

Software Engineering Principles

Instructor:

Frederick T. Sheldon

School of Electrical Engineering and Computer Science

Washington State University

Introduction to Software Engineering

AGENDA

- Definitions of Software Engineering & Terminology
- Introduce the Concepts of Software Product and Release
- System Engineering & Project Management
- Requirements & Specification
- Software Design
- Dependable Systems
- Verification & Validation
- CASE & Software Engineering Environments
- Management
- Evolution

DEFINITION OF SOFTWARE ENGINEERING

SE is concerned with the theories, methods and tools which are needed to develop the software for computer systems (e.g., aerospace, avionics, telecommunications, government, health care, etc.)

Different from other engineering disciplines because it is not constrained by materials governed by physical laws or by “manufacturing” processes.

Software Engineers model parts of the real world in software. Models are large and complex so they must be made visible in documents (e.g., requirements, design specifications, test reports, user manuals, etc.)

The goal is to produce practical *software* solutions in a cost effective way. Products that are reliable, robust, useable, flexible, maintainable, etc.)

SOFTWARE ENGINEERING BASIC TERMINOLOGY

Requirement. (1) A condition or capability needed by a user to solve a problem or achieve an objective. (2) A condition or capability that must be met or possessed by a **system** or system **component** to satisfy a contract, standard, **specification**, or other formally imposed **document**. The set of all requirements forms the basis for subsequent development of the system or system component. See also **requirements analysis**, **requirements phase**, **requirements specification**.

Requirements specification. A **specification** that sets forth the **requirements** for a **system** or system **component**; for example, a **software configuration item**. Typically included are **functional requirements**, **performance requirements**, **interface requirements**, **design requirements**, and development standards.

Specification language. A language, often a machine-processable combination of **natural** and **formal language**, used to specify the **requirements**, **design**, behavior, or other characteristics of a **system** or system **component**. See also **design language**, **requirements specification language**.

Design. (1) The process of defining the software architecture, components, modules, interfaces, test approach, and data for a software system to satisfy specified requirements. (2) The result of the design process.

SOFTWARE ENGINEERING BASIC TERMINOLOGY

(CONTINUED)

Design analysis. (1) The evaluation of a **design** to determine correctness with respect to stated **requirements**, conformance to design standards, **system efficiency**, and other criteria. (2) The evaluation of alternative design approaches.

Implementation requirement. Any **requirement** that impacts or constrains the **implementation** of a **software design**; for example, design descriptions, software development standards, **programming language** requirements, software **quality assurance** standards.

Implementation. (1) A realization of an **abstraction** in more concrete terms; in particular, in terms of **software**, or both. (2) A machine executable form of a **program**, or a form of a program that can be translated automatically to machine executable form. (3) The process of translating a **design** into **code** and **debugging** the code.

Walk-through. A process in which a designer or programmer leads one or more other members of the development team through a **segment** of **design** or **code** that he or she has written, while the other members ask questions and make comments about technique, style, possible **errors**, violation of development standards, and other problems. Contrast with **inspection**.

SOFTWARE ENGINEERING BASIC TERMINOLOGY

(CONTINUED)

Inspection. (1) A formal evaluation technique in which **software requirements, design, or code** are examined in detail by a person or group other than the author to detect **faults**, violations of development standards, and other problems. Contrast with **walk-through**. (2) A phase of quality control that by means of examination, observation or measurement determines the conformance of materials. (3) A phase of quality control that by means of examination, observation or measurement determines the conformance of materials, supplies, components, parts, appurtenances, **systems, processes** or structures to predetermined **quality requirements**.

Testing / Debugging. Testing is the process of exercising or evaluating a **system** or system **component** by manual or automated means to verify that it satisfies specified requirements or to identify differences between expected and actual results. Debugging is the process of locating, analyzing, and correcting suspected **faults**.

Integration / Integration testing. Integration is the process of combining **software** elements, **hardware** elements, or both into overall **system**. Integration testing is an orderly progression of **testing** in which **software** elements, **hardware** or both are combined and tested until the entire **system** has been integrated.

SOFTWARE ENGINEERING BASIC TERMINOLOGY

(CONTINUED)

Validation involves checking that the program as implemented meets the expectations of the software customer in such a way to ensure compliance with software **requirements**. See also **verification**.

Verification. (1) The process of determining whether or not the products of a given phase of the **software development cycle** fulfill the **requirements** established during the previous phase. See also **validation**. (2) Formal proof of **program** correctness. See **proof of correctness**. (3) The act of reviewing, inspecting, **testing**, checking, auditing, or otherwise establishment and documenting whether or not items, **processes**, services, or **documents** conform to specified **requirements**. (ANSI/ASQC A3-1978).

Independent verification and validation. (1) **Verification** and **validation** of a **software product** by an organization that is both technically and managerially separate from the organization responsible for developing the product. (2) **Verification** and **validation** of a **software product** by individuals or groups other than those who performed the original **design**, but, who may be from the same organization. The degree of independence must be a function of the importance of the **software**.

SOFTWARE ENGINEERING BASIC TERMINOLOGY

(CONTINUED)

Certification. (1) A written guarantee that a **system** or **computer program** complies with its specified **requirements**. (2) A written authorization that states that a **computer system** is secure and is permitted to operate in a defined environment with or producing sensitive information. (3) The formal demonstration of **system** acceptability to obtain authorization for its **operational** use. (4) The process of confirming that a **system, software subsystem, or computer program** is capable of satisfying its specified **requirements** in an operational environment. Certification usually takes place in the field under actual conditions, and is utilized to evaluate not only the software itself, but also the **specifications** to which the software was constructed. Certification extends the process of **verification** and **validation** to an actual or simulated operational environment. (5) The procedure and action by a duly authorized body of determining, verifying and attesting in writing to the qualifications of personnel, processes, procedures, or items in accordance with applicable requirements (ANSI/ASQC A3-1978).

Operation and maintenance phase. The period of time in the **software life-cycle** during which a **software product** is employed in its **operational** environment, monitored for satisfactory **performance**, and modified as necessary to correct problems or to respond to changing **requirements**.

SOFTWARE PRODUCTS ¹

Delivered to a customer with the documentation which describes how to install and use the system (may be packaged with hardware).

COTS - are *commercial-off-the-shelf* (generic) stand-alone systems which are sold on the open market to any customer.

Customized - systems commissioned by a particular customer and delivered by a particular contractor.

Product attributes include:

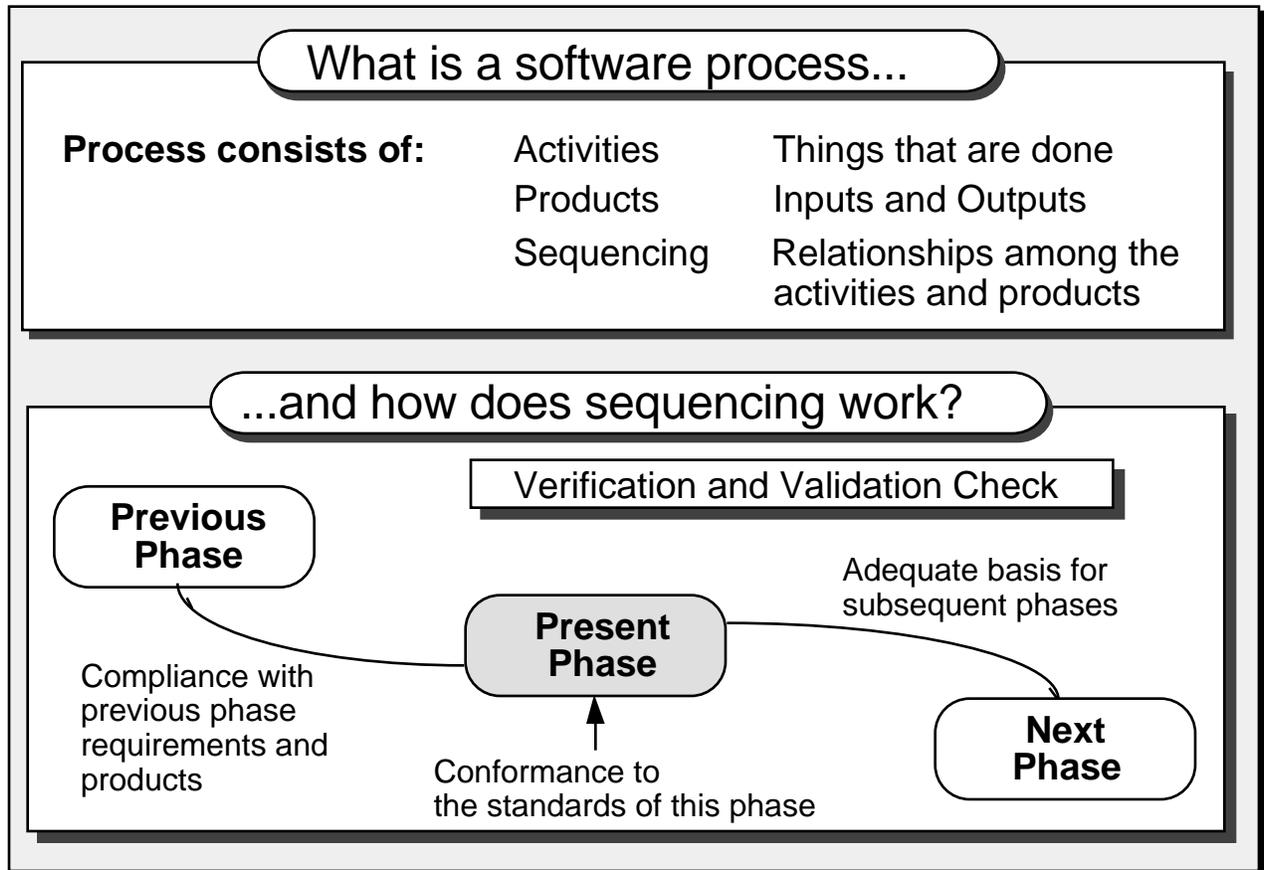
- ✱ Maintainability - how adaptable is the software to the changing needs of the customer.
- ✱ Dependability - how reliable, secure and safe (fault-tolerant) so that no physical or economic damage will occur in the event of a failure.
- ✱ Efficiency - how well does the system utilize the system resources.
- ✱ Useability - includes... learnability, speed of operation, robustness, recoverability, and adaptability (flexible to satisfy >1 work models).

¹ **Product Metrics.** Product metrics measure aspects relating to quality, customer satisfaction, and difficulty to produce, but "may not reveal anything about how the software has evolved into its current state." These indicators include: size - LOC, fault density/intensity, documentation, test sufficiency, prediction accuracy, and customer satisfaction.

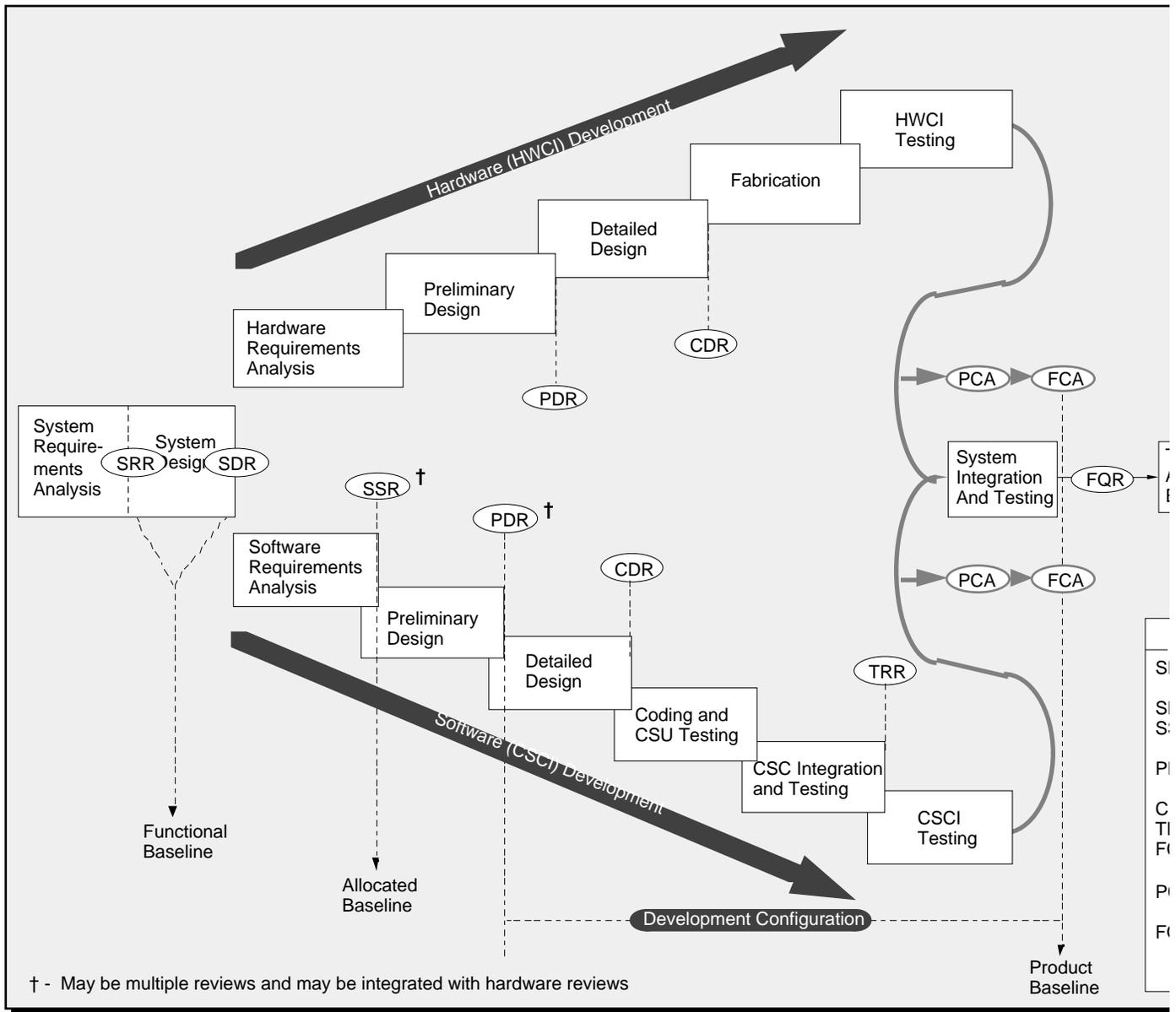
SOFTWARE PROCESS

Four fundamental process activities:

