Systems, the Design Process and Models

Objective

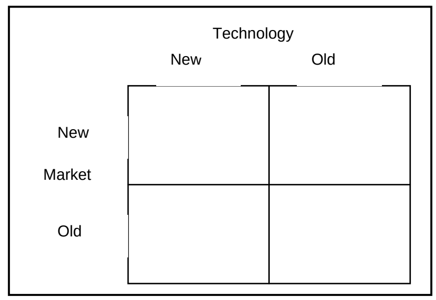
In design must look at things from many points of view

Customer
- Finances the development
  - Directly
  - Indirectly

Best design of little value if no one to buy

What kinds of things do we need to consider

Looking at products
- How to measure
  - Costs
  - Features
- Identifying need
  - Real
  - Perceived
- Tradeoff
  - Technology and market area
    - Difficult
      - New technology in new area
    - Easier
      - New technology in existing area
      - Existing technology in new area



Deadlines and Costs
- Product development
  - Based upon negotiated contract
    - Company and customer
      - Direct
      - Indirect
- Failure to respect
  - Development and delivery costs or schedules
  - Leads to loss of
    - Sales
    - Market share
    - Credibility

Quality
- Beyond obvious need to work properly
- Product must meet the following criteria
  - Robust
    - Reliability and tolerance of errors and use

Does it do what it's supposed to
How does it behave with unexpected

Clear and understandable documentation

Ease of use
Intuitive
Counter-intuitive

Post sales support
Including correction of bugs

Lack of quality
Two costs
Obvious and immediate
Cost to repair - often small
Hidden
Loss of customer confidence and sales - can be very large
Once confidence lost very difficult to regain

Problem Solving

When solving a problem
Begin with a set of requirements
Goal
Map the real to the abstract
Given
Problem Statement
Usually expressed in a natural language
Goal
Map
The problem statement
The real world
Through a series of transformations
Into the abstract world
Solution
Solution
Hardware
Software
Combination

System Design and Development

Getting Started
Development of systems require large number of decisions

Decisions require knowledge about
- Problem

- Tools and techniques that may be available
  - Techniques - how problem is solved
  - Tools - used to implement technique
- Methods for approaching solution

Based on a set of steps

Each has role of
    Transforming its input (specification) into an output (a selected solution)
Organization of the steps
    Done according to a design process model

Several process models have been suggested
    Waterfall model
    V model
    Spiral model,
    Contractual model

Which ever model chosen
    Most important aspects
        Meaning and intent or objective of the basic design steps
        Specification of inputs and outputs for each step

Effective approach proceeds in top-down (or modified top down) manner
    Potential overlap between steps
    Must support backwards or reverse flow - revisiting an earlier step
        Correct problems and/or enhance solutions

Five Steps to Design

Good system designers and designs proceed using a minimum of five steps

- ✓ Requirements definition
- ✓ System specification
- ✓ Functional design
- ✓ Architectural design
- ✓ Prototyping and test

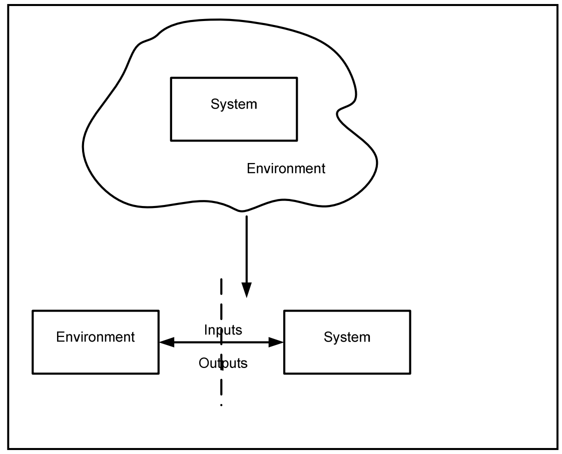
Contemporary design process must also enforce
    IP capitalization and reuse at every design stage

Traceability in both forward and reverse directions
    Captures the relationships between requirements and all subsequent design data
    Helps managing requirements changes

Requirements Definition

- Process of understanding the needs of all interested parties
- Documenting these needs as written definitions and descriptions

Focus is on
    What problem the system has to solve
    What needs to be done



Means examining
    System
    Environment in which it is operating

Primary step is focussing on
    World in which the system will operate
        Not immediately on the system itself
    How world is affecting system

