Software process

- **Process**: the activity of building a product (software program, house, IC, TV set, ...)
- Where do we start? What is the sequence of activities?
  - Most projects go through common phases (requirement gathering, specification, implementation, testing,...)
  - Lots of empirical and accumulated knowledge.
- How do we estimate the effort?
  - Cost/Time estimation
  - Ensuring that the resources are in place (personnel, facilities, tools,...)

The Waterfall Model (officially)

Start → Requirement Analysis → System Design → Program Design → Coding → Unit testing → Integration testing → System testing → Acceptance testing → Ship IT

The Waterfall Model ("builder's view")

Start → Gather Requirements → Partion (Design) → Implementation → Integration Testing → Ship IT

Gather Requirements

- What should the system do?
  - Talk to customer, user (if available), marketing
  - Note that they don’t always know what they want!
  - Wish list of features
  - Rough cost time estimates
**Specification**

- Write a specification of what the system should do
  - and of what it requires in order to work
- Specification must be complete
  - Describe behavior in all circumstances
  - Can be large
  - How are details filled in?

**Partition (Design)**

- Decide system architecture
- Divide the design into manageable pieces
  - According to functionality
  - Split along stability lines
- Specify the interfaces between modules

**Partition (Design)**

- Decide system architecture
- Divide the design into manageable pieces
  - According to functionality
  - Split along stability lines
- Specify the interfaces between modules

**Implementation**

- Make a plan, guided by:
  - Priority of features
  - Ability to test the design
  - Critical components first
- Implement the components
  - Or re-use components, if possible
- Test the components
  - May require scaffolding
  - Order of implementation is important

**Integration**

- Put everything together
- Test it
  - For functionality
  - According to interfaces
- What if...
  - Some components are late (can we integrate/fix the others?)
  - Errors in interfaces (more expensive to fix)

**System and Acceptance Testing**

- System testing
  - Does the system work according to the specifications?
  - Major effort
  - When to ship? (0 bugs may be unattainable)
- Acceptance testing
  - Does the customer like the system?
  - Beta-releases
**Ship it!**

**The Waterfall model**
- Top-down specifications
- Bottom-up implementation and testing

**The V Model (relating tests to specs)**
- Gather Requirements
  - Specification
  - Partition (Design)
- Implementation
  - Unit testing
  - System testing
  - Integration testing
- Acceptance testing

**The Waterfall model**
- Top-down specifications
- Bottom-up implementation and testing
  - Disadvantages?
    (your opinion here)

**Waterfall Model - Disadvantages**
- Little feedback from later phases on early ones.
  - If the specification is wrong and this is discovered during acceptance testing a lot of work needs to be redone.
- Long time before anything useful is obtained
  - Difficult to get user input
  - It is much easier for the user to react to a working system, and revise the requirements, than to give a consistent and complete set of requirements at the beginning.

**Waterfall Model - Shortcomings**
- Difficult to write all-encompassing specifications in the absence of an implementation.
- Little usable outcome if the project has to be shut/reoriented in midcourse.
- If delivery time is long:
  - User requirements are likely to change, causing instability (and more bugs)
  - Competing products coming out (another cause for requirement change)
Waterfall Model - Good Points

- Emphasis on specifications, and on the verification that the specifications are met.

How often is it used in practice?

The Waterfall Model in Practice

Rapid Prototyping

- Get a rough version of the requirements
- Write a quick prototype and show it to the user
  - To demonstrate functionality
  - Can be a mock-up only (not distributed, ...)
- Get refined requirements
- Proceed as in the Waterfall model
  - Throw away prototype
  - Design and implement new system

Rapid Prototyping - Disadvantages

- Time invested in the prototype
  - Tendency to over-do it to please the customer
  - Demos, demos
- Difficult to throw away the prototype and do a clean full-system design
  - Some code of the prototype tends to make it to the full system
  - And sub-optimal design choices (architectures, algorithms) with it
- Potentially very useful if done correctly
  - Easier to evolve requirements based on a prototype
  - Can be used to experiment with architectures

Iterative Models

- Evolve the system as a sequence of prototypes: from rough ones, used to evolve requirements, to advanced ones with more and more functionality.
Iterative Models

Start → Gather Requirements → Specification → Partition (Design) → Implementation → Integration → Testing → Demo it → Ship it

Gather Requirements

- Get preliminary requirements

Specification

- Write a specification of what the system should do
  - Focus at the main functionality first
  - Explore some "reasonable" assumptions for under-specified requirements

Partition (Design)

- Decide system architecture
- Divide the design into manageable pieces
  - According to functionality
  - Split along stability lines
- Specify the interfaces between modules

Implementation

- Make a plan, guided by:
  - Priority of features
  - Ability to test the design
  - Critical components first
- Implement the components
  - Do just the bare-bone functionality first
- Aim getting quickly a complete system build, that can do very little
Integration and Testing

- Put everything together
- Test it
  - Are the interfaces well chosen?
  - Test all the functionality present
- If functionality is substantial, can even be shipped

Get Feedback

- Get feedback from user and/or customer
- Use the feedback to revise the requirements, and hence the specifications

Revise Requirements and Increase Functionality

- Revise the prototype (fix incorrect requirements)
- Design and implement more functionality

Revise Requirements and Increase Functionality

- Revise the prototype (fix incorrect requirements)
- Design and implement more functionality
- Test it
- Continue until sufficient functionality is reached to ship (may ship versions with increasing functionality)