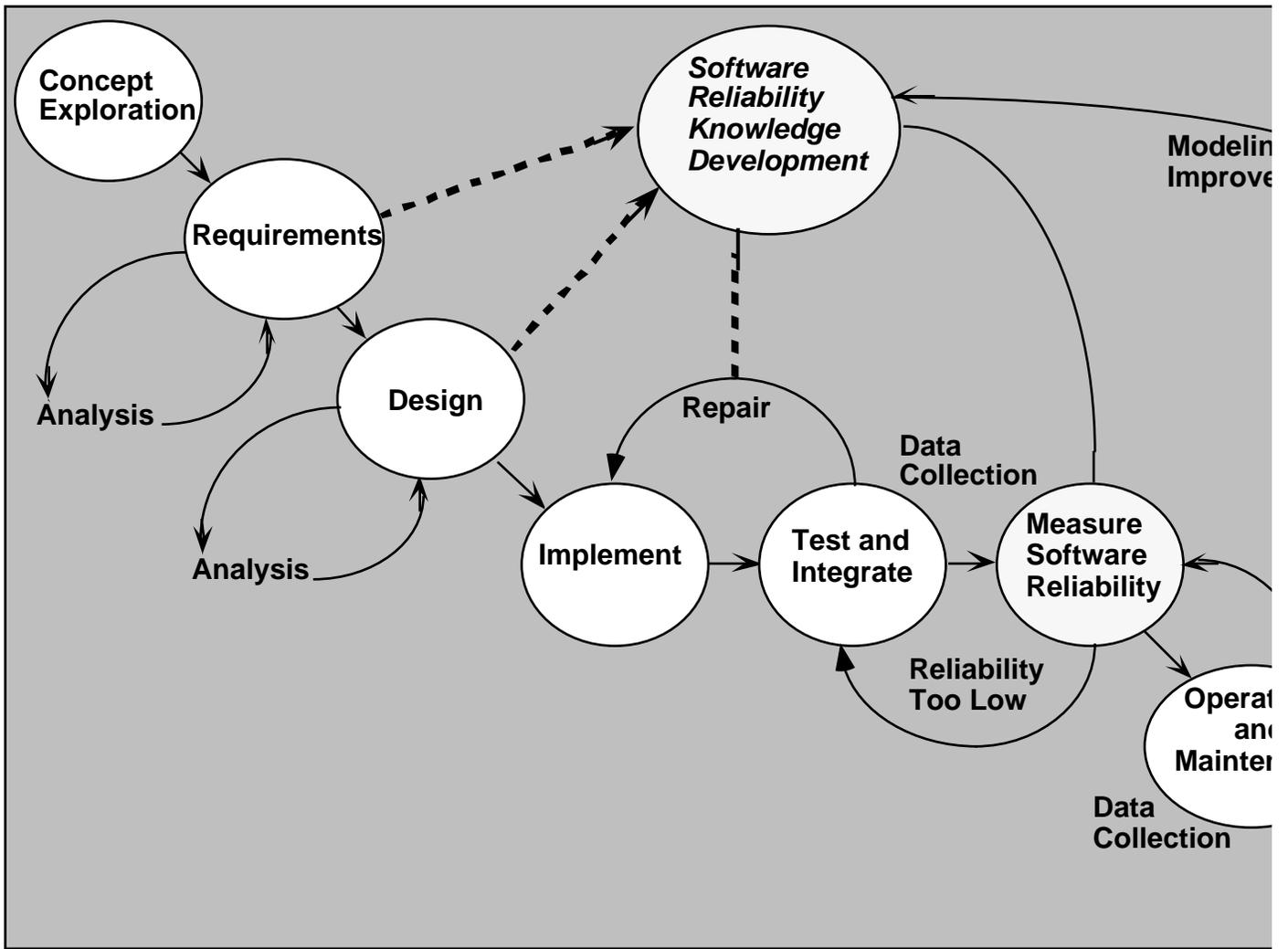
- * Software specification
- * Software development
- * Software validation
- * Software evolution



2167A PROCESS MODEL FOR DELIVERY OF CUSTOM HW & SW



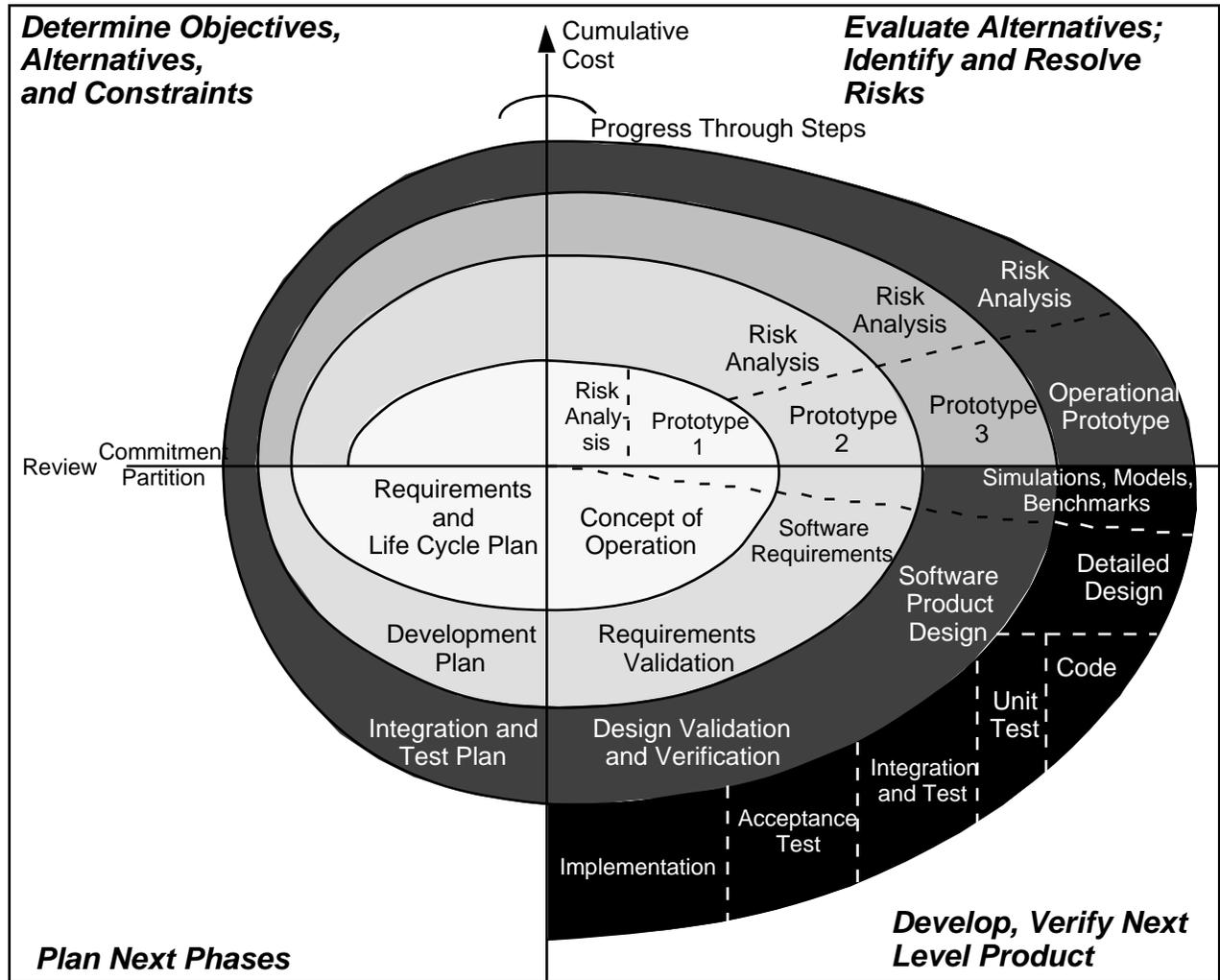
SOFTWARE PROCESS: WATERFALL MODEL ²



² Reliability measurement happens in conjunction with testing and integration, before the software is released into operations and maintenance. Reliability-model development is fed by activities in the requirements, design, repair, and operations and maintenance phases. During reliability-model development, you plan how to use the selected model, set a reliability objective, and initiate activities to support the level of sensitivity you need for data collection (calendar time, wall-clock time, or CPU execution time, for example). Reliability data collected from fielded software can be useful for evaluating the accuracy of predictions and recalibrating the reliability model. The reliability model, which incorporates project-specific constraints, tolerances, and sensitivities, should retain this information so that it yields more accurate measures when it is reused on future projects [Sheldon 92].

SOFTWARE PROCESS: SPIRAL MODEL

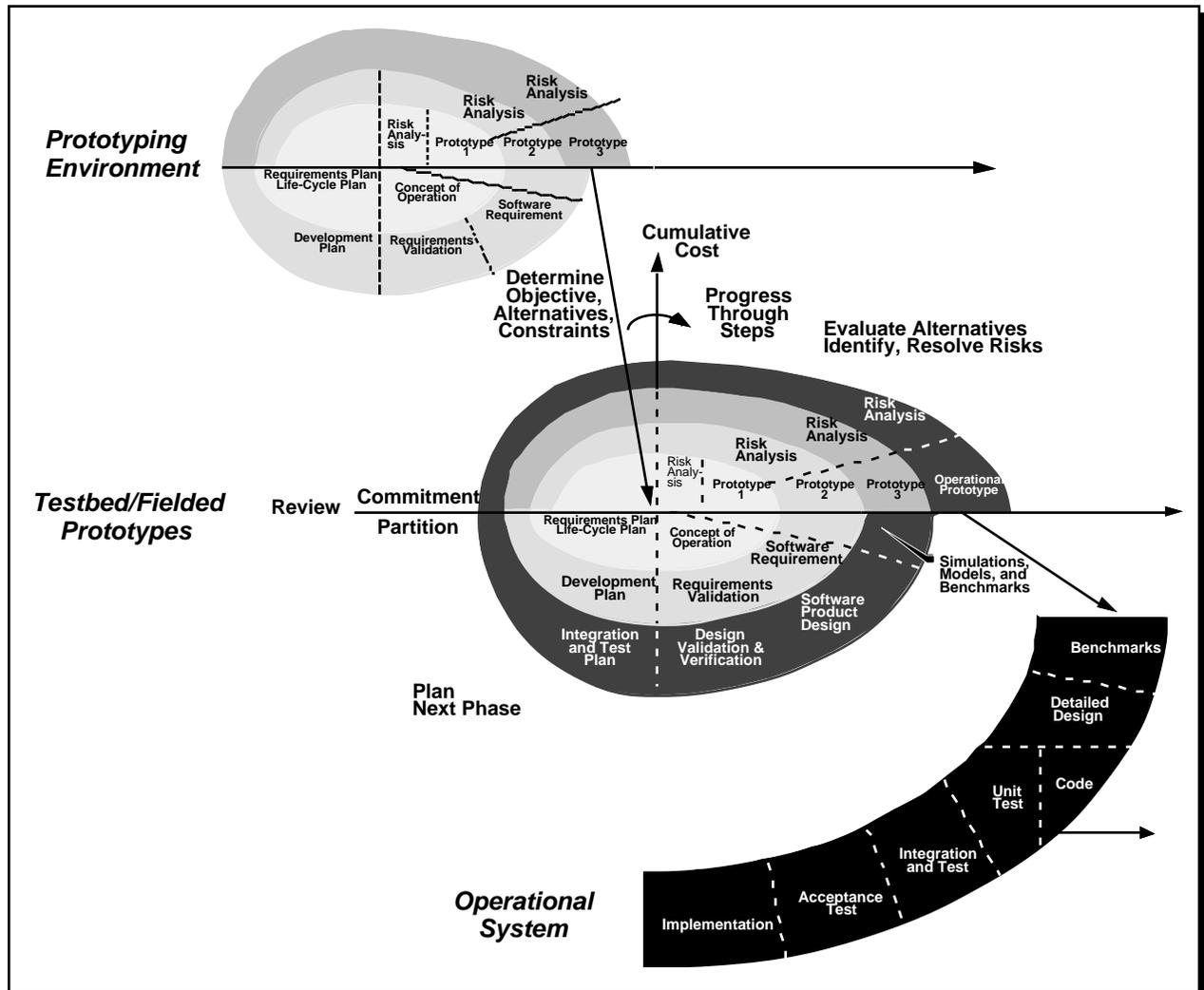
- ✧ Objective setting and validation
- ✧ Development and
- ✧ Risk assessment and reduction
- ✧ Planning