Follow with examination of combination

Are trying to capture the public interface to the system

There are no formal languages currently available
    Which are able to fully express such requirements

Many of tools developed for object centered design
    Find strong application here
    UML moving in this direction
        Use cases provide good starting tool

Natural language of experts often used
    Because the expression of requirements
        Forces the discussion of arbitrarily complex problems
            In many disciplines
            With many partners

System Specification

After first cut at identifying requirements
    Move into specification step

Remember this is an iterative process

Purpose of the *Specification* step is to
    Capture, express, and formalize purely external view of the system
        Identified during requirements definition

We have identified *WHAT* needs to be done
    Starting from needs and user requirements
    Now quantify those whats

Step requires solid understanding of

System behavior
Environment
System in environment

Appears obvious but not often used in companies

Formal specification must be written in precise language
Stating specific requirements of the system
May be or include
Tables
Equations or algorithms
State or Flow diagrams
Formal design language
Pseudo English

Does not include schematics, code, or parts lists
Unless exceptional and limited circumstances

Non-functional specifications have to be added
We use these to explain constraints such as

- Performance and timing constraints
- Dependability constraints
- Cost, implementation and manufacturing constraints

Many specification models and tools exist
Often far too many for designers to keep track of

Prelude to Functional Design

As we move from formal specification to design and implementation

Want to take disciplined approach to design of
System hardware and software

Goals

1. To attack complexity of problems by partitioning into modules and organizing modules into hierarchies
2. Use variety of tools to render complex systems understandable
3. Utilize strategies for developing design solution from well defined statement of problem
4. Establish criteria for evaluating quality of design

Partition into Modules

First step in controlling complexity
Partitioning or decomposing high level view into modules

Goals of such a process

- Each module should solve one well defined piece of the problem
- System should be partitioned so that function of each module is easy to understand
- Partitioning should be done so that connections between modules only introduced because of connection between pieces of problem
- Partitioning should assure that connections between modules are as independent as possible

When decomposing into modules

Begin with outside view of inter module relationships

Such relationships called *coupling*

Coupling

Analysis of coupling examines interdependence between modules

Objective

Minimize coupling

Want to make modules as independent as possible

Reducing coupling means

Reducing complexity of module interconnections

Low coupling between modules

Indicates well partitioned system

Achieved in 3 ways

1. Eliminate unnecessary relationships
2. Reduce the number of necessary relationships
3. Ease tightness of necessary relationships

Approaches

1. Create narrow (as opposed to broad) connections
   Breadth is measure of number of interconnections between modules
   Reduce the number of pieces of data that must flow between modules

2. Create direct vs indirect connections
   Don't require one module to go through second to get data from third

3. Create local rather than remote connections
   Have the connection with a second module
   Specified in parameter list
   Rather than through global data somewhere else in program

4. Create obvious rather than obscure connections

Express information in natural and expected way

5. Create flexible rather than rigid connections
   Don't hard code parameters to
   Particular memory location
   Specific data value

Cohesion

Idea related to *coupling* is *cohesion*

Coupling addresses partitioning a system
Cohesion addresses bringing things together

We stress modularity and encapsulation
Cohesion is measure of strength of functional relatedness
Elements in a module
Goal
Create strong highly cohesive modules
Whose elements are genuinely and strongly related to one another
Conversely
Elements should not be strongly related to elements in another module

Want to maximize cohesion and minimize coupling

Let's look at kinds of cohesion
Functional cohesion
Functionally cohesive module
Contains elements that all contribute to execution of
One and only one problem related task

Sequential Cohesion

Sequentially cohesive module
Contains elements that are involved in activity
Producing output data
That becomes input data to immediately successive task

Example

module formulate and cross validate data
usesraw data
format into raw record
cross validate fields in record
return formatted and cross validated record
end module

Communicational Cohesion

Communicational cohesive module
Contains elements that are involved in activity
Use the same input data

Example

module parse measurement command
uses raw data
find header field
find message length
find command
check parity
compute parity
return command or parity error
end module

Procedural Cohesion

Procedurally cohesive module
Contains elements that are involved in
Different and potentially unrelated activity
In which control flows from one activity to the next

Example

module read and modify record
uses output record
read input record
add parity to parity field
write output record
return
end module

