UTD 2012 REU Summer Program on Software Safety

Bhanu Kapoor, PhD
Adjunct Faculty, Department of Computer Science
UTD, Dallas, TX
bhanu.kapoor@utdallas.edu, 214-336-4973

June 04-05, 2012
UTD, Dallas, TX
Lecture Notes

Software Requirements

- Introduction to Software Requirements
  - How is Software Developed?
  - Software Development Life Cycle
- Problems with Software Requirements
  - Types of Requirements: Library System
  - Stakeholders: Tree Swing
  - Smartphone Requirements
- Tracking Requirements
  - Quality Function Deployment
  - Apple iPhone 4S Case Study

Software Requirements

- Requirements & Specification
  - Formal Approach
  - IEEE Standard: Software Requirement Spec.
- Non-functional Requirements
  - Software Security, Reliability, and Safety
  - Improving Software Safety with Fault-Tolerance

Software Requirements

- Introduction to Software Requirements
  - How is Software Developed?
  - Software Development Life Cycle
- Problems with Software Requirements
  - Types of Requirements: Library System
  - Stakeholders: Tree Swing
  - Smartphone Requirements
- Tracking Requirements
  - Quality Function Deployment
  - Apple iPhone 4S Case Study

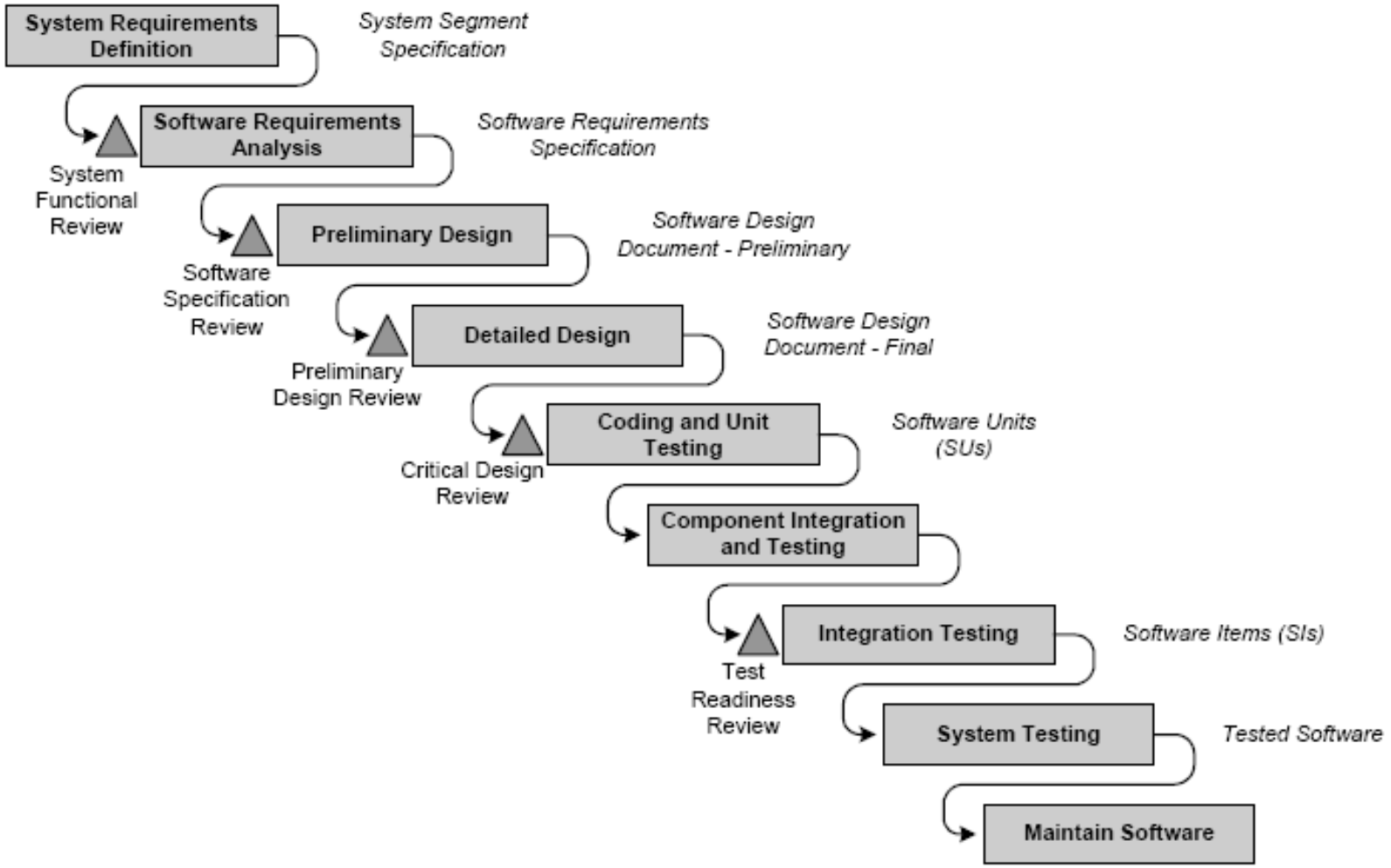
Software Development Life Cycle

- Need Determination
- Concept Definition and Demonstration
- Development
- Testing
- Deployment
- Operations and Maintenance

Software Development Life-Cycle (SDLC) Models

- Waterfall
- Incremental
- Evolutionary
- Spiral

Waterfall Model



Waterfall: Advantages

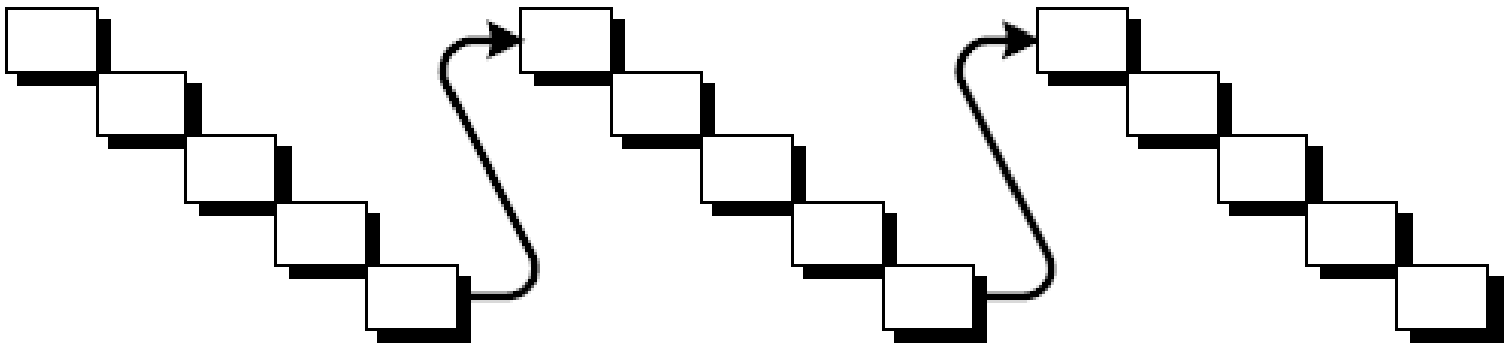
- System is well documented.
- Phases correspond with project management phases.
- Cost and schedule estimates may be more accurate.
- Details can be addressed with more engineering effort if software is large or complex.

Waterfall: Disadvantages

- All risks must be dealt with in a single software development effort.
- Because the model is sequential, there is only local feedback at the transition between phases.
- A working product is not available until late in the project.
- Progress and success are not observable until the later stages.
- Corrections must often wait for the maintenance phase.

Incremental

- A series of waterfalls
- Collect requirements initially
- Different builds address requirements incrementally



Incremental: Advantages

- Provides some feedback, allowing later development cycles to learn from previous cycles.
- Requirements are relatively stable and may be better understood with each increment.
- Allows some requirements modification and may allow the addition of new requirements.
- It is more responsive to user needs than the waterfall model.

Incremental: Advantages

- A usable product is available with the first release, and each cycle results in greater functionality.
- The project can be stopped any time after the first cycle and leave a working product.
- Risk is spread out over multiple cycles.
- This method can usually be performed with fewer people than the waterfall model.