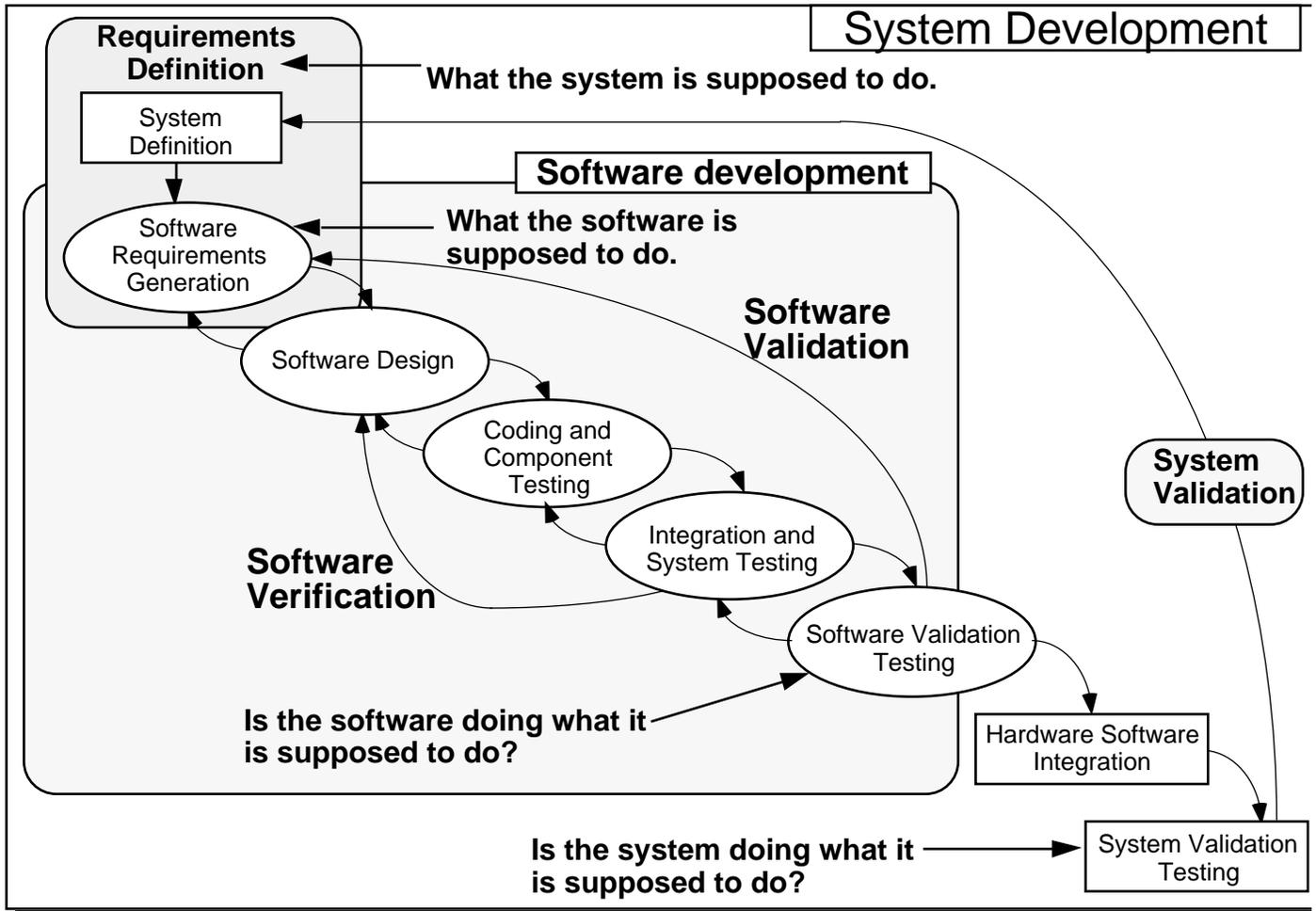
SOFTWARE PROCESS: STACKED SPIRAL MODEL

Risk Management

- 1) Objectives and constraints outcomes of strategies
- 2) Risk identification and risk resolution strategies
- 3) Results or
- 4) Plans for the next phase and commitment



SYSTEM VERSUS SOFTWARE V&V



SOFTWARE ENGINEERING INSTITUTE CAPABILITY MATURITY MODEL

Level	Characteristics	Key Challenges	Result
5 Optimizing	<ul style="list-style-type: none"> • Focus on process improvement • Data gathering is automated and used to identify weakest process elements • Numerical evidence used to justify application of technology to critical tasks • Rigorous defect-cause analysis and defect prevention • Focus on process optimization to reduce errors 	<ul style="list-style-type: none"> • Still human-intensive process • Maintain organization at optimizing level 	
4 Managed	(Quantitative) <ul style="list-style-type: none"> • Measured process: estimates/actuals, error-cause analysis • Minimum set of quality and productivity measurements established • Process database established & resources to analyze & maintain its data • Focus on technology management and insertion 	<ul style="list-style-type: none"> • Changing technology • Problem analysis • Problem prevention 	
3 Defined	(Quantitative) <ul style="list-style-type: none"> • Process defined and institutionalized • Software Engineering Process Group established to direct improvements • Focus on Software design skills, design tracking • Focus on various types of traceability 	<ul style="list-style-type: none"> • Process measurement • Process analysis • Quantitative quality plans 	
2 Repeatable	(Intuitive) <ul style="list-style-type: none"> • Process dependent on individuals • Established basic project controls with strength in doing similar work • Process faces major risk when presented with new challenges • Lacks orderly framework for improvement • Focus on collecting various types of "trend" data • Focus on management and tight project control 	<ul style="list-style-type: none"> • Training • Technical practices (reviews, testing) • Process focus (standards, process groups) 	
1 Initial	(Ad hoc/chaotic process) <ul style="list-style-type: none"> • No formal procedures, cost estimates, project plans • No management mechanism to ensure procedures are followed, tools not well integrated, and change control is lax • Senior management does not understand key issues 	<ul style="list-style-type: none"> • Project management • Project planning • Configuration management • Software quality assurance 	

To make orderly improvement, development and maintenance organizations should view their process as one that can be controlled, measured, and improved. This requires that they follow a traditional quality-improvement program such as that described by W. E. Deming. For software, this involves the following six steps:

1. Understand the current status of their process.
2. Develop a vision of the desired process.
3. Establish a list of required process-improvement actions in priority order.
4. Produce a plan to accomplish these actions.
5. Commit the resources and execute the plans.
6. Start over at step 1.

The SEI has developed a framework to characterize the software process across five maturity levels. By establishing their organization's position in this framework, software professionals and their managers can readily identify areas where improvement actions will be most fruitful. Many software organizations have found that this framework provides an orderly set of process improvement goals and a helpful yardstick for tracking progress.

COMPUTER BASED SYSTEMS ENGINEERING

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

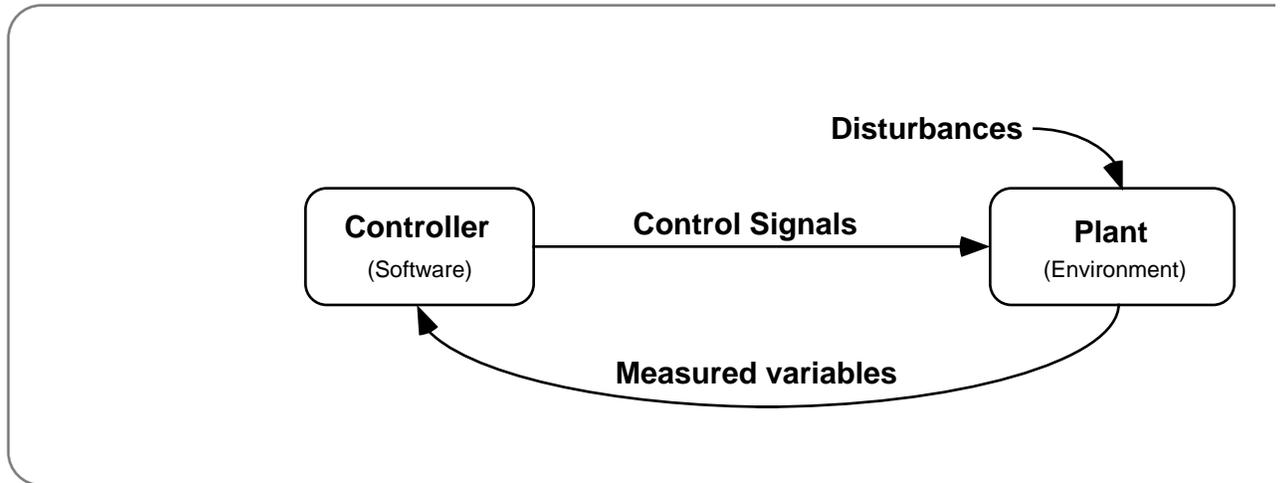
Complex relationships among system components involve emergent properties such as:

- 1) Deadlock, liveness, and safety
- 2) Reliability and performance
- 3) Usability

Understands the environment within which the system will be operating

- 1) Domain knowledge (e.g., Air traffic control)
- 2) For example embedded real-time systems (see next slide)
- 4) Reasoning about system models

EMBEDDED REAL-TIME SYSTEM ENVIRONMENT



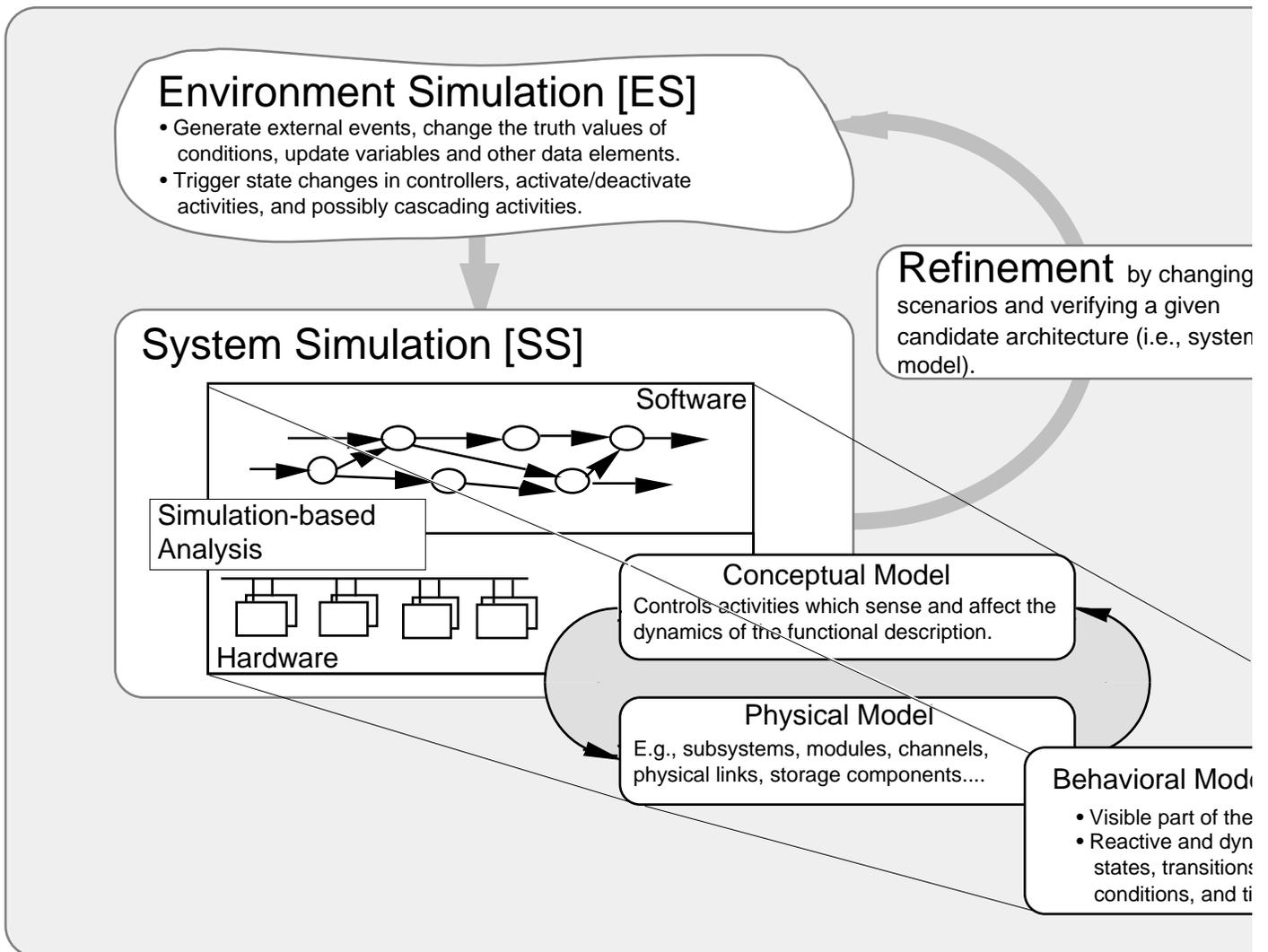
The "open-loop" behavior of the plant (without the controller) is usually unsatisfactory in some important respect.

The lag time or response time of the controller is determined by the (physical) nature of the processes in the plant.

The primary goal is to ensure the correct behavior of the plant.

This is achieved by designing a controller that will interact with the plant in such way that the correct functioning of the plant is ensured.

REASONING ABOUT SYSTEM MODELS



FRAMEWORK FOR REASONING ABOUT SYSTEM MODELS

		Stages of development			
		Specification	Design	Implementation	Testing
Views of the System	Functional	X			
	Structural	X	X		
	Behavioral	X	X	X	X
	Action-Semantic		X	X	X

Various views of the system are needed to describe its intended and actual operation [Harel 86]. These views are relevant at each stage of the system development.