Temporal Cohesion

Temporally cohesive module
Contains elements that are involved in activities
Related in time

Example

module initialize serial interface
updates wordCount, rBaudRate, tBaudRate, direction, parity
reset wordCount
set rBaudRate 9600
set tBaudRate 9600
set direction receive

set parity even
return
end module

Co-incidental Cohesion

Coincidentally cohesive module
Contains elements that are involved in activities
No meaningful relation to one another

Such cohesion – or lack of cohesion should not be used

Comparison

Cohesion	Coupling	Cleanliness	Ease of Modification	Ease of Understanding	Ease of Maintenance
Functional	Good	Good	Good	Good	Good
Sequential	Good	Good	Good	Good	Fairly
Communicational	Medium	Medium	Medium	Medium	Medium
Procedural	Variable	Poor	Variable	Variable	Bad
Temporal	Poor	Medium	Medium	Medium	Bad
Logical	Bad	Bad	Bad	Poor	Bad
Co-incidental	Bad	Poor	Bad	Bad	Bad

Functional Design

With metrics of reduced coupling and enhanced cohesion as guides
Now examine process of functional decomposition

With functional design / decomposition phase
Begin to move inside system
Formulate first cut at implementation of system
That meets desires and specification

Specifies the *how* of the design not the *what*
Move from desired to implementation of what is desired

Functional design based upon

- System requirements documentation
- Formal system specification

Purpose is to find an appropriate internal architecture for the system
Which explains the *HOW* the requirements are implemented
According to an application-oriented viewpoint

Design expressed
In designer's language
From designers point of view

Must also serve as bridge between
- Customer and designer

The description based on a
- Functional structure and the behavior of each function
- Must be technology-independent

A first functional decomposition is carried out based upon
- Search of and for
  - ✓ Major functional blocks
  - ✓ Essential internal variables
  - ✓ Events in the system

Depending upon the nature and the complexity of the problem
- The initial model can be based on
  - Data-flow diagrams
  - Structured Analysis and Structures Analysis Requirements Techniques
  - FSMs
  - State Charts
  - Structured Design Languages
  - UML activity and sequence diagrams

The design process then utilizes successive refinements or decompositions
- For each functional block
  - Using exactly the same process
  - Until elementary or leaf functions are obtained

- Such functions have a behavior
  - That can be expressed by a purely sequential description

The functional description model is appropriate and sufficient
- To verify the design quality and to evaluate system behavior and performance

Ideally functional models should be executable
- Simulation is often used to verify that the model
  - Does in fact
    - Model the right problem
    - Model the problem properly
- To permit verification of conformance to the specification
- Verify model accuracy
- Once again UML beginning to make this possible

During modeling and verification
- Performance characteristics can be allocated to internal functions
- Relations between such functions defined

Allows one to estimate the expected performance of the system

The functional model is different from
- A specification model and also from the physical architecture
  - The specification model normally describes
    - *External* behavior of the system

  - The functional model describes
    - *Internal* implementation of the system

Partitioning

- Partitioning a system is an essential step
  - In the process of developing a good functional design
    - Particularly in design of embedded systems
- Process is important during the early stages of the development
  - Help us attack the complexity of a large system

- During the later stages
  - Guide us in arriving at a sound architecture
- Let's look briefly at partitioning and things to consider

- Partitioning is the process of decomposing system
  - First into progressively more detailed functional modules
  - Ultimately into hardware and software components
- Consequently natural ste4p before architectural mapping

- Should begin to get comfortable seeing a partition from
  - External view and from an internal view
- Modules in each case are
  - Somewhat different yet also somewhat the same

- Remember also
  - Not a one time process
  - Don't have to be perfect the first time
    - Will probably have to redo the partitioning several times
      - Until you're satisfied with it

- As we begin to partition system
  - Let criteria for inter module coupling and intra module cohesion
    - Serve as guide
  - Keeping those thoughts in mind remember

  1. Each module should solve one well-defined piece of the problem.
     This is the philosophy we've been teaching throughout all of our courses
       Mixing functionality across modules

Makes all aspects of the development and support process
Much more difficult
We now have object oriented spaghetti code
Future changes to such modules
Easily lead to unexpected side effects
Unrelated pieces of our system suddenly not working

