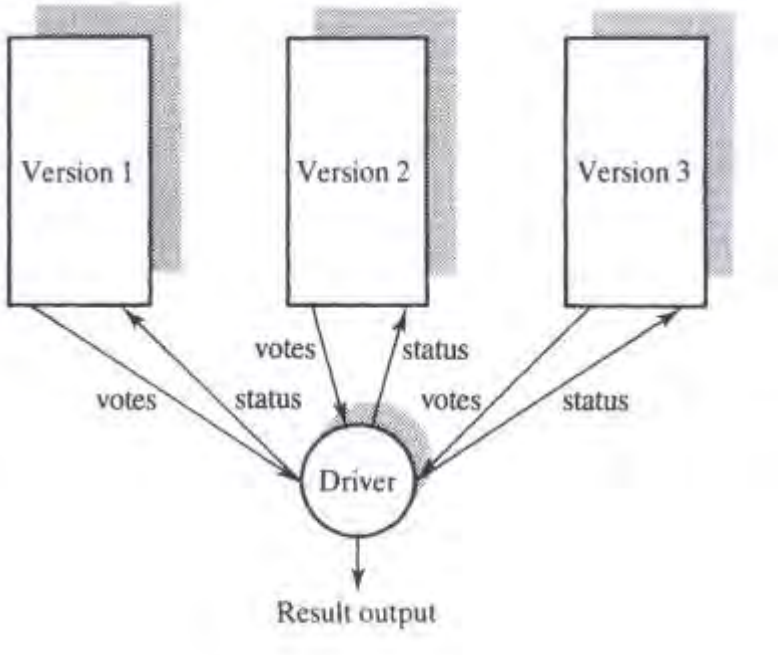
Using development test data vs simulated operational data – best to use any and all data combined with some weighting factor for the realistic operational hours w/o failure.

Start simple – finish complex. Textbook examples were just the simple end of the S/W reliability spectrum.

5.9 **Software Redundancy**

N-Version Programming – the TMR of the software world. The s/w modules must be different for the N-Versions to be independent and produce outputs for meaningful voting.

If independent then the software versions are comparable to the hardware modules used in the TMR mathematical formulations and the same equations are applicable.



The different versions come from the same set of requirements (hopefully correct reqt's)

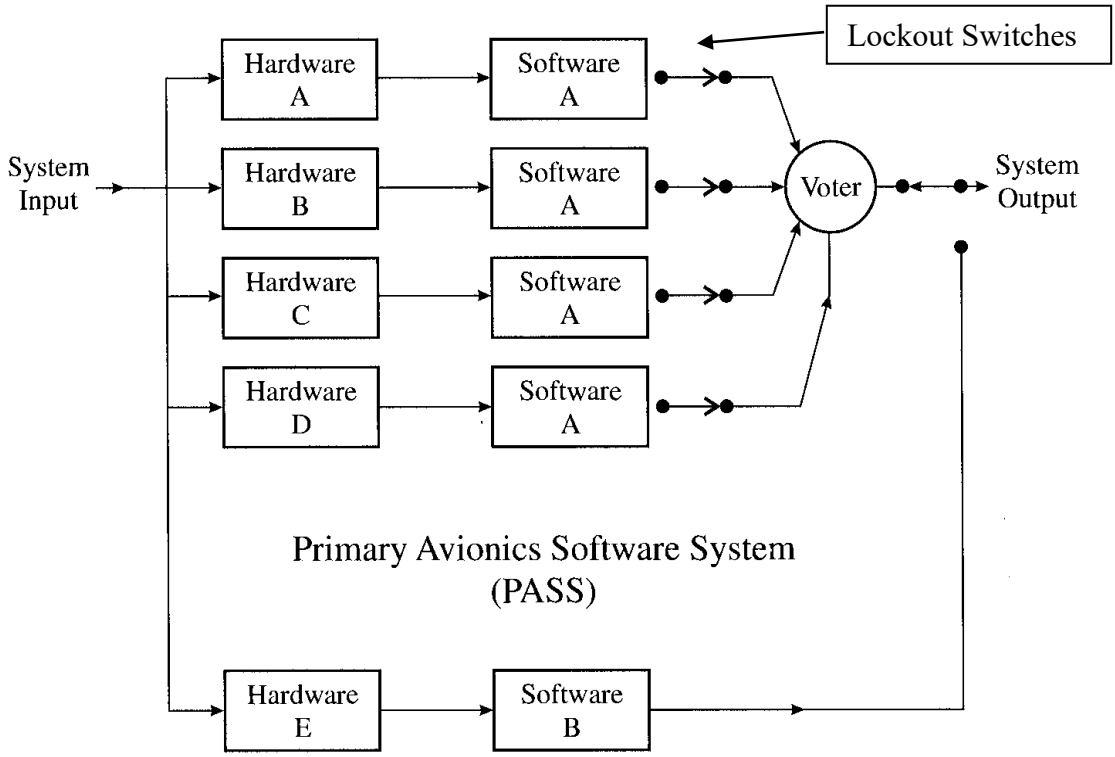
Diverse Programming – independently developed versions concurrently executed
Problems: cost, lengthly development, how to resolve differences (independently)
Atomic Processes – modules don't interfere with each other