Incremental: Advantages

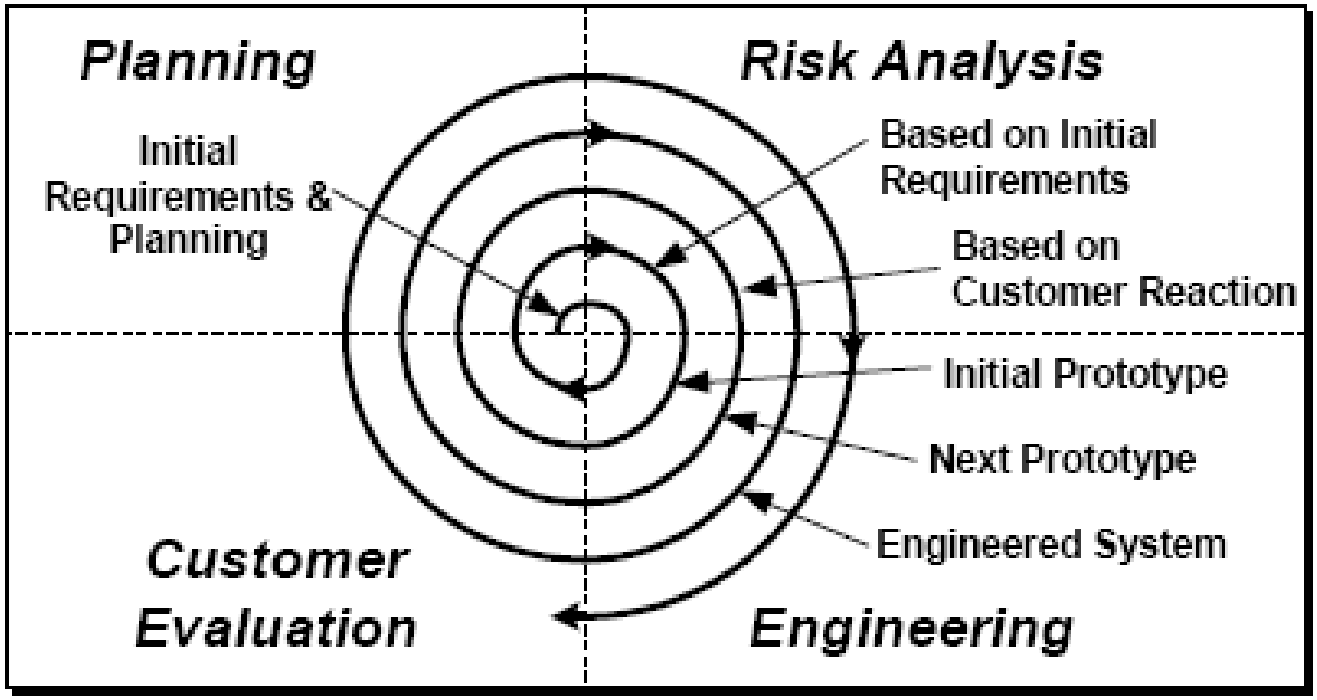
- Return on investment is visible earlier in the project.
- Project management may be easier for smaller, incremental projects.
- Testing may be easier on smaller portions of the system.

Incremental: Disadvantages

- Formal reviews may be more difficult to implement on incremental releases.
- Interfaces between modules must be well-defined in the beginning.
- Cost and schedule overruns may result in an unfinished system.
- Operations are impacted as each new release is deployed.
- Users are required to learn how to use a new system with each deployment.

Evolutionary

- Requirements evolve as system is used



Evolutionary: Advantages

- Project can begin without fully defining or understanding requirements.
- Final requirements are improved and more in line with real user needs.
- Risks are spread over multiple software builds and controlled better.
- Operational capability is achieved earlier in the program.
- Newer technology can be incorporated into the system as it becomes available during later prototypes.

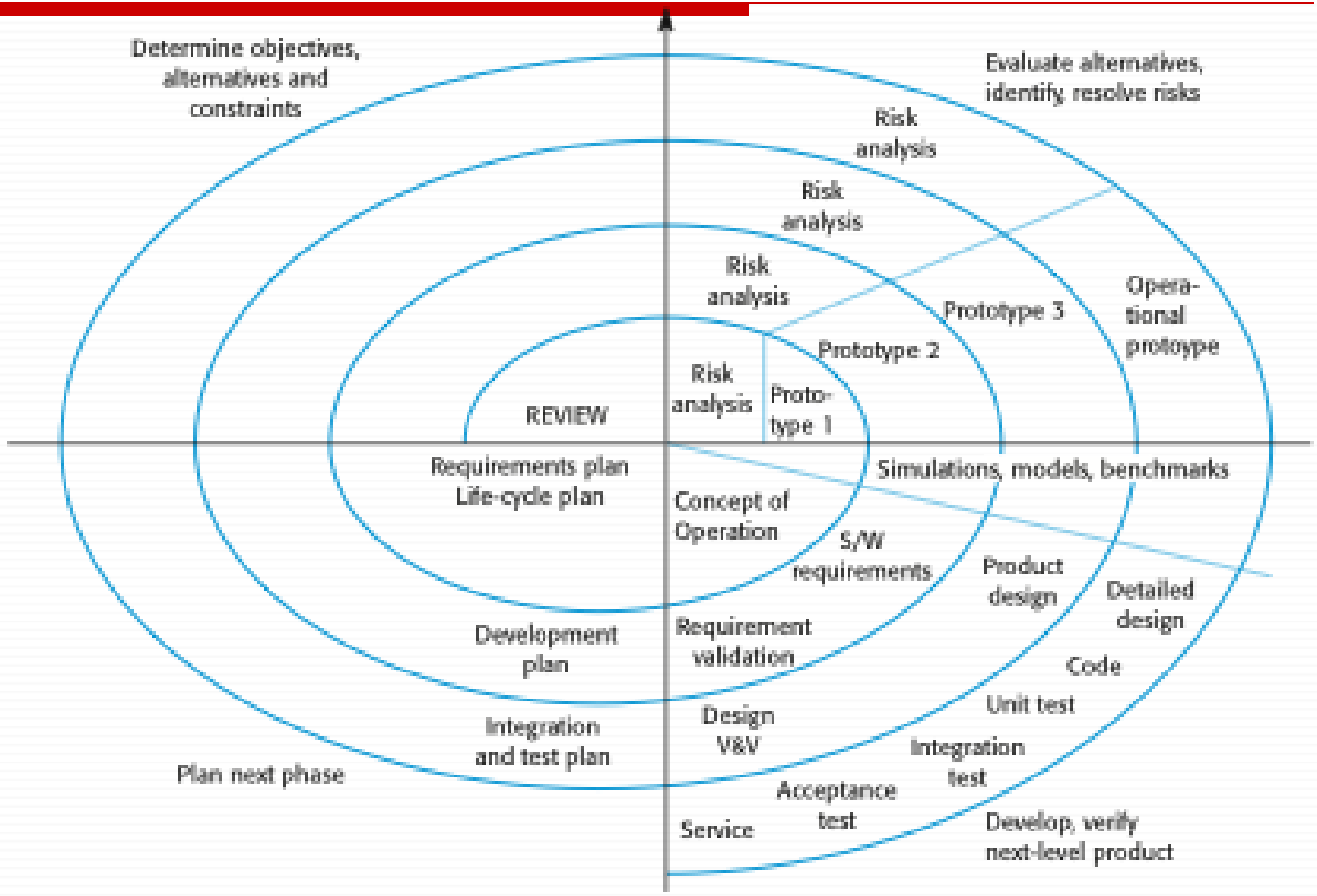
Evolutionary: Disadvantages

- Usually an increase in both cost and schedule over the waterfall method.
- Management activities are increased.
- Configuration management activities are increased.
- Greater coordination of resources is required.
- Prototypes change between cycles, adding a learning curve for developers and users.

Spiral

- Addresses risk incrementally
- Determines objectives and constraints
- Evaluate alternatives
- Identify risks
- Resolves risks by assigning priorities
- Develop a series of prototypes for identified risks, start with highest risk
- Waterfall for each prototype development
- Progress with risk resolution, else end.

Spiral


Determine objectives, alternatives and constraints
Evaluate alternatives, identify, resolve risks
Risk analysis
Risk analysis
Risk analysis
Risk analysis
Prototype 3
Opera-tional prototype
Prototype 2
Proto-type 1
REVIEW
Requirements plan
Life-cycle plan
Simulations, models, benchmarks
Concept of Operation
S/W requirements
Product design
Detailed design
Development plan
Requirement validation
Code
Unit test
Design V&V
Integration and test plan
Integration test
Acceptance test
Plan next phase
Service
Develop, verify next-level product

Spiral Model

Advantages

- It provides better risk management than other models.
- Requirements are better defined.
- System is more responsive to user needs.

Disadvantages

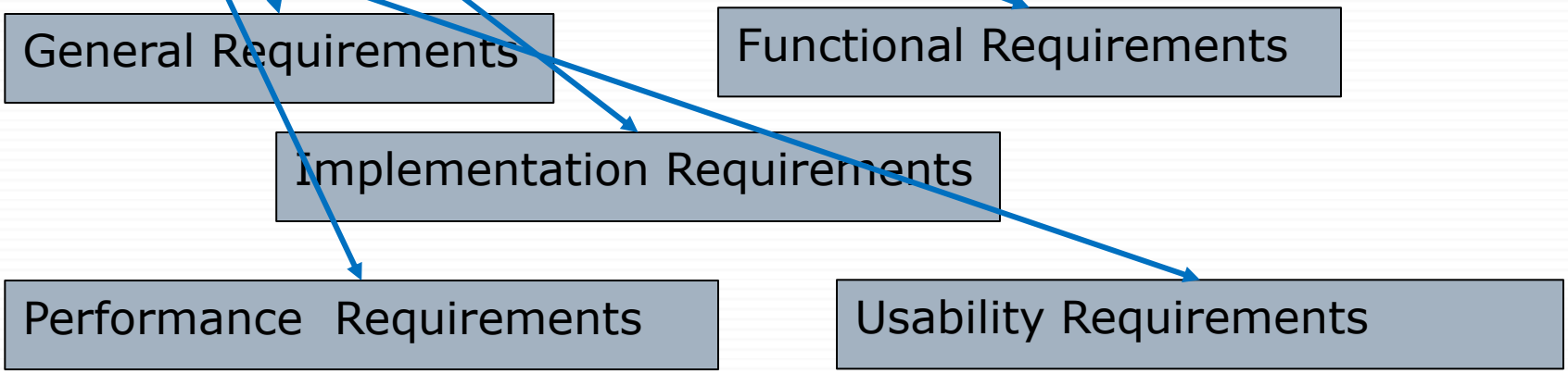
- The spiral model is more complex and harder to manage.
- This method usually increases development costs and schedule.

