

9. System design techniques



⌘ Design methodologies.

⌘ Requirements and specification.

Design methodologies



- ⌘ Process for creating a system.
- ⌘ Many systems are complex:
 - ☑ large specifications;
 - ☑ multiple designers;
 - ☑ interface to manufacturing.
- ⌘ Proper processes improve:
 - ☑ quality;
 - ☑ cost of design and manufacture.

Typical specifications



- ⌘ Functionality
- ⌘ Manufacturing cost
- ⌘ Performance

Design process goals



⌘ Time-to-market:

- ☑ beat competitors to market;
- ☑ meet marketing window (back-to-school).

⌘ Design cost.

⌘ Quality

- ☑ Correctness
- ☑ Reliability
- ☑ Usability

⌘ A good methodology is critical to building systems that works properly

Mars Climate Observer

⌘ Lost on Mars in September 1999.

☒ Most likely exploded as it heated up in the atmosphere

⌘ Requirements problem:

☒ Requirements did not specify units.

☒ Lockheed (contractor) Martin used English : pound-force

☒ JPL (user) wanted metric: N

☒ $1 \text{ pound-force} \equiv 0.45359237 \text{ kg} \times 9.80665 \text{ m/s}^2$
= 4.45 N
= 32.17405 lbm·ft/s²

⌘ Not caught by manual inspections.

Mars Climate Orbiter

- ⌘ NASA started Mars Surveyor Program in 1993.
- ⌘ Mars Climate Orbiter was launched on Dec. 11, 1998.
- ⌘ Mars Polar Lander was launched on Jan. 3, 1999.



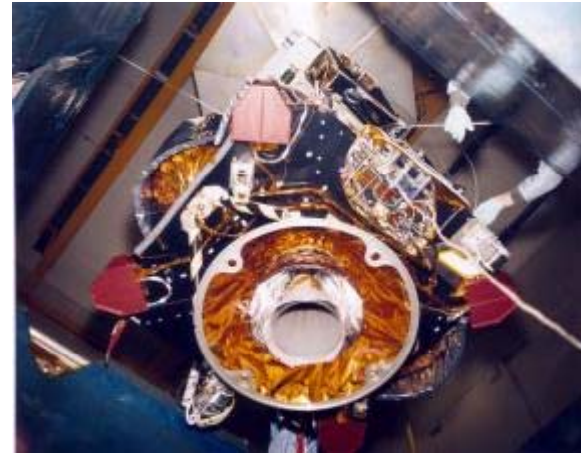
MCO Scope

- ⌘ Develop and launch two spacecrafts to Mars during the 1998 Mars transfer opportunity.
- ⌘ Development cost was estimated at \$183.9 Million.
- ⌘ Collect and return to Earth, science data resulting from the water and remote investigations of the Martian environment by the Lander.

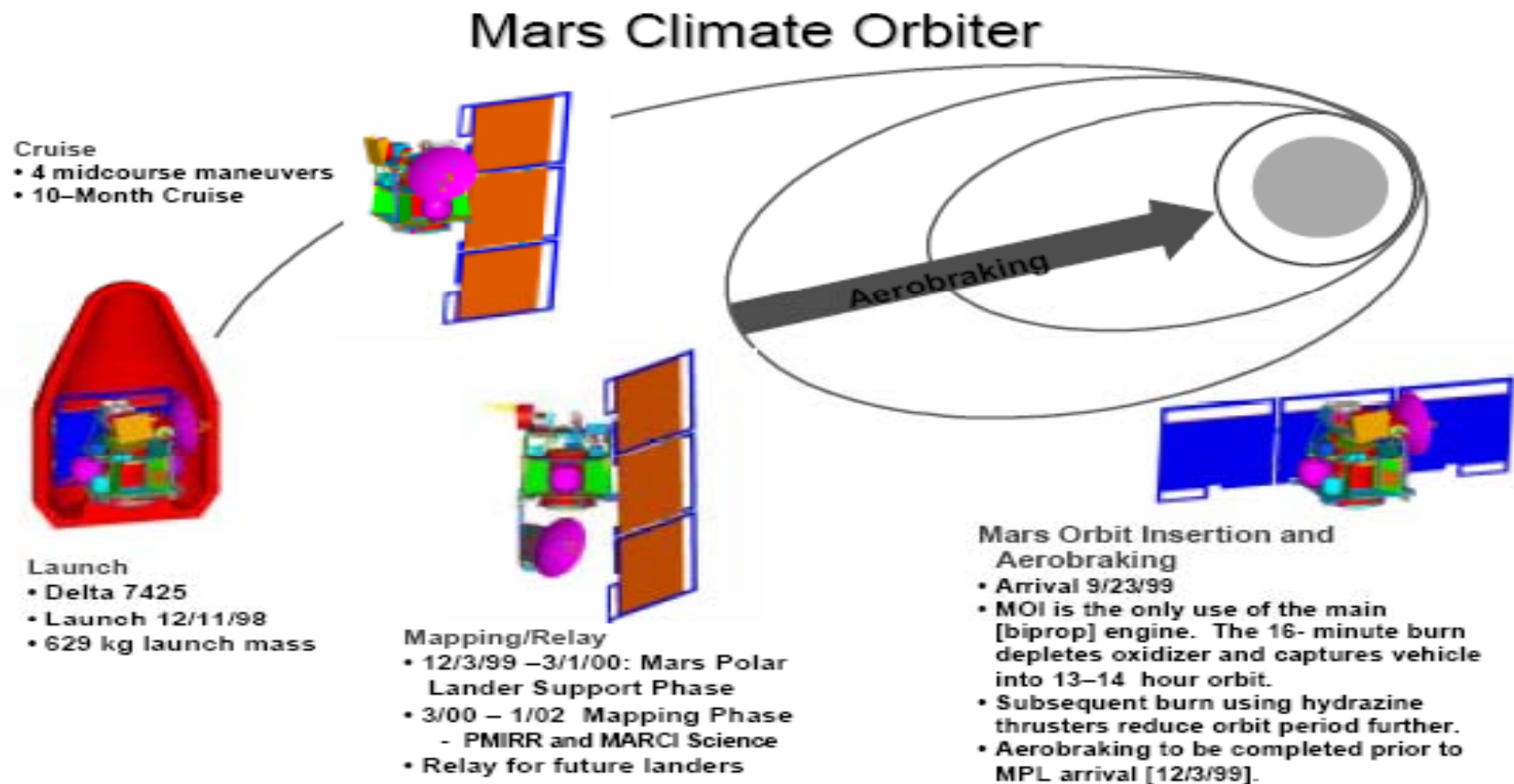


MCO Scope

- ⌘ Orbiter should act as a relay station for five years.
- ⌘ Assist in data transmission to and from the Mars Polar Lander.
- ⌘ Provide detailed information about the atmospheric temperature, dust, water vapor, and clouds on Mars.
- ⌘ Provide valuable information about the amount of carbon dioxide (CO₂) in Mars.



MCO Scope



Aerobraking



- ⌘ Aerobraking is a spaceflight technique wherein an orbiting spacecraft brushes against the top of a planetary atmosphere.
- ⌘ The friction of the atmosphere against the surface of the spacecraft slows down and lowers the craft's orbital altitude.
- ⌘ The solar panels are used to provide the maximum drag in a symmetrical position that allows some control as the spacecraft passes through the atmosphere.

MPL Scope

- ⌘ A second spacecraft Mars Polar Lander will be launched.
- ⌘ Perform daily recording of the sound and images of Mars for one Martian year (687 days).
- ⌘ The Purpose of the mission is to gather atmospheric data of each of the seasons on Mars.
- ⌘ The mission's projected end date is December 1, 2004.

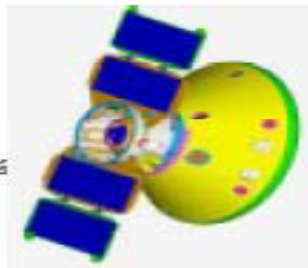


MPL Scope

Mars Polar Lander

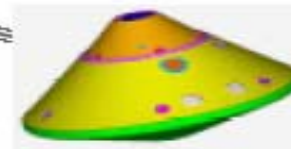
Cruise

- RCS attitude control
- Four trajectory correction maneuvers, Site Adjustment maneuver 9/1/99, Contingency maneuver up to Entry – 7 hr.
- 11 Month Cruise
- Near-simultaneous tracking w/ Mars Climate Orbiter or MGS during approach



Entry, Descent, and Landing

- Arrival 12/3/99
- Jettison Cruise Stage
- Microprobes sep. from Cruise Stage
- Hypersonic Entry (6.9 km/s)
- Parachute Descent
- Propulsive Landing
- Descent Imaging [MARDI]

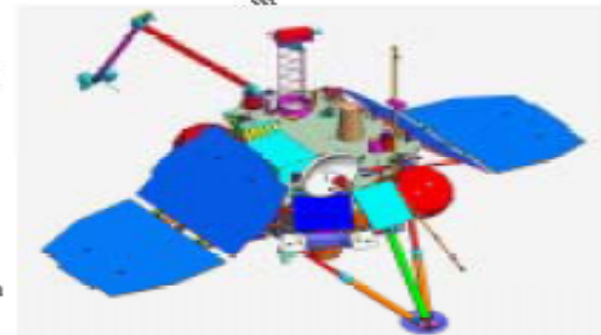


Launch

- Delta 7425
- Launch 1/3/99
- 576 kg Launch Mass

Landed Operations

- 76° S Latitude, 195° W Longitude
- Ls 256 (Southern Spring)
- 60–90 Day Landed Mission
- MVACS, LIDAR Science
- Data relay via Mars Climate Orbiter or MGS
- Commanding via Mars Climate Orbiter or direct-to-Earth high-gain antenna



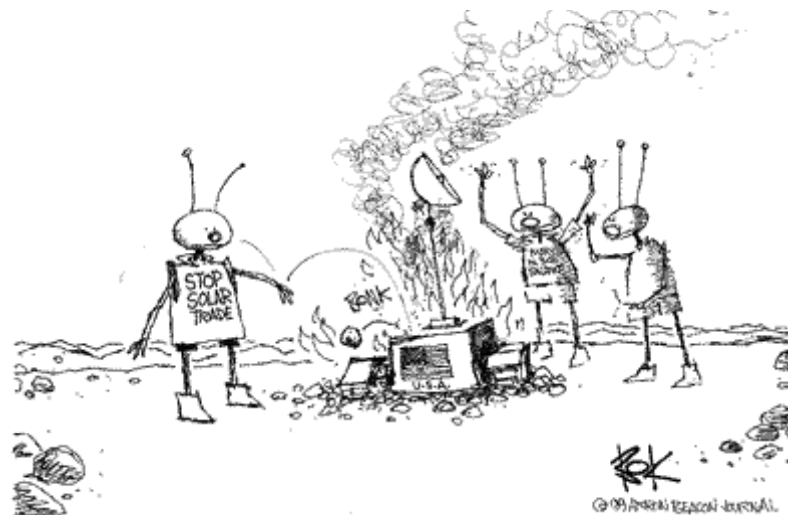
MCO Cost

- ⌘ A total of \$327.6 million was allocated for the Mars '98 Project (which included the Mars Climate Orbiter and the Mars Polar Lander)
 - ⊠ \$193.1 million for spacecraft development,
 - ⊠ \$91.7 million for launch, and
 - ⊠ \$42.8 million for mission operations
- ⌘ \$80M of the \$193.1M went toward the building of the MCO spacecraft
- ⌘ \$5M of the \$42.8M used to operations the MCO
- ⌘ \$35.5M went toward the launching of the MCO
- ⌘ According to a later assessment, '98 Mars Project was at least 30% under-funded.

Project Management

So What Went Wrong?

WHAT REALLY HAPPENED



Project Management



- ⌘ The official report cited the following “contributing factors” to the loss of the spacecraft
 - ⊠ undetected errors in ground-based models of the spacecraft the
 - ⊠ the operational navigational team was not fully informed on the details of the way that Mars Climate Orbiter was pointed in space
 - ⊠ a final, optional engine firing to raise the spacecraft’s path relative to Mars before its arrival was considered but not performed

Project Management



⌘ Summary

☒ One Technical Problem

- failed conversion of unit

☒ Many Process and Social Problems

- No review (e.g. verification), insufficient training, informal processes in place, formal processes ignored

☒ Led to a destroyed spacecraft

Design flow




- ⌘ **Design flow**: sequence of steps in a design methodology.
- ⌘ May be partially or fully automated.
 - ☒ Use tools to transform, verify design.
- ⌘ **Design flow is just one component of methodology.**
- ⌘ Methodology also includes management organization, etc.

Design methodology



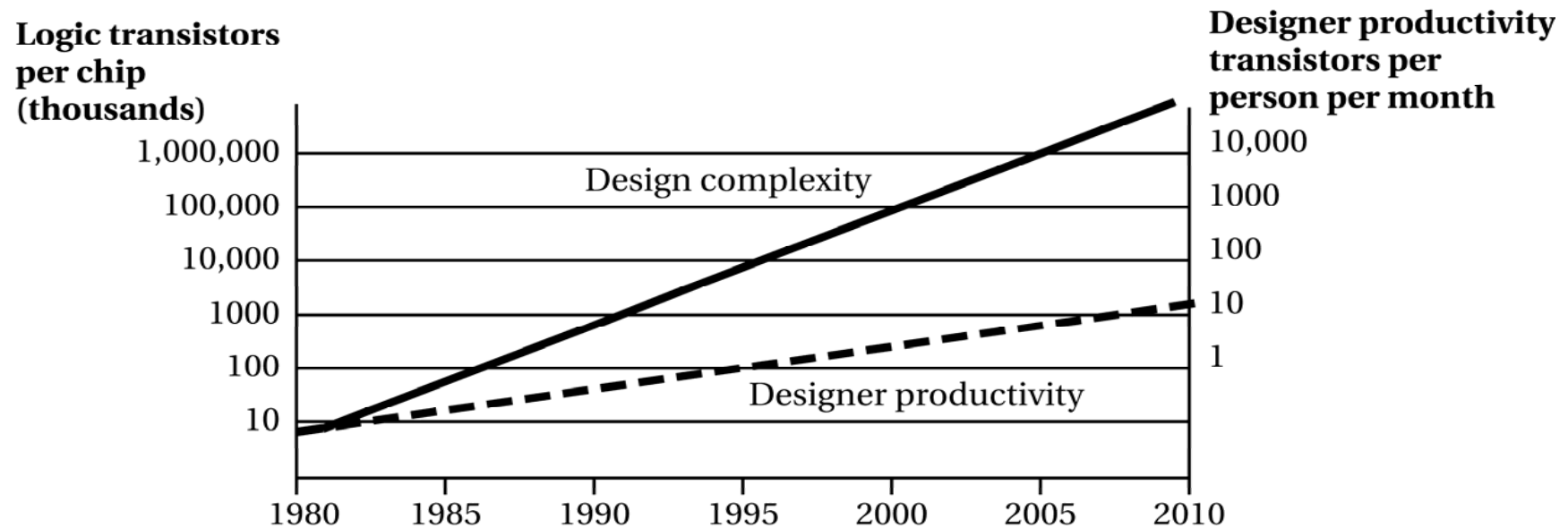
- ⌘ Design methodology: a procedure for creating an implementation from a set of requirements.
- ⌘ Methodology is important in embedded computing:
 - ☑ Must design many different systems.
 - ☑ We may use same/similar components in many different designs.
 - ☑ Design time, results must be predictable.