Programmers of the different versions must work independently of the others

Each version of the s/w is run through the same acceptance tests.

Common-Mode failures – errors that affect all versions/modules. Example: Errors in the requirements can create common-mode failures in all the versions. Or unexpected data value inputs (not considered) can cause common-mode failures.

Space Transportation System (STS) – Shuttle (Orbiter)



Five Identical Computers (CPU and I/O) – essentially NMR/Simplex arrangements

Only one computer's output is used – lockout switches are used at the inputs to the voter after the voting process has competed (actually a time sync comparison process)

BFS – protects against a generic software failure since the same software is run on the PASS computers so an independent version (just critical phases) is used on the BFS. It is developed using the same set of requirements (common-mode failure ??). The BFS is manually engaged so if the Commander waits to long to engage the BFS → it could result in loss of the vehicle. Testing was done to determine how long was too long.

NASA's Mission Control Center (MCC) has a lot of insight into the internal operations of each computer via telemetry which should increase the reliability with more "eyes" on the operations. MCC could also act as an independent voting source during critical phases. This made the communications system the most critical system on the vehicle. Overall vehicle design protects for total loss of communications, since the vehicles could operate independently from the ground with their on-board systems in addition to hand held backup solutions (programmable calculators on Apollo/laptops on Shuttle).

Built-In Test (BIT) and Self-Test Features of the Shuttle's AP-101 (catch most faults)

Watchdog Timers – processors set a timer at the start of a process and if the timer counts down to zero before being reset – the computer is labeled as failed and is locked out.

Comparisons – check sum is computed and two successive miscompares fail the computer

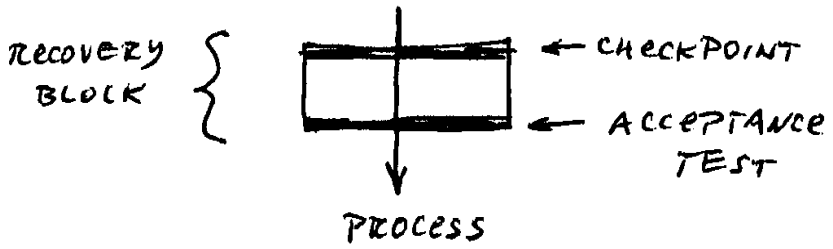
Bus Time-Out Tests – if the computer does not perform a periodic operation on the bus and the timer has expired, the computer is considered failed (and/or the computer bus has failed).

Rollback and Recovery

These type of recovery techniques are a **backward** error recovery. Generally assume that the software problems are transient errors and thus the software can be 're-exectued'.

Forward Error Recovery – continue operation tolerating loss that is dealt with later,
e.g. reconfiguration.

