Software Development Processes

Sequential, Prototype-based RAD, Phased, Risk-based Spiral

Software Life-Cycle Models

Breaking projects down into pieces for ...

- Planning
  - ``What do I do next?"
- Process visibility
  - ``Are we on schedule?"
- Intellectual manageability
- Division of labor

Process Models in Other Fields

- Reliable, efficient production
  - Process improvement for quality, efficiency
- Predictable production
  - Ability to plan, schedule, and budget production
- Standardization
  - Economic advantage of standard processes and components
- Automation

Inadequacy of Industrial Process Models

- Software is primarily an intellectual, design-based process
  - Unlike fabrication of physical things
  - More like designing an automobile than building it
- Software is "unstable"
  - Malleability is a major advantage of software over hardware, but
  - Changing requirements and design make controlled processes more difficult

The "Code and Fix" Model (or, Software through Chaos)

- Process steps:
  - Write some code
  - Fix and enhance
  - Repeat until satisfied, or until unmanageable
- Characteristics of code-and-fix model
  - Suitable when: Developer is the user (no formal requirements), schedule is short (no planning), quality need not be high (fix as needed)
  - Highly unstable: Software structure deteriorates over time, or collapses as complexity increases

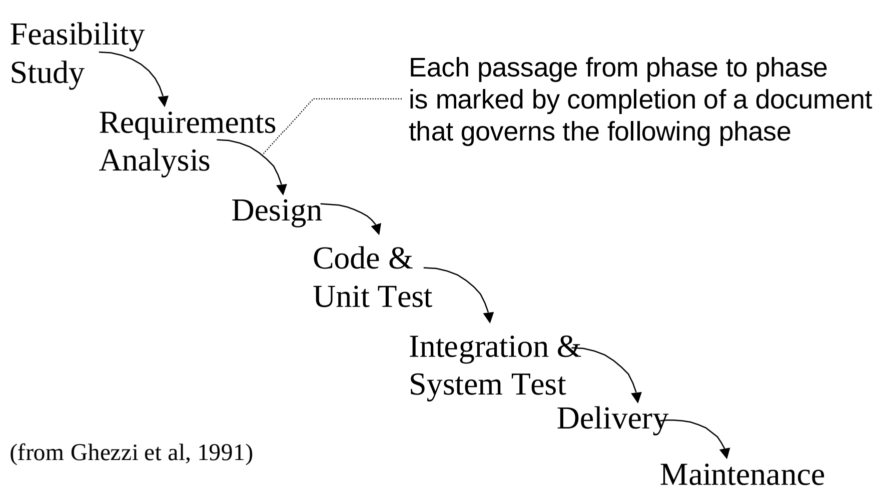
Changes Motivating Defined Processes

- Non-technical users, distinct from developers
  - Problem of "building the wrong system"
  - Need for careful analysis of requirements, distinct from design and implementation
- Scale and complexity => Team development
  - Organizational structure and coordination
  - Control of communication complexity
  - Need for design phase, unit & integration testing
- Need for predictability => Scheduling
- Quality requirements => Checkpoints

The "Waterfall" model

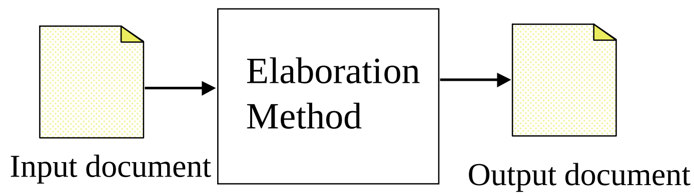
- Inspired by industrial product development cycles, esp. aircraft
- A document-based model
  - Stages in development are marked by completion of documents
  - Feedback and feed-forward are through documents
- Several variations

Waterfall Model (example)



(from Ghezzi et al, 1991)

Waterfall Model Phase



- Goal is an output document consistent with the input document; an "error" is an inconsistency
- Phase is complete when document is finished
- Each phase has specific methods

Example waterfall stages ...

Feasibility Study

- Evaluate costs and benefits of a proposed application
  - Required for go/no-go decision or choice among competing projects
  - Ideally requires complete analysis and design; Practical reality: Limited time and resources
  - Results in problem definition, alternative solution sketches, and approximate resource, cost, and schedule

Example waterfall stages ...

Requirements Analysis

- Produce specification of what the software must do
  - User requirements; may be divided into problem analysis and solution analysis
  - Suppress the "how" until design phase
  - Must be understandable to user, which in practice means it is necessarily somewhat informal
  - To the extent possible, should be precise, complete, unambiguous, and modifiable; Should include object acceptance tests and a system test plan

Example waterfall stages ...