Embedded system design challenges



- ⌘ Design space is large and irregular.
- ⌘ We don't have synthesis tools for many steps.
- ⌘ Can't simulate everything.
- ⌘ May need to build special-purpose simulators quickly.
- ⌘ Often need to start software development before hardware is finished.

Design complexity vs. designer productivity

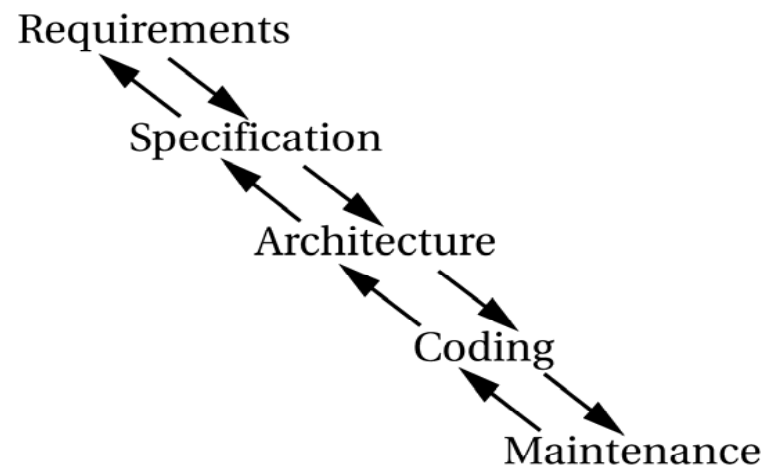


Basic design methodologies

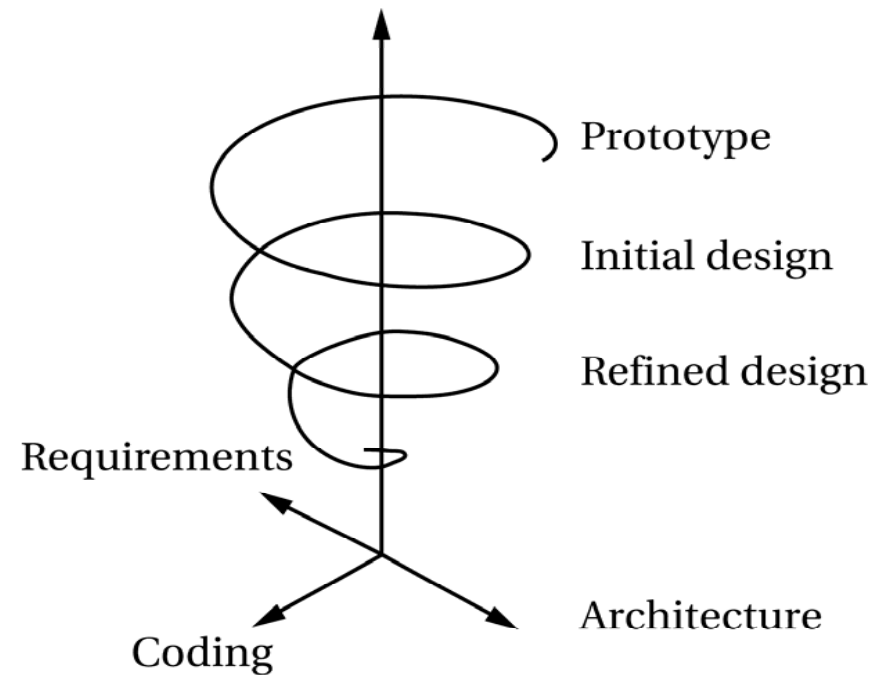


- ⌘ Figure out flow of decision-making.
- ⌘ Determine when bottom-up information is generated.
- ⌘ Determine when top-down decisions are made.

Waterfall and spiral models



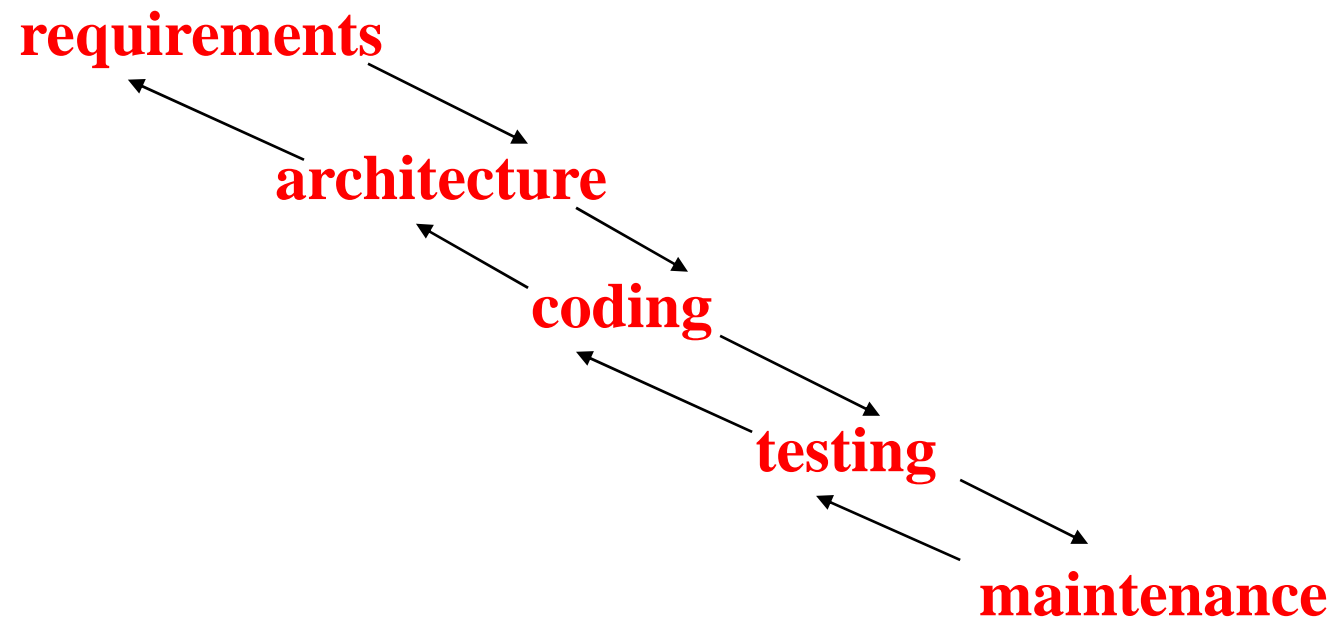
Waterfall



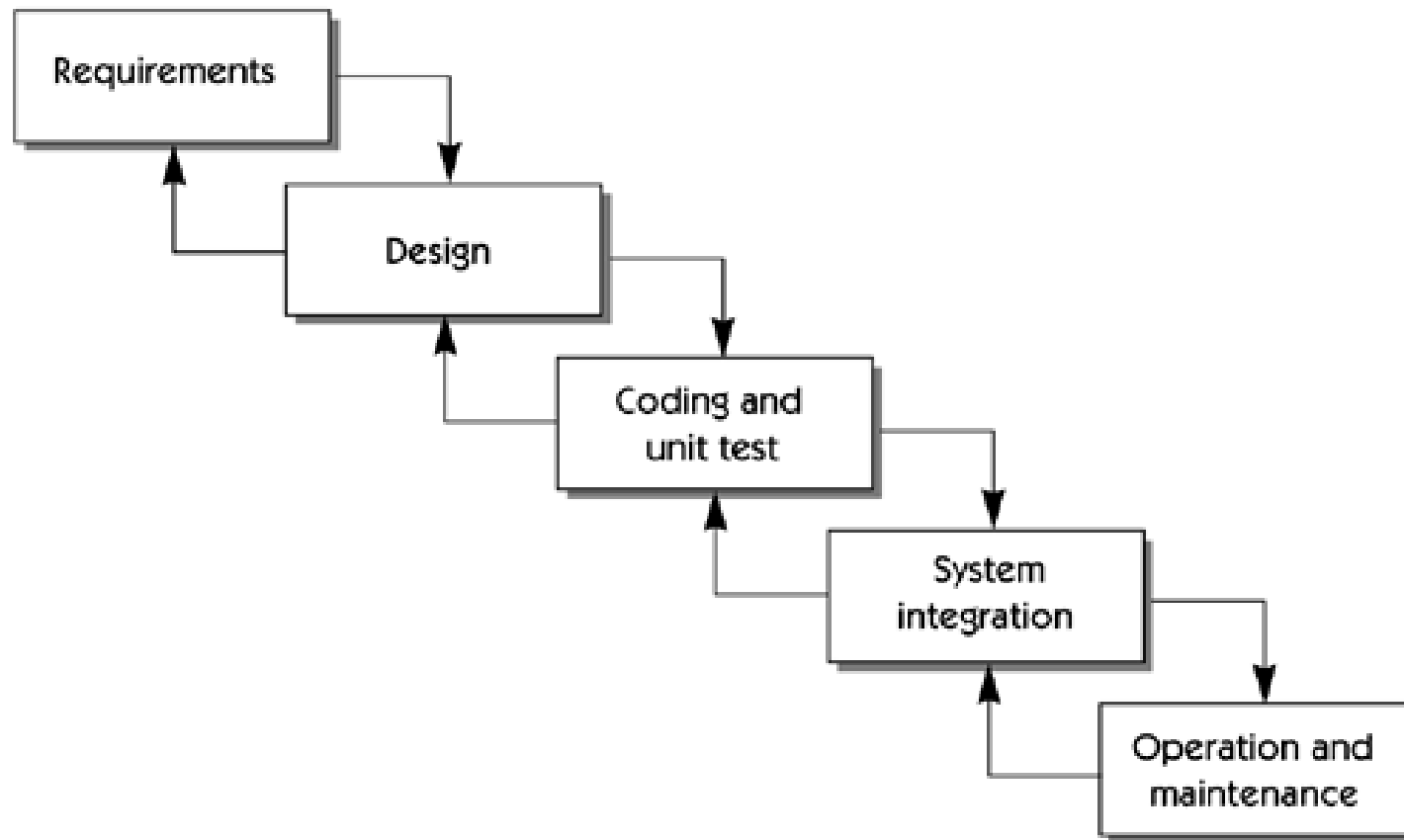
Spiral

Waterfall model

⌘ Early model for software development:



Waterfall model



Waterfall model steps



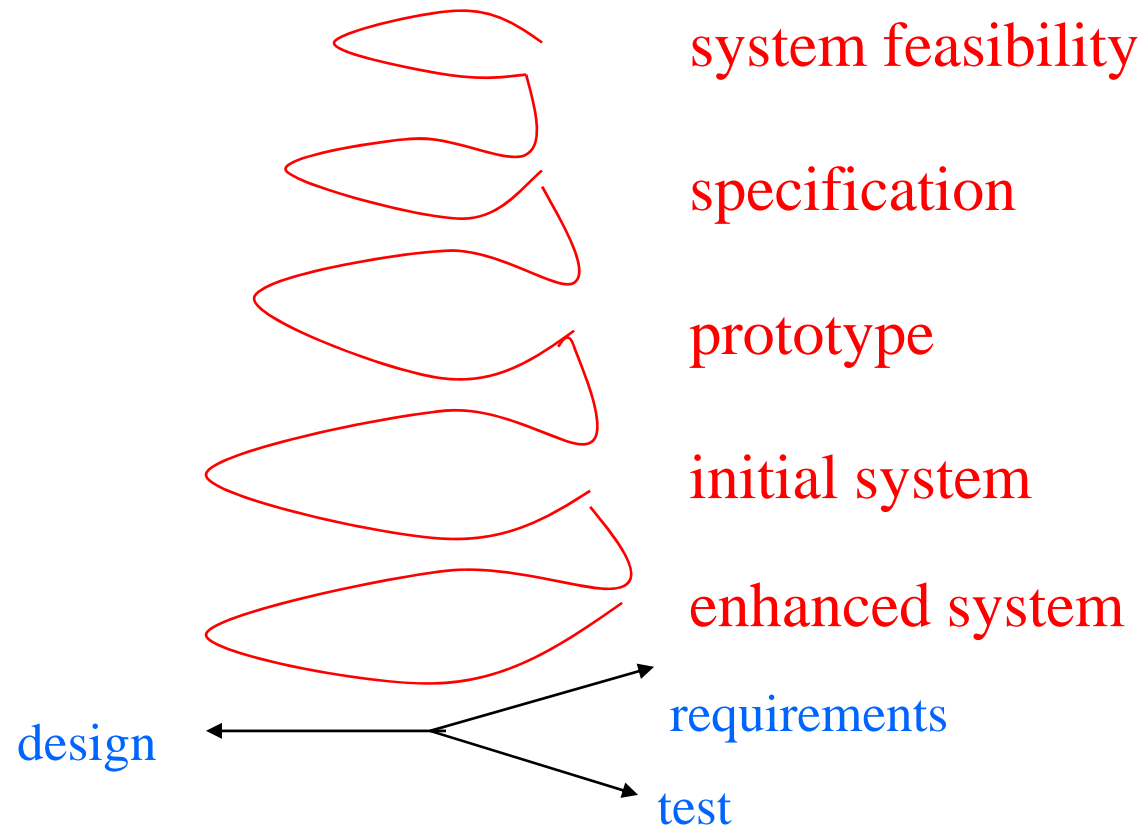
- ⌘ Requirements: determine basic characteristics.
- ⌘ Architecture: decompose into basic modules.
- ⌘ Coding: implement and integrate.
- ⌘ Testing: exercise and uncover bugs.
- ⌘ Maintenance: deploy, fix bugs, upgrade.

Waterfall model critique



- ⌘ Unrealistic
- ⌘ Only local feedback---may need iterations between coding and requirements, for example.
- ⌘ Doesn't integrate top-down and bottom-up design.
- ⌘ Just for software development
 - ☑ Assumes hardware is given.