Software Requirements

- Introduction to Software Requirements
  - How is Software Developed?
  - Software Development Life Cycle
- Problems with Software Requirements
  - Types of Requirements: Library System
  - Stakeholders: Tree Swing
  - Smartphone Requirements
- Tracking Requirements
  - Quality Function Deployment
  - Apple iPhone 4S Case Study

Problem with Requirements

- Library System
  - System maintains record of all library items
  - Allows users to search by title, author, ISBN
  - User interface via web browser
  - System supports 20 transactions per second
  - Facilities demonstrable in 10 minutes or less



Problems with Requirements

- We have trouble understanding the requirements that we do acquire from the customer
- We often record requirements in a disorganized manner
- We spend far too little time verifying what we do record
- We allow change to control us, rather than establishing mechanisms to control change
- Most importantly, we fail to establish a solid foundation for the system or software that the user wants built

(Source: Pressman, R. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 2005)

Problems with Requirements

- Many software developers argue that
  - Building software is so compelling that we want to jump right in
  - Things will become clear as we build the software
  - Things change so rapidly that requirements engineering is a waste of time
  - The bottom line is producing a working program and that all else is secondary
- All of these arguments contain some truth, especially for small projects
- However, as software grows in size and complexity, these arguments begin to fail

Problems with Requirements

- Many different kind of requirements
- No standard way of writing requirements
  - Application domain dependent
  - Writer dependent
  - Reader dependent
  - Organization practices
- What is required of system may include
  - General information about type of system
  - Information about standards to adhere to
  - Information about other interacting systems

Problems with Requirements

- Requirements at the root of software engineering problems
  - Real needs of customer not reflected
    - Misunderstanding between customer, marketing, and developer
  - Inconsistent or incomplete requirements
    - Allows users to search by title, author, ISBN
- Requirement problems are universal
  - Human issues, impossible to be accurate
  - Good practices reduce issues
  - Requirements engineering is about good practices

Requirements Process

- Requirements in Software Lifecycle
  - Initial phase
  - May span the entire life cycle
- Essential Requirements Process Steps
  - Understand the problem
    - elicitation
  - Formally describe the problem
    - specification, modeling
  - Attain agreement on the nature of problem
    - validation, conflict resolution, negotiation
    - requirements management - maintain the agreement!
  - Sequential or iterative/incremental

Requirements Elicitation

- Four Dimensions
  - Application Domain Knowledge
    - Cataloguing System → Knowledge of Library
    - Knowledge can be present in multiple places
  - Problem Understanding
    - Cataloguing System → How Library organizes?
    - People who understand the problem are busy
  - Business Understanding
    - Organization issues may influence the requirements
  - Needs of Stakeholders
    - General knowledge, difficult to articulate

Problems with Elicitation

- Scope
- Volatility
- Understanding

Scope

- Boundary of system ill-defined
- Unnecessary design information may be given
- Focus on creation of requirements and not on design activities
  - Users may not understand design language
  - Such a focus may not reflect user needs

Scope

- Organizational Factors
  - Input providers
  - Users of target system
  - Managers of users
  - How target system will change organization's means of doing business?
- Environmental Factors
  - Accurate description of users
  - Accurate description of environment
  - H/W or S/W constraints imposed
  - Interfaces to the larger system
  - Role in larger system

Volatility

- Requirements Change
  - User needs may change over time
  - They may evolve over time
    - Iterative nature of RE process
    - Conflicting and changing needs of stakeholders
    - Political climate may change
- You cannot complete requirements capture before the design stage

Understanding

- Understanding issues lead to requirements that are:
  - Ambiguous
  - Incomplete
  - Inconsistent
  - Incorrect
- Reasons
  - Variety of background
  - Experience levels
  - Language too formal or informal
  - Amount of information

Understanding

- Stakeholders
  - Sponsors
  - Users
  - Developers
  - Quality Assurance
  - Requirement Analysts
  - Managers of users

Tree Swing

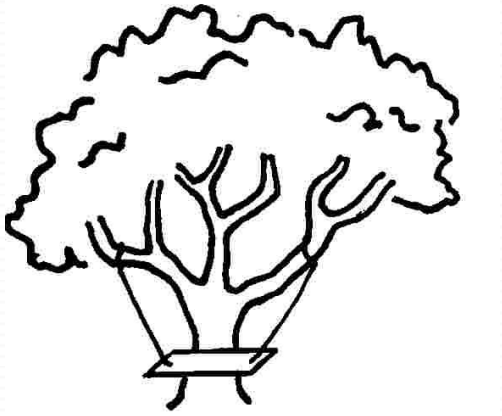
- What marketing suggested

- What management approved

Tree Swing

- What engineering designed

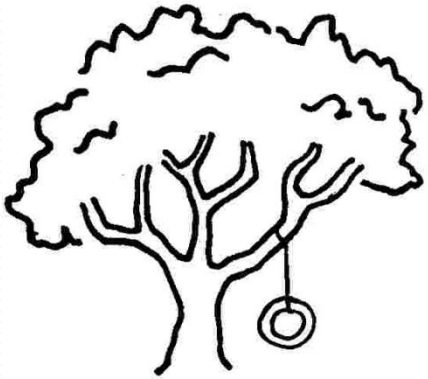
- What was manufactured



Tree Swing

- As maintenance installed it

- What the customer wanted



Importance of Requirements

- Engineering Argument
  - A good solution can only be developed if the engineer has a solid understanding of the problem.
- Economic Argument
  - Defects are cheaper to remove if are found earlier.
- Empirical Argument
  - Failure to understand and manage requirements is the biggest single cause of cost and schedule over-runs.
- Safety Argument
  - Safety-related software errors arise most often from inadequate or misunderstood requirements
- … …

Software Requirements

- Introduction to Software Requirements
  - How is Software Developed?
  - Software Development Life Cycle
- Problems with Software Requirements
  - Types of Requirements: Library System
  - Stakeholders: Tree Swing
- Tracking Requirements
  - Quality Function Deployment
  - Apple iPhone 4S Case Study

Quality Function Deployment (QFD)

- Developed in Japan in the mid 1970s
- Introduced in USA in the late 1980s
- Toyota was able to reduce 60% of cost to bring a new car model to market
- Toyota decreased 1/3 of its development time
- Used in cross functional teams
- Companies feel it increased customer satisfaction

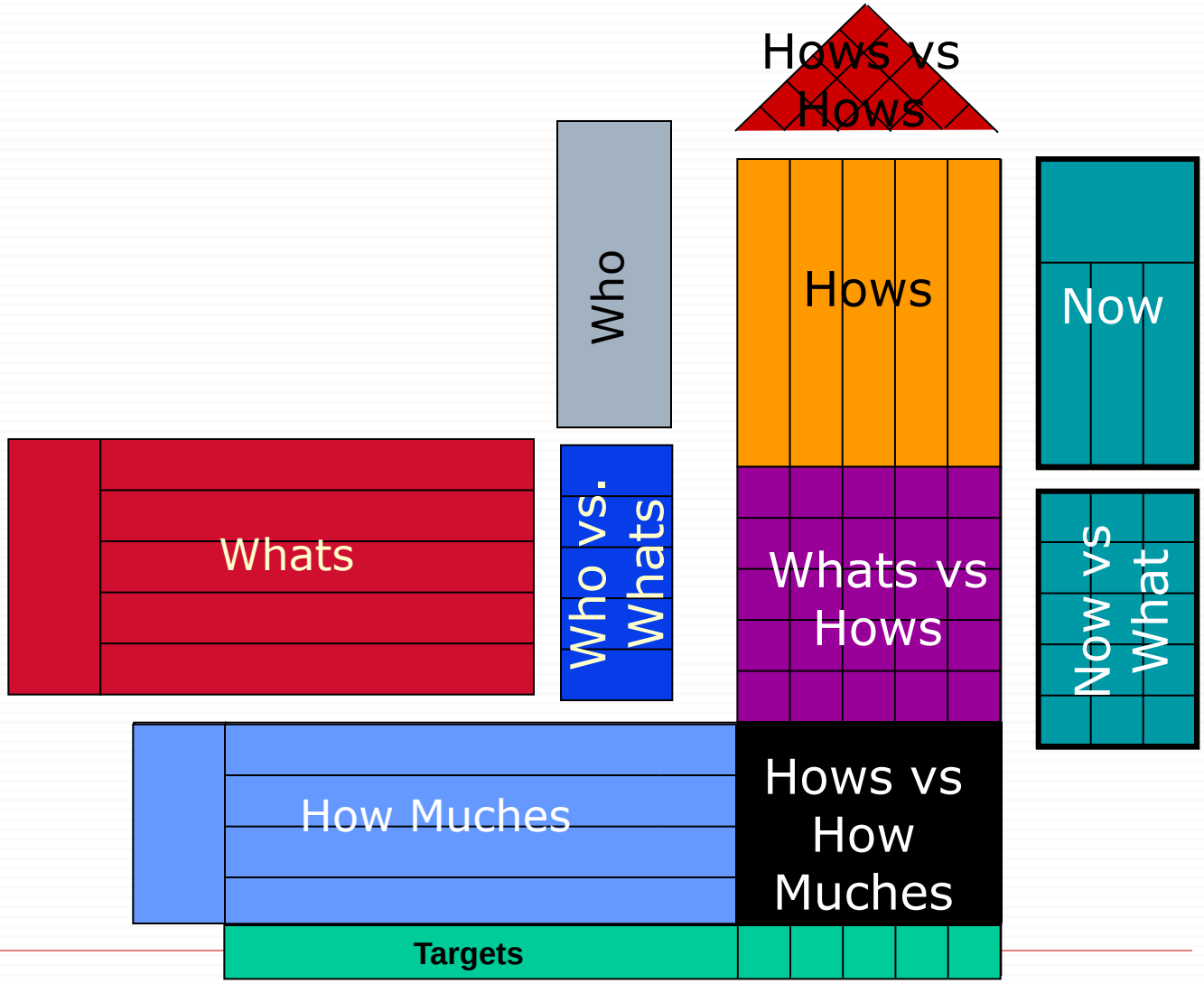
Why?