The functional view shows the system as a set of entities performing relevant tasks. Usually the description of the tasks is abstract.

The structural view shows how the system is put together: the components, interfaces, and flow between them. This view also shows the environment and its interfaces, and information flows between it and the system. Ideally the structural view is an elaboration of the functional view.

The behavioral view shows the way the system will respond to specific inputs: what states it will adopt, what outputs it will produce, what boundary conditions exist on the validity of inputs and states. This includes a description of the environment that produces the inputs and consumes the outputs.

Action-Semantics View describes the domain of inputs, range of the outputs, and the meaning of the input-output transformation during each state transition (including side effects, and accuracy). Action-semantics aids in validation of the implementation (as per requirements).

SYSTEMS ENGINEERING PROCESS

Distinctions from Software Engineering Process

- 1) Interdisciplinary involvement
- 2) Reduced scope for iterations between phases
- 3) Usability

System requirements definition: functions, properties, characteristics

System design: partition requirements, identify sub-systems and assign requirements, specify sub-systems functionality, define sub-system interfaces

Sub-system development: design and implementation

System integration: incrementally add new sub-systems into the system at large

System installation must address: environment (platform, OS), human resistance, coexistence with other systems / versions, physical installation problems

System evolution: evaluate proposed changes, decide to fix or replace, etc.

System decommissioning: taking the system out of service

PROJECT MANAGEMENT

Motivation: Software is intangible
 No standard production process
 Typically software is over budget and
behind schedule (late)

Activities:

- 1) Project costing, planning and scheduling
- 2) Risk analysis and resolution
- 3) Project monitoring and reviews
- 4) Personnel selection and evaluation
- 5) Report writing and presentation

REQUIREMENTS AND SPECIFICATION

Requirements engineering

Requirements analysis

System models

Requirements definition and specification

Software prototyping

Formal specification

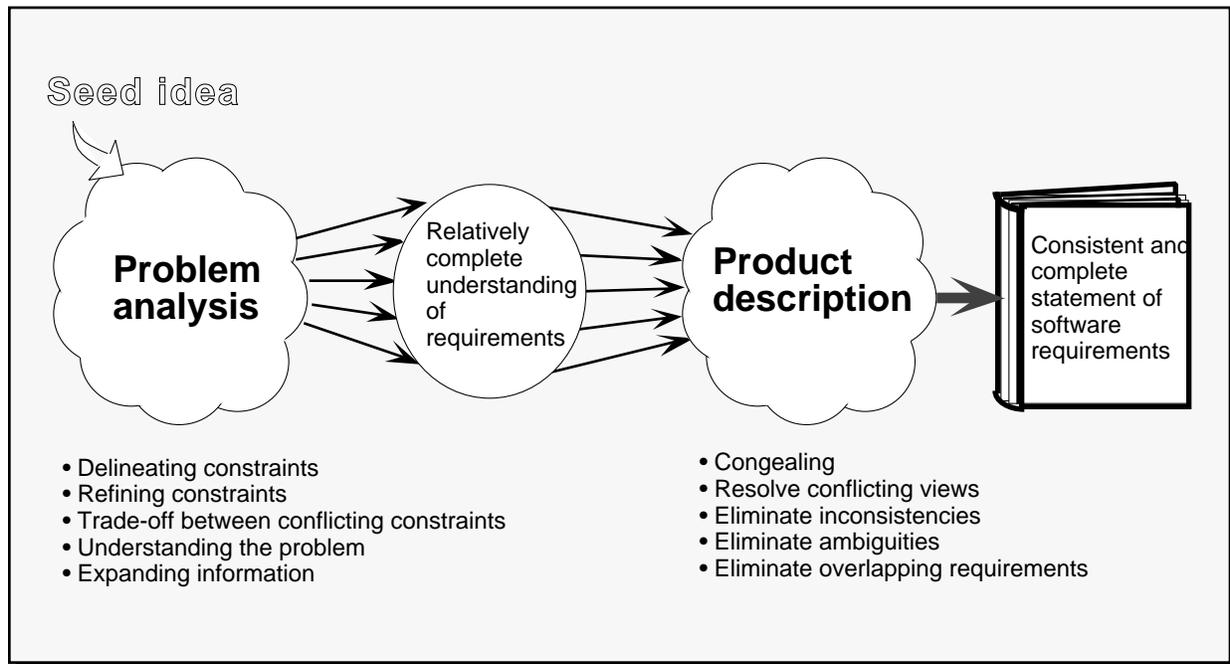
Algebraic specification

Model-based specification

REQUIREMENTS ENGINEERING

Four principle stages in this process:

- 1) Feasibility study
- 2) Requirements analysis specification
- 3) Requirements definition
- 4) Requirements



REQUIREMENTS ENGINEERING ISSUES

First phase of large-scale software system development.

Requirements must be written at different levels of detail for different readers.

Organized for presentation to customers, users and engineers!

Requirements validation “ensures” that the system will meet the customers needs:

- 1) Verifiability (is the requirement as stated testable?)
- 2) Traceable (is the origin of the requirements clearly stated?)
- 3) Comprehensible (is the requirement properly understood by procurers or end-users?)
- 4) Adaptable (can the requirement be changed without large-scale effects?)

Requirements always change / evolve: Some are *enduring* and others are *volatile* (e.g., *mutable* [due to environmental changes], *emergent* [due to customer understandings], *consequential* [organizational processes are changed] and *compatible* [with other business systems or processes]).

REQUIREMENTS ANALYSIS

Is an iterative process which involves domain understanding, requirements collection, classification, structuring, prioritization and validation.

Viewpoints may be based on sources or sinks of data, different models of the system expressed using different notations or external interaction with the system.

Methods include a set of activities, associated notations, rules for governing the use of notations, guidelines for defining good practice and standard forms or reports used to document the analysis (e.g., viewpoint-oriented method is based on data and control requirements for services delivered to a particular viewpoint).

Important to define the boundaries between a system and its environment. Consequently, as part of the analysis process, the system's environment or context is studied.

SYSTEM MODELS

An abstract view of a system which ignores some system details. Complementary system models can be developed which present different information about the system.

Data-flow diagrams model the data processing. The system is usually modeled as a set of data transformations with functions acting on the data.

Semantic data models describe the logical structure of the data.

Object models:

- 1) Describe logical entities, classification and aggregation
- 2) Combine data with processing models
- 3) Describe interfaces in an abstract way
- 3) End users find hard to understand

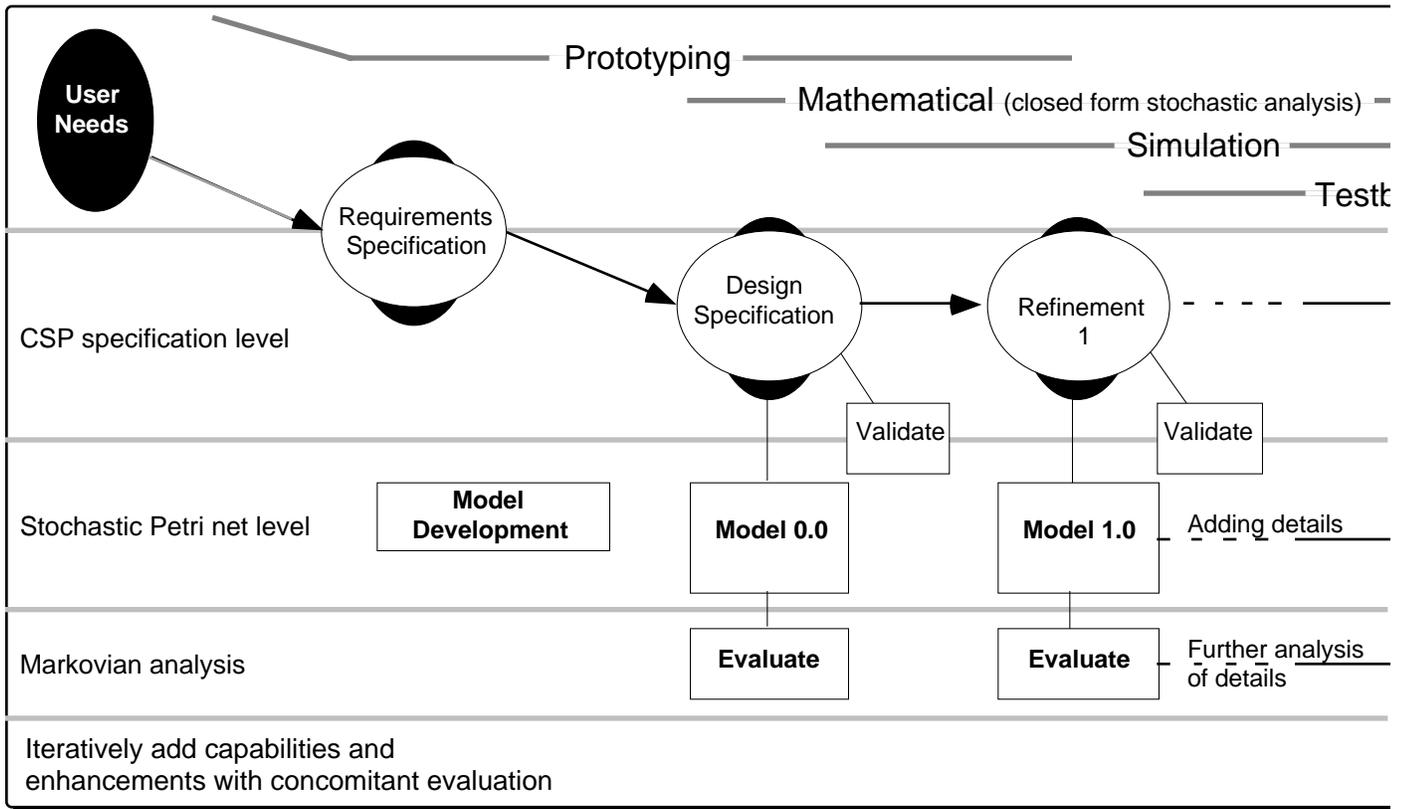
Data dictionary is an important tool for maintaining information about the system entities throughout the lifetime of a project. Supports any kind of system model.

ITERATIVE REFINEMENT OF SYSTEM MODELS

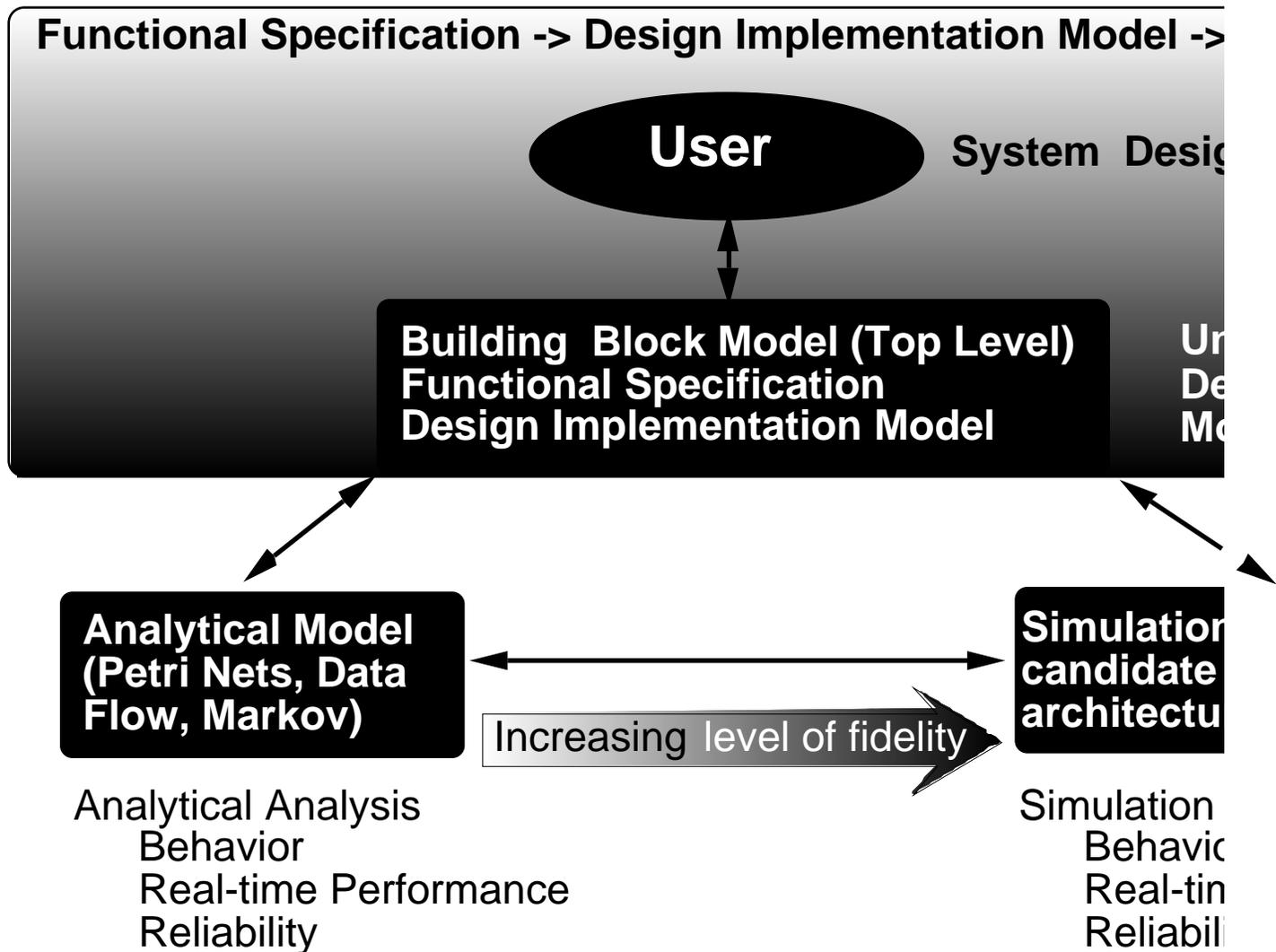
Specify Model Evaluation Feedback and
refinement

An Example: CSP Petri nets Markovian analysis

analysis Annotate the specification



APPROACH SPECIFICATION AND ANALYSIS FOR DEPENDABLE SYSTEMS



Framework – *Specification, Design, and Refinement*

APPROACH OBJECTIVES

Overall goal is to identify strengths and weaknesses of a set of candidate software architectures.