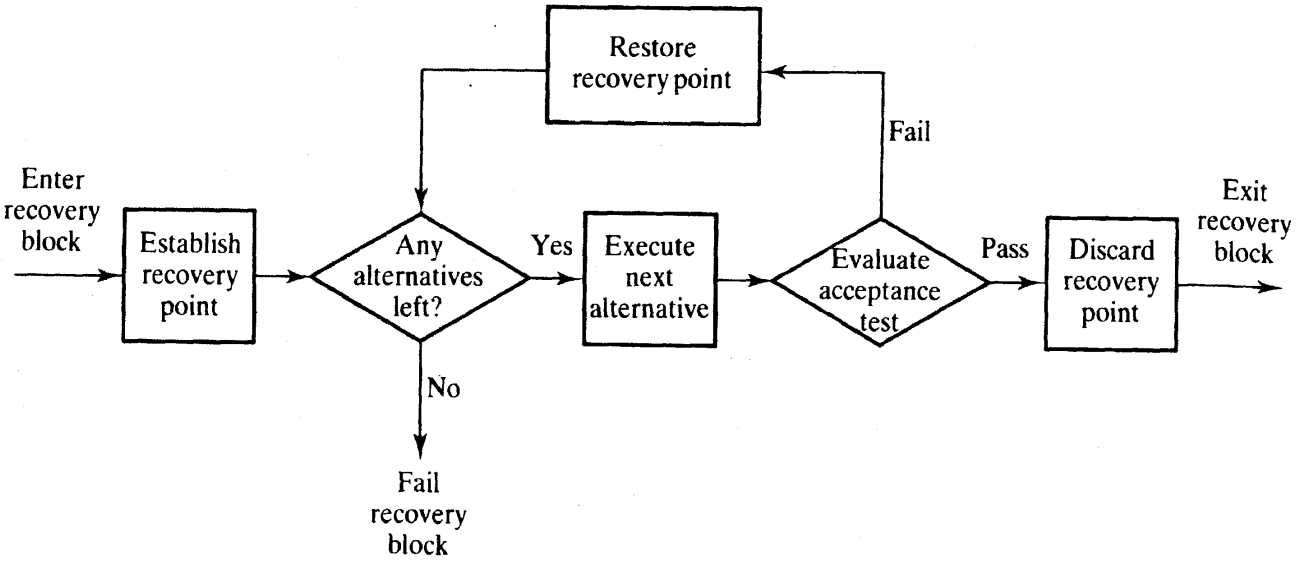
Recovery Blocks



The failure of the Acceptance Test cycles the program back to the Checkpoint and an alternative module is executed through the same process

Code Example:

```
ensure <acceptance test>
by
        <primary module>
else by
        <alternative module>
else by
        <alternative module>
.....
.....
else by
        <alternative module>
else
        <error handling module>
```



Recovery Blocks vs N-Module Programming

N-Module – static redundancy

Recovery Blocks – dynamic redundancy, error is detected

Overhead – both methods incur extra development costs

Diversity – both schemes are diverse but both susceptible to the same specification errors

Error Detection – N-Module Voting → voting (less overhead)

Recovery Blocks → acceptance testing (more flexible)

5.10.2 **Rebooting**

A weak and sometime futile recovery technique (who says the problem isn't going to happened again?) But it is a simple (brute force) technique if you have the time (rebooting on the Space Station takes 10 minutes).

When the entire software process is lost (watchdog timer ?), execute a last-ditch recovery technique like rebooting or go back to the beginning process (start over/restart)
Example: Autonomous submarine – back to the surface and stop to await recovery

5.10.4 **Journalization Techniques**

All transactions (inputs) are saved during execution available to be re-executed

The state of the process is saved (stored) at the beginning so the transactions can be rerun, possibly just up to the point that something went 'bad'

5.10.5 **Retry**

Usually implemented in hardware, when an error is detected, a retry is begun immediately or possibly a small execution delay awaiting a transient to decay.

