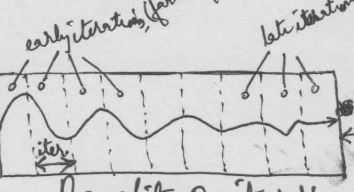
Unified Process (UP):

- iterative software development process
- for OO systems
- flexible & open for (Extreme Programming (XP), Scrum, etc.
- iterative (having iteration = a series of short, fixed-length mini projects) and incremental development
- iterative and evolutionary (because feedback and adaptation evolve the specifications and design) development

(Detailed refinement of UP) = RUP (Rational)



iterative and evolutionary development

iterative feedback and evolution leads towards the desired system.
The requirements and design instability lowers over time.

Benefits of iterative development:

- less project failure, better productivity, lower defect rates,
- early mitigation of high risks (technical, requirements, objectives, usability, ...)
- early visible progress
- early feedback, user engagement, and adaptation, leading to a refined system that more closely meets the real needs of the stakeholders
- managed complexity
- the learning, iteration by iteration

How long iteration? = (2-6 weeks)

* iterations are timeboxed (fixed in length), so date slippage is illegal. (de-scope - remove tasks or requirements from the iteration, and include them in a future iteration, rather than slip the completion date.)

The need for feedback and adaptation (key ingredients for success)

- feedback from early development, programmers trying to read specifications, and client demos to refine the requirements
- feedback from test and developers to refine the design or models
- feedback from the progress of the team tackling early features to refine the schedule and estimates
- feedback from the client and marketplace to re-prioritize the features to tackle in the next iteration.

* risk-driven iterative planning (goals of early iterations are chosen to identify and drive down the highest risks.)
* client-driven " " (" " " " " " to build visible features that the client cares most about)

* because not having a solid architecture is a common risk, so risk-driven iterative development includes more specifically the practice of architecture-centric iterative development (early iterations focus on building, testing, and stabilizing the core architecture.)

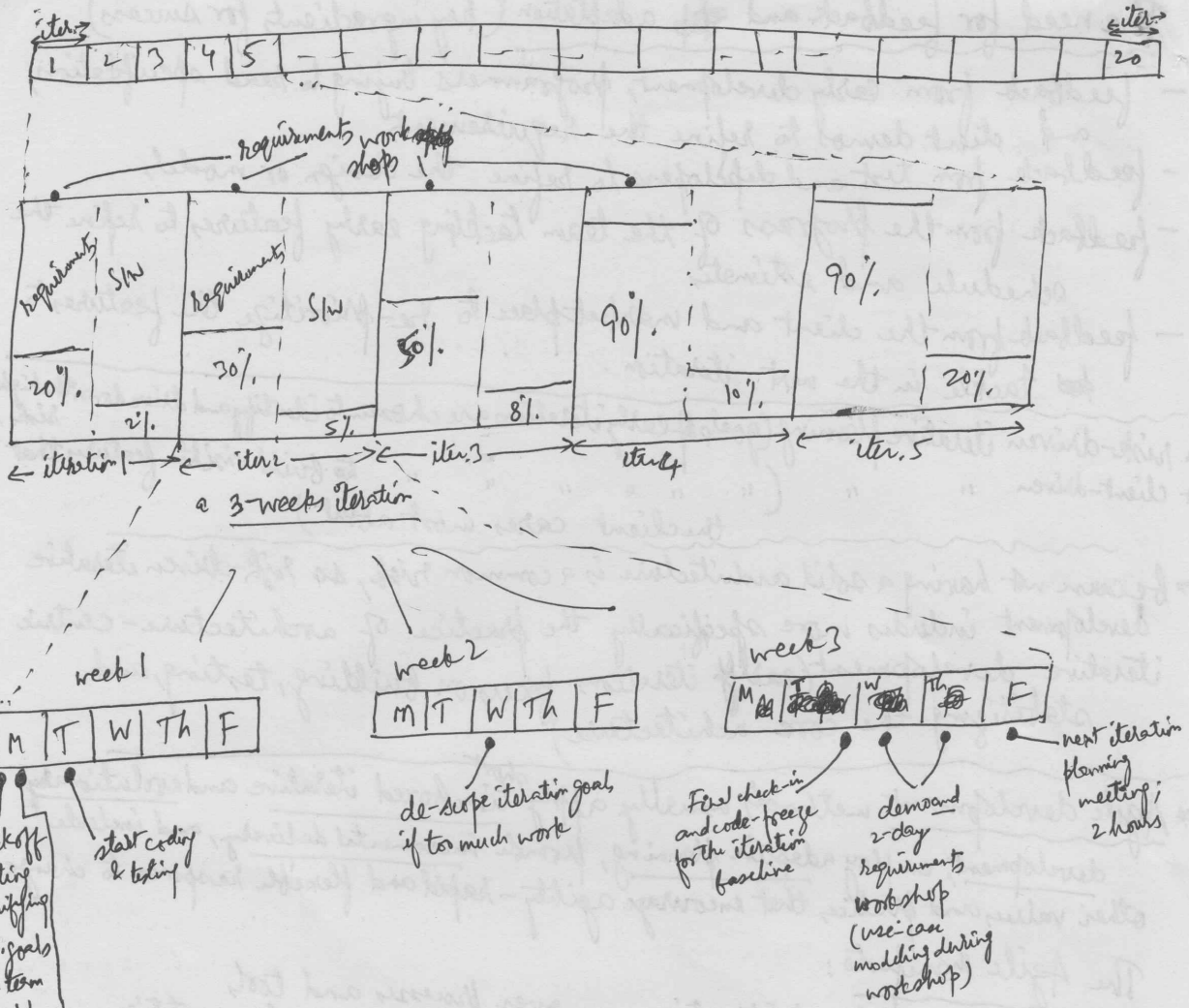
* Agile development methods usually apply short timeboxed iterative and evolutionary development, employ adaptive planning, promote incremental delivery, and include other values and practices that encourage agility – rapid and flexible response to change.