Spiral model

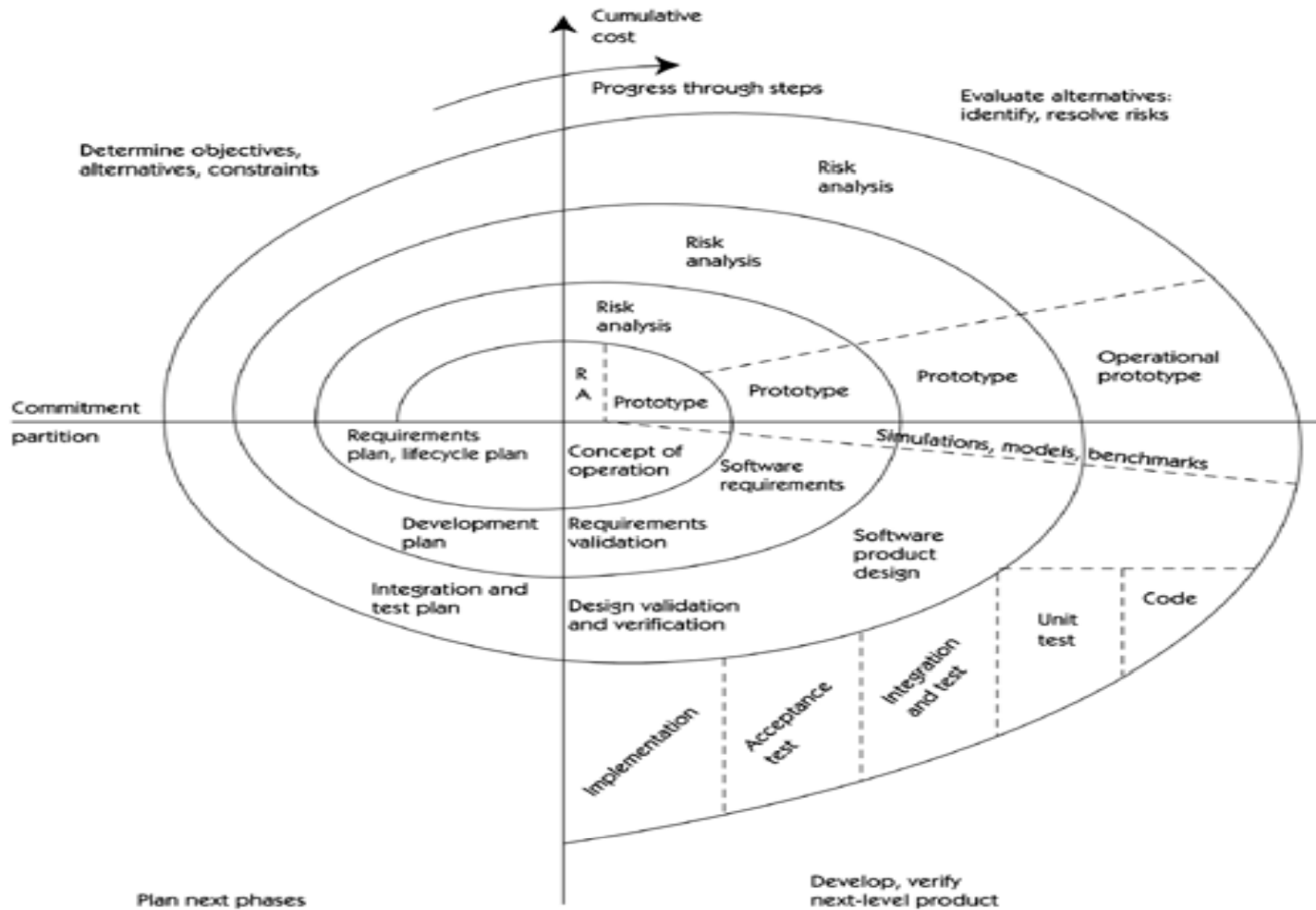


Spiral model



- ⌘ Assumes that several versions of the system will be built.
- ⌘ **Successive refinement** of system.
 - ☑ Start with mock-ups, move through simple systems to full-scale systems.
- ⌘ Provides bottom-up feedback from previous stages.
- ⌘ Working through stages may take too much time.

Spiral model

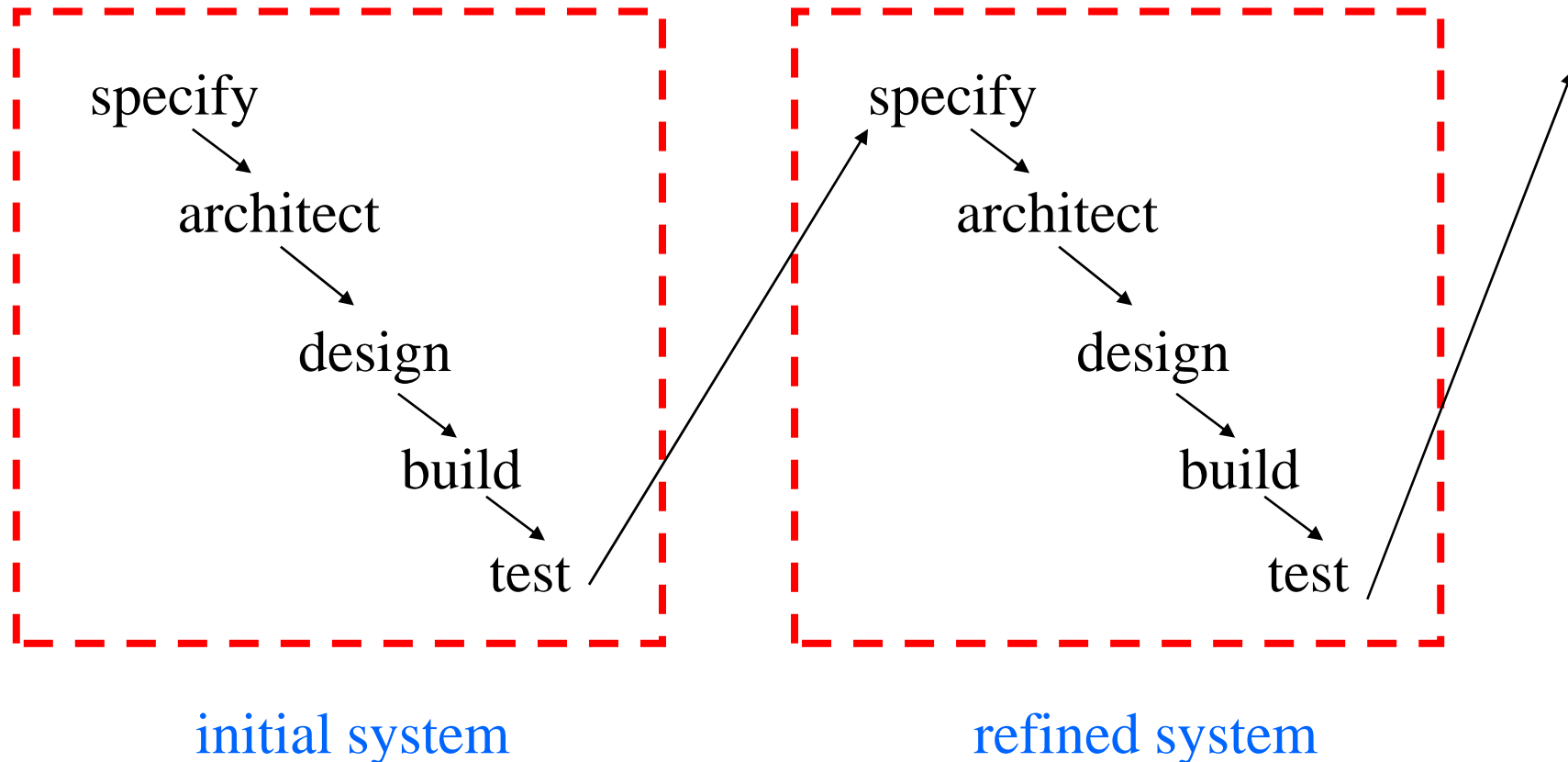


Spiral Model Sectors



- ⌘ Objective setting
 - ☑ Specific objectives for the phase are identified.
- ⌘ Risk assessment and reduction
 - ☑ Risks are assessed and activities put in place to reduce the key risks.
- ⌘ Development and validation
 - ☑ A development model for the system is chosen which can be any of the generic models.
- ⌘ Planning
 - ☑ The project is reviewed and the next phase of the spiral is planned.

Successive refinement model



Successive refinement model



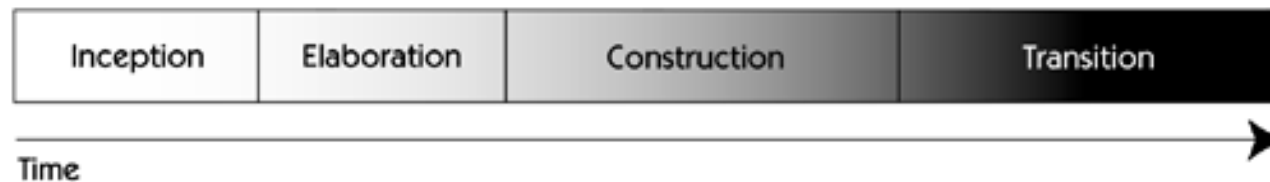
- ⌘ More make sense when you are unfamiliar to the application domain.
- ⌘ **Successive refinement** of system.
 - ☑ Start with mock-ups, move through simple systems to full-scale systems.
- ⌘ Provides bottom-up feedback from previous stages.
- ⌘ Working through stages may take too much time.

Iterative Approach



⌘ In the iterative approach, the lifecycle phases are decoupled from the logical software activities that occur in each phase, allowing us to revisit various activities, such as requirements, design, and implementation, during various iterations of the project.

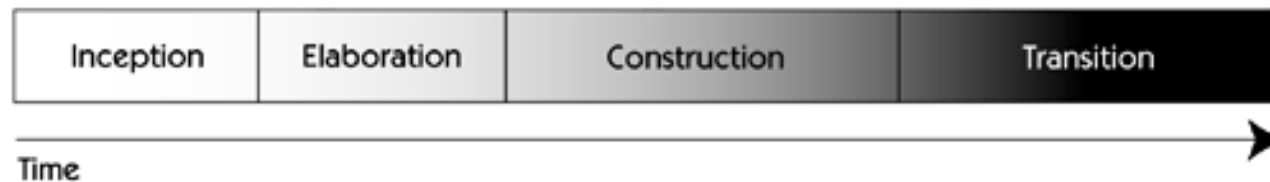
Iterative Approach: Lifecycle Phases



1. Inception phase

- ☒ The team is focused on understanding the business case for the project, the scope of the project, and the feasibility of an implementation.
- ☒ Problem analysis is performed, the vision for the solution is created, and preliminary estimates of schedule and budget, as well as project risk factors, are defined

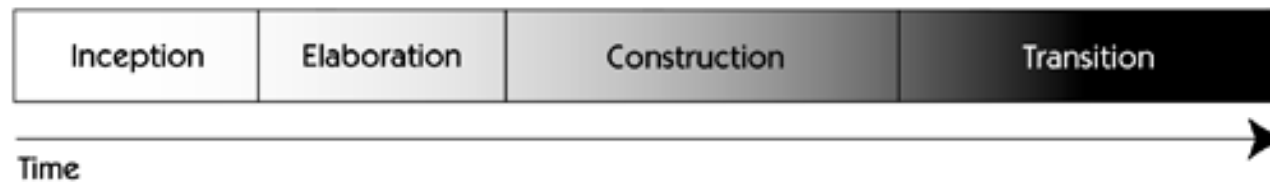
Iterative Approach: Lifecycle Phases



2. Elaboration phase

- ☒ The requirements for the system are refined, an initial, perhaps even executable, architecture is established, and an early feasibility prototype is typically developed and demonstrated.

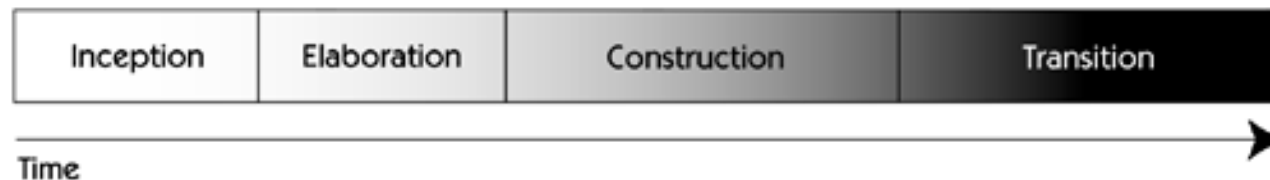
Iterative Approach: Lifecycle Phases



3. Construction phase

- ☒ The focus is on implementation.
- ☒ Most of the coding is done in this phase, and the architecture and design are fully developed.

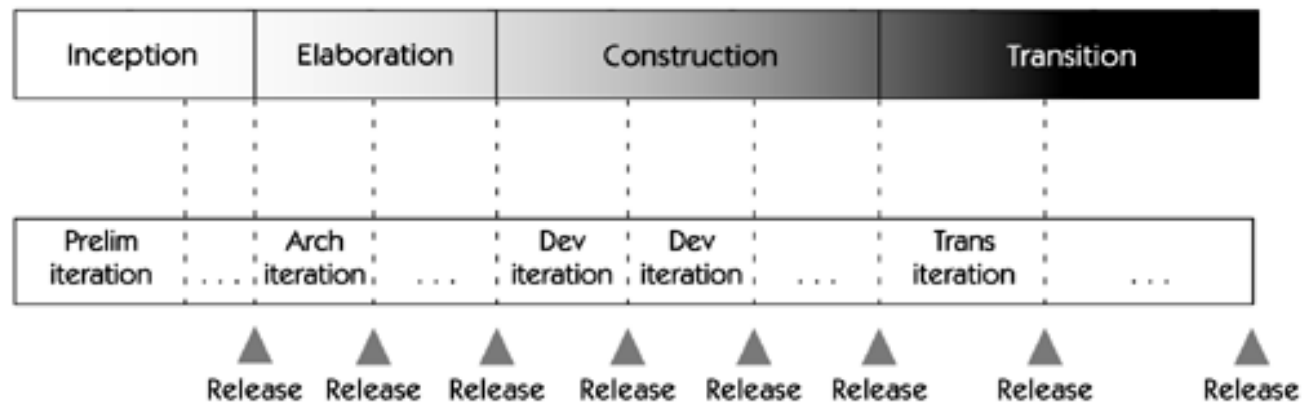
Iterative Approach: Lifecycle Phases



4. Transition phase

- ☒ Beta testing
- ☒ The users and maintainers of the system are trained on the application.
- ☒ The tested baseline of the application is transitioned to the user community and deployed for use.