Convert a formal description of the system into the information needed for simulation.

Converting a formal description of a system into the information needed for a simulation

Develop a model which can predict system *behavior* as a function of observable parameters.

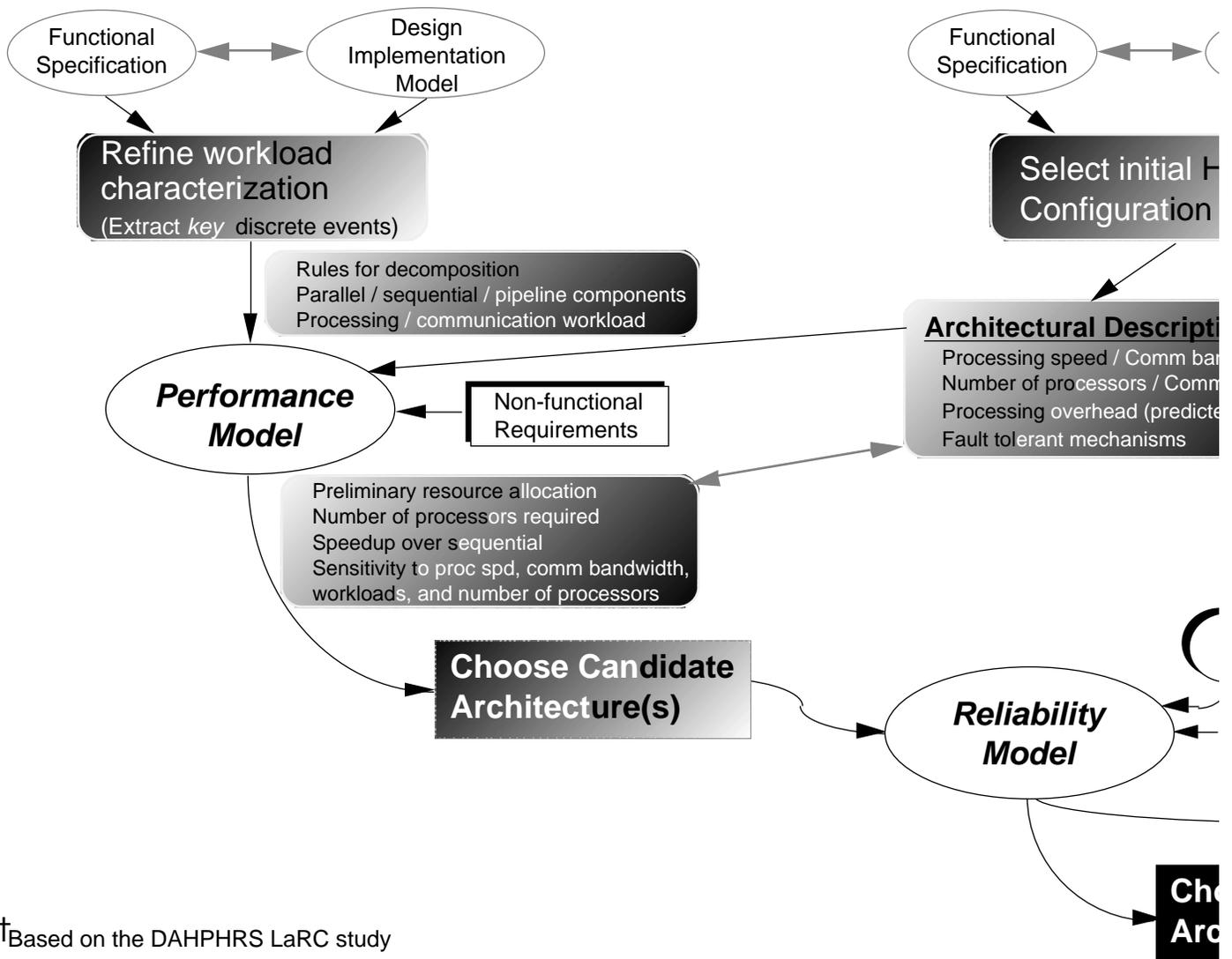
- **Performance analysis** - worst case latencies, system overhead as a function of workload, recovery performance, etc.
- **Reliability analysis** - probability of system failure / mission time.

Carefully enumerate modeling assumptions.

Estimate and measure model parameters.

Solve the model for specific values of the parameters.

DESIGN IMPLEMENTATION MODELING FRAMEWORK



†Based on the DAHPHS LaRC study

Framework – *Specification, Design, and Refinement*

Build-Up – *Integration and Experimentation*

EXAMPLE: HOW THE GCS SIMULATION AND SPECIFICATION EVOLVED

GCS: Guidance and Control Software

Start with an external view of the target vehicle system.

GCS is decomposed in a subsystem block diagram.

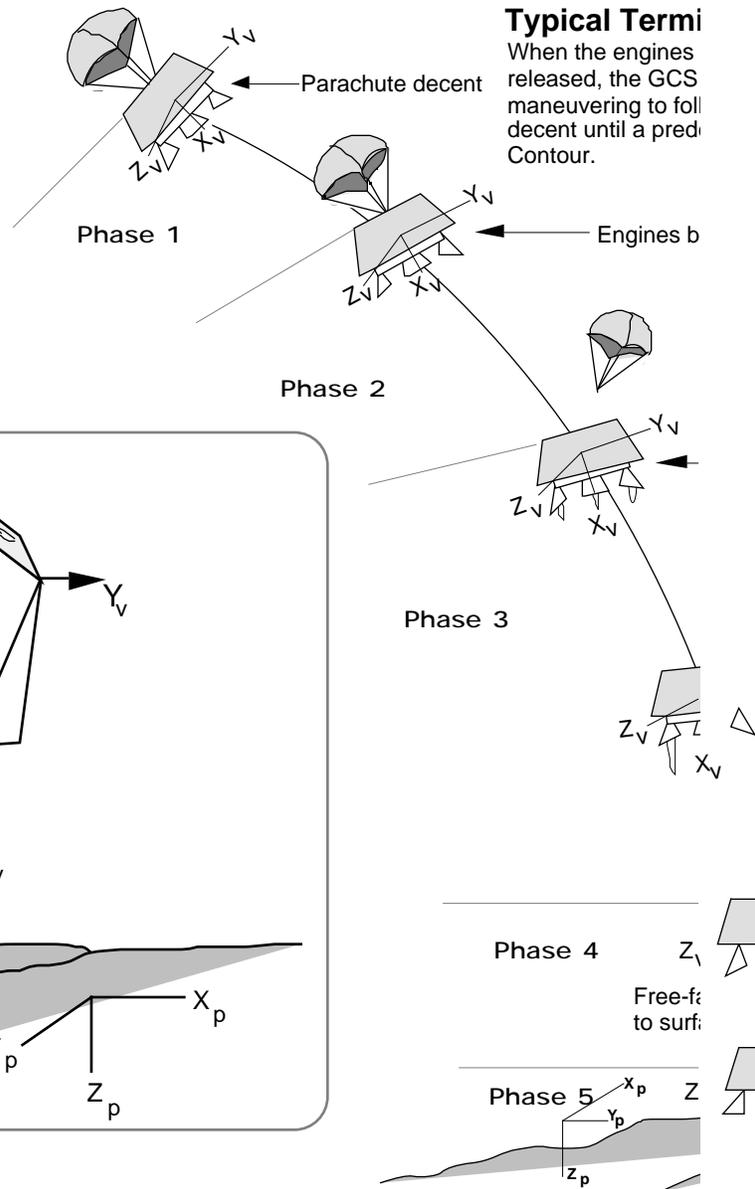
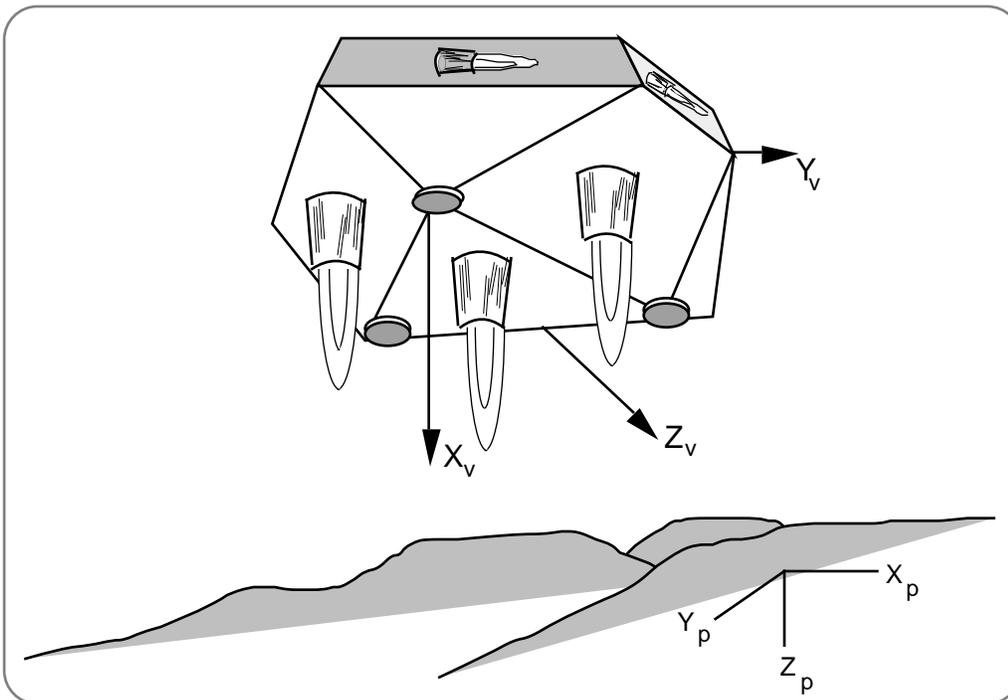
Simulation model is abstracted and specified using a simulation tool (e.g., SES/Workbench).

Processing rate schedule is developed from timing analysis results

EXAMPLE TARGET SYSTEM: ON-BOARD NAVIGATIONAL SOFTWARE VIKING MARS LANDER

Guidance and Control Software:

1. Provide guidance and engine control of the vehicle during its terminal phase of descent onto a surface.
2. Communicate sensory information about the vehicle and its descent to some other receiving device.

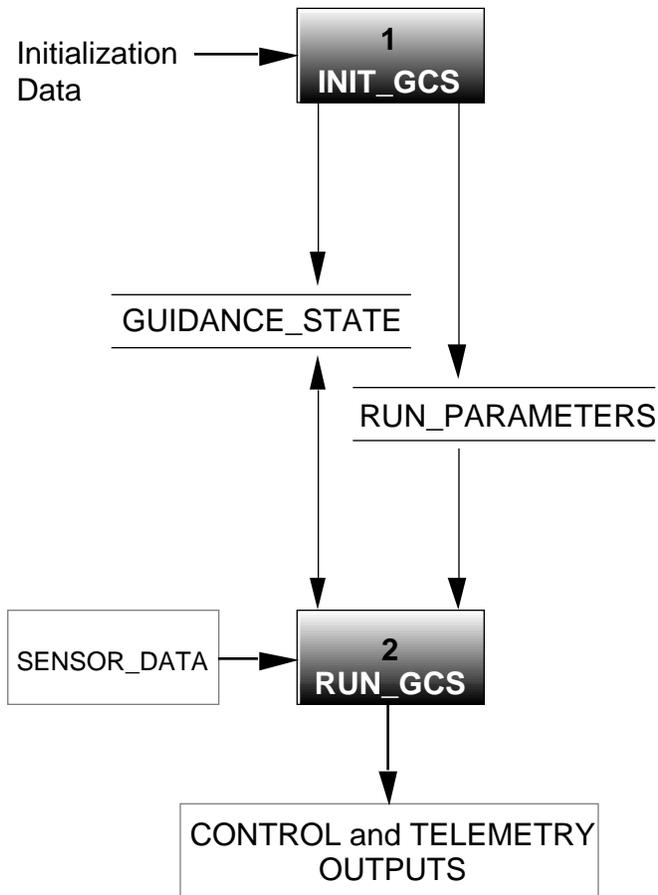


Typical Termi

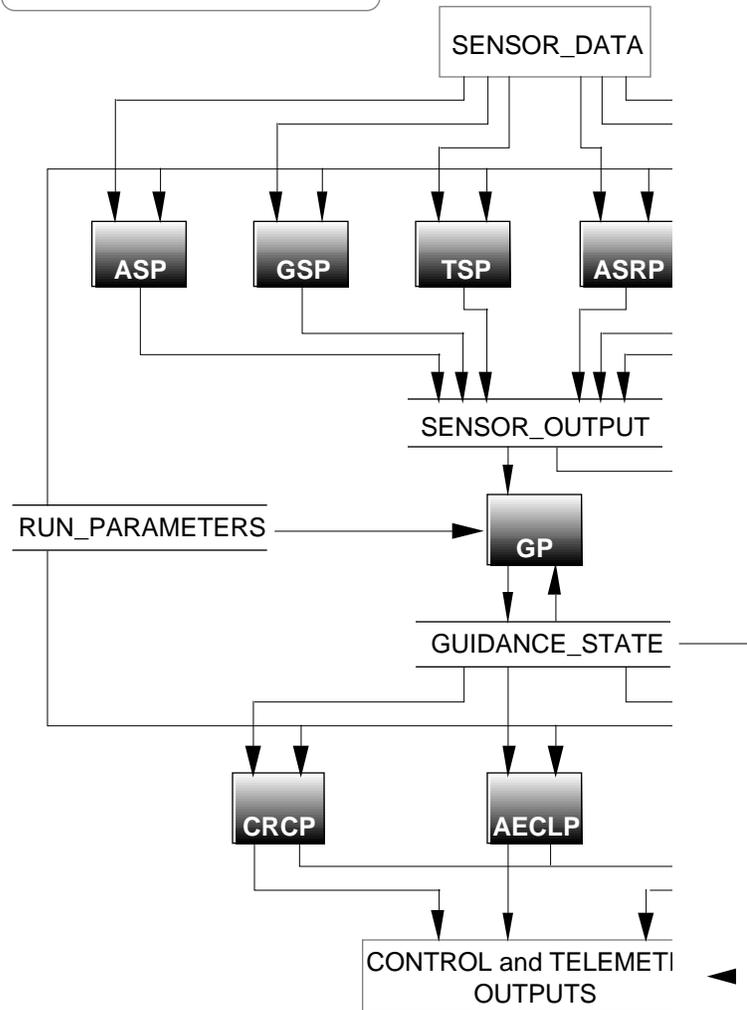
When the engines released, the GCS maneuvering to foll decent until a pred Contour.