5.10.6 **Checkpointing**

A software technique - Saving the system state, rollback on error which infers error latency, process restarted again from the checkpointed state. If this fails, can possibly go back to an even earlier checkpoint. Example: the Windows OS restore points.

If the 'error' has permanently modified the state of the system, the checkpoint technique fails and an error recovery process is most likely initiated (or you give up and go home).

5.10.7 **Distributed Systems**

Client/Server systems, connected by LANs.
Reliability thru network connectivity which provides extensive redundancy.

Client – user at a PC, the local environment
Server – connected to many clients and for redundancy other servers

In Distributed Systems, designs must avoid collisions (for data) and lockups (A waiting on B, waiting on C who is waiting on A)

A Fault Tolerant Program Description

implemented by Pat Lorenzo, CENG 5334 Graduate student Spring 2003
improved by Dave Ayan, CENG 5334 Teaching Assistant Spring 2004

Implementation Requirements

- A simple fault tolerant system that will sort an array of integers.
  - ✓ 3 Different Sort Algorithms
  - ✓ 2 Fault Tolerant Schemes
  - ✓ Adjudicator (Recovery Block's Acceptance Test)
  - ✓ Voter (TMR Scheme, data alignment realized by using integers versus floating point)
- Use Error Injection to evaluate the two schemes
- Use Objected Oriented Programming (OOP) techniques in the software design and implementation (encapsulation of Methods and Data Members using derived classes with inherited functions)
- Design system for possible future use in parallel processing environments