Design and Specification

- May be divided into external design (and/or system specification), preliminary design, and detailed design
- Results in (semi-)formal diagrams and text defining structure and function of the software, ready for programming individual units
- Many notations, methods, and tools for different "styles" of design

Example waterfall stages ...

Coding and Module Testing

- Individual programmers produce program "units," which are assembled into subsystems and the final system
- Includes unit testing and debugging, and may include inspections
- Often includes much non-product code, called "scaffolding"

Example waterfall stages ...

Integration and System Testing

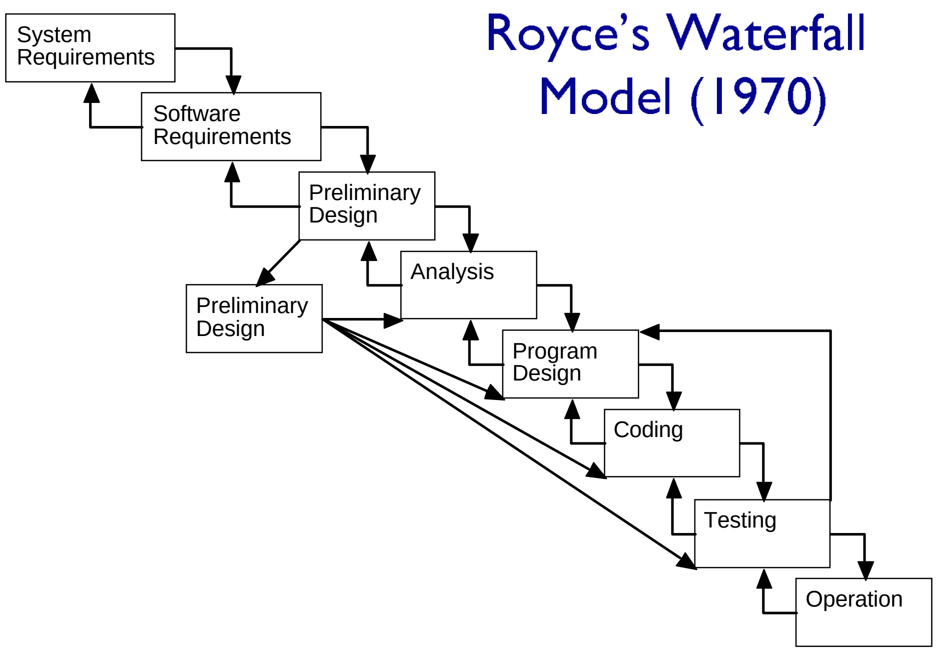
- Assembly of units into larger and larger substructures
- Proceeds according to a "build plan" which is typically "top-down" or "bottom up"
- Subsystem test followed by system, apha, and beta test; purpose of testing shifts from debugging to acceptance, and may involve an independent test team

Example waterfall stages ...

Delivery and Maintenance

- Beta test: controlled release to a small (or adventurous) real-world clientele
- Alternative: single-client and critical applications "run parallel"
- After delivery, further change to sofware is called "maintenance" (of which most is NOT fixing bugs)

Royce's Waterfall Model (1970)



Characteristics of the Waterfall Model

- Limited iteration
  - Naive version is purely sequential; more commonly there is some iteration and adjustment, but the model is highly sequential
  - Well-suited to a "contract" mode of application
- "Big bang" development
  - Beginning from nothing
  - Ending with a single delivery of a single product

RAD: Rapid Application Development

A variant of "evolutionary prototyping"

Based partly on: The Impact of the development context on the implementation of RAD approaches by D. Fulton, 1996

(was: www.cs.ucl.ac.uk/staff/D.Fulton/interim.html)

Main characteristics of RAD

- Rapid ≈ 6 weeks to 9 months
- Small, flat, highly skilled teams
- Intense user participation
- Iterative prototyping (with less paper-based documentation)

Origins

- Evolutionary prototyping
  – vs. throw-away prototypes: closer to incremental build, but more dynamic
- DuPont (mid-80s) Rapid Iterative Production Prototyping
- IBM Joint Application Development method (JAD)
- Popularized by J. Martin (1991) and others