EXAMPLE: GCS LEVEL 1 AND LEVEL 2 SPECIFICATION

Level 1 Specification 3.1 PROCESS 0.GCS



Level 2 Specification 4.1 PROCESS 2.RUN_GCS



REQUIREMENTS DEFINITION AND SPECIFICATION

Definition is for users and procurement and is organized / written in a natural language, tables and diagrams.

Rationale provides understanding of the consequences with respect to changes

Must be clear and verifiable.

Requirements specifications communicate a precise and unambiguous description of system functionality.

Non-functional requirements - may constrain both the software process and the software product.

SOFTWARE PROTOTYPING

Developed to give the end user a concrete impression of the system capabilities

Establishes and validates system requirements.

May be *Throw-away* or *Evolutionary*....

System structure in a evolutionary prototype becomes corrupted by constant change. Changes and/or updates become increasingly difficult.

Initially, develop the parts you understand least or best depending on the type of prototype approach used: Throw-away or Evolutionary (respectively).

Rapid development is important (leave out some functionality and details, relax the non-functional constraints).

Very useful for such application fragments as Graphical User Interface (GUI pronounced “goo-ee”).

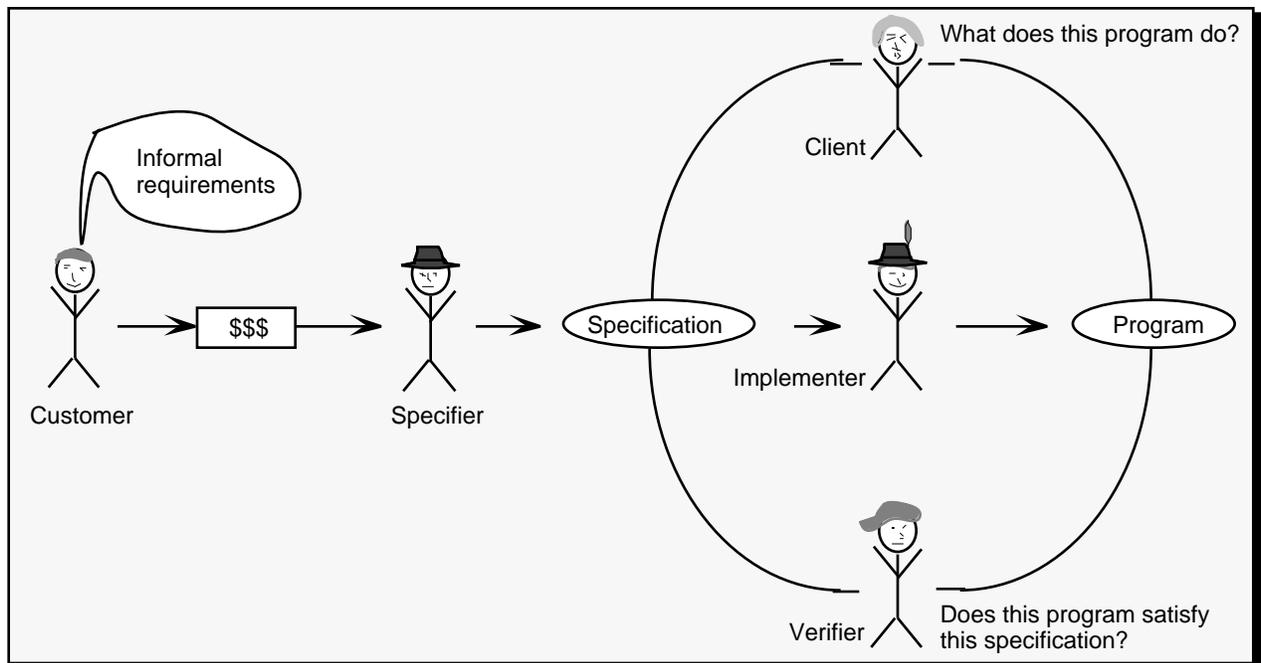
FORMAL SPECIFICATION

Compliment informal specification techniques by removing areas of doubt (since they are precise and unambiguous).

Forces an analysis of system requirements at an early stage.

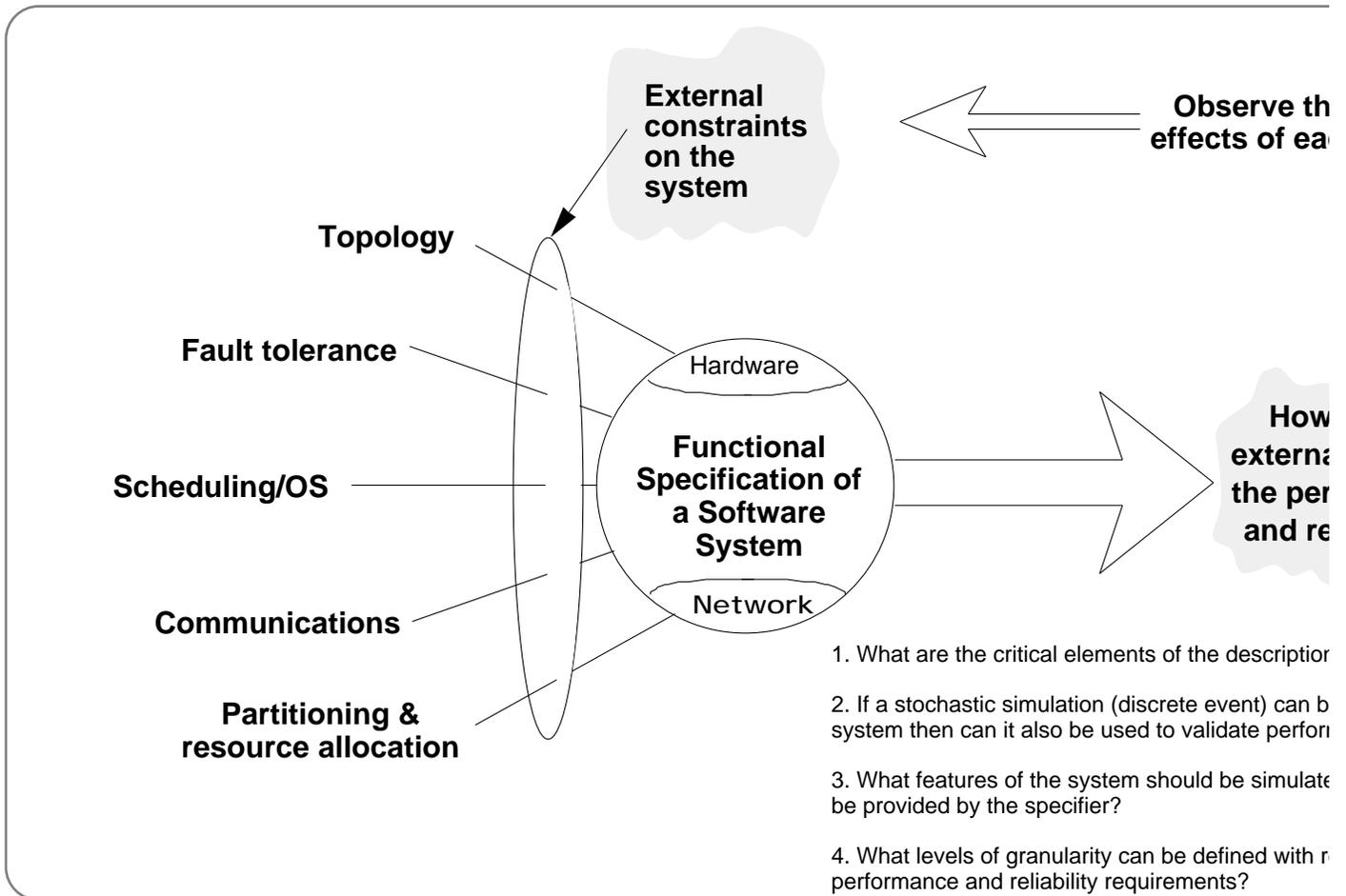
Unlikely to be cost effective in foreseeable future for the typical interactive application.

Most applicable in the development of safety critical systems and standards.



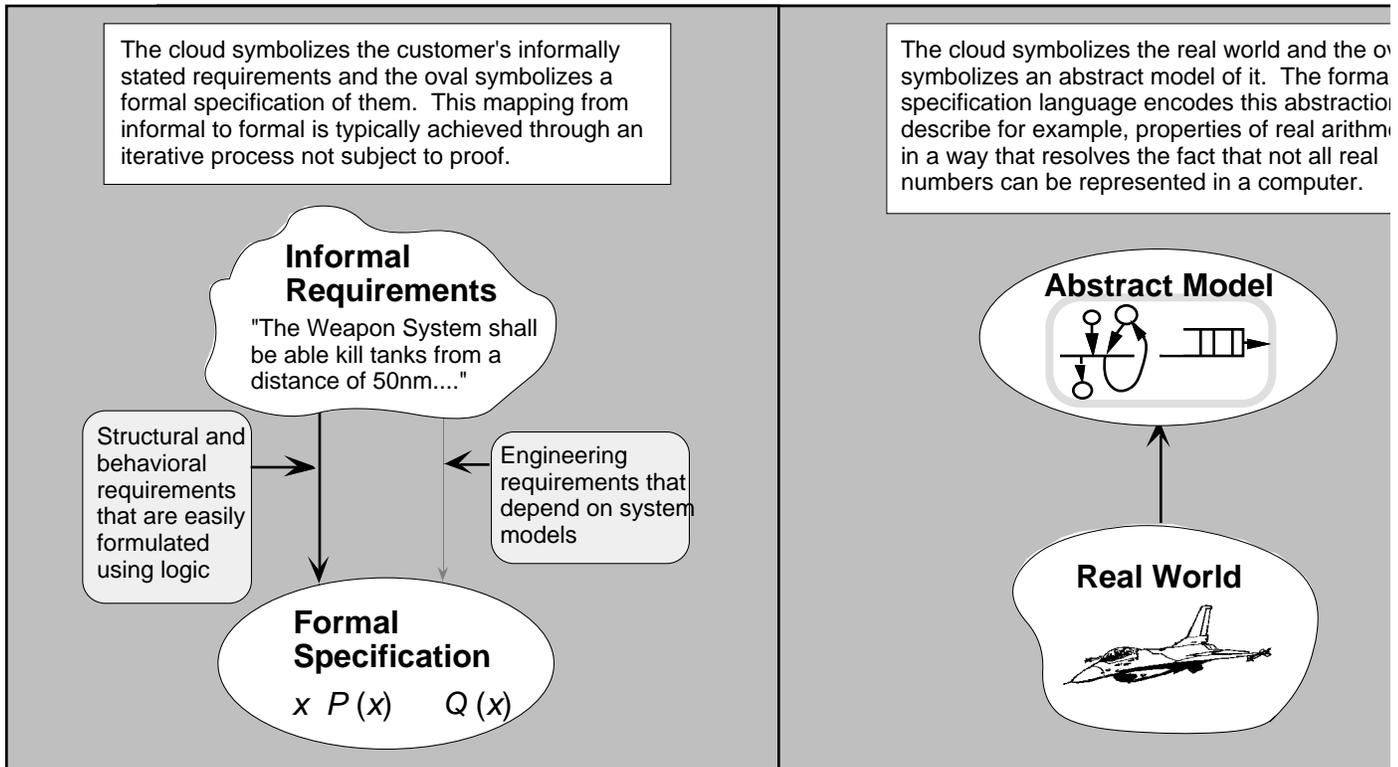
FORMAL SPECIFICATION SUPPORTS ANALYSIS

Convert a formal description of a system into the information needed for simulation to predict system behavior as a function of observable parameters.



FORMAL SPECIFICATION SUPPORTS HIGH ASSURANCE SYSTEM DEVELOPMENT

Informal requirements -> formally specified -> abstract
model -> real world



ALGEBRAIC SPECIFICATION

Is a particularly appropriate technique when interfaces between software systems must be specified.