- Product should be designed to reflect customers' desires and tastes.
- House of Quality is a kind of a conceptual map that provides the means for inter-functional planning and communications
- To understand what customers mean by quality and how to achieve it from an engineering perspective.
- HQ is a tool to focus the product development process

QFD Key Points

- Should be employed at the beginning of every project (original or redesign)
- Customer requirements should be translated into measurable design targets
- It can be applied to the entire problem or any sub-problem
- First worry about what needs to be designed then how
- It takes time to complete

Components of House of Quality



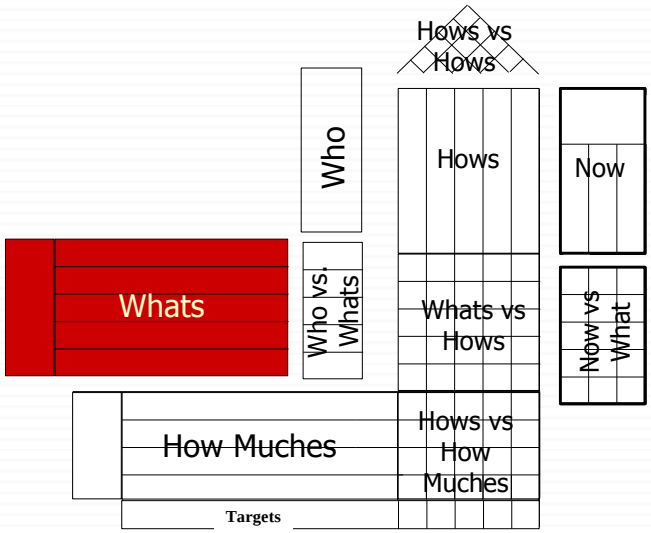
Step 1: Who are the customers?

- To "Listen to the voice of the customer" first need to identify the customer
- In most cases there are more than one customer
  - consumer
  - regulatory agencies
  - manufacturing
  - marketing/Sales

Customers drive the development of the product, not the designer

Step 2: Determine the customers' requirements

- Need to determine what is to be designed
- Consumer
  - product works as it should
  - lasts a long time
  - is easy to maintain
  - looks attractive
  - incorporated latest technology
  - has many features



List all the demanded qualities at the same level of abstraction

Step 2: cont…

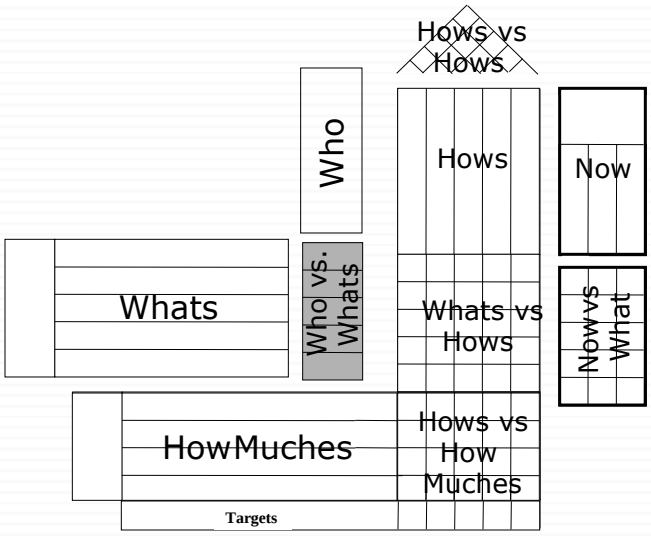
- Manufacturing
  - easy to produce
  - uses available resources
  - uses standard components and methods
  - minimum waste
- Marketing/Sales
  - Meets customer requirements
  - Easy to package, store, and transport
  - is suitable for display

How to determine the “Whats”?

- Customer survey (have to formulate the questions very carefully)
- If redesign, observe customers using existing products
- Combine both or one of the approaches with designer knowledge/experience to determine “the customers’ voice”

Step 3: Who vs. What

- Need to evaluate the importance of each of the customer's requirements.
  - Generate weighing factor for each requirement by rank ordering or other methods



Rank Ordering

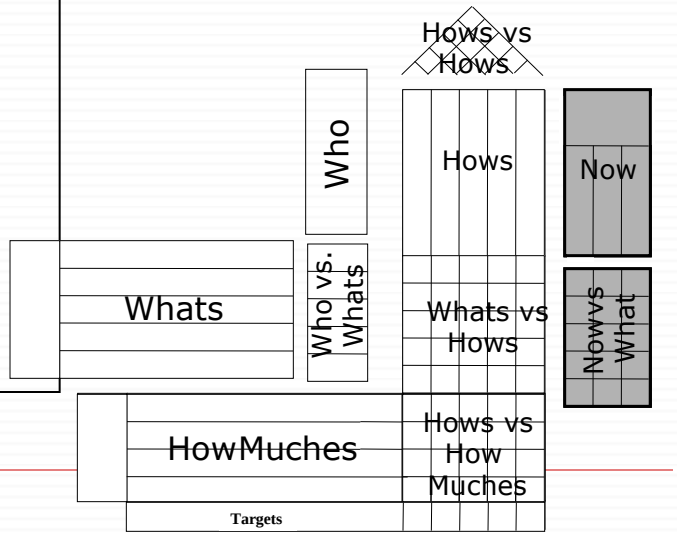
- Order the identified customer requirements
- Assign “1” to the requirement with the lowest priority and then increase as the requirements have higher priority.
- Sum all the numbers
- The normalized weight

  Rank/Sum
- The percent weight is: Rank*100/Sum

Step 4: How satisfied is the customer now?

- The goal is to determine how the customer perceives the competition's ability to meet each of the requirements
  - it creates an awareness of what already exists
  - it reveals opportunities to improve on what already exists

The design:
1. does not meet the requirement at all
2. meets the requirement slightly
3. meets the requirement somewhat
4. meets the requirement mostly
5. fulfills the requirement completely

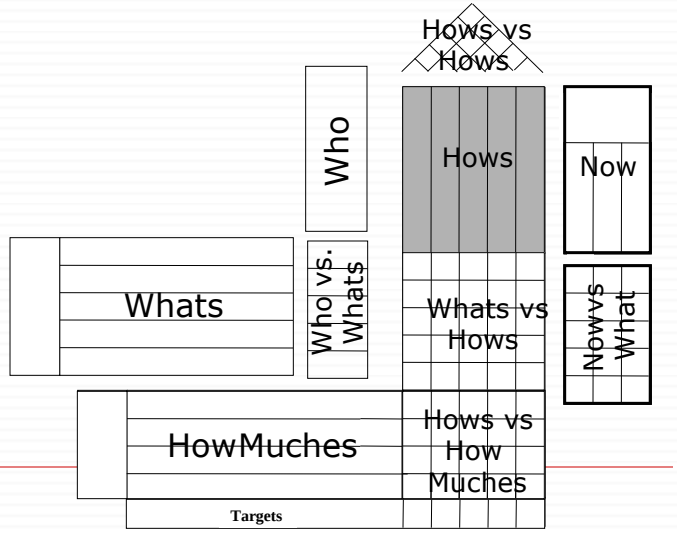


Step 5: How will the customers' requirements be met?

- The goal is to develop a set of engineering specifications from the customers' requirements.

Restatement of the design problem and customer requirements in terms of parameters that can be measured.

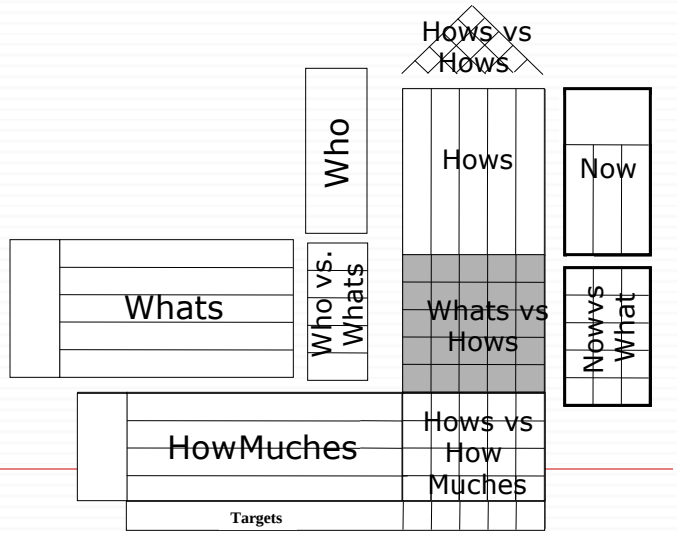
Each customer requirement should have at least one engineering parameter.



Step 6: Hows measure Whats?

- This is the center portion of the house. Each cell represents how an engineering parameter relates to a customers' requirements.

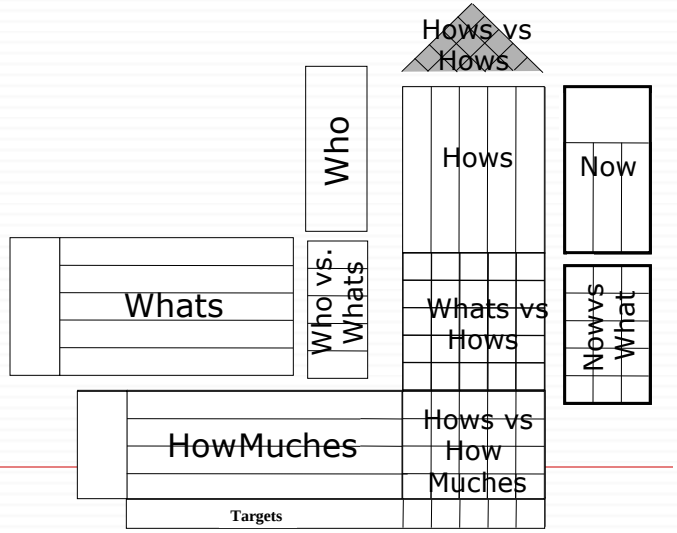
9 = Strong Relationship
3 = Medium Relationship
1 = Weak Relationship
Blank = No Relationship at all



Step 7: How are the How's Dependent on each other?