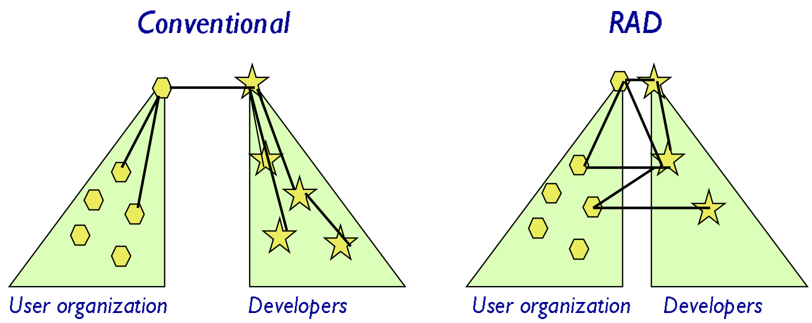
RAD "philosophy"

- Initially fix:
  - high-level requirements,
  - project scope
  - plan (schedule)
- Then iteratively build the product
  - with intense user involvement to negotiate requirements and test deliverables

Joint Application Development "Workshops"

- Objective: Scope the project
- Participants:
  - Development team
  - User representatives
  - Facilitator
- Intense negotiation to create stable scope and plan
  - similar to "design to schedule," applied to requirements

RAD communication structure



- Peer-to-peer communication between users and developers
- Intense user involvement (and commitment) in negotiating requirements and testing prototypes

RAD team structure

- Small teams of highly-skilled developers
- Fixed team through full development
  - Less specialization; each developer must fill several roles
  - Less reliance on formal documents to record requirements and design
- Requires *stable* staffing
  - Loss of a developer is a larger risk than in document-based process models
  - Loss of user representatives is also a danger

Timeboxing

- If functionality not delivered by date, scale back or abandon
  – Radical application of "design-to-schedule"
- The build-plan is stable; the product functionality is fluid within bounds of project scope
  – What is actually built depends on technical feasiblity as well as user wants

Prototype-based requirements elicitation

- Cycle: Build, demo, revise design
  – Scheduled review meetings with demos and feedback
  – Additional internal prototype build cycles
  – Additional ad hoc user demos
- "Shopping list" replaces detailed requirements document
  – Broad list of desirable functions can change depending on user feedback

Reduced Paper Documentation

- Emphasis on rapid delivery and change
  - Not on preserving information for a longer period
  - Fixed personnel (including user representatives) reduces need for documents as orientation and communication
  - Active, intense user participation
- Reliance on computerized documentation
  - CASE tools, databases and application generators
  - The prototype itself as "documentation"
- Developer "logs" of design rationale

RAD on Contract?

- Requires stronger relationship than typical contracts
  - Since requirements are not fully known when contract is let
- May be based on fixed effort, rather than fixed functionality

RAD tools

- RAD projects typically rely on strong tool support
  - application generators, database engines (including interface builders, etc.)
  - CASE tools
  - ...
- Reported success is mostly within well-understood and supported domains, esp. information systems

"Super designers"?

- Small, flat teams require multi-talented individuals
  - Technical, inter-personal, and managerial skills
  - Overall view of project, not only pieces
- Vague requirements require strong motivation to do more than "enough"
- Strong management needed to hold human resources
  - Loss of a developer can be disastrous
  - Loss of adequate user involvement can be nearly as bad

When is RAD appropriate?

- Requirements are not clear or stable
- Technical pre-requisites available: adequate tool and facility support
- Developer expertise in domain and tools
  - especially: able to anticipate likely change
- Strong facilitator/manager
  - able to keep project appropriately scoped
  - able to hold resource (people) for duration of project

RAD issues

- Quality: Little process control, little documentation on which to base measurement and acceptance
  - Quality measured by "the smile on the user's face"
- Lifetime cost: What will it cost to maintain RAD projects?
  - BUT if initial build cost is comparable to a revision cycle, a "disposable" system may be acceptable
- Heavy reliance on individuals
  - Risk may be too high for critical projects

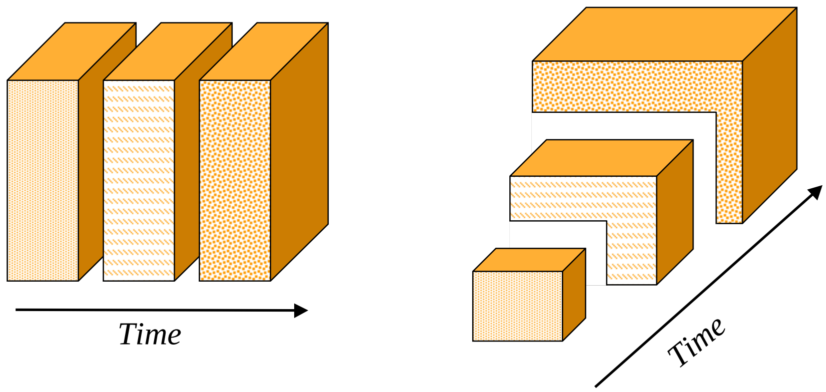
Summary: RAD

- Evolutionary prototyping method
  - with particular management features like "timeboxing"
- Small team, limited scope approach
- Intense, continuous user involvement
- "Programming in the small" at its outer limits?
  - Most of what has been omitted (documents, clear process, etc.) are the measures we use to cope with multiple people and long schedules