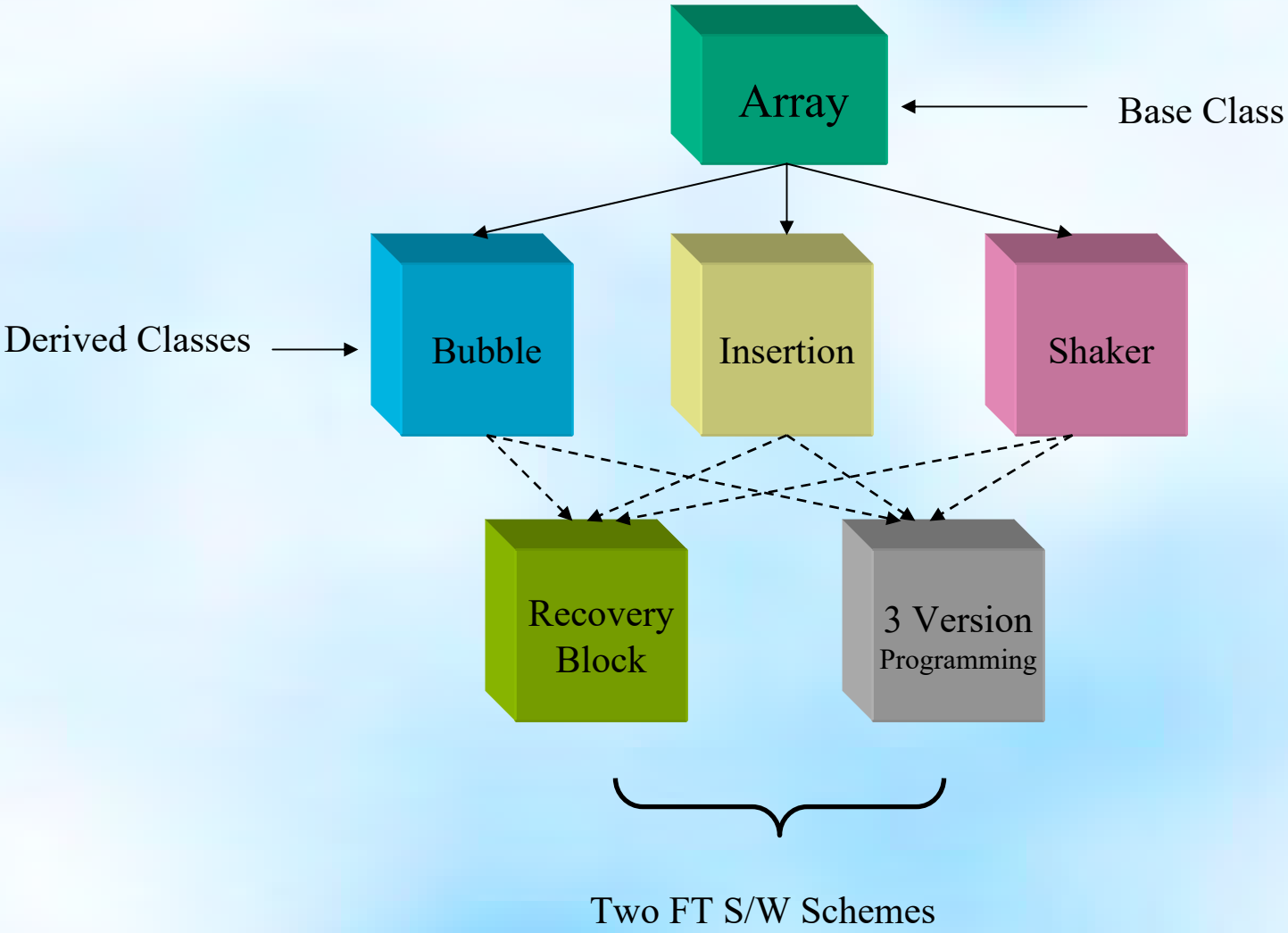
Fault Tolerant Program Description

implemented by Pat Lorenzo, CENG 5334 Graduate Student Spring 2003
improved by Ayan Dave, CENG 5334 Teaching Assistant Spring 2004

Implement

- ✓ A simple fault tolerant system that will sort an array of integers.
- ✓ 3 Different Sort Algorithms
- ✓ 2 Fault Tolerant Schemes
- ✓ Adjudicator (Recovery Block's Acceptance Test)
- ✓ Voter (TMR Scheme, input data alignment realized by using integers versus floating point numbers)
- ✓ Use Error Injection to evaluate the two schemes
- ✓ Use Objected Oriented Programming (OOP) techniques in the software design and implementation (encapsulation of Methods and Data Members using derived classes with inherited functions)
- ✓ Design system for future use in parallel processing

Modules/Classes



Array Class

```
class Array {
public:
    Array(int user_array[], int = 0);
    ~Array();
    virtual void Sort() = 0;
    void Print();
protected:
    int *List;
    int Size;
};
```