Involves designing the operations on an abstract data type or object and specifying them in terms of their interrelationships.

Consists of 2 formal parts: A signature part (operations and their parameters are defined); and an axiom part (relationships between the operations are defined).

Formal specifications are (should be) associated with informal descriptions to make the formal semantics more understandable.

Complex formal specifications are constructed from simple building blocks. Specifications can be developed from simpler specifications by instantiating a generic specification, incremental specification development and specification enrichment.

MODEL-BASED SPECIFICATION

Relies on building a model of the system using mathematical entities such as sets which have a formal semantics. Z (pronounced “zed”) is based on typed sets.

Z specifications consist of a mathematical model of the system state and a definition of operations on that state.

A Z specification is presented as a number of schemas, where a schema introduces some typed names and defines predicates over these names. Schema in Z may be distinguished from surrounding text by graphical highlighting.

Schemas are combined and used in other schemas. The effect of including a schema A in schema B is that B inherits the names and predicates of A.

Operations may be specified in Z by defining their effect on the system state. It is normal to specify operations incrementally and then combine the specification fragments to produce the complete specification.

Z functions are sets of pairs where the domain of the function is the set of valid inputs. The range is the set of associated outputs. If ordering is important, sequences can be used for ordering (sets are unordered).

SOFTWARE DESIGN

Software design principles - activities and decomposition objectives.

Architectural design - deriving an overall structural model of the system.

Object-oriented design - a means of designing with information hiding.

Function-oriented design - identify functions which transform their inputs to create their outputs.

Real-time systems design - hardware and software co-design, where correctness depends on both timeliness and integrity of outputs.

User interface design - a user centered cognitive process.

SOFTWARE DESIGN PRINCIPLES

Design is a creative and innovative process (good judgment and flair are required)

Main activities:

- Architectural design - subsystems identified, defined and documented

- System specification - subsystem services and constraints

- Interface design - define common subsystem boundaries

- Component design - services allocated to components

- Data structure design - detailed elaboration of supporting structures

- Algorithm design - detailed elaboration

Functional decomposition involves modeling the system as a set of interacting functional units. Object-oriented decomposition models the system as a set of objects where an object is an entity with state and functions to inspect and modify that state.

Function-oriented and object oriented design are complementary rather than opposing design strategies.

Different perspectives may be applied at different levels of abstraction.

SOFTWARE DESIGN: COHESION

Cohesion is a measure of how closely the parts of a component relate to each other.

- 1) Coincidental cohesion - unrelated parts bundled together!
- 2) Logical association - related components bundled together (e.g., input and error handling).
- 3) Temporal cohesion - all elements are activated at a single time.
- 4) Procedural cohesion - elements in a component make up a single control sequence.
- 5) Communication cohesion - All elements of a component operate on the same input -> output.
- 6) Sequential cohesion - Output from one element in the component serves as input for another element.
- 7) Functional cohesion - each component part is necessary for the execution of a single function.

SOFTWARE DESIGN: COUPLING AND MAINTAINABILITY

Coupling is a measure of the strength of component interconnections. Designers should aim to produce strongly cohesive and weakly coupled design.

- 1) Tightly coupled modules use shared variables or exchange control information (common and control coupling).
- 2) Loose coupling is achieved by ensuring that details of the data representation are held within a component.
- 3) Component interface with other components through a parameter list.
- 4) If shared information is necessary, the sharing should be limited to those components which are need access to the information.
- 5) Globally accessible information should be avoided when ever possible.

Maintainability is an important design quality attribute. Maximizing cohesion and minimizing the coupling between modules / components makes them easier to change. Understandability and adapatability are also important for maintainability.

ARCHITECTURAL DESIGN

Deriving an overall structural model of the system which identifies sub-systems and their relationships. Architects may also design a control model for the system and decompose sub-systems into modules.

Large systems are usually heterogeneous and incorporate different models at different levels of abstraction.

Decomposition includes: 1) repository models, 2) client-server and 3) abstract machine (or layered) models.

Control models use centralized control and event models.

Modular decomposition models include data-flow and object models.

Domain specific architectural models are abstractions over an application domain. They may be generic models which are constructed bottom up from existing systems or reference models which are idealized, abstract models of the domain.

OBJECT-ORIENTED DESIGN

A means of designing with information hiding. Information hiding allows the information to be changed without other extensive system modifications.

An object is an entity which has a private state. It will have constructor and inspection functions allowing its state to be inspected and modified. The object provides services (operations using state information) to other objects.

Object identification is a major problem (consider the nouns [objectives] and verbs [operations]). Otherwise identify tangible entities using behavioral and/or scenario analysis.

Object interfaces must be defined precisely (use of an OO programming language)

A hierarchy chart showing objects and their sub-objects is developed.

An object interaction network may be created to show which objects call on the services of which other objects.

Objects may be implemented sequentially or concurrently.

FUNCTION-ORIENTED DESIGN

Relies on identifying functions which transform their inputs to outputs and share some global system state.

Business / transaction processing systems are naturally functional.

Process identifies transformations and decomposes those functions into sub-functions describing the operation and interface of each including flow of control.

Data flow diagrams (DFDs) are a means of documenting end-to-end flow. Structure charts represent hierarchical organization. Control can be documented using PDL.

DFDs can be implemented directly as a set of cooperating sequential processes. Each transform in the DFD is a separate process (which may realized as a separate procedure in the sequential program).

Often a heterogeneous (functional / object-oriented) approach is used.

REAL-TIME SYSTEMS DESIGN

Such systems are responsive / reactive in relationship with external events. Correctness is a function of both logical and timing accuracy.

Delay partitioning of a design into hardware and software until as late as possible in the design process.

Architectural design involves organizing the system as a set of interacting, concurrent processes.

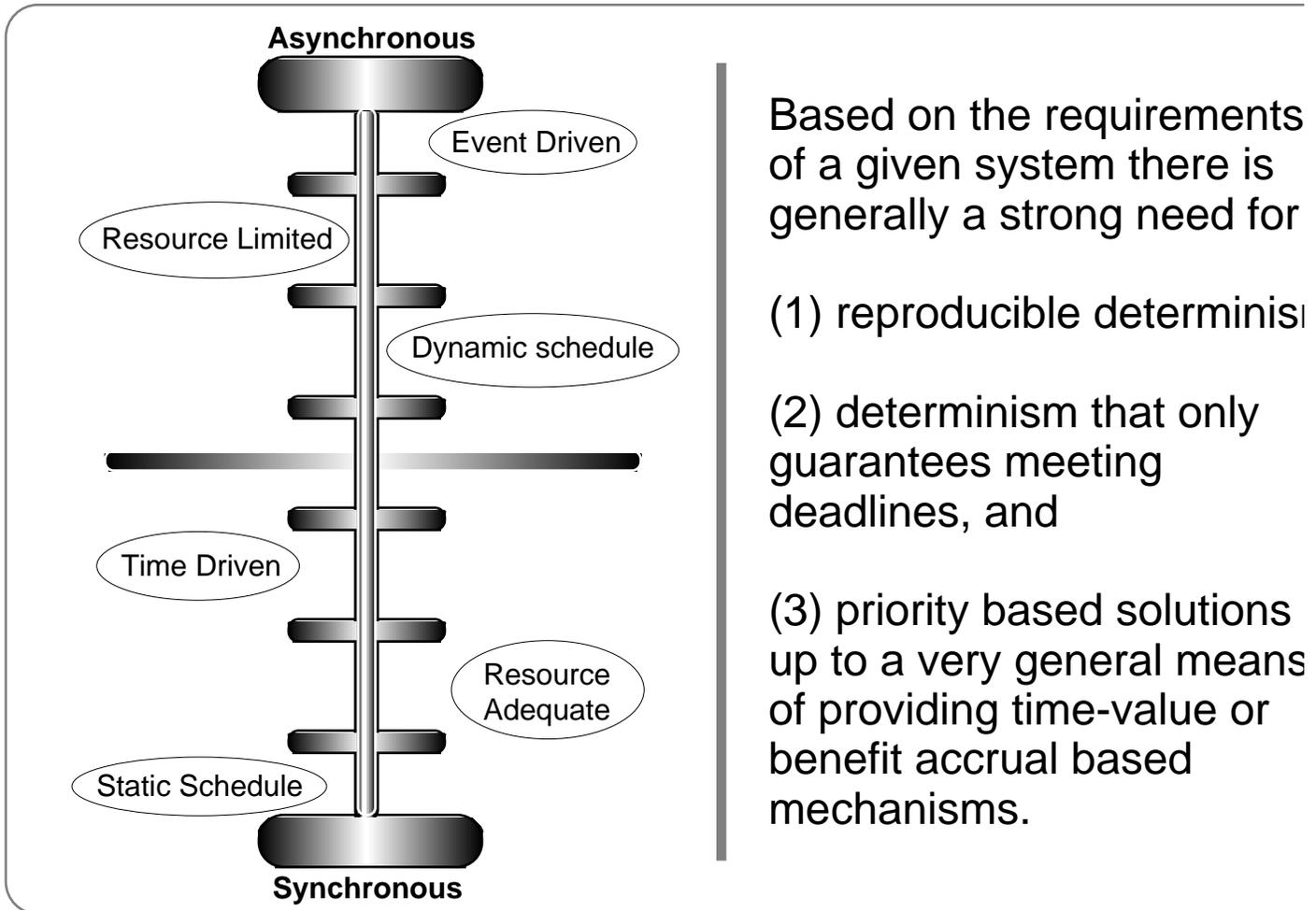
Associate a process with each class of sensor or actuator device. Other coordination processes may also be required.

A real-time executive is responsible for process and resource management (e.g., a scheduler/dispatcher decides which process to run based on their priority).

Monitoring and control systems periodically poll the sensors to acquire a snapshot of the systems environment and issue commands to the actuators.

Data acquisition systems are developed around the consumer/producer model.

PHILOSOPHIES AND GENERAL PROPERTIES OF REAL-TIME SYSTEMS



Based on the requirements of a given system there is generally a strong need for

(1) reproducible determinism

(2) determinism that only guarantees meeting deadlines, and

(3) priority based solutions up to a very general means of providing time-value or benefit accrual based mechanisms.

USER INTERFACE DESIGN

A user-centered process to develop an interface in which humans interact. Facilities to help users with the system and recover from their mistakes.

Menu systems provide a low learning overhead.

Graphical and digital information is combined and displayed to the user. Color must be used sparingly (many people are color blind).

Two kinds of help are “Help! I’m in trouble” and “Help! I need information.”

Error messages should not suggest the user is to blame. They should offer suggestions on how to repair the error.

User documentation should include beginner’s and reference manuals. Separate documents for the system administrators should be provided.

DEPENDABLE SYSTEMS

Software reliability - most important dynamic characteristic
(cost of failure)

- 1) Software reliability metrics and specification.
- 2) Statistical testing.
- 3) Reliability growth modeling.

Programming for reliability

- 1) Fault avoidance and fault-tolerance.
- 2) Exception handling and defensive programming (against corrupted data or links).

Software reuse

- 1) Software with and for reuse.
- 2) Generator based reuse.
- 3) Application system portability.

Safety critical software

- 1) Primary and secondary safety critical software categories.
- 2) Hazard analysis is a key activity in the safety specification process.
- 3) Safety proofs show that an identified hazard can never occur.

TESTING, VERIFICATION AND VALIDATION

The testing process

- 1) Unit/module, sub-system, system testing and finally acceptance testing

Test planning and strategies

- 1) Requirements traceability, tested items, testing schedule and recording procedures.
- 2) Hardware and software testing requirements and constraints
- 3) Top-down / bottom-up testing, thread testing, stress testing and back-to-back testing.

Defect testing

- 1) Black-box / white-box testing
- 2) Interface testing.

Static verification

- 1) Program inspections and mathematically based verification
- 2) Static analysis tools (e.g., Cyclomatic complexity).
- 3) Cleanroom software development.

COMPUTER-AIDED SOFTWARE ENGINEERING

CASE classification - According to functionality, activities and task size supported.