2. The system should be partitioned so that the function of each module is easy to understand.
   If we someone else can understand our design
   Will be able to easily maintain it and to extend it
   Throughout the products life time
   Such is an obvious benefit

   During development
   Easy to understand designs
   Lead to fewer surprises as the design nears completion
   All interested parties should be able to
   Follow the design
   Comment as the process unwinds
   Design that is too complex
   Quickly discourages early criticism
   People won't take the time to learn what the system is to do
   Unfortunately, such early acceptance
   Often is replaced by later rejection
   Potentially major redesign efforts
   Although we should be proud of our work
   Should seek out others constructive ideas

3. Partitioning should be done so that connections between modules are only introduced because of connections between pieces of problem.
   Don't put a piece of functionality into a module
   Just because there's nowhere else for it to go

4. Partitioning should assure that connections between modules are as independent as possible
   Once again, keep like things together
   Such a practice helps to reduce errors.

When we partition system
We decompose system into
Hardware and software components

As we do so must consider process from following viewpoints
Taking only single point of view or neglecting any one
Can have significant long-term affects

System does not meet customer or performance specifications

Functional
Result - functional architecture
Criteria - effectiveness, cohesiveness

Spatial
Result - distributed functional architecture
Criteria - performance, communication costs

Resource
Result - resource architecture
Criteria - performance, cost, dependability

Hardware / software
Result - hardware architecture
Criteria - performance

Architectural Design

When initial functional decomposition complete
Time to map functional blocks onto physical hardware and software
Keep in mind
Functional decomposition and architectural mapping
Iterative processes

In executing an architectural design
Goal is to define or develop the detailed solution

Consists of searching
Firstly for the executive support or hardware architecture
Secondly for the organization of the software
Onto each programmable processor
CPLD or FPGA

- First the functional description must be
  Enhanced and detailed to take into account the technological constraints
  Geographical distribution
  If necessary
  Physical and user interfaces
  Timing constraints

- Performance requirements are then analyzed
  Determine the hardware/software partitioning

Hardware portion is specified by an executive structure or physical architecture

Mapping to an architecture includes function allocation to physical blocks
Such a mapping completely describes
Implementation of functional description onto the executive structure

Behavior of the architectural solution
Can be verified by macroscopic co-simulation.

The result of the architectural design phase is selection of
The most appropriate solution to original problem
Based upon
- Exploration of variety of architectures and selection of best suited
- Hardware/software partitioning and allocation of functionality

Criteria for any decision must consider the non-functional constraints such as
- System performance
- Any real-time constraints
- Cost constraints
- Any legacy components and available technologies.

Prototyping

The prototype phase leads to an operational system prototype

Prototype implementation includes
- Testing,
- Debugging
- Validation

Prototyping is naturally a bottom-up process
Since it consists of assembling individual parts
Fleshing out more and more abstract functionalities
Each level of the implementation must be validated
Checked for compliance with the specifications
Of the corresponding level in the top-down design

Hardware and software implementations
Developed simultaneously
Involve specialists in both domains
Hopefully reducing the total implementation time
Often doesn't happen
Software usually leads hardware

Complete solution can be generated and/or synthesized both for
Hardware
ASICs CPLDs and standard cores
Software

HW/SW interfaces

Resulting prototype is verified by
- Co-simulation and emulation
- Other means

Activities in this step highly dependent on the technology used

Remember
- Prototype is tool for understanding and confirming system design
- Major mistake to transformed into
  - Mass-produced and marketed product.

Design Process Description Models

The design process briefly described above

Based on an approach in 4 conceptual steps

- A *specification* approach - A look from the outside
  - To characterize
    - External behavior of the system
    - Associated set of constraints
- A *functional* approach - A look from the inside
  - To identify
    - Internal functions
    - The relationships between them
      - Which are necessary to describe the solution to the initial problem
- An *operational* approach - Inside - Outside relationship
  - To explain the behavior of all internal functions and actions
    - On the system environment
  - To identify / allocate non-functional requirements
    - Such as performances and dependability
- A *technological* approach - Making it work
  - To define the hardware architecture and mapping of the functional solution onto it
    - Considering all technological constraints.

Each step
- Uses a description (specification) as its input then
- Transforms that description into an output
  - A selected solution

Each input or output is a description
- Which needs to be well defined and formalized

A sound methodological process is then based on
- Hierarchy of description models ranging from
  - The preliminary idea to the final product.