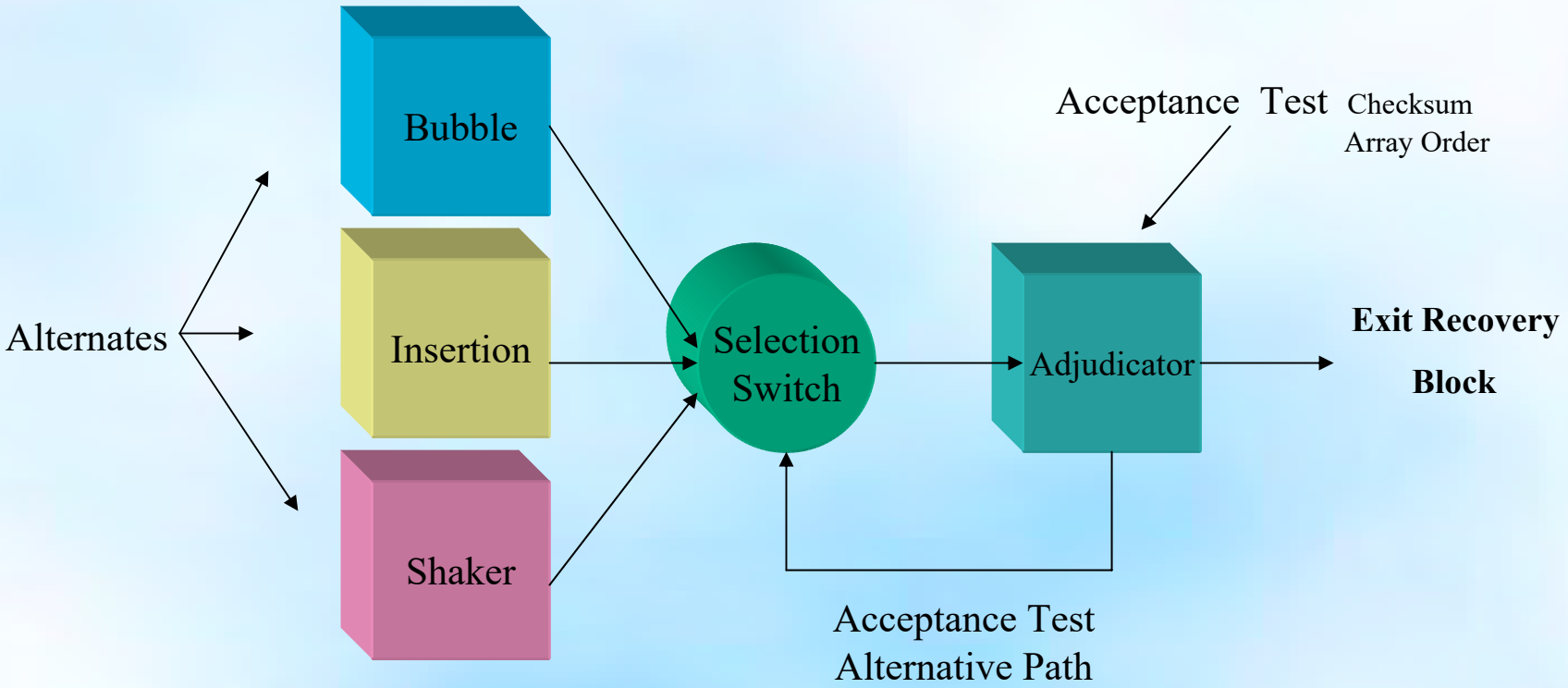
```
Array::Array(int user_array[], int
    user_size)
{
    Size = user_size;
    List = new int[Size];
    for(int i = 0; i < Size; i++)
         List[i] = user_array[i];
}
void Array::Print()
{
    for(int i = 0; i < Size; i++)
         cout << setw(4) << List[i];
         cout << endl;
}
Array::~Array()
{
    delete [] List;
}
```

Bubble Class

```
class Bubble: public Array {
public:
    Bubble(int *, int = 0);
    virtual void Sort();
};
```

```
Bubble::Bubble(int *user_array, int user_size)
:Array(user_array, user_size)
{
    Sort();
    for(int i = 0; i < Size; i++)
        user_array[i] = List[i];
}
void Bubble::Sort()
{
    int temp;
    for(int a=1; a < Size; ++a) {
        for(int b = Size-1; b >= a; --b) {
            if(List[b-1] > List[b]) {
                temp = List[b-1];
                List[b-1] = List[b];
                List[b] = temp;
            }
        }
    }
}
```

Recovery Block



Recovery Block (RB) Code

```
void RBarray::Sort()
{
     int *temp = new int[Size];
     int CheckSum = 0;

     for(int i = 0; i < Size; i++) CheckSum += List[i];

     for(int version = 0; version < 3; version++) {
          if(!Success) {

               for(i = 0; i < Size; i++)  temp[i] = List[i];

               if     (version == 0) Bubble    version1(temp, Size);
               else if(version == 1) Insertion version2(temp, Size);
               else if(version == 2) Shaker    version3(temp, Size);

               Success = Adjudicator(temp, CheckSum);
          }
     }
     if(Success) {
          for(i = 0; i < Size; i++) List[i] = temp[i];
     }
     delete [] temp;
}
```