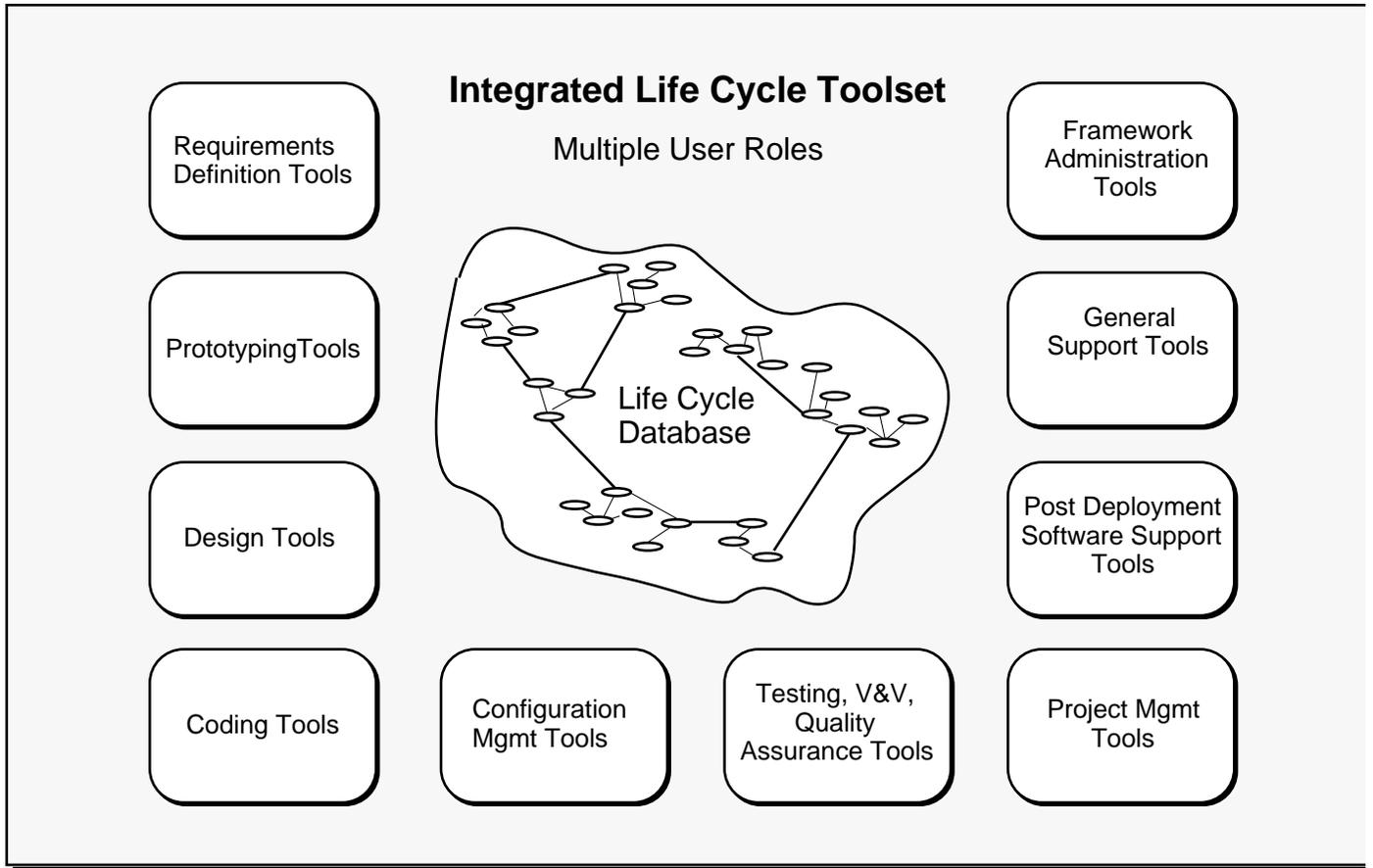
Integrated CASE - Five levels of integration (platform, data, presentation, control and process integration).

The CASE life-cycle - Six stages: procurement, tailoring, introduction, usage, evolution, and obsolescence.

CASE workbenches - An integrated set of tools intended to support a coherent software process activity (e.g., design or management). Used for programming, analysis and design, testing, etc.

Software Engineering Environments - Provides support for a wide range of software process activities. Distinguish it from other CASE systems which provide less comprehensive support. ECMA PCTE is widely accepted as a framework defining Software Engineering Environment services and interfaces.

SOFTWARE ENGINEERING ENVIRONMENT INSTANCE



MANAGEMENT

Managing people

- 1) Software management is principally concerned with people.
- 2) Small cohesive development groups and good communication is promoted.

Software cost estimation

- 1) Based on factors that affect productivity and quality (and possibly risk management).
- 2) COCOMO costing model accounts for project, product, HW and personnel attributes.

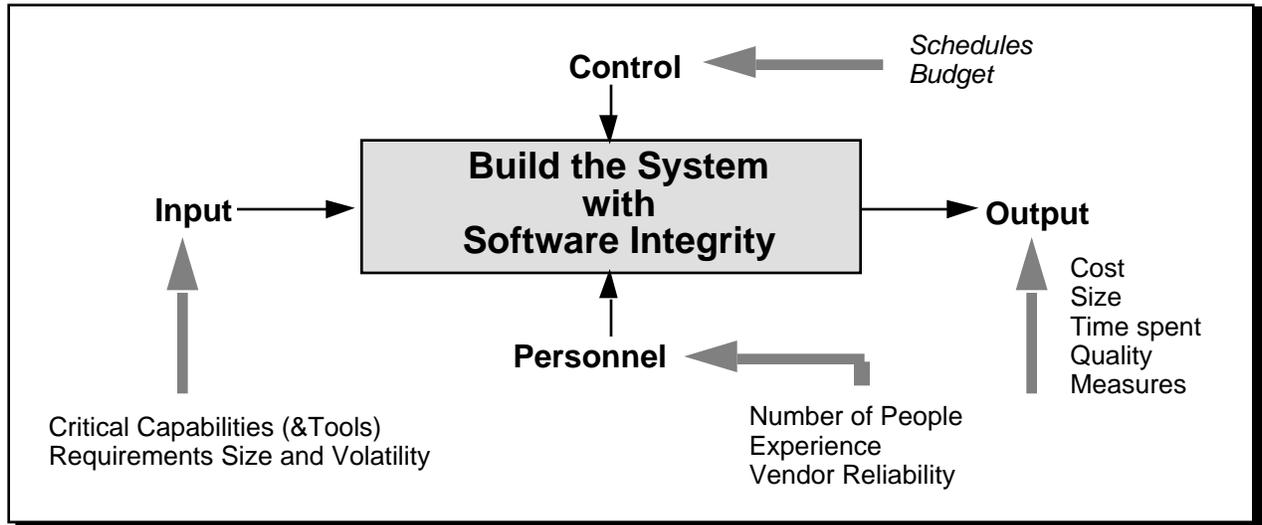
Quality management

- 1) Process quality assurance, reviews standards.
- 2) Documentation standards, software metrics and product quality metrics.

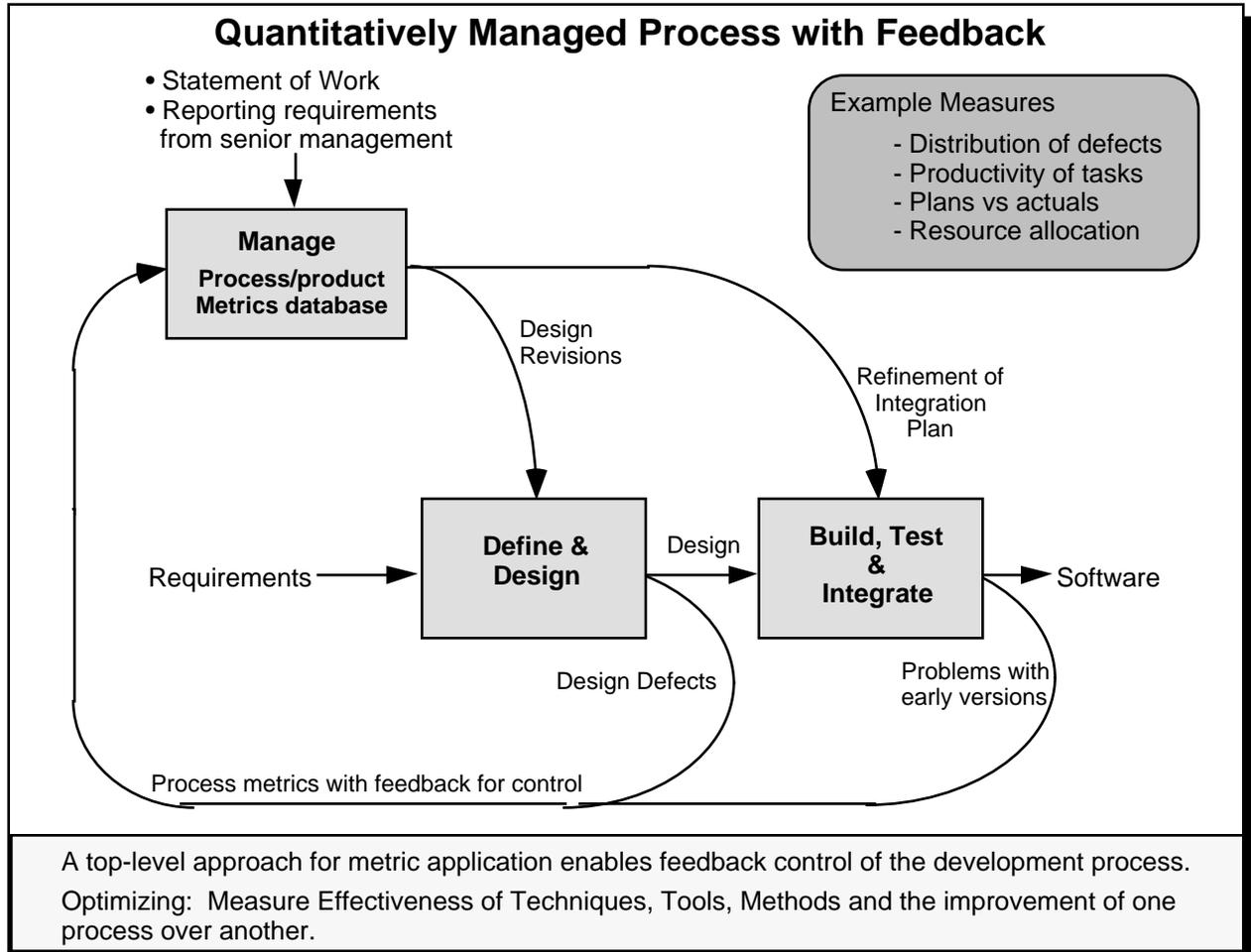
Process improvement

- 1) Process and product quality are tightly related.
- 2) Process analysis, modeling and measurement.
- 3) SEI process maturity model.
- 3) Proposed process classifications: informal, managed, methodological & improving.

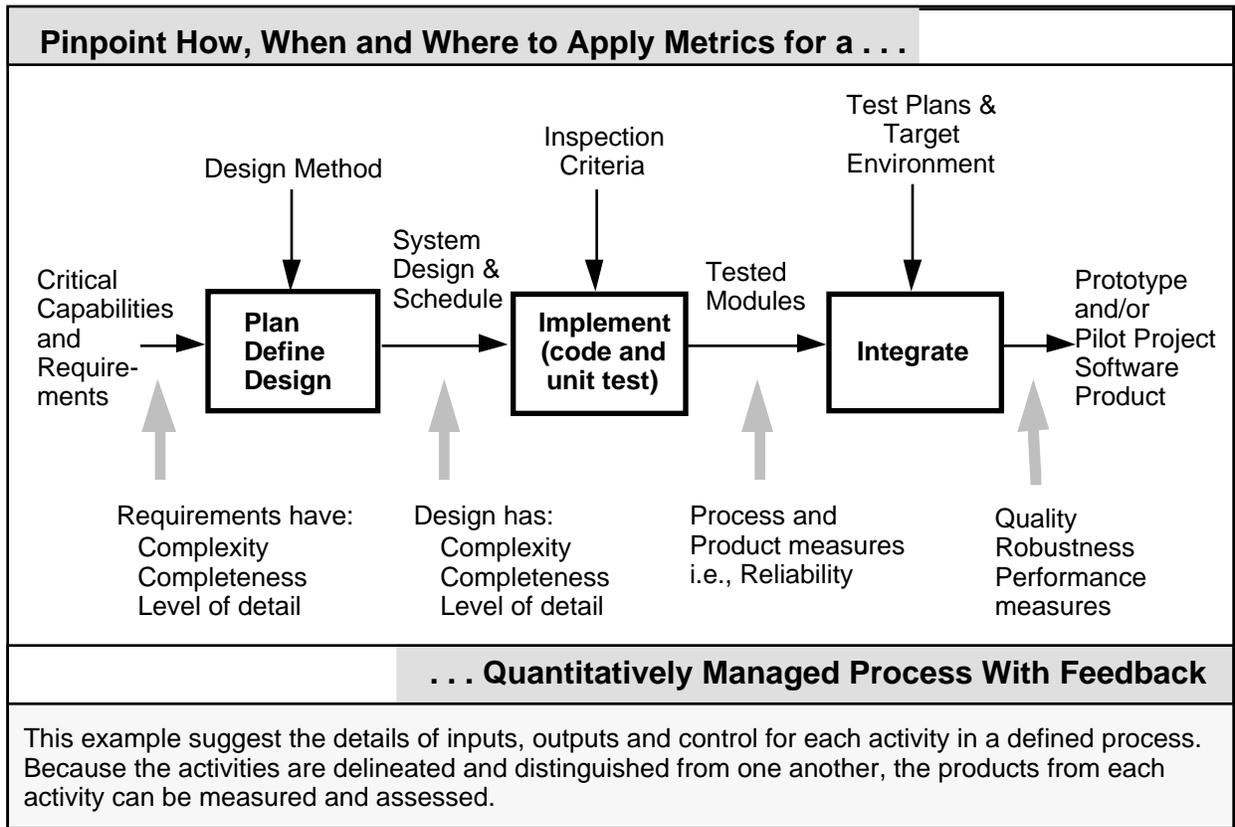
INPUTS/OUTPUTS OF PROCESS MANAGEMENT



OVERVIEW OF PROCESS MANAGEMENT METHOD



PINPOINTING AND MANAGING PROCESS AND PRODUCT METRICS



EVOLUTION

Software maintenance

- 1) Corrective, adaptive and perfective types.
- 2) Depends on a process, documentation, program evolution dynamics.
- 3) Maintenance costs are usually the greatest (e.g., 60%) cost component.
- 4) Maintainability measurement is an important issue.

Configuration management

- 1) Planning, policies and tools to support change management.
- 2) Version and release management.
- 3) System building.

Software Re-engineering

- 1) Source code translation and program restructuring.
- 2) Data re-engineering may be needed due to inconsistent data management.
- 3) Reverse engineering derives the systems design and specification from its source code.