Engineering Process

We need to understand the steps that take us from an idea to a product.

- What do we do?
- In what order do we do it?
- How do we know when we're finished each step?

Production process

Typical steps in production process for any engineering project:

1. Requirements analysis — what problem are we to solve?
2. Design — how will we solve the problem?
3. Analysis — will our proposed solution solve the problem?
4. Implementation/construction — build the solution.
5. Validation — did we solve the problem (and will the customer buy it)?
6. Production — make copies of the product.
7. (Maintenance — make sure the solution keeps running.)

Is Software Different?

- Requirements are less stable.
- Nature (physics) plays a smaller role.
  - Doesn't (necessarily) fail in predictable ways.
  - Interpolation and extrapolation are rarely valid.
- No obvious natural decomposition.
- Complexity can be very high.
- Production (not implementation) is trivial.

"Traditional" Software Process

1. Requirements — describe "what" not "how"

   Analysis/elicitation What does the client really want?

   Specification Precisely describe the behaviour of the system.

2. Design

   Architectural design — what classes/components/modules will make up the system?

   - What role does each play?
   - What are its relationships (interactions, sub-classing, associations) with other modules?
   - What is its interface (including behaviour)? Abstract: independent of the implementation.
   - How can it be tested?

   Module design — how will each module be implemented.

3. Implementation — write the code.

Traditional Software Process (cont'd)

4. Verification
   - Unit test — behaviour of each module.
   - Integration test — interaction between modules.
   - System test — behaviour of the whole system.
5. Maintenance — modify the system.

- Each phase results in a product (document, code), which is the input to the next phase.
- If a phase is carried out rigorously, its product can be verified against the products of the previous phase(s) (analysis).
- Validation (Is this what the customer wants?) can be carried out early and after each phase.

When does the "Traditional" Process work well?

- Large teams — Components can be reasonably worked on in parallel.
- Few unknowns
  - Requirements are stable.
  - Technology is well known.
  - The team has solved similar problems before.
- Implementation and design are clearly distinct tasks.
- Implementation is a significant portion of development.
- Components can be effectively tested independently.

Incremental/Unified/Spiral Software Development

- Software system is developed through a sequence of short, fixed length (e.g., four week) increments.
  - Each includes requirements analysis, design, implementation, testing
  - The endpoint of each increment is a **functioning system**.
  - You can objectively determine if you've met the goals of the increment by the **behaviour** of the system.
  - Usually all behaviour of previous increments is also exhibited in later increments — there is progress towards the final goal.
- Documents are updated as part of each increment so they remain accurate. (live documents)
- Having an executable system makes it possible to get early and frequent feedback (validation).
- Feedback allows the design to be adapted as requirements are better understood or change.

Major phases of Incremental Development

The whole developement period is divided into four major phases, each of which may contain one or more increments.

Inception	approximate vision, scope, vague estimates
Elaboration	refined vision, incremental implementation of core architecture, resolution of high risks
Construction	implementation of lower risk elements
Transition	beta test, deployment

How to plan for and execute incremental development

- Identify a set of distinct system behaviours (use cases).
- Assign each system behaviour to a particular increment.
- Candidates for early increments:
  - Highest priority (from the customer's point of view).
  - Highest risk (i.e., least well understood).
- Focus on what is needed for the current increment only (if it's not needed in this increment, *then don't do it*).
- Constantly track progress
  - If it looks like you won't meet a target date, drop behaviour rather than move the date.

When and Why to use Incremental Development

- Requirements are not well understood.
  - Developers get a better understanding by solving parts of the problem.
  - Users can give feedback on early increments.
- Technological uncertainty.
  - Early increments used to test if/how well technology is working to solve the problem.
- Schedule uncertainty.
  - Lack of experience makes it difficult to know how long it will take to do some parts.
  - Constant reflection makes adjusting the schedule easy.

When and Why (cont'd)

- Design/interface uncertainty.
  - Constant integration ensures that interfaces are understood.
- Important to deliver something quickly.
  - Time to market can be critical in some industries.
  - Gives management/customers confidence that progress is happening.
- Small team.
  - Parallel development of components isn't feasible.
  - Directs energy, avoids "thrashing."

Common mistakes

- An actual progress report from a previous student:

  "Increment 1 is 80% complete, increment 2 is 50% complete and increment 3 is 20% complete."

  - This student clearly missed the point.
  - Each increment should be finished before the next is started.
- Failure to refactor.
  - At each increment consider the design and how it can evolve into the next increment.
  - If it isn't right for the next increment fix it **now** before it becomes an albatross around your neck.