Specifications versus Requirements

First consider that requirements and specifications
Are fundamentally different types of descriptions

Requirements
A description of something wanted or needed
A set of needed properties

Specification
Is a description of some entity
Which has or implements those properties

We consider a specification to be
A precise description of the system
Which meets stated requirements

As such it should be as
Independent of the users and expected environment as possible

Ideally a specification document should be

- Complete
- Consistent
- Comprehensible
- Traceable to the requirements
- Unambiguous
- Modifiable
- Writable

It should be
Expressed in as formal a language or notation as possible
Ideally it should be executable

A specification should
Focus precisely on the system itself
Provide a complete description of its externally visible characteristics
Its public interface
Externally visible clearly separates those aspects which are
Functionally visible to the environment in which the system operates
from
Those aspects of the system which reflect its internal structure

A specification is not concerned with the internal organization
A specification is supposed to describe
What a system must do and *how well* it has to do it
Not how it does it

Therefore the *Specification* step is a more formal process
Translating the description of needs
Into a more formal structure and model

So far research in system specifications has concentrated on
Language capabilities and graphical notations
Necessary to express and verify specifications

Many specification models and methods have been proposed
Can be found in the available literature

Some are useful for
Hardware such as ASICs
Software
System level specification and design

Functional Model versus Architectural Model

The internal organization of any system is based on
Components and interconnections between them
An appropriate model has to include elements both at
Functional level and at the *architectural* (or executive) level
To be able to represent and evaluate hardware/software systems

Means that in such systems
Hardware component such as a microprocessor
May be used to implement several software tasks
Therefore several functions

The model also has to be as generic as possible
Permits it to be sized or scaled during the design process

The Functional Model

Describes a system by
Set of interacting functional elements
The behavior of each of them

Described with a hierarchical and graphical model

Functions interact using relations of one of the following three types

- The shared variable relation
  Defines a data exchange without temporal dependencies
- The synchronization relation
  Specifies temporal dependency
- The message transfer by port
  Implies a producer/consumer relationship

The Executive - Architectural Model

Describes the physical architecture based on

- Active components
  - Microprocessors
  - DSP specific processors,
  - Analog and digital components

- Interconnections between them

The Need for Both Models

These two views when considered separately
 Are not sufficient to completely describe the design of contemporary systems

It is necessary to add
 Mapping between the *functional* viewpoint and the *executive* one
  Defining a (functional) partition and allocation (of functional components) correspondence
   Also called architectural configuration

 The result is expressed as a triple

$$\text{Architectural solution} = \{\text{Functional solution, Executive Structure}, \lambda\}$$

$$\lambda: \text{Functional Solution} \rightarrow \text{Executive Structure}$$

 The *functional model*
  Located between specification model and the architectural model
  Suitable to represent the internal organization of a system
   By explaining
    All necessary functions and couplings between them
    Expressed from the point of view of the original problem

  Using such a scheme leads to a technology-independent solution
   In particular with this kind of model
    All or part of the description can be implemented either in software or hardware

   Consequently such a model is appropriate as a basis for Hw/Sw CoDesign

   The functional model is the basis for coarse-grain partitioning
    Such a partitioning leads naturally to the selection of
     Which functions to implement in hardware or software

 The *executive structure* (hardware architecture)
  Derived from the partitioning or can be imposed a priori

The allocation of functions to executive blocks
Is also derived
Specifies the mapping between the
Functional description
Executive structure

Other Considerations

Several other factors must also be taken into consideration
Product must be able to be
Manufactured
Tested
Factors must be addressed early in definition and design process

The two additional complementary and concurrent activities need to be considered
Capitalization and Reuse
Requirements and Traceability Management

Capitalization…

Capitalization and reuse are essential activities to the contemporary design process
Proper and efficient exploitation of IPs

Consideration of component reuse is an activity to be done during
Functional and architectural design
Can be considered sometimes during prototyping as well

Its purpose is to help designers shorten the design process

Component reuse is facilitated in two ways: *present* and *future*

- Present
  By identifying a set of external (existing) functional or architectural components
  Which can satisfy some parts of desired functionality
- Future
  By identifying components of the solution under design
  Which will be reusable in other projects or products.