Iterative Approach: Iterations

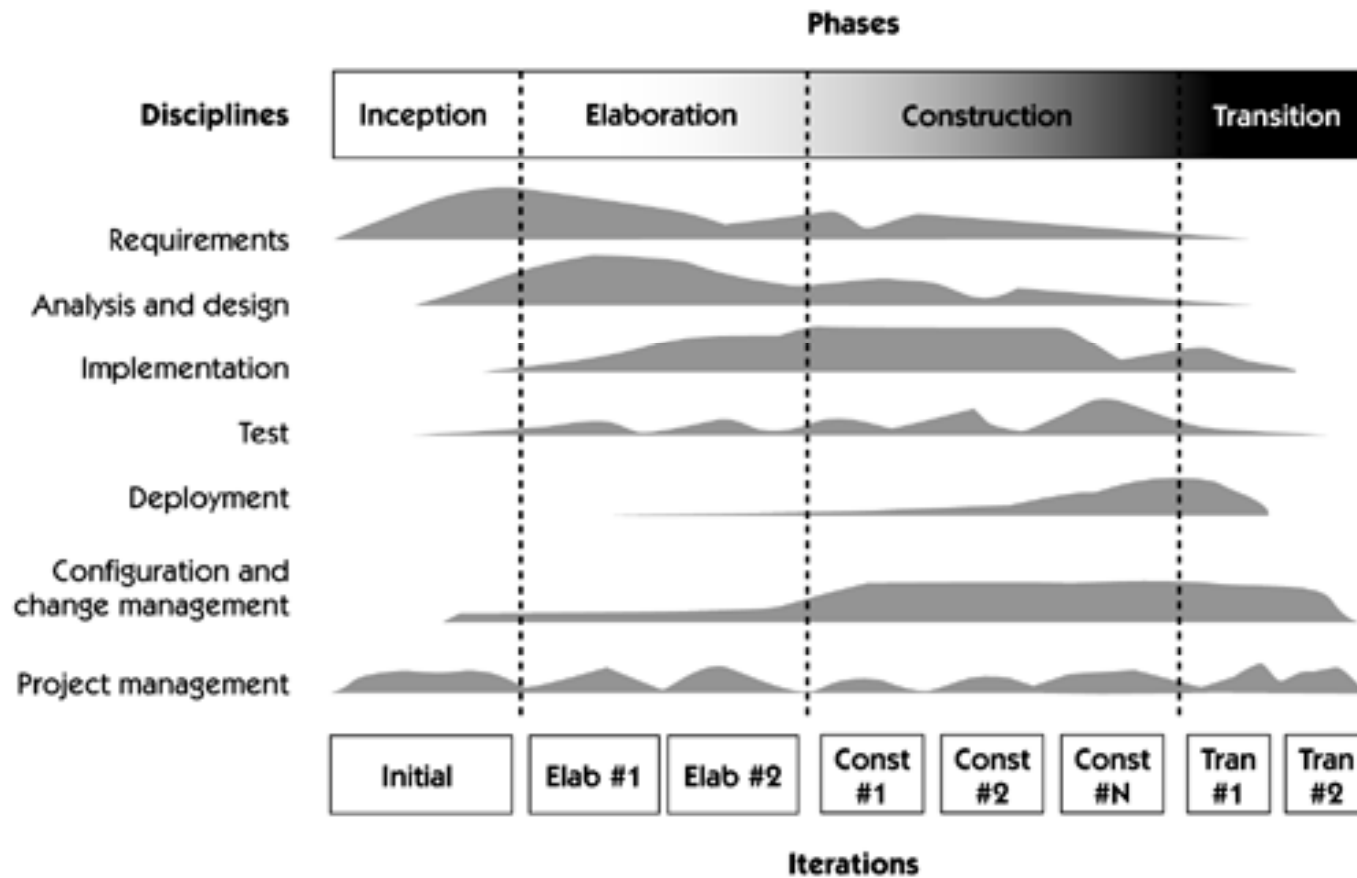


Iterative Approach: Iterations



- ⌘ Within each phase, the project typically undergoes multiple iterations.
- ⌘ An *iteration* is a sequence of activities with an established plan and evaluation criteria, resulting in an executable of some type.
- ⌘ Each iteration builds on the functionality of the prior iteration; thus, the project is developed in an "iterative and incremental" fashion.

Iterative Approach: Disciplines



Iterative Approach: Disciplines



- ⌘ In the iterative approach, the activities associated with the development of the software are organized into a set of disciplines.
- ⌘ Each discipline consists of a logically related set of activities, and each defines how the activities must be sequenced to produce a viable work product (or artifact).
- ⌘ Although the number and kind of disciplines can vary, based on the company or project circumstances, there are typically at least six disciplines.

Iterative Approach: Disciplines



- ⌘ During each iteration, the team spends as much time as appropriate in each discipline.
 - ☒ Thus, an iteration can be regarded as a mini-waterfall through the activities of requirements, analysis and design, and so on, but each mini-waterfall is "tuned" to the specific needs of that iteration.

- ⌘ The size of the "hump" indicates the relative amount of effort invested in a discipline.
 - ☒ For example, in the elaboration phase, significant time is spent on "refining" the requirements and in defining the architecture that will support the functionality.

- ⌘ The activities can be sequential (a true mini-waterfall) or may execute concurrently, as is appropriate to the project.

Requirements in the Iterative Model



⌘ From the requirements management perspective, the iterative approach provides two major advantages:

1. Better adaptability to requirements change.
2. Better scope management.

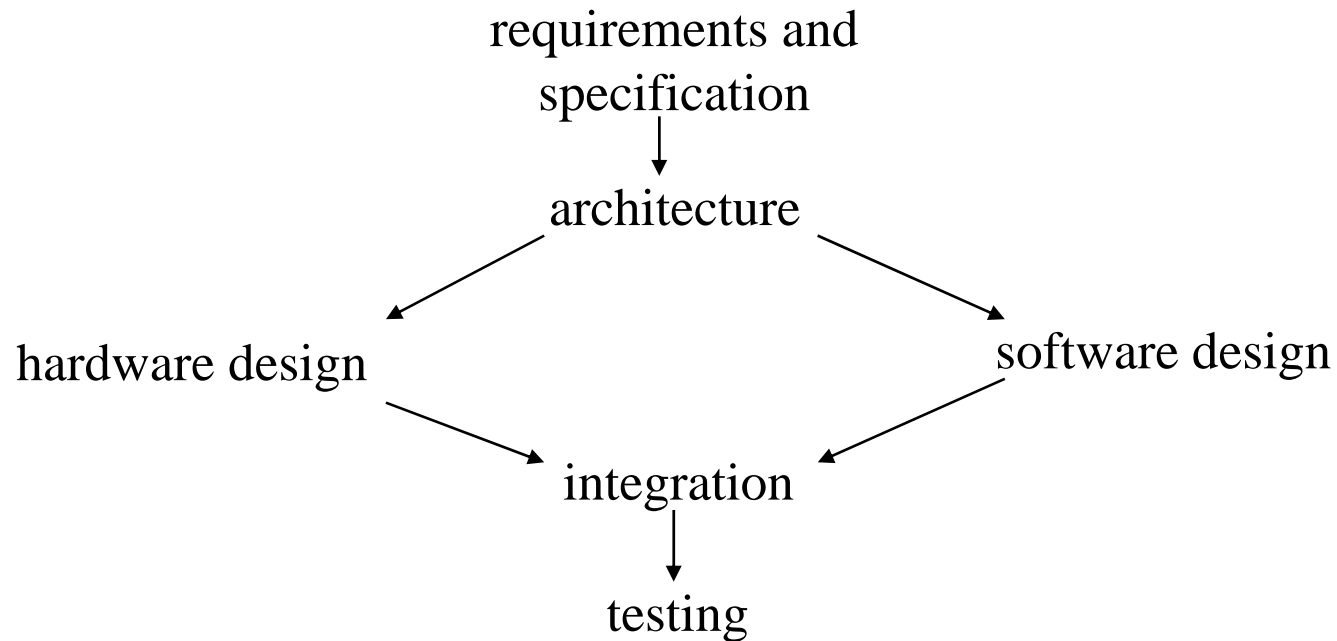
Modeling in hardware



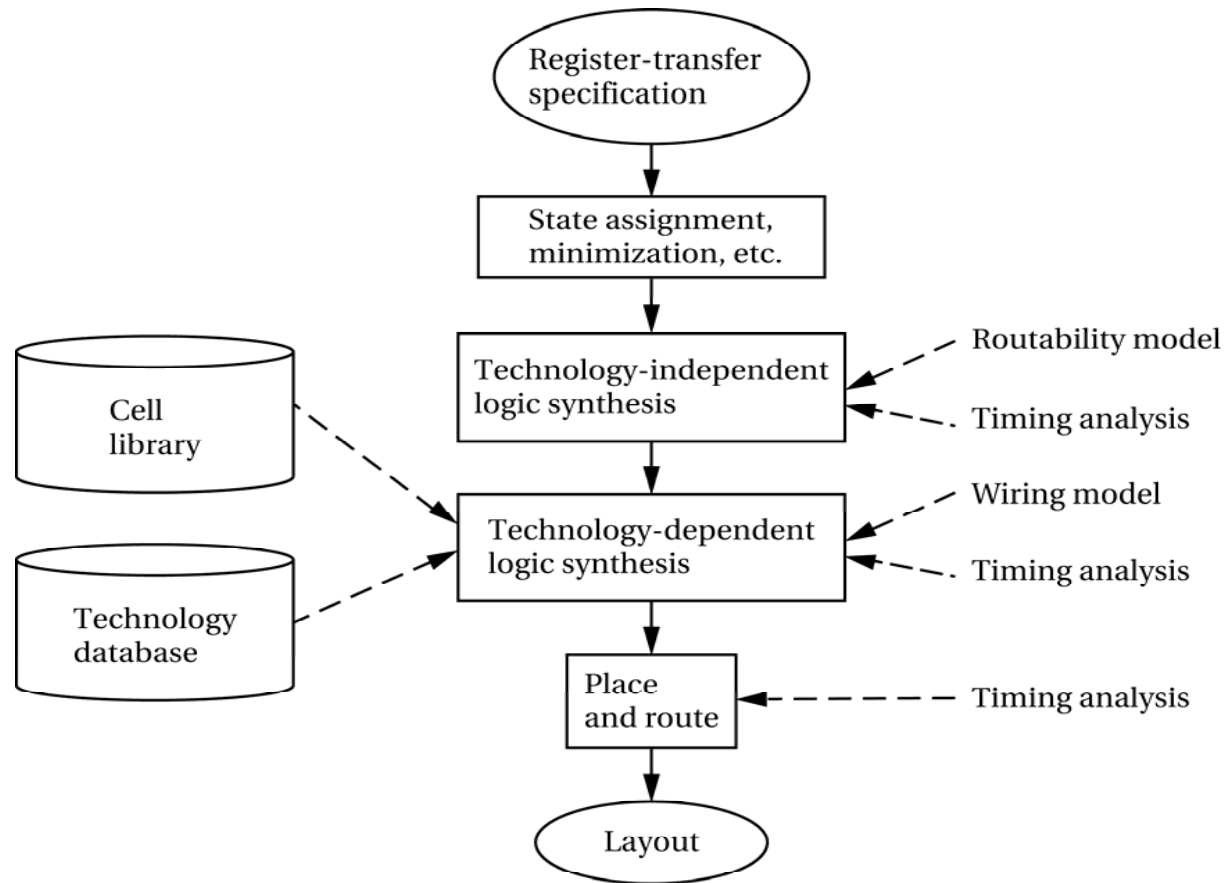
- ⌘ Technology databases capture manufacturing process information.
- ⌘ Cell libraries describe the cells used to compose designs.
- ⌘ Logic synthesis systems use routability and timing models.

Embedded system

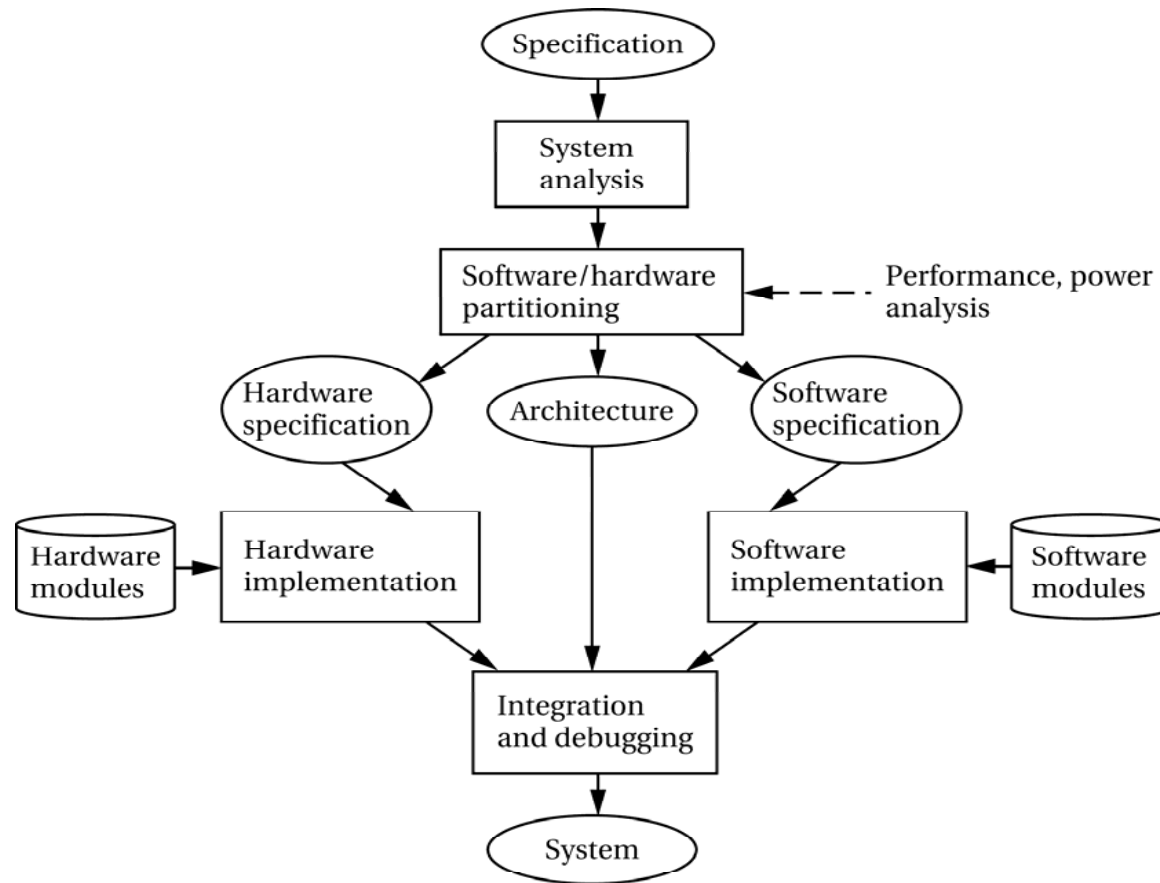
⌘ Often involve the design of hardware as well as software



Hardware design flow



Hardware/software co-design flow



Co-design methodology



- ⌘ Must architect hardware and software together:
 - ☑ provide sufficient resources;
 - ☑ avoid software bottlenecks.
- ⌘ Can build pieces somewhat independently, but integration is major step.
- ⌘ Also requires bottom-up feedback.