Rapid Prototyping

Advantages:
- Much easier to get correct requirements
  - Hopefully, they won’t change as much after the first few prototypes
  - Feedback on experimental features
  - Experiment with architectural choices early,
  - Refine design (algorithms, ...) only where needed.
  - Even if development is stopped early, some functionality may be available.
  - Quantifiable (75% of functionality implemented)

Disadvantages:
- Much easier to get correct requirements
  - Hopefully, they won’t change as much after the first few prototypes
  - Feedback on experimental features
  - Experiment with architectural choices early.
  - Refine design (algorithms, ...) only where needed.
  - Even if development is stopped early, some functionality may be available.
  - Quantifiable (75% of functionality implemented)
In practice

- **Waterfall model:**
  - Systems that must work correctly from the beginning
  - Systems that are difficult to test, or demo
    - Spacecrafts, air traffic control, banking systems, ...
- **Iterative model:**
  - Consumer software (office productivity, etc)
  - Can be easily demos, previewed
  - When requirements are fuzzy...

Cost/Time/Performance Estimation

- Early on, customers typically want:
  - Cost estimate
  - Time estimate
  - Performance estimate
- One of the hardest problems!
  - And one that is most often gotten wrong:
    - DMV registration: 87-93, 6.5x cost estimate, 2x time estimate
    - Denver airport baggage system: after one year slippage, cost of $600 millions a day
    - FAA AAS (Advanced Automation System): 5 years late, $1 billion over budget

The "Lines of Code" Fad

- Is it really a good criterion??
  - Many languages require different amount of lines to do the same
  - Not robust:
    - Loop unrolling
    - Style of variable declaration, coding
    - COBOL is a high productivity language (stear away from python or ML, if you want high productivity)
  - It reminds me of alchemists, that were trying to measure the similarity of elements using completely inappropriate measurements.
  - Still, what to use otherwise? Procedure interaction? Flow graph analysis? ...

The "Lines of Code" Fad

- In my personal experience:

```
<table>
<thead>
<tr>
<th>Loc</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code</td>
<td>Think</td>
</tr>
<tr>
<td>Test</td>
<td>Cleanup</td>
</tr>
</tbody>
</table>
```

- What about you?

Lines of Code - The Light Side

- From Alexander Dumas, "The Three Mosqueteers":
  - Qui? demande Son Eminence
  - Elle et lui.
  - Lo reine et le duo? s'ecrit Richelieu.
  - Qui.
  - Et au ceb?
  - Au Louvre.
  - Vous enetes sur?
  - Parfumement sur.
  - Qui vous l'dit?
  - ...

Cost/Time/Performance Estimation

- How to do estimates?
- **Empirical models:**
  - Measuring parameters
    - How much red-time code?
    - How much distributed processing?
    - How big a user interface?
  - Training a model based on previous experiences
  - Using trained model to do forecasts.
Cost/Time Estimation

Activity graphs:

- Dig hole
- Lay foundations
- Build ext walls
- Main plumbing
- Build int walls
- Main electrical
- Floors
- Done

Scheduling activity graphs: LDF (Latest Deadline First)

- Build the schedule from last to first, by recursively:
  - Let Q be the subset of activities whose descendants have been scheduled.
  - Pick the activity from Q that has the latest deadline.
  - Add p to the front of the schedule.
- Should be really called LDL (Latest deadline last).
- Optimal w.r.t. maximum lateness: end_time - deadline

LDF Example

earliest start time duration deadline

J_6

J_5

J_4

J_3

J_2

J_1

0 2 0 0 1 3 0 2 0

16 0 2 4 1 0 10 2 0

earliest start time duration deadline

J_6

J_5

J_4

J_3

J_2

J_1

0 2 0 0 1 3 0 2 0

16 0 2 4 1 0 10 2 0
LDF Example

earliest start time
duration
deadline

J_3 J_4 J_5

LDF: Proof of Optimality

<table>
<thead>
<tr>
<th>S</th>
<th>A</th>
<th>J_4</th>
<th>B</th>
<th>J_5</th>
</tr>
</thead>
<tbody>
<tr>
<td>S'</td>
<td>A</td>
<td>B</td>
<td>J_4</td>
<td>J_5</td>
</tr>
</tbody>
</table>

J_6 should be last according to LDF

L_{\text{max}} = \max \{L_A, L_B, L_C, L_D, L_{\text{max}}\}

L_A \text{ unchanged}
L_B \leq L_A \text{ as } B \text{ starts earlier}
L_D \leq L_{J_6} \text{ as } J_6 \text{ starts earlier}
L_C = f_a - d_a \leq f_b - d_b \text{ as } d_a > d_b = L_b
So \ L_{\text{max}} \leq L_{\text{max}}.

The argument is concluded by induction.

Main Risk Factors [Boehm]

- Personal shortfalls
- Unrealistic schedules and budgets
- Developing the wrong functionality
- Developing the wrong user interface
- Gold plating
- Continuing stream of requirement changes
- Shortfalls in externally performed tasks
- Shortfalls in externally furnished components
- Real-time performance shortfalls
**Main Risk Factors [Boehm][LdA]**

- Personnel shortfalls
- Unrealistic schedules and budgets
- Developing the wrong functionality
- Developing the wrong user interface
- Gold-plating
- Continuing stream of requirement changes
- Shortfalls in externally performed tasks
- Shortfalls in externally furnished components
- Real-time performance shortfalls

*Building something the customer doesn't want!*