Adjudicator Code

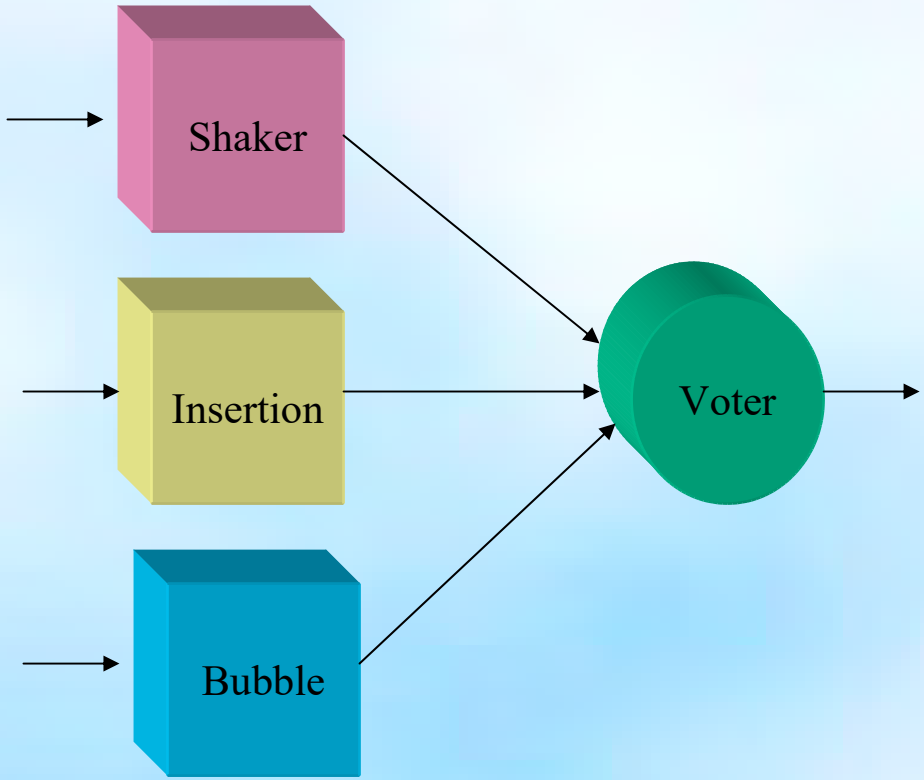
two fold test – checksum and array order

```
bool RBarray::Adjudicator(int *temp, int CheckSum)
{
    int reCheckSum = 0;
    for(int i = 0; i < Size-1; i++) {
          if(temp[i] > temp[i+1]) return false;
          reCheckSum += temp[i];
    }
    reCheckSum += temp[Size-1];
    if(reCheckSum != CheckSum) return false;

    return true;
}
```

N-Version Programming

3 Modules (TMR)



3 Version Code

```
void TMRarray::Sort()
{
     int *ver1 = new int[Size];
     int *ver2 = new int[Size];
     int *ver3 = new int[Size];
     int i, select = 0;

     for(i = 0; i < Size; i++) ver1[i] = List[i];
     for(i = 0; i < Size; i++) ver2[i] = List[i];
     for(i = 0; i < Size; i++) ver3[i] = List[i];

     Bubble    Version1(ver1, Size);
     Insertion Version2(ver2, Size);
     Shaker    Version3(ver3, Size);

     select = Voter(ver1,ver2,ver3);
     switch(select) {
          case 1: for(i=0;i<Size;i++) List[i] = ver1[i]; break;
          case 2: for(i=0;i<Size;i++) List[i] = ver2[i]; break;
          case 3: for(i=0;i<Size;i++) List[i] = ver3[i]; break;
          default: Success = false;
     }
     delete [] ver1;
     delete [] ver2;
     delete [] ver3;
}
```