Hierarchical design flow



⌘ Embedded systems must be designed across multiple levels of abstraction:

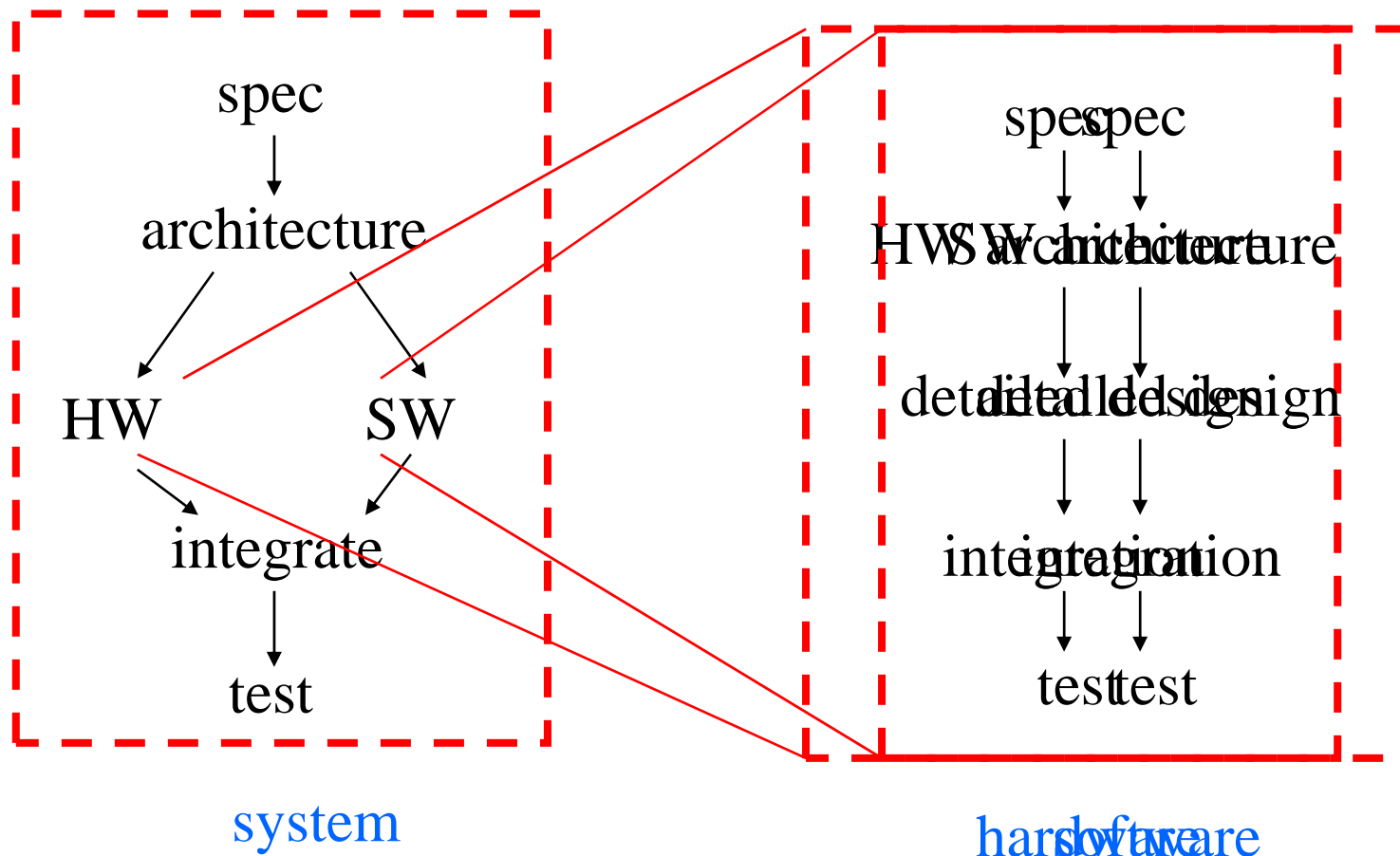
- ☑ system architecture;

- ☑ hardware and software systems;

- ☑ hardware and software components.

⌘ Often need design flows within design flows.

Hierarchical HW/SW flow



Concurrent engineering



- ⌘ Large projects use many people from multiple disciplines.
- ⌘ Work on several tasks at once to reduce design time.
- ⌘ Feedback between tasks helps improve quality, reduce number of later design problems.
- ⌘ Tries to eliminate “over-the-wall” design steps with little interaction between the two.

Concurrent engineering techniques



- ⌘ **Cross-functional teams**: include members from various disciplines
- ⌘ **Concurrent product realization**: doing several things to reduce design time.
- ⌘ **Incremental information sharing**: as soon as new information is available, it is shared and integrated into the design. Cross-functional teams are important to the effective sharing of information in a timely fashion.

Concurrent engineering techniques



- ⌘ **Integrated product management**: ensures that someone is responsible for the entire project, and that responsibility is not abdicated once one aspect of the work is done.
- ⌘ Early and continual **supplier involvement**
- ⌘ Early and continual **customer focus**

AT&T PBX concurrent engineering



- ⌘ AT&T applied concurrent engineering to the designs of PBXs
- ⌘ They re-engineered their process to reduce design time and make other improvement to the end product
- ⌘ They used seven step process described below:
 1. Benchmark against competitors.
 2. Identify breakthrough improvements.
 3. Characterize current process.
 4. Create new process.
 5. Verify new process.
 6. Implement.
 7. Measure and improve.

AT&T PBX concurrent engineering

1. Benchmark against competitors.

- ☒ They found that it took them 30% longer to introduce a new product than their best competitors.
- ☒ They decided to shoot for a 40% reduction in design time

2. Identify breakthrough improvements.

- ☒ Identify the factors that would influence their effort
- ☒ Three major factors were identified:
 - ☒ increased partnership between design and manufacturing,
 - ☒ continued existence of the basic organization of design labs and manufacturing, and
 - ☒ support of managers at least two levels above the working levels

AT&T PBX concurrent engineering



- ☒ As a result three groups were established to help manage effort
 - ☒ A steering committee formed by mid-level managers
 - ☒ A project office formed by a manager and an operation analyst
 - ☒ A core team of engineers and analysts formed to make things happens

3. Characterize the current process.

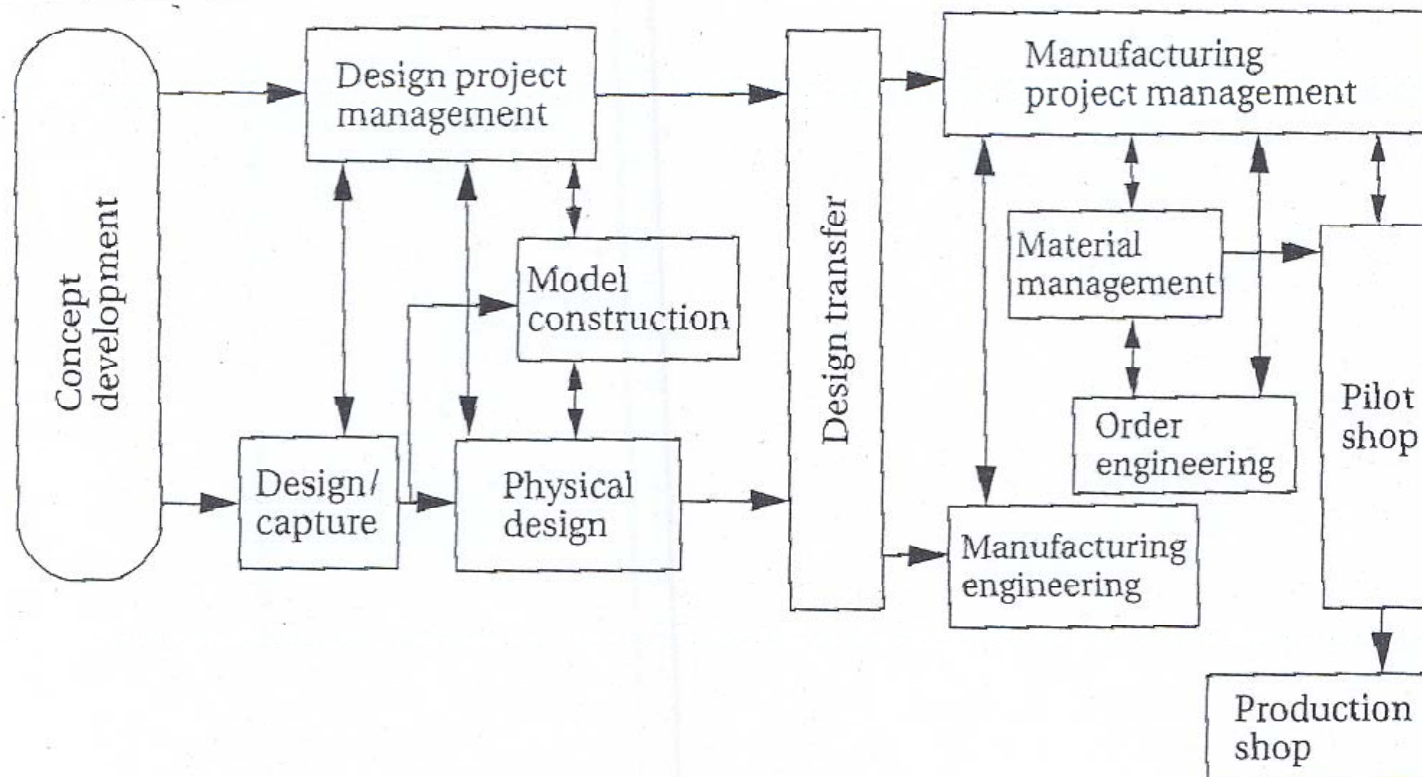
- ☒ They built flowcharts and identified several root causes of delays
 - ☒ Design and manufacturing tasks were performed sequentially

AT&T PBX concurrent engineering

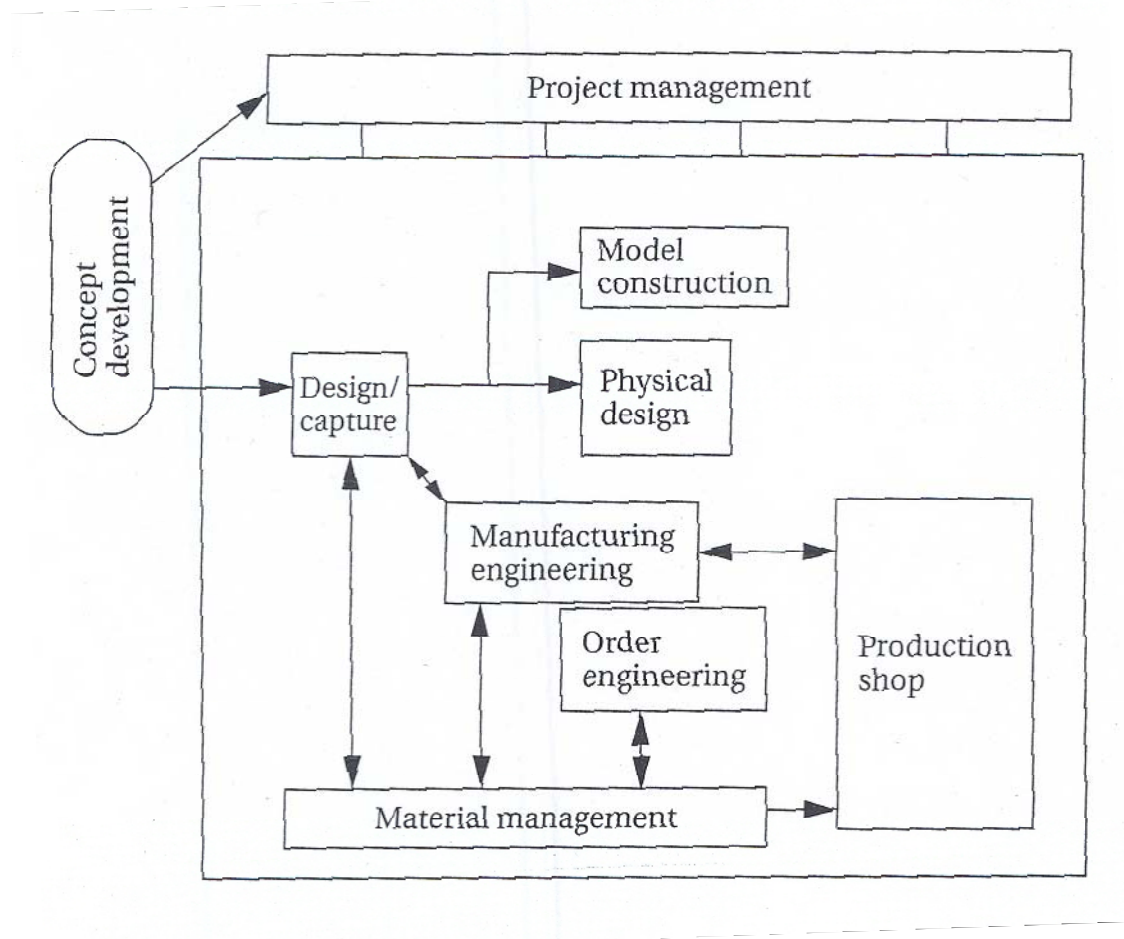


- ☒ Groups tended to focus on intermediate milestones related to their narrow job descriptions, rather than trying to take into account the effects of their decisions on other aspects of the development
- ☒ Too much time was spent waiting in queues – jobs were handed off from one person to another very frequently. In many cases, the recipient didn't know how to best prioritize the incoming tasks. It's a typical managerial problem
- ☒ Too many groups have their own design databases, creating redundant data that had to be maintained and synchronized.