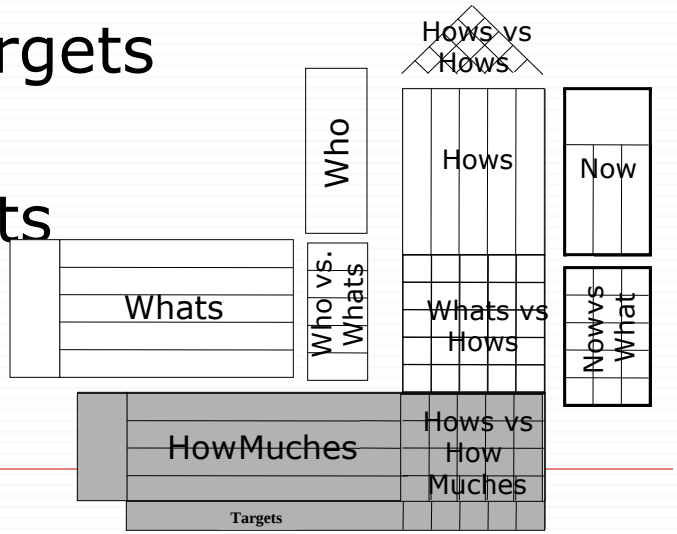
- Engineering specifications maybe dependent on each other.

9 = Strong Relationship
3 = Medium Relationship
1 = Weak Relationship
-1 = Weak Negative Relationship
-3 = Medium Negative Relationship
-9 = Strong Negative Relationship
Blank = No Relationship at all



Step 8: How much is good enough?

- Determine target value for each engineering requirement.
  - Evaluate competition products to engineering requirements
  - Look at set customer targets
  - Use the above two information to set targets

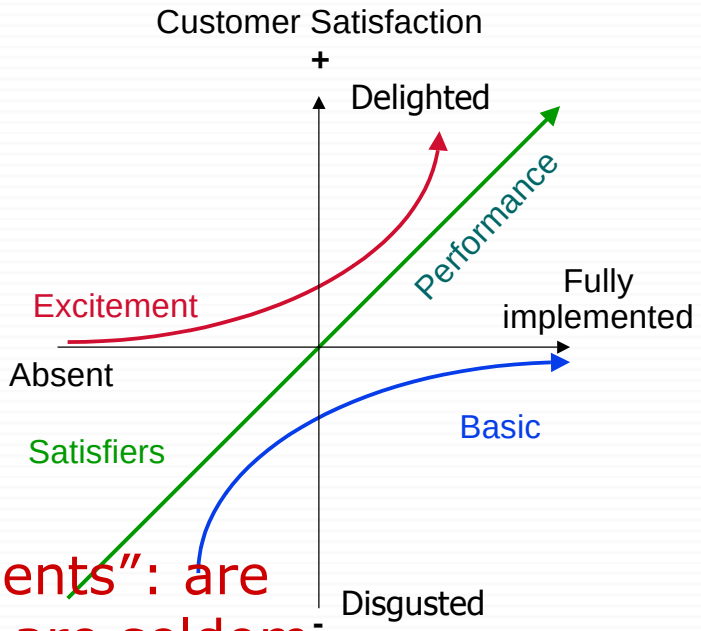


Kano Model

Basic Quality: These requirements are not usually mentioned by customers. These are mentioned only when they are absent from the product.

Performance Quality: provides an increase in satisfaction as performance improves

Excitement Quality or "wow requirements": are often unspoken, possibly because we are seldom asked to express our dreams. Creation of some excitement features in a design differentiates the product from competition.



Software Requirements

- Requirements & Specification
  - Formal Approach
  - IEEE Standard: Software Requirement Spec.
- Non-functional Requirements
  - Software Security, Reliability, and Safety
  - Improving Software Safety with Fault-Tolerance

Meaning of Requirements

- For now, Requirements => Functional requirements
- Requirements are located in environment, which is distinguished from the machine/software to be built.
- Distinction between requirements and specifications
- A specification is a restricted form of requirement, providing enough information for the implementer to build the machine without further environment knowledge
- Requirements need appropriate description

The Machine and The Environment

- The requirements do not directly concern the machine, they concern the environment into which it will be installed.
- The environment is the part of the world with which the machine will interact, in which the effects of the machine will be observed and evaluated
- Example
  - Machine: Lift-control system
  - Environment: floors served, lift shaft, motor, doors and etc.
- Environment: What is given
- Machine: What is to be constructed

Shared Phenomena

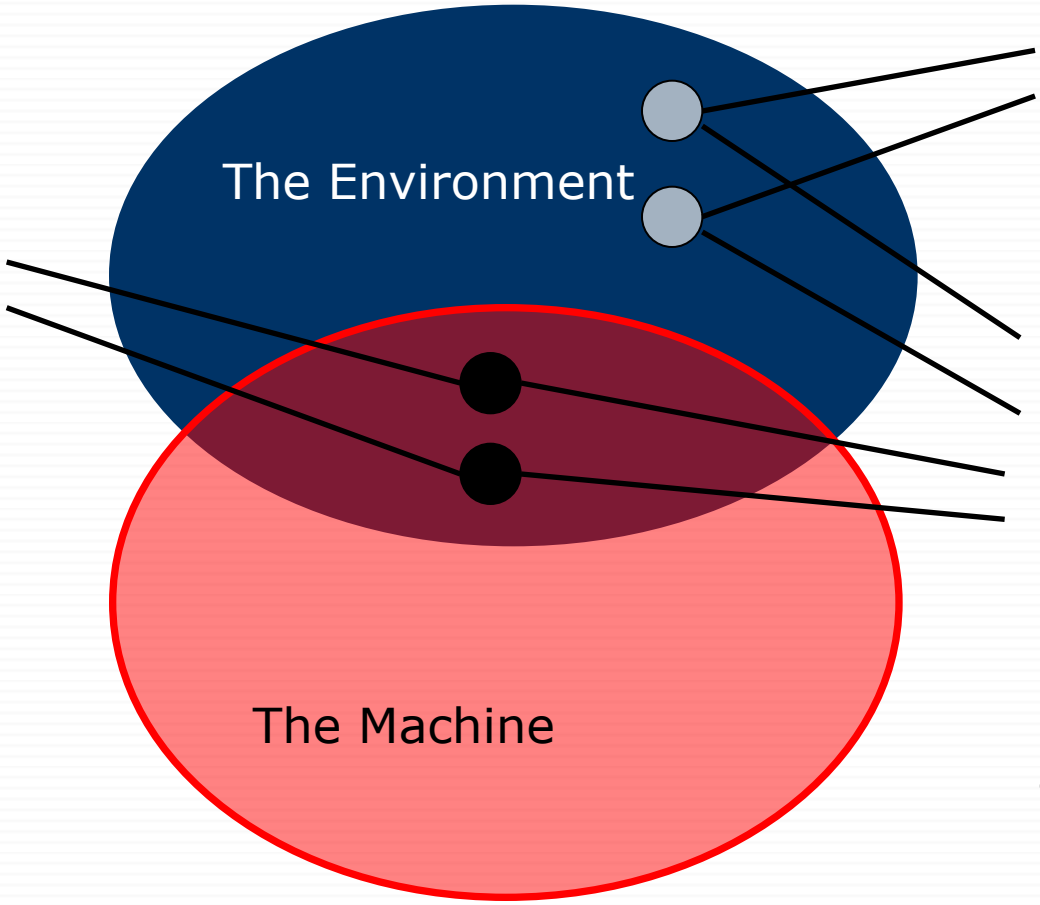
- The machine can affect, and be affected by, the environment only because they have some shared phenomena in common (events and states)
- Example in the lift system
  - Shared event: turn-motor-on (between motor and machine)
  - Shared state: up-sensor-2-on (between a sensor located in the lift at floor 2 and machine's store)
- In considering shared phenomena, it is essential to distinguish between those that are controlled by the machine and those that are controlled by the environment
  - turn-motor-on event is controlled by machine
  - up-sensor-2-on state is controlled by environment

Optative and Indicative

- The full description of a requirement consists of at least two parts:
  - We must describe the requirement itself
    - The desired condition over the phenomena of the environment (optative) guaranteed by the machine
  - We must also describe the given properties of the environment (indicative) guaranteed by the env. (environment assertions)
    - By virtue of which it will be possible for a machine, participating only in the shared phenomena, to ensure that the req. is satisfied

Requirements in Environment

Machine behaviour is about these shared phenomena only



Requirements are typically about these private phenomena

Environment properties relate all of these shared and private phenomena, and so relate the requirements to the machine bahaviour

Requirements and Specifications

- To show that the requirements are satisfied by some machine, we derive a *specification S* of the machine.
- If a machine whose behavior satisfies *S* is installed in the environment and the environment has the properties described in E, then the environment will exhibit the properties described in R:

$$E , S \vdash R \quad \text{Or} \quad E \wedge S \Rightarrow R$$

The Importance of Requirements

- Reasons for failure:
  - Straightforward programming errors
  - Mismatch between the designed behavior and the effects in the environment
  - Errors in requirements
    - Incorrectly identified
    - Imprecisely expressed
    - Based on faulty reasoning about the environment
    - Based on faulty approximations to the reality of the phenomena and properties of the environment

Turnstile Example

- Control of turnstile at the entry of zoo
  - Turnstile consists of
    - Rotation of Barrier
    - A coin slot
    - Electrical Interfaces
    - Mechanical part exists
  - Development: S/W that controls
  - M/C: Small computer on which the s/w runs
  - Environment: Turnstile mechanism itself and its use by visitors

Turnstile

- Visitor who wants to enter the zoo
  - Must push on the turnstile barrier
  - Move it to an intermediate position
  - It rotates on its from that position letting visitor in
  - Returns to original position
  - Has a locking device when locked prevents barrier being pushed to the intermediate position

Designations

- Write a designation set
- Each designation informally described
- Terms to denote the phenomena
- What's happening in the environment?
  - Pushing the barrier into intermediate
  - Insertion of coins in the slot
  - Entry into the zoo
  - Locking of the barrier
  - Unlocking of the barrier

Designations: Shared and Unshared

- Designations, all predicates
  - Push(e)
  - Lock (e)
  - Coin (e)
  - Unlock (e)
  - Enter (e)

All designations are specific to the environment.