NMR Voter Code

voting performed on checking array order amongst all 3 schemes

```
int TMRarray::Voter(int *ver1, int *ver2, int *ver3)
{
    bool same12,same13,same23;
    int i;

    same12 = true;
    same13 = true;
    same23 = true;

    for(i = 0; i < Size; i++) {
          if(ver1[i] != ver2[i]) { same12 = false; break;}
    }
    if(same12) return 1;
    for(i = 0; i < Size; i++) {
          if(ver1[i] != ver3[i]) { same13 = false; break;}
    }
    if(same13) return 1;
    for(i = 0; i < Size; i++) {
          if(ver2[i] != ver3[i]) {same23 = false; break;}
    }
    if(same23) return 2;
    return 0;
}
```

Conclusion

- An Object Oriented Approach (OOP) allowed better isolation of modules into objects.
- In the Recovery Block Scheme, the Adjudicator (acceptance test) is always application *dependent* while the Voter in an N-Version Programming scheme is *independent* of the application.
- The complexity of the design of the adjudicator is proportional to complexity of the application. The voter design in the N Version Programming scheme remains the same although it may require some individualized alignment based on the data members and the module outputs. Having integers in this FT Software demonstration avoided this problem.
- N-Version Programming is the generally the simpler of the two.
- The Recovery Block scheme infers some ability to go back in time by performing another scheme if a previous scheme fails the acceptance test. Processing speed may make this concern moot for real-time systems.

No Error Case

Forming the arrays ... Bubble Sorting the Array ... Sorting Done ...

Bubble Algorithm

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Insertion Sorting the Array ... Sorting Done ...

Insertion Algorithm

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Shaker Sorting the Array ... Sorting Done ...

Shaker Algorithm

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

--- RECOVERY BLOCK ---

The real unsorted array ... 3 7 9 1 11 2 16 8 6 12 14 5 13 4 10 15

Checksum of the List is : 136

Trying Version 1 ... Bubble Sort ...

Bubble Sorting the Array ... Sorting Done ... Adjudicator : Success

Recovery Block (Success)

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

--- TMR VOTING ---

Bubble Sorting the Array ... Sorting Done ...

Insertion Sorting the Array ... Sorting Done ...

Shaker Sorting the Array ... Sorting Done ...

N Version Programming (Success)

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Error in Bubble Sort

Forming the arrays ... Bubble Sorting the Array ... Sorting Done ...
Bubble Algorithm
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

Insertion Sorting the Array ... Sorting Done ...
Insertion Algorithm
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Shaker Sorting the Array ... Sorting Done ...
Shaker Algorithm
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

--- RECOVERY BLOCK ---

The real unsorted array ...
3 7 9 1 11 2 16 8 6 12 14 5 13 4 10 15
Checksum of the List is : 136
Trying Version 1 ... Bubble Sort ... Bubble Sorting the Array ... Sorting Done ...
Adjudicator : Fail ... try next version

Trying Version 1 ... Insertion Sort ... Insertion Sorting the Array ... Sorting Done ...
Adjudicator : Success

Recovery Block (Success)
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

--- TMR VOTING ---

Bubble Sorting the Array ... Sorting Done ...
Insertion Sorting the Array ... Sorting Done ...
Shaker Sorting the Array ... Sorting Done ...

N Version Programming (Success)
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

```
VAL Parameter 2 – Error in Shaker Sort

Forming the arrays ...
Bubble Sorting the Array ...
Sorting Done ...

 Bubble Algorithm
  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
Insertion Sorting the Array ...
Sorting Done ...

 Insertion Algorithm
  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
Shaker Sorting the Array ...
Sorting Done ...

 Shaker Algorithm
  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17


            --- RECOVERY BLOCK ---

The real unsorted array ...
 3  7  9  1  11  2  16  8  6  12  14  5  13  4  10  15
Checksum of the List is : 136
      Trying Version 1 ... Bubble Sort ...
Bubble Sorting the Array ...
Sorting Done ...
      Adjudicator : Success

 Recovery Block  (Success)
  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16


            --- TMR VOTING ---

Bubble Sorting the Array ...
Sorting Done ...
Insertion Sorting the Array ...
Sorting Done ...
Shaker Sorting the Array ...
Sorting Done ...

 N Version Programming  (Success)
  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
```