Current process



New process

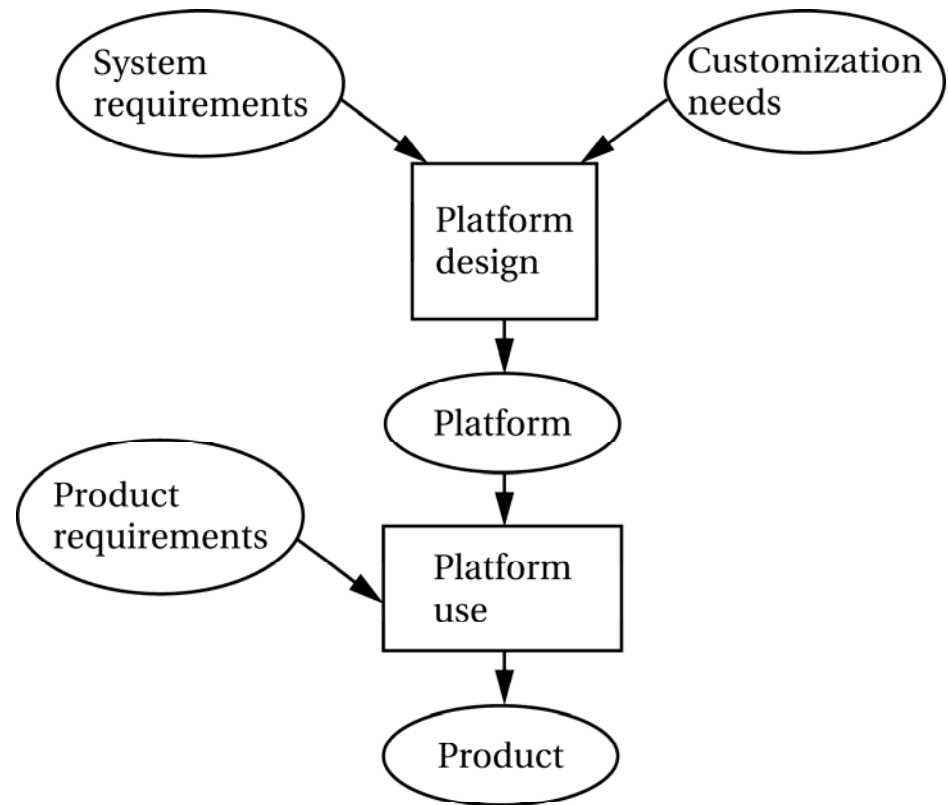


AT&T PBX concurrent engineering

4. **Creating the target process**: Based on its studies, the core team created a new development process.
5. **Verify the new process**: the team undertook a pilot product development project to test a new process
 - ☒ Mechanical enclosures and PCB boards
6. **Implement across the product line**
 - ☒ This activity required training of personnel, documentation of the new standards and procedures, and improvement to information systems
7. **Measure results and improve**: the team found that product development time had been reduced from 18-30 months to 11 months.

Platform-based design

- ⌘ Platform includes hardware, supporting software.
- ⌘ Two stage process:
 - ☑ Design the platform.
 - ☑ Use the platform.
- ⌘ Platform can be reused to host many different systems.



Platform design



- ⌘ Turn system requirements and software models into detailed requirements.
 - ☑ Use profiling and analysis tools to measure existing executable specifications.
- ⌘ Explore the design space manually or automatically.
- ⌘ Optimize the system architecture based on the results of simulation and other steps.
- ⌘ Develop hardware abstraction layers and other software.

Programming platforms



- ⌘ Programming environment must be customized to the platform:
 - ☑ Multiple CPUs.
 - ☑ Specialized memory.
 - ☑ Specialized I/O devices.
- ⌘ Libraries are often used to glue together processors on platforms.
- ⌘ Debugging environments are a particular challenge.

Standards-based design methodologies



- ⌘ Standards enable large markets.
- ⌘ Standards generally allow products to be differentiated.
 - ☑ Different implementations of operations, so long as I/O behavior is maintained.
 - ☑ User interface is often not standardized.
- ⌘ Standard may dictate certain non-functional requirements (power consumption), implementation techniques.

Reference implementations




- ⌘ Executable program that complies with the I/O behavior of the standard.
 - ☑ May be written in a variety of language.
- ⌘ In some cases, the reference implementation is the most complete description of the standard.
- ⌘ Reference implementation is often not well-suited to embedded system implementation:
 - ☑ Single process.
 - ☑ Infinite memory.
 - ☑ Non-real-time behavior.

Designing standards-based systems



- ⌘ Design and implement system components that are not part of the standard.
- ⌘ Perform platform-independent optimizations.
- ⌘ Analyze optimized version of reference implementation.
- ⌘ Design hardware platform.
- ⌘ Optimize system software based on platform.
- ⌘ Further optimize platform.
- ⌘ Test for conformity to standard.

H.264/AVC



- ⌘ Implements video coding for a wide range of applications:
 - ☑ Broadcast and videoconferencing.
 - ☑ Cell phone-sized screens to HDTV.
- ⌘ Video codec reference implementation contains 120,000 lines of C code.

Design verification and validation



- ⌘ Testing exercises an implementation by supplying inputs and testing outputs.
- ⌘ Validation compares the implementation to a specification or requirements.
- ⌘ Verification may be performed at any design stage; compares design at one level of abstraction to another.

Design verification techniques



- ⌘ Simulation uses an executable model, relies on inputs.
- ⌘ Formal methods generate a (possibly specialized) proof.
- ⌘ Manual methods, such as design reviews, catch design errors informally.

A methodology of methodologies



- ⌘ Embedded systems include both hardware and software.
 - ☑ HW, SW have their own design methodologies.
- ⌘ Embedded system methodologies control the overall process, HW/SW integration, etc.
 - ☑ Must take into account the good and bad points of hardware and software design methodologies used.

Joint algorithm and architecture development



- ⌘ Some algorithm design is necessarily performed before platform design.
- ⌘ Algorithm development can be informed by platform architecture design.
 - ☒ Performance/power/cost trade-offs.
 - ☒ Design trends over several generations.

Requirements analysis



- ⌘ **Requirements**: informal description of what customer wants.
- ⌘ **Specification**: precise description of what design team should deliver.
- ⌘ Requirements phase links customers with designers.
 - ☑ marketing

Types of requirements



⌘ **Functional**: input/output relationships.

⌘ **Non-functional**:

- ☑ timing;
- ☑ power consumption;
- ☑ manufacturing cost;
- ☑ physical size;
- ☑ time-to-market;
- ☑ reliability.

Good requirements



- ⌘ A good set of requirements should meet seven tests: correctness, unambiguousness, completeness, variability, consistency, modifiability, traceability
- ⌘ **Correctness**: avoid over-requiring
- ⌘ **Unambiguousness**: clear, only one interpretation
- ⌘ **Completeness**: all requirements should be included
- ⌘ **Verifiability**: there should be a cost-effective way to ensure that each requirement is satisfied in the final system.
 - ⌘ "attractive" : without some agreed definition
- ⌘ **Consistency**: requirements should not contradict each other.

Good requirements, cont'd.



- ⌘ **Modifiability**: the requirement document should be structured so that it can be modified to meet changing requirements without losing consistency, verifiability, etc.
- ⌘ **Traceability**: each requirement should be traceable in the following ways.
 - ☑ trace backward to know why each requirement exists;
 - ☑ trace forward to go from source documents (marketing memos) to requirements;
 - ☑ trace forward to go from requirement to implementation;
 - ☑ trace back from implementation to requirement.

Setting requirements



- ⌘ Customer interviews.
- ⌘ Comparison with competitors.
- ⌘ Sales feedback.
- ⌘ Mock-ups, prototypes.

- ⌘ Next-bench syndrome (HP): design a product for someone like you.
 - ⌘ the ability to turn to the engineer at the next bench—or desk—and ask what features they want.

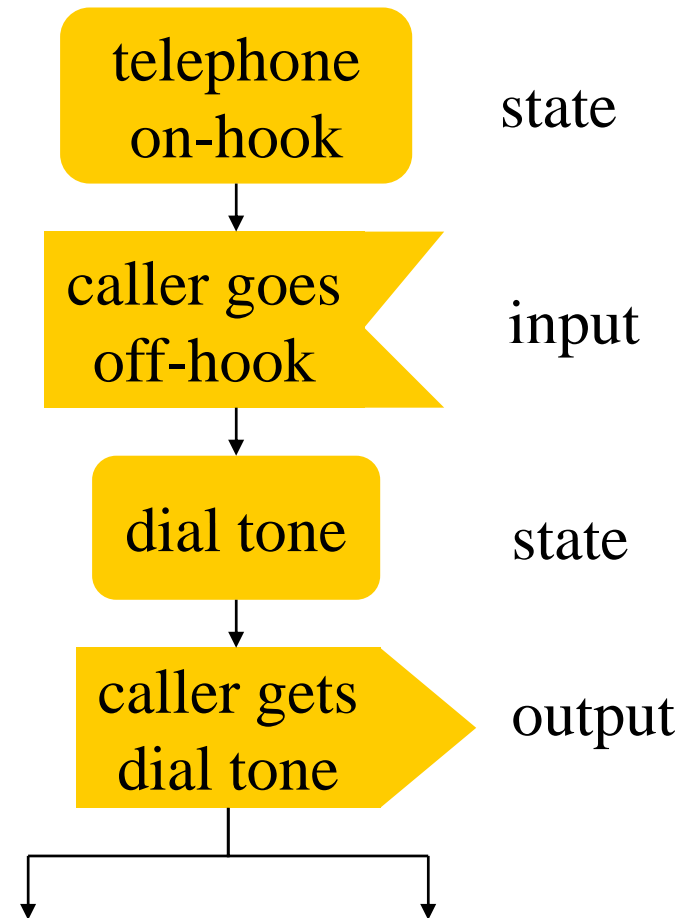
Specifications



- ⌘ Capture functional and non-functional properties:
 - ☑ verify correctness of spec;
 - ☑ compare spec to implementation.
- ⌘ Many specification styles:
 - ☑ control-oriented vs. data-oriented;
 - ☑ textual vs. graphical.
- ⌘ UML is one specification/design language.

SDL

- ⌘ Used in telecommunications protocol design.
- ⌘ Event-oriented state machine model.



SDL



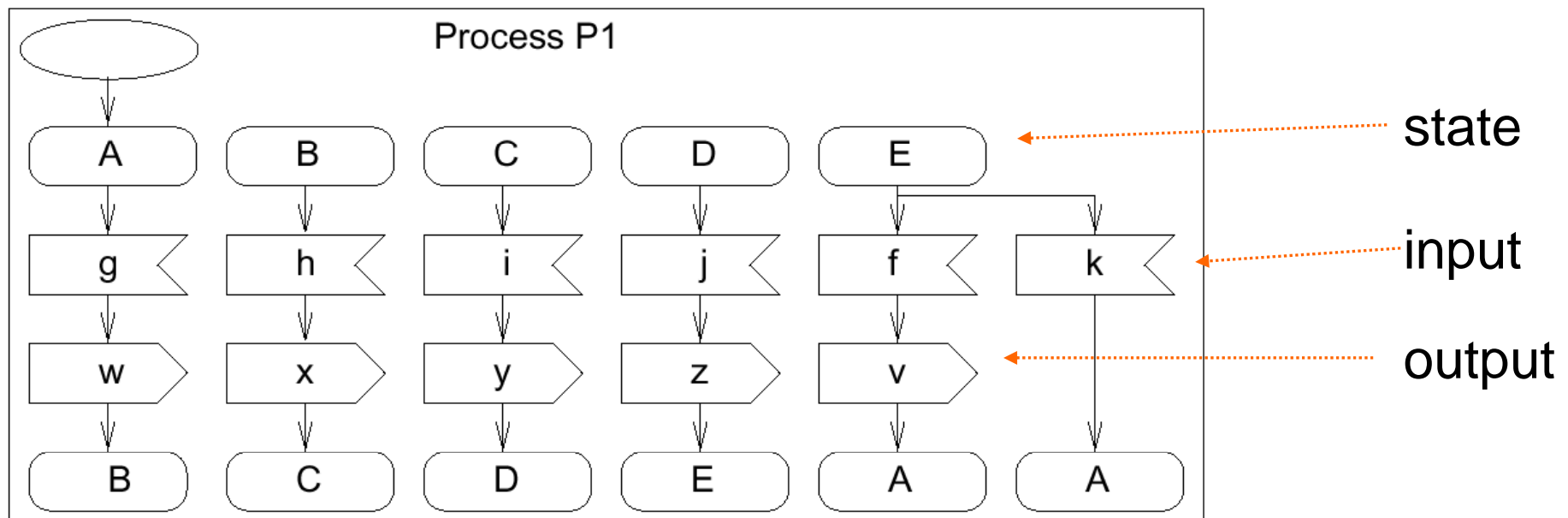
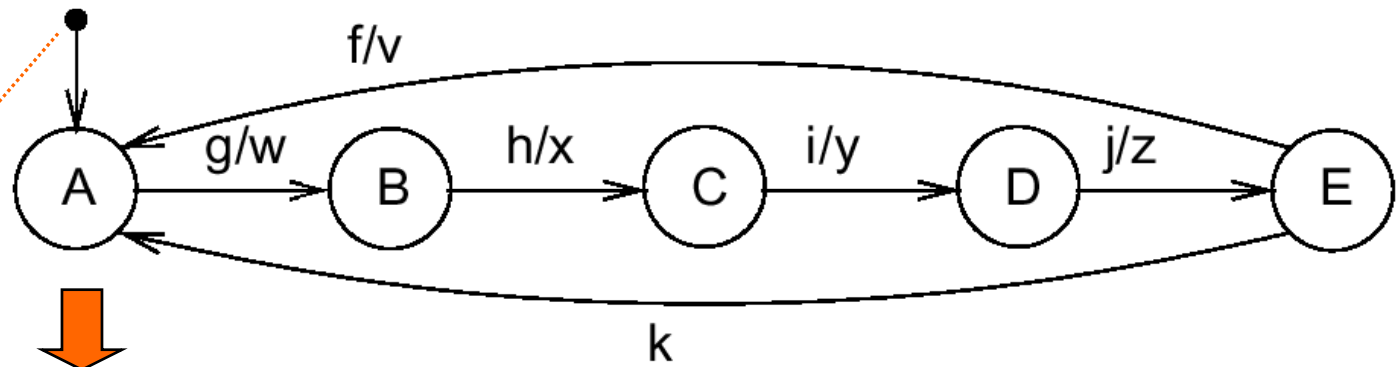
- ⌘ Language designed for specification of distributed systems.
- ⌘ Dates back to early 70s,
- ⌘ Formal semantics defined in the late 80s,
- ⌘ Defined by ITU (International Telecommunication Union): Z.100 recommendation in 1980
Updates in 1984, 1988, 1992, 1996 and 1999

SDL



- ⌘ Provides textual and graphical formats to please all users,
- ⌘ Just like StateCharts, it is based on the CFSM model of computation; each FSM is called a **process**,
- ⌘ However, it uses message passing instead of shared memory for communications,
- ⌘ SDL supports operations on data.