To be reused
Component needs to be
- Well-defined
- Properly modularized
- Conform to some interchange standard

Requirements Traceability

Refers to the ability to follow the life of a requirement (from the original spec)
In both forward and reverse directions
Through the whole design process

Traceability is potentially a one-to-many relationship
Between a requirement and the components it relates or traces to
(or that implement it)

An accurate and complete record of traceability
Between requirements and system components
Provides means for the project manager (potentially) the customer
To monitor the development progress

Requirements Management

Addresses

- Requirement modifications
- Changes
- Improvements
- Corrections

During the design

Such changes are difficult to avoid for many reasons
Therefore a clear procedure which facilitates accommodating such modifications
Has to be used during the whole design process.

Analyzing the System Design

Motivation

Verify the solution meets specs
Make various architectural vs. functional trade-offs
According to well defined criteria
Retain knowledge gained during early development
Work towards an implementation

Static Analysis

Should consider 3 areas
We begin with interdependence between modules
Called *coupling*
Want to maximize cohesion and minimize coupling

Coupling

Related to number and complexity of relationships
Coupling also a measure of implications of a change

Cohesiveness

Measure of functional homogeneity of included elements

Applies to both components and relations
Can be external or internal
External

Appropriate name and meaning for elements
Internal
Structure and relationships among components
Coupling through shared data more coherent than messages
Messages imply temporal dependency

Complexity

Functional complexity characterized by number of

- Internal functions
- Relational components
- Interconnections

Behavioral complexity characterized by number of

- Inputs
- Outputs
- Description length
- Structure of the control
- Number and structure of state variables
- Readabiliy

Dynamic Analysis

Dynamic analysis considers the following

Behavior verification

Ensures the behavior of system within its environment
Meets functional specification

Performance analysis

Ensures the behavior of system within its environment
Meets operating specification

Trade-off analysis

Necessary to determine optimal solution
Given constraints and objectives
Analysis based upon small set of performance criteria
May make product succeed or fail

Test

There are four main reasons to test

- To verify that any of the following performs as we had intended
  Code module or hardware prototype
  Subsystem or collection of subsystems
  System
- To ensure the system meets specification
- To ensure that any changes to system
  Work
  Do not alter other intended functionality

To ensure the system functions properly after being built

Each of these tests
Is different
Has different objective and scope
Tests different things

System Test Specification
Must include
Description of and specification for
Each set of tests

System Test Plan
Describes in general terms
How test will be carried out
Testing order within each type of test
Assumptions made
Algorithms that may be used

System Test Procedure
Gives detailed steps of each test

Testing formality increases towards latter phases of development
Testing to ensure design functionality
Can be reasonably informal
Should still have plan
As system begins to come together
Formality must increase
Systems becoming too complex
Too easy to miss critical yet subtle point

Testing for Yourself

Let's look first at testing during the early phase

Egoless Design

Early phase of testing begins with specification
Among first steps
Design reviews
Code walk-throughs
Code inspection
Future work
Execution of specification
Some tools exist now

Debugging

This is the phase we call debugging

- Code is written
- Prototype available
- Now must ‘turn it on’ as expression goes
- To effectively debug we must know
  - What we are looking for
  - How we are going to produce the appropriate stimui
  - How to analyze results

First Steps

- Never wait until a module or subsystem completely
  - Built
  - Coded
- Never complete entire system then try to debug

Test Case Design

- Test case design is essential for testing at any level
- Content of test cases will vary
  - Nature and intent of test

Early Stages

Test Values

- During early stages of test must test for following three kinds of values
  - Expected values
  - Unexpected values
  - Boundaries of expected values
    - Inside
    - Outside
    - At
- May be random or statistically based patterns
  - Reasonable for Combinational logic
  - Fall down on sequential

Test Coverage

- Must ensure every line of code executed
  - At least once
- Each path through code executed

Module Test

- Three kinds of module testing
- Based upon assumed knowledge of system internals
  - Black box
  - Gray box
  - White box
- Apply to hardware or software

Black Box Test

- Black box tests are data driven
- Module tested from external point of view
  - Assumes no knowledge of system or subsystem internals
- Test cases generated and applied
- Test failure aborts test and fault identified and fixed
- Testing resumes from beginning
- Black box testing requires module interfaces
  - Be clearly defined
- Weaknesses
  - Potentially exhaustive test
    - Very time consuming
  - May miss
    - Certain paths
    - Dead code