Mistakes (cont'd)

- Focusing on components rather than behaviour.
  - Increments are defined by the behaviour they deliver, not the development that goes into them.
  - Keep your eyes on the bottom line (behaviour), not the components.
- Over design.
  - There is a strong tendency to want to make an "elegant design" (e.g., more flexible, configurable . . .).
  - The best designs are the simple designs (KISS = Keep It Simple, Stupid).
- Failure to make a plan and communicate it.
  - Write down what behaviour is in each increment.
  - Make sure everybody has the plan.
  - Keep the plan current.

Software Qualities: Exernal

Qualities: What makes good software?
External qualities: What the user cares about.

- Usefulness — does it do what the user wants?
- Performance — speed, size.
- Correctness — does it meet its specification? (assumes precise specification)
- Reliability — does it do what the user wants most of the time?
- Robustness — does it respond well to error/failures of other systems?
- Usability — is it easy to use?
- Interoperability — does is conform to relevant standards/formats etc.?
- User documentation.

Software Qualities: Internal

What future developers will care about. Attributes of source code and documentation.

- Correctness — internal documentation consistent with itself and the code.
- Changeability — Can the most likely changes be made easily? Can other changes be made reasonably?
- Understandability — Is code & documentation understandable?
- Reusability — Can components be reused in other projects?

Software Qualities: Process

What managers care about.

- Cost — production and maintenance.
- Timeliness — When will it be ready?
- Traceability — Can the progress of the production be monitored?

Principles to achieve quality

Rigour/Formality

- Precision and exactness in descriptions and processes.
- Software is only correct when it is clear what it means to be correct.
- Should not inhibit creative process—sketch then design.
- *Formal* means that content and meaning (syntax and semantics) are governed by mathematical laws.

Separation of Concerns

- Time: concentrate on different aspects of system at different times.
- Qualities: e.g., verify correctness without considering efficiency.
- Views: Choose appropriate technique for examining different aspects of system (e.g., class diagram vs. interaction diagram).
- Components: modularity.

Quality Principles (cont'd)

Modularity (a.k.a., classes)

- Simplify a difficult task by breaking it into smaller components.
- Correct parts imply a correct whole.
- Keep each part simple.
- High *cohesion* — all elements of the same module are strongly related.
- Low *coupling* — elements in a module do not depend heavily on elements in other modules (except what is in the interface).

Abstraction

- Hide details to enable understanding (separation of concerns)
- The interface of a component is more simple than its implementation.

Quality Principles (cont'd)

Anticipation of change

- Change is inevitable and often predictable.
- Encapsulate anticipated changes in modules (classes).
- Unanticipated changes ruin modularity.
- Unanticipated changes will be much harder to get right.

Incrementality

- Identify useful subsets of system.
- Get something running early and keep it running.

Software Architectural Design

Goal is to define a set of components (classes, packages, modules) and their interelations/interactions.

- Information hiding — define components in terms of their *secret(s)*: information that the implementer of the component needs to know, but implementers of other components don't need to know.
- Design for change — choose the component secrets to be those that are most likely to change.
  - algorithms
  - data representation
  - underlying machine (hardware, OS)
  - peripheral devices/external interfaces
  - external environment (e.g., bussiness rules)
  - increments
  - product families

Relations On Components

Uses – if M_1 requires the presence of M_2 then M_1 *uses* M_2 (a.k.a. is a client of).

- not the same as *calls* or *associated with* (these are specializations of uses)
- if it is not a hierarchy then all components in cycle must be implemented together (strong coupling)
- sub-trees in hierarchy indicate possible increments/family members
- aim for low fan-out and high fan-in

is part of – (a.k.a. aggregation)

passes data to

specialization of — (a.k.a. inheritance)

Component Interface

The *interface* to *M* is the set of assumptions about *M* that can be made by programmers/designers of other components.

- public methods, their signatures and *externally observable* behaviour
- public data, constants etc.
- other assumptions (e.g., response time, side effects, limitations)

Try to make interface small, simple and effective.
Make sure that the interface is well documented. This is particularly important for aspects that aren't obvious from the syntax.
In incremental development pay particular attention to interfaces to components that will be enhanced in later increments.

Stepwise Refinement

In *stepwise refinement* problem is solved by defining a sequence of steps to solve the solution.
The steps are refined by adding details until the right level of detail is reached.

- Emphasis on the algorithm not the data (objects).
- Sub-problems are considered in isolation.
- Information hiding is not used (except w.r.t. algorithm steps).
- 'Top level' algorithm is assumed.
- Early commitment to control structure.

Object Oriented Design

Solution is developed as a model of the problem domain.

- Key nouns in the problem are realized by *objects*.
- Objects have *behaviour* and *attributes* that model their role in the solution.

Strong emphasis on information hiding and on particular relations on classes:

Generalization/Specialization (a.k.a., inheritance)

Association

Aggregation (is composed of)