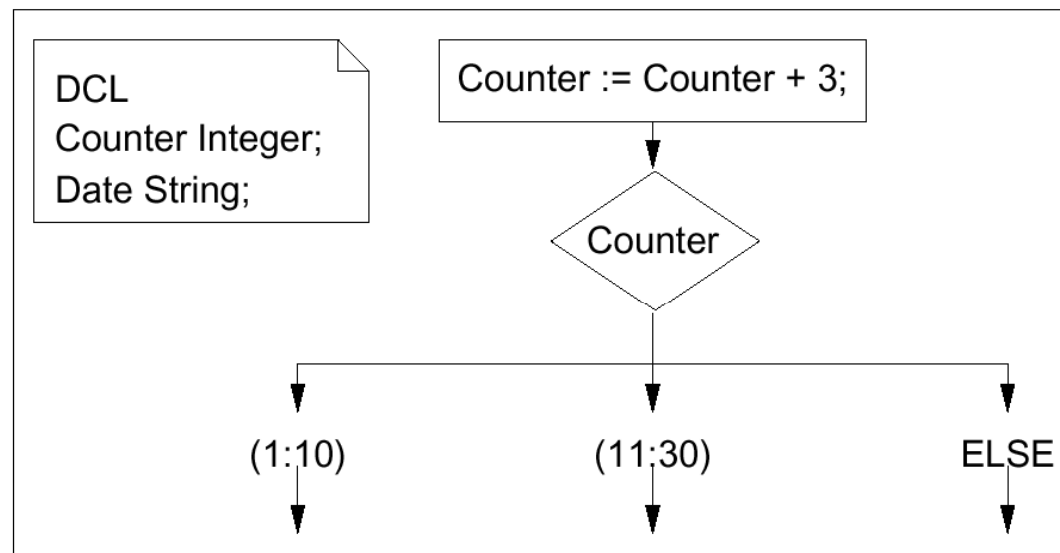
SDL-representation of FSMs/processes



Operations on data

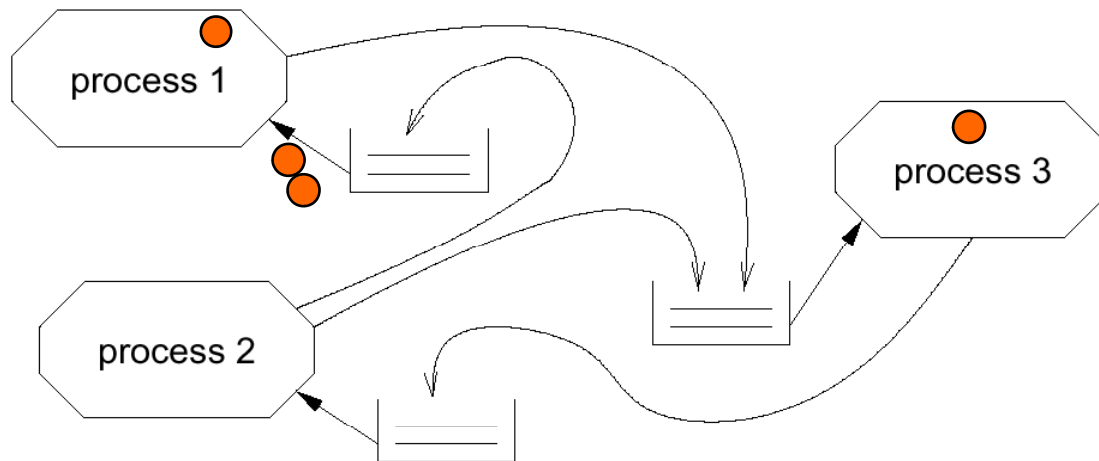
- ⌘ Variables can be declared locally for processes.
- ⌘ Their type can be predefined or defined in SDL itself.
- ⌘ SDL supports abstract data types (ADTs).

Examples:



Communication among SDL-FSMs

⌘ Communication between FSMs (or „processes“) is based on message-passing, assuming a potentially indefinitely large FIFO-queue.



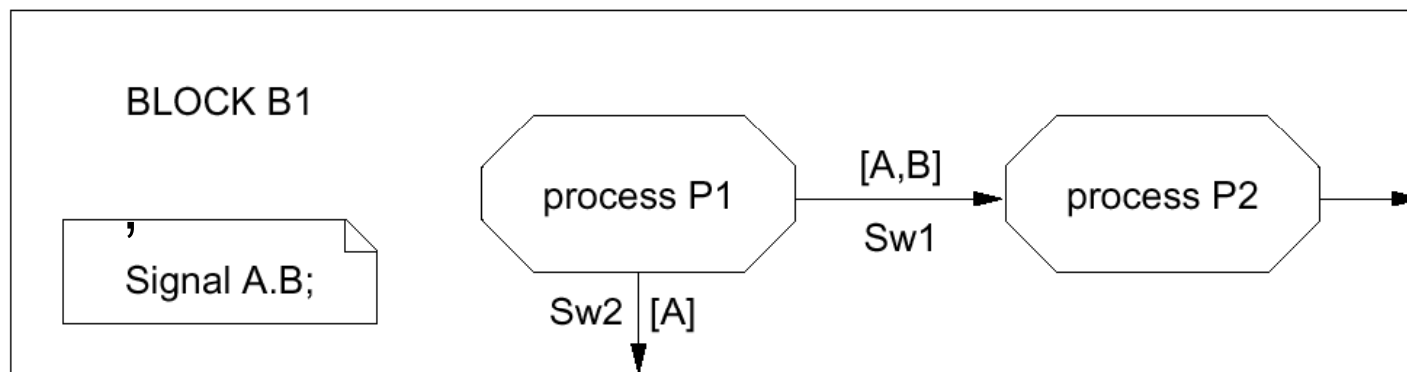
- Each process fetches next entry from FIFO,
- checks if input enables transition,
- if yes: transition takes place,
- if no: input is ignored (exception: SAVE-mechanism).

Process interaction diagrams

⌘ Interaction between processes can be described in process interaction diagrams (special case of block diagrams).

⌘ In addition to processes, these diagrams contain channels and declarations of local signals.

⌘ Example:

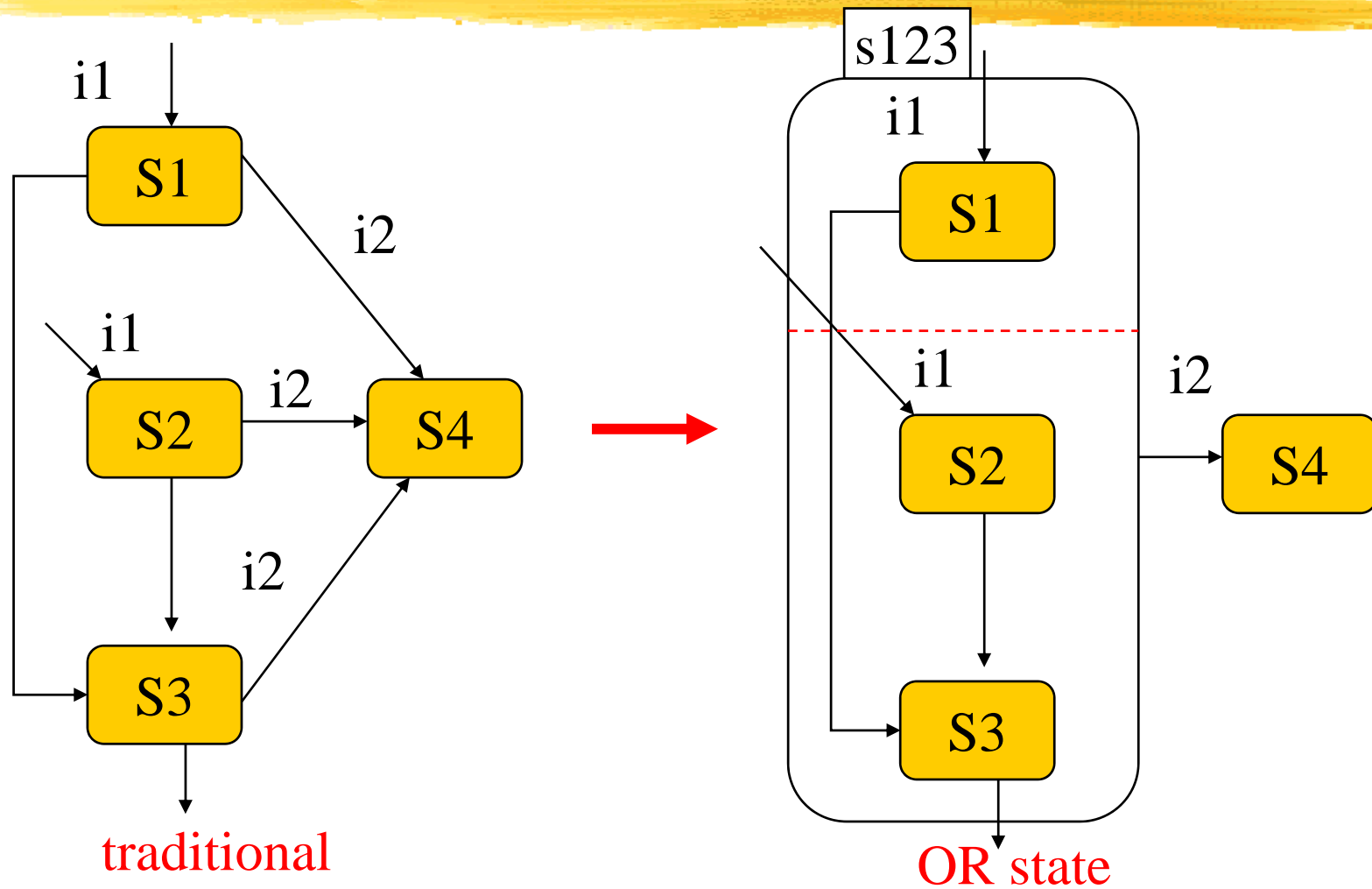


Statecharts

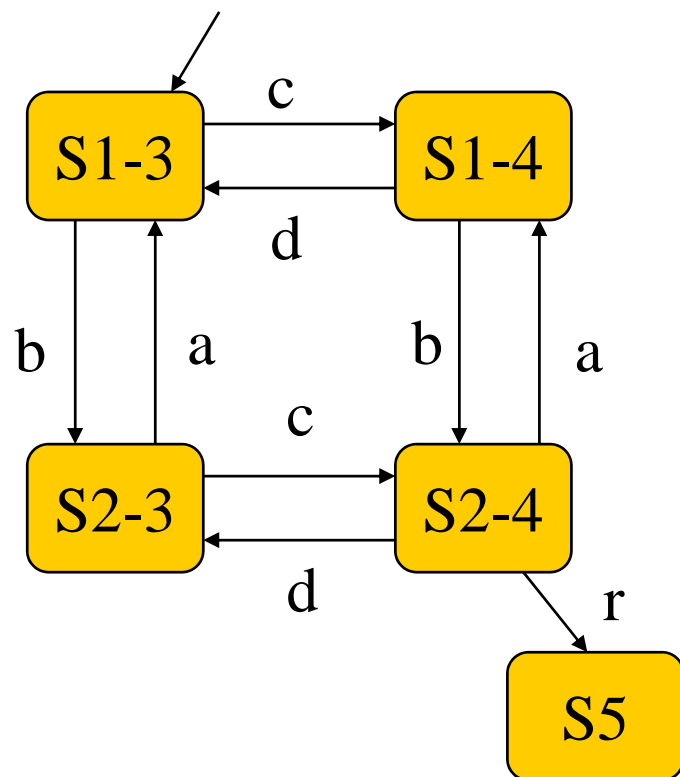


- ⌘ Ancestor of UML state diagrams.
- ⌘ Provided composite states:
 - ☑ OR states;
 - ☑ AND states.
- ⌘ Composite states reduce the size of the state transition graph.

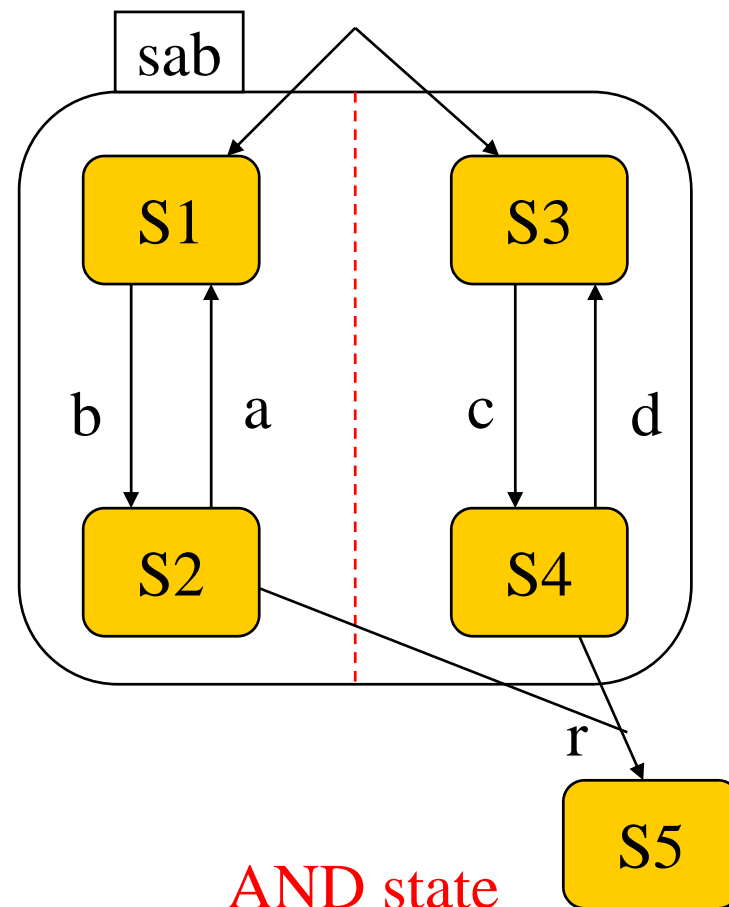
Statechart OR state



Statechart AND state



traditional



AND state

AND-OR tables

⌘ Alternate way of specifying complex conditions:

cond1 or (cond2 and !cond3)

	OR	
cond1	T	-
cond2	-	T
cond3	-	F

CRC cards



- ⌘ Well-known method for analyzing a system and developing an architecture.
- ⌘ CRC:
 - ☒ **classes**;
 - ☒ **responsibilities** of each class;
 - ☒ **collaborators** are other classes that work with a class.
- ⌘ Team-oriented methodology.

CRC card format



Class name:
Superclasses:
Subclasses:
Responsibilities: Collaborators:

front

Class name:
Class's function:
Attributes:

back

CRC methodology



- ⌘ Develop an initial list of classes.
 - ☑ Simple description is OK.
 - ☑ Team members should discuss their choices.
- ⌘ Write initial responsibilities/collaborators.
 - ☑ Helps to define the classes.
- ⌘ Create some usage scenarios.
 - ☑ Major uses of system and classes.

CRC methodology, cont'd.



- ⌘ Walk through scenarios.

 - ☑ See what works and doesn't work.

- ⌘ Refine the classes, responsibilities, and collaborators.

- ⌘ Add class relationships:

 - ☑ superclass, subclass.

CRC cards for elevator



⌘ Real-world classes:

- ☑ elevator car, passenger, floor control, car control, car sensor.

⌘ Architectural classes:

- ☑ car state, floor control reader, car control reader, car control sender, scheduler.