Identify phenomena using which Requirements and Specifications can be expressed

- Shared - Some phenomena must be shared.
  - Push(e)
  - Lock (e)
  - Coin (e)
  - Unlock (e)

Control of Phenomena

- Where does the control of shared phenomena reside?
- Environment Controlled – initiated here
  - Push
  - Coin
  - Enter
- Machine Controlled – initiated here
  - Lock
  - Unlock
- Machine can prevent Push and Enter through locking of turnstile

Safety and Liveness

Safety: something "bad" will never happen

Liveness: something "good" will happen
(but we don't know when)

Safety: the program will never produce a wrong result ("partial correctness")

Liveness: the program will produce a result ("termination")

Requirements

- No entry without payment.
- Anyone paid should be allowed to enter.

- Needs designated environment phenomena based on previously designated phenomena.
- Push#(v, n)
- Enter# (v, n)
- Coin# (v, n)

Question

- Assume (a OR b) is the logic expression that captures the environment properties for a machine and (a OR c) is the logical expression that captures the overall specification of the machine. It turns out that the requirements for this machine can be refined to the logic (~a OR (b AND c)), where ~ represents the negation operation. Do the satisfaction of environment properties and the specification imply the satisfaction of requirements for this machine?

$$E \wedge S \Rightarrow R$$

Software Requirements

- Requirements & Specification
  - Formal Approach
  - IEEE Standard: Software Requirement Spec.
- Non-functional Requirements
  - Software Security, Reliability, and Safety
  - Improving Software Safety with Fault-Tolerance

Software Development Life Cycle

Project initiation
Needs
Requirements
Specifications
Prototype design
Prototype test
Revision of specs
Final design
Coding
Unit test
Integration test
System test
Acceptance test
Field deployment
Field maintenance
System redesign
Software discard

Software flaws may arise at several points within these life-cycle phases.

Software Importance

- Software is becoming central to many life-critical systems
- Software is created by error-prone humans
- In the real world, software is executed by error-intolerant machines
- Software development and maintenance is affected more by budget and schedule concerns than by a concern of reliability

Faults and Failures

- A software is said to contain a fault if for some input data the output is incorrect
- For each execution of the software program where the output is incorrect, we observe a failure
- Error, bug, mistake, malfunction, defect etc.

What Does Software Reliability Mean?

- Major structural and logical problems are removed very early in the process of software testing
- Flaws appear less frequently afterwards
- Software usually contains one or more flaws per thousand lines of code, with < 1 flaw considered good (linux has been estimated to have 0.1)
- If there are f flaws in a software component, rate of failure occurrence per hour, is kf, with k being the constant of proportionality which is determined experimentally
- Software reliability: $R(t) = e^{-kft}$
- The only way to improve software reliability is to reduce the number of residual flaws through more rigorous verification and/or testing

Comparison (cont'd)

- Once a software fault is removed it will never cause the same failure again.
  - Software reliability can be improved by testing whereas, for hardware one has to use better material, improved design, and increased strength etc.
- Software redundancy does not make any sense unless multi-version

Reliability Improvement

- Fault Avoidance
- Fault Detection and Removal
- Fault Tolerance

Fault Tolerance

- Exception handling
- Recovery Block Schemes
- N-version programming
- Self checking programs

Exception Handling

- Framework within which each phase of FT can be implemented
- Software system is a hierarchy of modules
- Hierarchy represented by acyclic graph
  - Arrow from module M to N, if M uses N
  - Successful completion of M depends upon N's success
- Response of each module
  - Normal
  - Abnormal [Exceptions]
- EH framework signals & handles [mask] exceptions

Programming with Exceptions

Traditional piece of code:

Open a file, do something with it, close the file.

```
void use_file (const char* fn)
{
   FILE* f = fopen(fn,"r");
   // use f
   fclose(f);
}
```

Something goes wrong in “use f” segment then possible to exit code without closing file f.

Programming with Exceptions

A typical first attempt to make use_file() fault-tolerant looks like this:

```
void use_file(const char* fn)
{
    FILE* f = fopen(fn,"r");
    try {
    // use f
}
catch (...) {
    fclose(f);
    throw;
}
    fclose(f);
}
```

Catches exception, closes file, re-throws exception

Exception Handling

- Section of code in which exception may occur, enclosed in a try statement
- Something that causes exception and triggers emergency procedures through a throw statement
- Exception handling code inside a catch block

Recovery Block Scheme

The software counterpart to standby sparing for hardware

Suppose we can verify the result of a software module by subjecting it to an acceptance test

ensure	*acceptance test*	
by	*primary module*	e.g., sorted list
else by	*first alternate*	e.g., quicksort
		e.g., bubblesort
.		.
.		.
.		.
else by	*last alternate*	e.g., insertion sort
else fail		

The acceptance test can range from a simple reasonableness check to a sophisticated and thorough test

Design diversity helps ensure that an alternate can succeed when the primary module fails

RBS: Example

- Sorting program
  - Ensure A[j+1] > A[j] for j=1,2,...,n-1
  - by Sort A using quick sort
  - by Sort A using insertion sort
  - by Sort A using bubble sort
  - else ERROR

RBS: Acceptance-Test Design

- Design of acceptance tests (ATs) that are both simple and thorough is can be difficult
- Simplicity is desirable because acceptance test is executed after the primary computation, thus lengthening the critical path
- Thoroughness ensures that an incorrect result does not pass the test (of course, a correct result always passes a properly designed test)
- Some computations do have simple tests (inverse computation) Examples: square-rooting can be checked through squaring, and roots of a polynomial can be verified via polynomial evaluation
- At worst, the acceptance test might be as complex as the primary computation itself

RBS: Deadline Mechanism

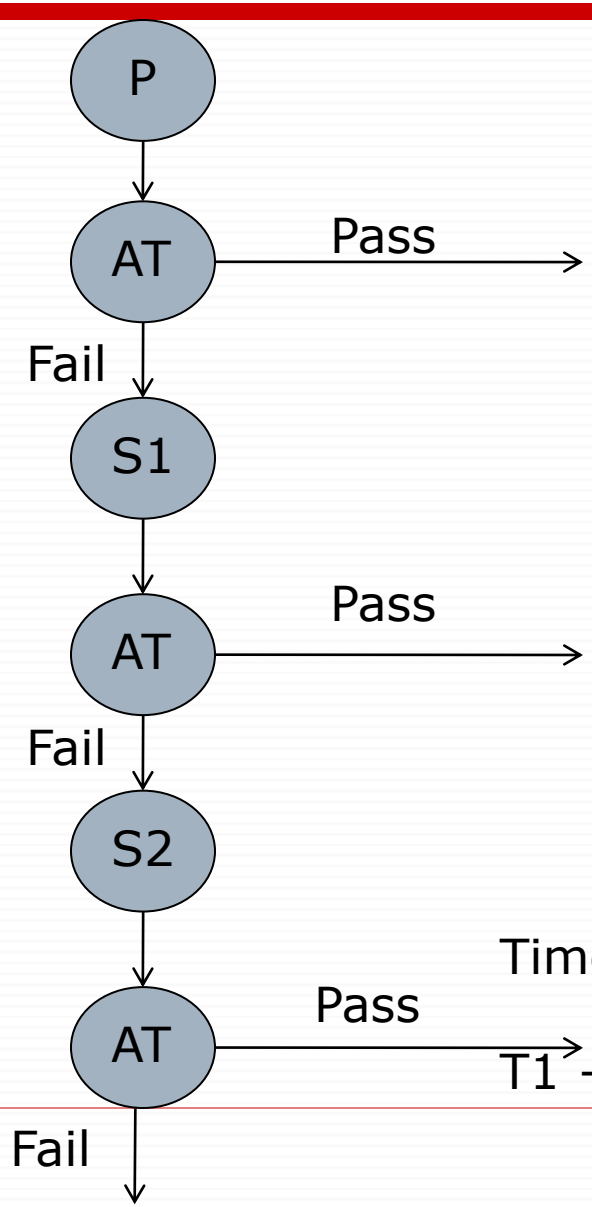
Based on Recovery Block approach to avoid timing failures

Service <service-name>

Within <response-period>

By
  Primary Algorithm
Else by
  Alternate Algorithm

RBS: Performance & Reliability



P1 = Prob. of P's success

P2 = Prob. of S1's success

P2 = Prob. of S2's success

Prob. that scheme successful =

P1 + (1 – P1) (P2 + (1 – P2) (P3 + ….))