White Box Testing

- White box tests are logic driven
- Module tested from internal point of view
  - Assumes perfect knowledge of system or subsystem internals
- Test cases generated and applied
  - Designed to exercise every internal path and code segment
- Test failure aborts test and fault identified and fixed
- Testing resumes from beginning
- White box testing requires module interfaces
  - Be clearly defined

Gray Box Testing

- Mix of white and black box testing
- Applies when we have modules we did not design
  - Complex LSI or gate arrays
  - Library modules

Subsystem and System Test

- Once individual components tested
  - They need to be integrated and tested in larger subsystems
    - Until system comes together
- Many of techniques used at module level still apply
- Several other things considered at this level
  - One interesting approach called *fault seeding*
- *Fault Seeding*
  - Intentionally plants number of faults into system
  - Testing proceeds as normal
  - Count number of seeded faults identified
  - Assume test cannot distinguish between seeded and nonseeded faults
    - If x% of seeded faults remain then x% of nonseeded remain as well

Regression Test

Testing at system level should result in regression test suite
Regression tests
Used later to ensure changes to system
Don’t unintentionally alter behavior
Updated
Changes discovered during
Alpha
Beta
Verification / Validation testing
To reflect system changes as it evolves
Upon completion of testing at this level
Focus of testing shifts from design to production

Testing for your Customer

Testing for customer begins at the specification stage of design
Distinguish between
Production tests and ongoing testing for product support

Testing

Testing at this stage
3 pronged attack
Alpha and Beta tests
Verification Test
Validation Test

Alpha and Beta Tests

Intent of these test is to get real world experience on system
Either given to
Select customers
Internal users
Goal is to apply product as it is expected to be used
<u>Alpha tests</u>
Occur shortly after system has completed
Comprehensive internal test suite
<u>Beta tests</u>
Follow incorporation of fixes discovered during alpha test
Both alpha and beta tests series may be repeated number of times

Verification

Verification testing is designed to prove product
Meets specification
Is not as comprehensive as some of earlier system testing
The efficacy of this test suite is only as good as the specifications
May want to develop reduced regression suite for verification tests

- Validation
  - Intent of validation testing
    - Prove test suite is
      - Testing what its supposed to test
      - Make required measurements
        - Within required tolerance
      - It can catch faults
  - Executed
    - Prior to releasing tests to production
    - Whenever product or tests modified

- Acceptance Test
  - The acceptance test suite is set of tests customer uses
    - When accepting product
  - May include any or all of verification and validation tests
  - During production
    - May be randomly applied to ensure quality standards

Production Test

- Assumes system design is correct and meets specs
  - Is not developed to verify integrity of design
- May be subset of verification tests
- Goal two fold
  - Test system least amount to ensure quality system
    - Meets spec and will not be dead on arrival
  - Test as quickly as possible
    - Production testing does not add anything to product
      - It is a cost
    - If one could guarantee
      - Quality
        - Parts
        - Production
    - Production testing could go away

Self Test

- Series of built in tests system can execute
- Goal are to
  - Ensure system working
  - Basis for action if
    - Element fails
    - System locks up
- Two general categories
  - Those invoked on demand
    - Command
    - Push button somewhere
  - Those running in background

Demand

These are a sanity check to ensure system basically operational

Often done at power up

Report a status on completion

Word of caution when developing such tests

Process of simply executing test often requires

Most of system to be working

Background

Can be as simple as watchdog timer

Must be periodically reset by CPU

If it expires

Forces action

Extreme as system reset

Benign as warning or error message

Complex as test suite running in background

Can check

Busses - stuck lines

Memory

ROM - signature

RAM - failed or stuck bits

Math processing

Built in

A/D

Measure a known reference

D/A

Convert at cardinal points

Test A/D against D/A

Timers

Caution

Anything added to system for testing can fail as well

Summary

Design is process of translating customer requirements into working system

Complexity of contemporary systems

Demands more formal approach and methods

Following formal specification

Formulate a functional model

Then develop and refine the model

Eventually map functional model on to architectural design

Also called executive structure

Conclude with a working prototype
Test against original specification and requirements

The functional and executive models are graphical and hierarchical

Each constituent component is
An encapsulation unit of behavioral and structural characteristics

These models naturally facilitate
Capitalization and reuse of functional and architectural IPs.