Phased Projects

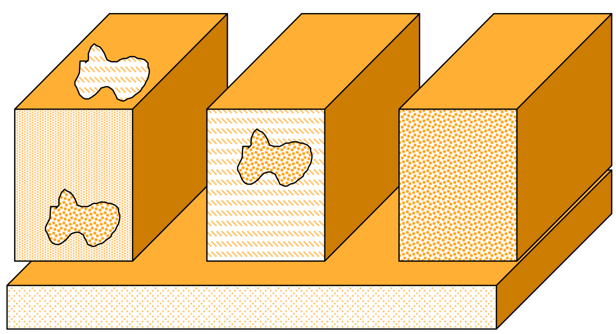
- Develop & Deliver in Increments
  - May repeat entire waterfall model in each increment
- Goals:
  - Keep clients/customers happy
  - Improve requirements through feedback
  - Improve process visibility through more frequent milestones

Dividing a Large Project into Phases



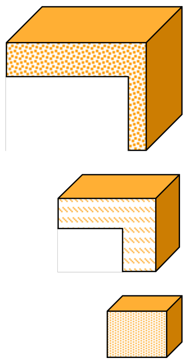
- Division by function
- Incremental Delivery

Functional Division in Practice



- Some shared infrastructure is developed (incrementally)
- Some revision to previous phases is required

Planning Incremental Development
What is a good increment?



- Identify system subsets
  - Minimal usable feature sets
  - Encapsulated functions (limit scope of change)
- Choice driven by:
  - Schedule
    (opportunity cost, time-to-market)
  - Decomposability
    (minimize duplicated work)
  - Risk control

Spiral Model
(Risk-driven evolutionary development)

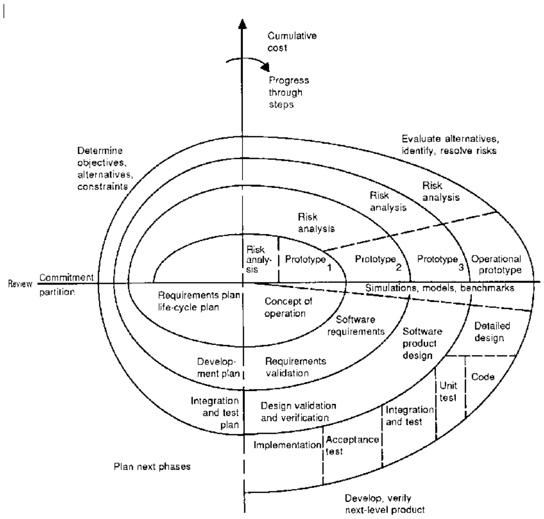


Figure 7.4 The spiral model. (From Boehm [1988], ©1988 IEEE, by permission of IEEE.)

In each "turn" of the spiral

- Problem definition
  - Determine objectives (qualities to achieve)
  - Identify alternatives and constraints
- Risk analysis
  - Determine risks
  - Gain information (typically through prototyping)
- Develop & verify next level "product"
  - may be only requirements, or design
- Plan next phase

Prototypes vs. Incremental Deliveries

- The primary goal of a prototype is *information*
  - Should address the most significant risks
- Incremental deliveries should be useful
  - May avoid the highest risks
- These goals are in conflict!
  - It is sometimes possible to serve both purposes
  - *but* ... Many "prototypes" fail to serve either purpose, because developers fail to distinguish goals and plan accordingly

Prototyping for Information

- Requirements clarification
  - Users "learn what they want" by using the prototype
  - Implicit requirements are identified through failure
  - Human interface can be assessed and refined
- Design alternatives
  - Performance, complexity, capacity, ...
  - Requires evaluation plan *before* implementation

Choosing a Process Model

- No single "best" model
  - Depends on many factors, including the experience of a particular organization in a particular application domain
- Larger team, larger product
  => More elaborate process
- More risk, less experience
  => More iteration