T1 = Time taken by P + AT

T2 = Time taken by S1 + AT

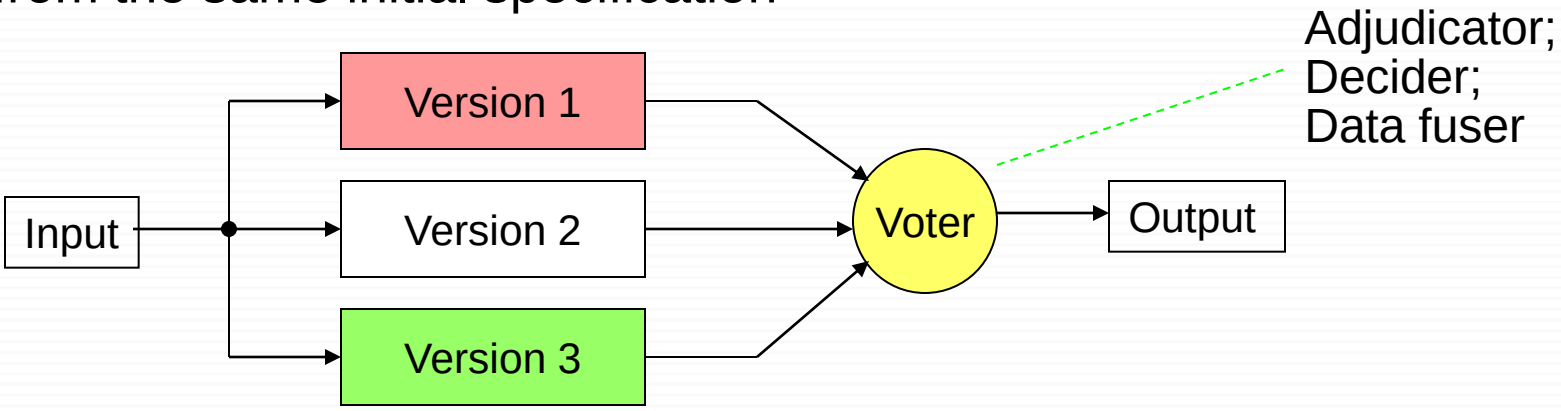
T3 = Time taken by S2 + AT

Time taken by RBS scheme =

T1 + (1 – P1) (T2 + (1 – P2) (T3 + ….))

N-Version Programming

Independently develop *N* different programs (known as “versions”) from the same initial specification



The greater the diversity in the *N* versions, the less likely that they will have flaws that produce correlated errors

Diversity in:

1. Programming teams (personnel and structure)
2. Software architecture
3. Algorithms used
4. Programming languages
5. Verification tools and methods
6. Data (input re-expression and output adjustment)

NVP: Key Points

- Independent generation of n>2 functionally equivalent programs from the same initial specification.
- Independent generation - programs developed by N different groups that do not interact.
- Multiple versions must be run
- Versions can run in parallel
- Construction of voting mechanism

Airbus A320/330/340 flight control: 4 dissimilar hardware/software modules drive two independent sets of actuators.

RBS vs. NVP

- In RBS if the error escapes the AT, no recovery action is initiated
- In NVP if a majority of versions have the same fault recovery will not be initiated
- In recovery blocks, production cost low, since earlier versions of the software can be used as alternates
- Combination schemes are attractive.

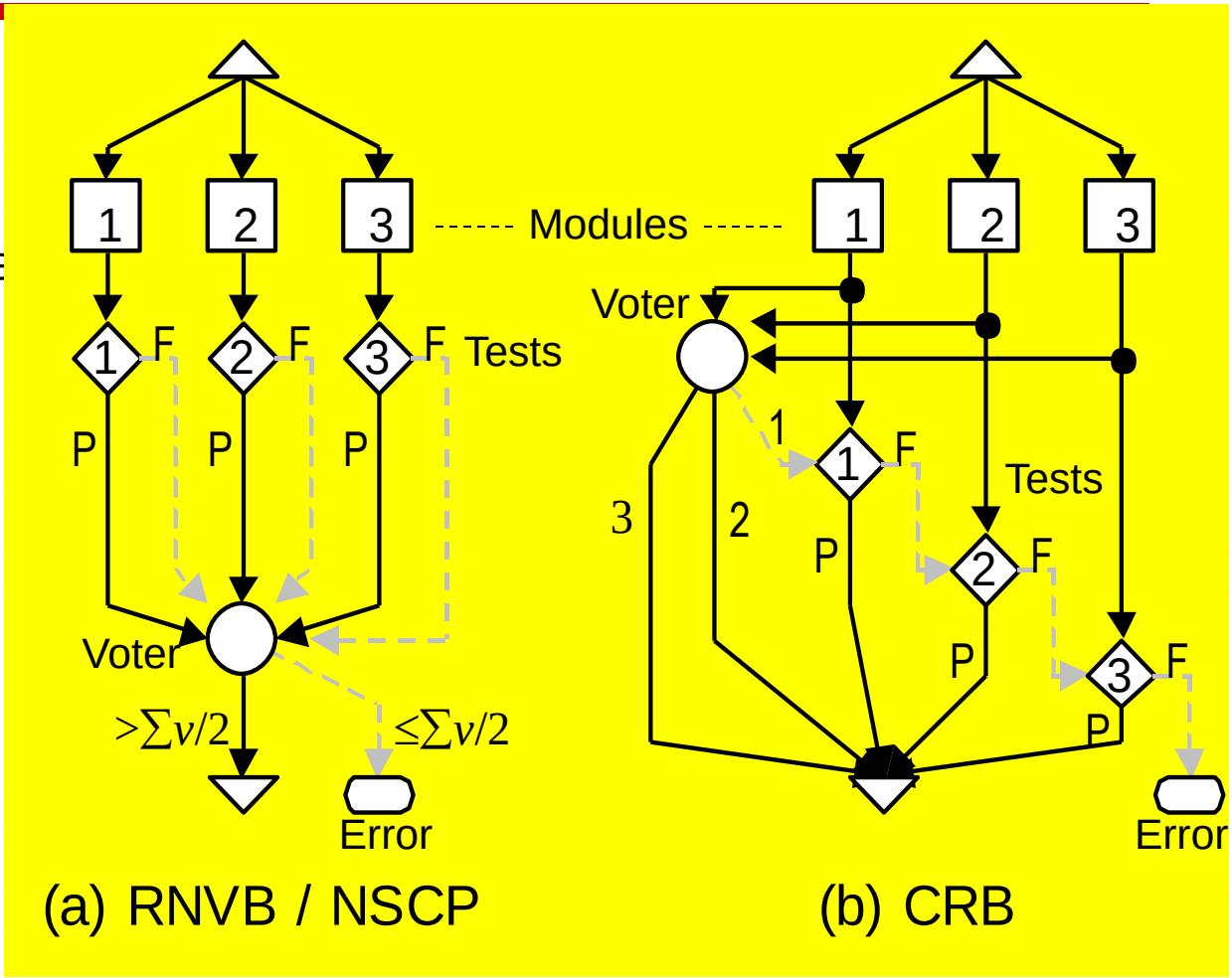
RBS & NVP Combinations

Recoverable N-version block scheme = N-self-checking program

Voter acts only on module outputs that have passed an acceptance test

Consensus recovery block scheme

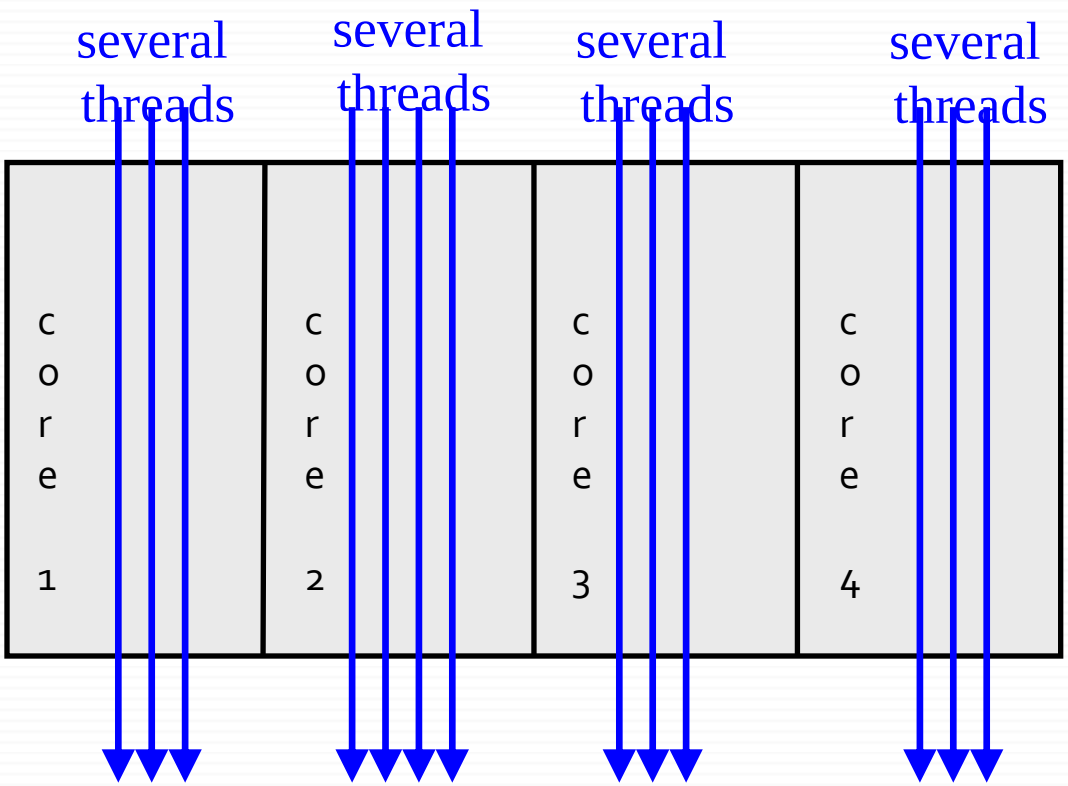
Only when there is no majority agreement, acceptance test applied (in a prespecified order) to module outputs until one passes its test