Elevator responsibilities and collaborators



class	responsibilities	collaborators
Elevator car*	Move up and down	Car control, car sensor, car control sender
Passenger*	Pushed floor control and car	Car control, floor control
Floor control*	Transmits floor requests	Passenger, floor control reader
Car state	Records current position of car	Scheduler, car sensor

Quality assurance



⌘ **Quality** judged by how well product satisfies its intended function.

☑ May be measured in different ways for different kinds of products.


⌘ **Quality assurance (QA)** makes sure that all stages of the design process help to deliver a quality product.

Therac-25 Medical Imager (Leveson and Turner)



- ⌘ Six known accidents: radiation overdoses leading to death and serious injury.
- ⌘ Radiation gun controlled by PDP-11.
- ⌘ Four major software components:
 - ☑ stored data;
 - ☑ scheduler;
 - ☑ set of tasks;
 - ☑ interrupt services.

Therac-25 tasks



- ⌘ Treatment monitor controlled and monitored setup and delivery of treatment in eight phases.
- ⌘ Servo task controlled radiation gun.
- ⌘ Housekeeper task took care of status interlocks and limit checks.

Treatment monitor task

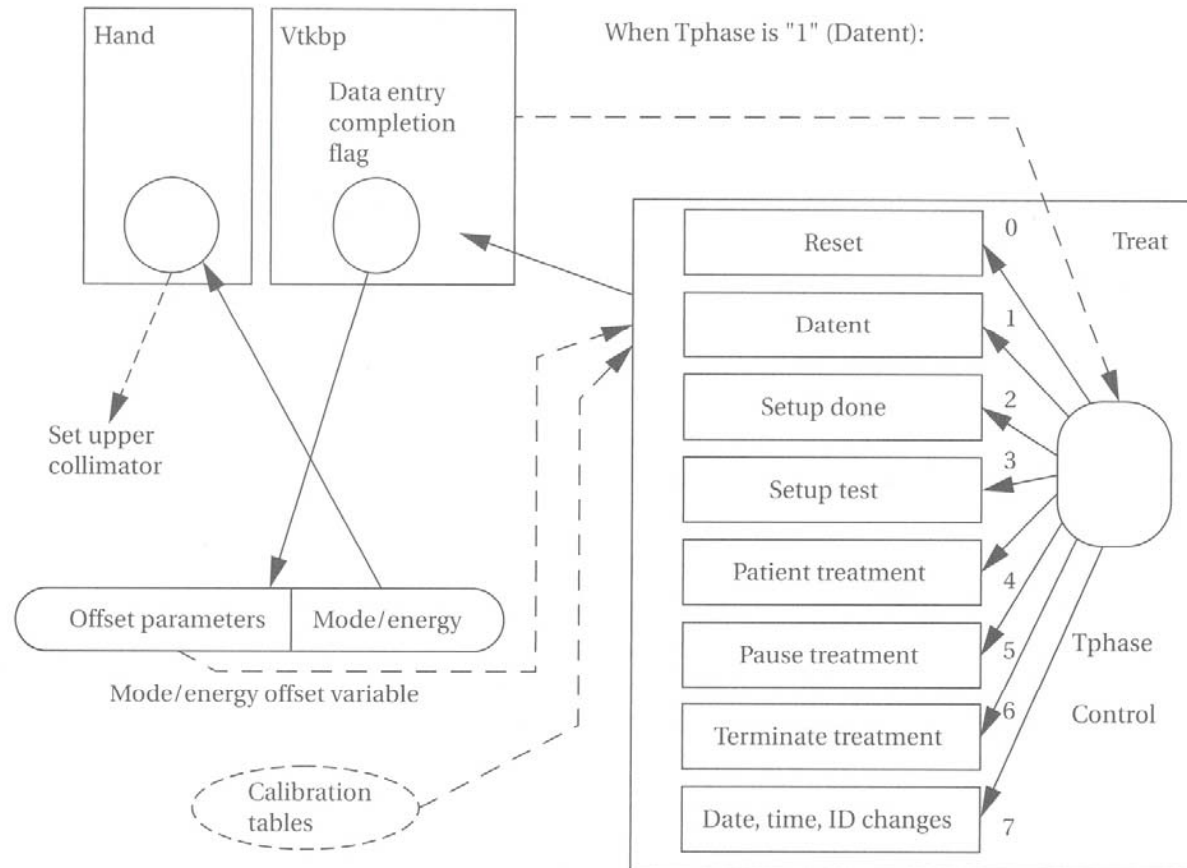


⌘ Treat was main monitor task.

☑ Eight subroutines.

☑ Treat rescheduled itself after every subroutine.

Treatment monitor task



Software timing race



⌘ Timing-dependent use of mode and energy:

☑ if keyboard handler sets completion behavior before operator changes mode/energy data, Datent task will not detect the change, but Hand task will.

Software timing errors



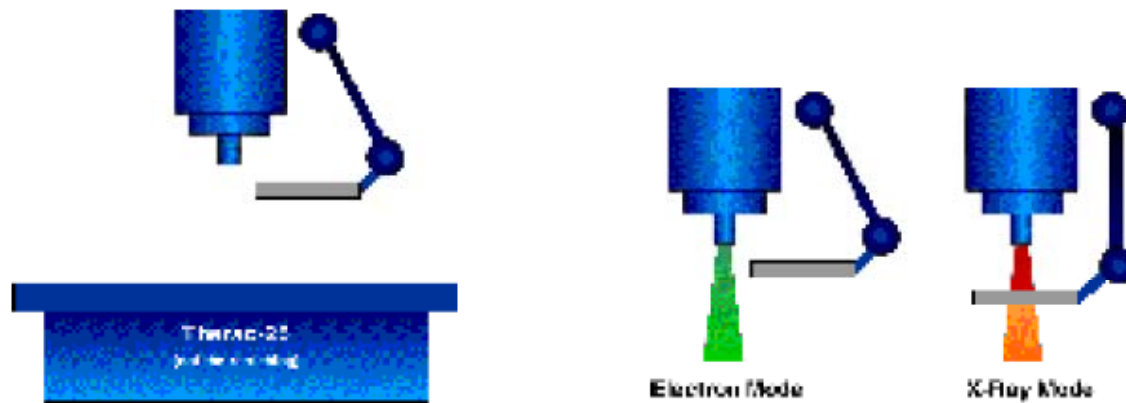
- ⌘ Changes to parameters made by operator may show on screen but not be sensed by Datent task.
- ⌘ One accident caused by entering mode/energy, changing mode/energy, returning to command line in 8 seconds.
- ⌘ Skilled operators typed faster, more likely to exercise bug.

Leveson and Turner observations



- ⌘ Performed limited safety analysis: guessed at error probabilities, etc.
- ⌘ Did not use mechanical backups to check machine operation.
- ⌘ Used overly complex programs written in unreliable styles.

Therac-25



Low energy mode
200 rads electrons

High energy mode
25 Mev x-ray

Therac-25 fault



Consequences of the over dosage

1	The breast was removed, complete loss of arm and shoulder mobility and constant pain
2	Patient severely burned, died November 1985
3	Tissue necrosis, chronic ulcerations and pain in the treated area and several skin grafts were made and the patient is alive today (1995)
4	Pain in neck and shoulders, periodic nausea and vomiting, radiation induced myelitis of the cervical cord, paralysis of the left arm and both legs, left vocal cord and left diaphragm Died after five months due to the complications
5	Patient died one month later
6	The patient died within weeks from a severe terminal form of cancer but the over-dosage probably shortened his life

Characteristics of the accidents



- ⌘ Three cases involved carousel rotation prior to treatment (confirmed)
- ⌘ The accelerator malfunctioned shortly after “beam on”, reporting a malfunction code at the console
 - ⊞ The codes were cryptic and not recognized by the operator as indicating a serious error
- ⌘ In several cases, the operator repeated the exposure one or more times
- ⌘ Following treatment, the patients complained of burning sensations, sometimes accompanied by a feeling of electric shock
- ⌘ In each case, the patients received doses of between 4 and 25 KRADs in a very brief exposure (1-3 seconds)

Summary of causes of accidental exposure



⌘ Manufacturer recycled software

- ⊞ Earlier model functioned somewhat differently, so software was not entirely suitable
- ⊞ Newer model relied entirely on software for safety, whereas older model had mechanical and electrical interlocks
- ⊞ The safety of the newer system was not evaluated as a whole, only the hardware was evaluated since software had been in use for years...

⌘ The manufacturer had no mechanism for investigating and reporting accidents

- ⊞ After the first accident, the manufacturer refused to believe the equipment was at fault
- ⊞ The FDA was not notified, nor were other users
- ⊞ The vendor kept their opinion that this machine was safe

Lessons: Suggested by computer science consultants



- ⌘ Documentation key from beginning
 - ☑ Use established software engineering practices
 - ☑ Keep designs simple
 - ☑ Build in software error logging & audit trails
- ⌘ Extensive software testing and formal analysis at all levels
 - ☑ Revalidate re-used software
- ⌘ Don't rely only on software for safety
- ⌘ Do incorporate redundancy
- ⌘ Pay careful attention to human factors
- ⌘ Involve users at all phases

The continuation

- ⌘ The 11 machines were refitted with the safety devices required by the FDA and remained in service
- ⌘ No more accidents were reported from these machines
- ⌘ After the Therac-25 deaths, the FDA made a number of adjustments to its policies in an attempt to address the breakdowns in communication and product approval.
 - ☒ In 1990, health care facilities were required by law to report incidents to both the manufacturer and the FDA.



Image supplied by Larry Watts

ISO 9000



- ⌘ Developed by International Standards organization.
- ⌘ Applies to a broad range industries.
- ⌘ Concentrates on process.
- ⌘ Validation based on extensive documentation of organization's process.

CMU Capability Maturity Model

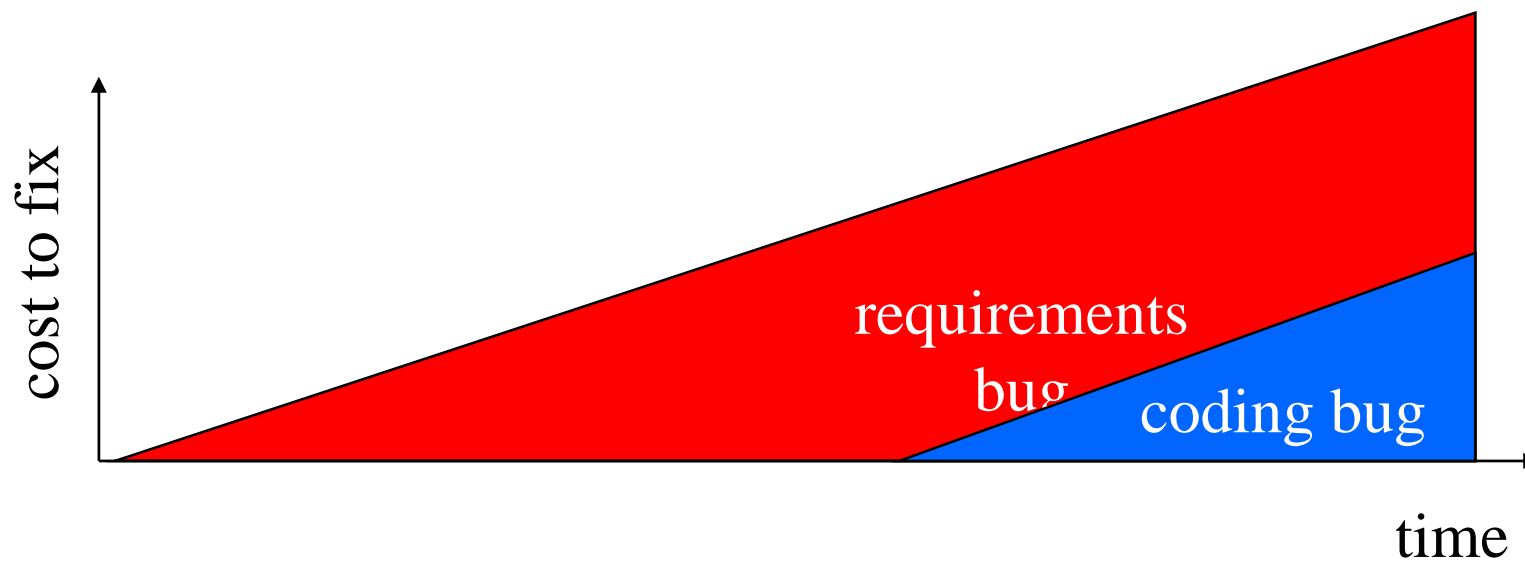


⌘ Five levels of organizational maturity:

- ☒ **Initial**: poorly organized process, depends on individuals.
- ☒ **Repeatable**: basic tracking mechanisms.
- ☒ **Defined**: processes documented and standardized.
- ☒ **Managed**: makes detailed measurements.
- ☒ **Optimizing**: measurements used for improvement.

Verification

- ⌘ Verification and testing are important throughout the design flow.
- ⌘ Early bugs are more expensive to fix:



Verifying requirements and specification



⌘ Requirements:

- ☑ prototypes;
- ☑ prototyping languages;
- ☑ pre-existing systems.

⌘ Specifications:

- ☑ usage scenarios;
- ☑ formal techniques.

Design review



- ⌘ Uses meetings to catch design flaws.
 - ☑ Simple, low-cost.
 - ☑ Proven by experiments to be effective.
- ⌘ Use other people in the project/company to help spot design problems.

Design review players



- ⌘ **Designers**: present design to rest of team, make changes.
- ⌘ **Review leader**: coordinates process.
- ⌘ **Review scribe**: takes notes of meetings.
- ⌘ **Review audience**: looks for bugs.

Before the design review



- ⌘ Design team prepares documents used to describe the design.
- ⌘ Leader recruits audience, coordinates meetings, distributes handouts, etc.
- ⌘ Audience members familiarize themselves with the documents before they go to the meeting.

Design review meeting



- ⌘ Leader keeps meeting moving; scribe takes notes.
- ⌘ Designers present the design:
 - ☑ use handouts;
 - ☑ explain what is going on;
 - ☑ go through details.

Design review audience



⌘ Look for any problems:

- ☑ Is the design consistent with the specification?
- ☑ Is the interface correct?
- ☑ How well is the component's internal architecture designed?
- ☑ Did they use good design/coding practices?
- ☑ Is the testing strategy adequate?

Follow-up



- ⌘ Designers make suggested changes.
 - ☑ Document changes.
- ⌘ Leader checks on results of changes, may distribute to audience for further review or additional reviews.

Measurements



⌘ Measurements help ground our beliefs:

- ☑ Do our practices really work?

- ☑ Do they work where we think they work?

⌘ Types of measurements:

- ☑ bugs found at different stages of design;

- ☑ bugs as a function of time;

- ☑ bugs in different types of components;

- ☑ how bugs are found.