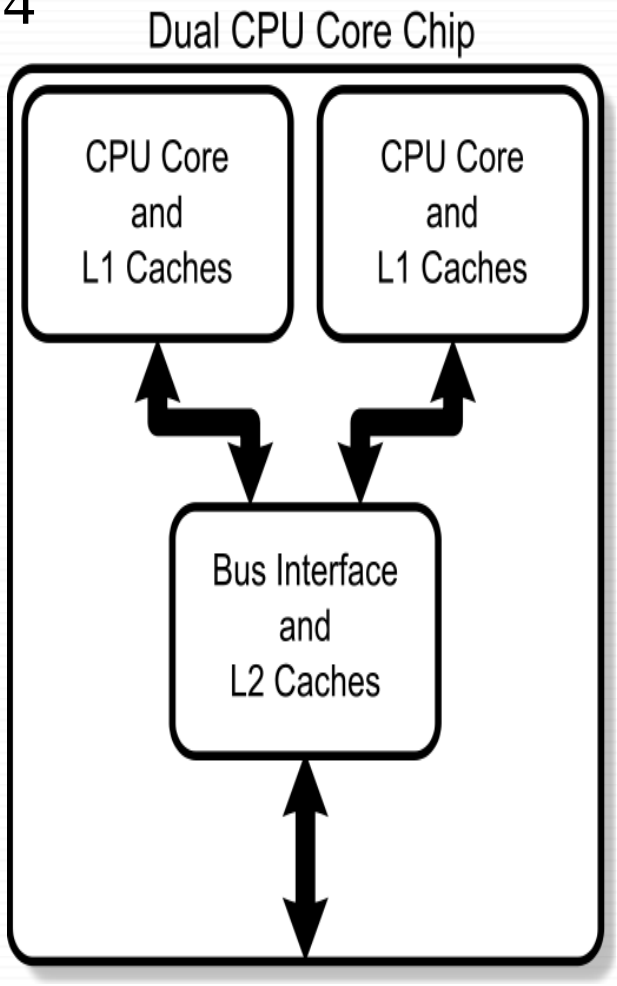


Source: Parhami, B., "An Approach to Component-Based Synthesis of Fault-Tolerant Software," *Informatica*, Vol. 25, pp. 533-543, Nov. 2001.

Multicore & FT

- Dual-core/Quad-core processor contain 2/4 independent microprocessors.





Multicore & FT

- N-Version programming can utilize multiple cores to improve performance
- Ideal for any FT schemes that use voting
- Requires parallel programming
  - OpenMP (Shared Memory MP)
  - MPI (message passing)
- Helps both software and hardware FT

Algorithm-Based Fault Tolerance

- Encode the input data stream
- Redesign of the algorithm to operate on the coded data
- Generally more suitable for computationally intensive applications
  - Matrix operations
    - Transposition, Addition
    - Multiplication
  - FFT

ABFT: Matrix Multiplication

- Use column and row checksum encoding

$$A = \begin{vmatrix} 2 & 1 \\ -1 & 0 \end{vmatrix} \quad B = \begin{vmatrix} 1 & 0 \\ 3 & 2 \end{vmatrix}$$

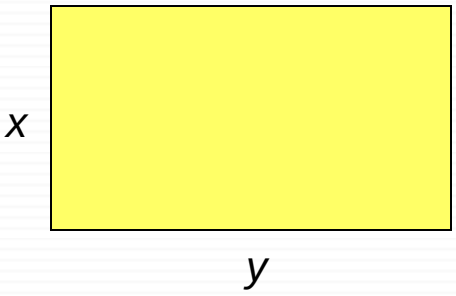
$$Ac \times Br = \begin{vmatrix} 2 & 1 \\ -1 & 0 \\ 1 & 1 \end{vmatrix} \times \begin{vmatrix} 1 & 0 & 1 \\ 3 & 2 & 5 \end{vmatrix} = \begin{vmatrix} 5 & 2 & 7 \\ -1 & 0 & -1 \\ 4 & 2 & 6 \end{vmatrix}$$

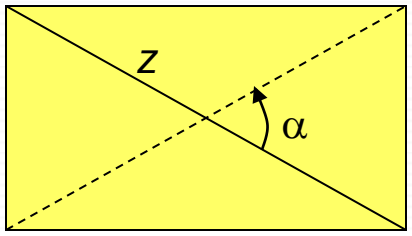
Data Diversity

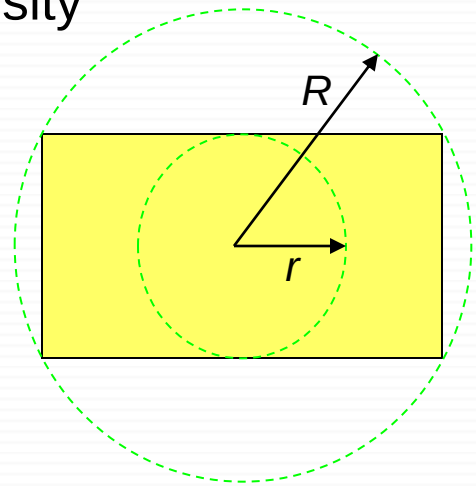
Alternate formulations of the same information (input re-expression)

Example: The shape of a rectangle can be specified:
By its two sides *x* and *y*
By the length *z* of its diameters and the angle α between them
By the radii *r* and *R* of its inscribed and circumscribed circles

Area calculations with computation and data diversity







$A = xy$

$A = \frac{1}{2} z^2 \sin \alpha$

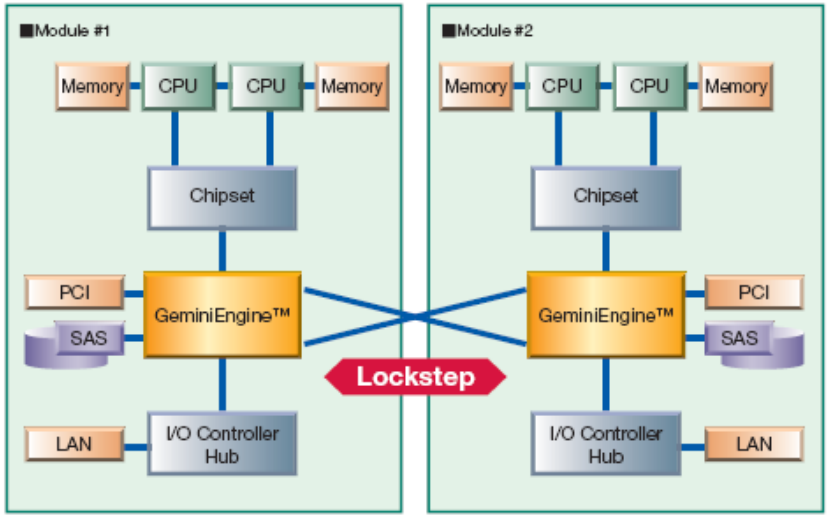
$A = 4r(R^2 - r^2)^{1/2}$

FT Cloud

- A single moment of downtime: Not an option in today's business
- A single server failure could result in enormous loss of business opportunities
- Minimize risk of downtime: keep systems up and running
- FT Servers: Fully Redundant Servers
- Address planned and unplanned downtime
- HP, NEC, DELL, ….

High Availability

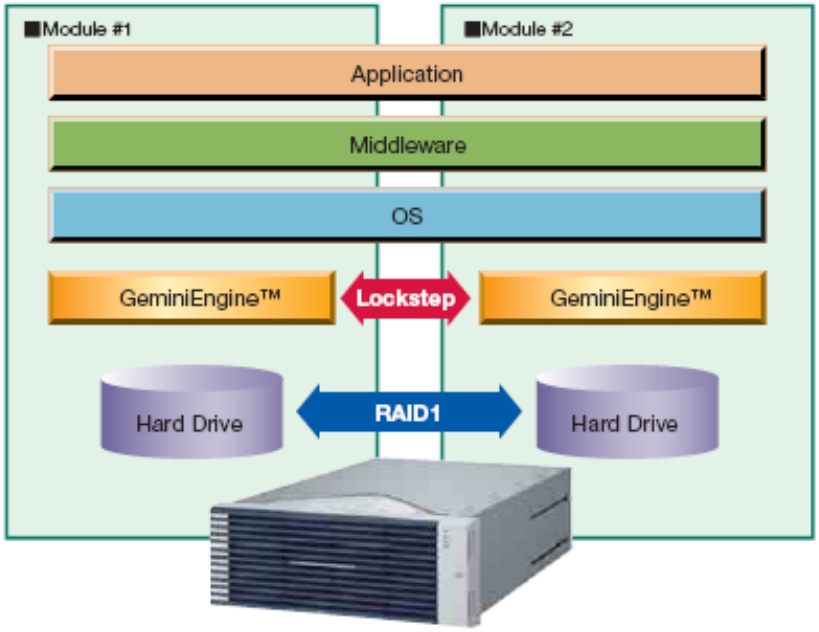
- Servers engineered for transparent failover and system integrity: NEC FT Servers



- Hardware components replicated
- Redundancy chipset controls redundant h/w
- Redundant modules provide lockstep processing

Single Server View

- Perform as single servers running single operating system



- No need to modify any middleware or applications