The Agile Manifesto:

individuals and interactions	over processes and tools
working S/W	over comprehensive documentation
Customer collaboration	over contract negotiation
Responding to change	over following a plan

The Agile Principles:

① highest priority is to satisfy customer through early and continuous delivery of S/W
② welcome changing requirement, anytime. Agile process harness change for the customer's..
③ deliver working S/W frequently, from a couple of weeks to a couple of months, with a preference to the shorter time scale.
④ Business people and developers must work together daily throughout the project.
⑤ Build projects around motivated individuals. Give them environment and support, trust them to get the job done.
⑥ face to face conversation within development team.
⑦ working software is primary measure of progress.
⑧ Agile processes promote sustainable development.
⑨ sponsors, developers, users should maintain constant pace indefinitely.
⑩ Continuous attention to technical excellence and good design enhances agility.
⑪ Simplicity
⑫ best architectures, requirements, and designs emerge from self-organizing teams.
⑬ At regular intervals, the team reflects on how to effective, then tunes and adjusts its behavior accordingly.


iter.
1
2
3
4
5
20
requirements
workshop
requirements
S/W
20%
2%
requirements
S/W
30%
5%
50%
8%
90%
10%
90%
20%
iteration 1
iter 2
iter 3
iter 4
iter. 5
a 3-weeks iteration
week 1
M
T
W
Th
F
week 2
M
T
W
Th
F
week 3
M
T
W
Th
F
Kickoff meeting clarifying iter. goals with team (1 hour)
team Agile modeling & design, UML whiteboard sketching (5 hours)
start coding & teting
de-scope iteration goals if too much work
Final check-in and code-freeze for the iteration baseline
demoand 2-day requirements workshop (use-case modeling during workshop)
next iteration planning meeting; 2 hours
most OOA/D and applying UML during this period

Agile Modeling: (the very act of modeling can and should provide a way to better understand the problem or solution space)
(to quickly explore alternatives, and the path to a good OO design).

- does not mean avoiding any modeling.
- to support understanding/communication
- simple design
- use simplest tools possible.
  (sketch UML on whiteboard, and capture diagrams with digital camera)
- model in pairs (or trios) at the whiteboard. Rotate the pen sketching across the members so that all participate.
- create models in parallel. (example, on one whiteboard start sketching a dynamic-view UML interaction diagram, and on another whiteboard, start sketching complementary static-view UML class diagram.)
- simple notation
- Know that all models will be inaccurate, except tested code model.
- developers themselves should do the OO design modeling, for themselves.

Agile UP: - prefer a small set of UP activities and artifacts. (simple)
- requirements and designs adaptively emerge through a series of iterations, based on feedback.

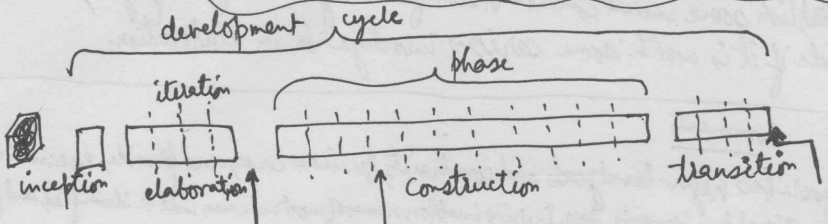
How this applies

- Apply UML with agile modeling practices.
- There is not a detailed plan for the entire project. There is a high-level plan (called the Phase plan) that estimates the project end date and other major milestones, but it does not detail the fine-grained steps to those milestones. A detailed plan (called the iteration plan) only plans with greater detail one iteration in advance. Detailed planning is done adaptively from iteration to iteration.
- a relatively small number of artifacts, and iterative development, in the spirit of an agile UP.

Best practices and key concepts in the UP:

- tackle high-risk, high-value issues in early iterations
- continuously engage users for evaluation, feedback, and requirements
- build a cohesive, core architecture in early iterations
- continuously verify quality; test early, often, and realistically
- apply use cases where appropriate
- do some visual modeling (with the UML)
- carefully manage requirements
- practice change request and configuration management

UP phases:
1- inception (approximate vision, business case, scope, vague estimates) = beginning
2- Elaboration (refine vision, iterative implementation of the core architecture, resolution of high risks, identification of most requirements and scope, more realistic estimates) = detailed development
3- Construction (iterative implementation of the remaining lower risk and easier elements, and preparation for deployment)
4- transition (beta tests, deployment)



milestone
an iteration end-point when some significant decision or evaluation occurs.

release
A stable executable subset of the final product. The end of each iteration is a minor release.

increment
the difference (delta) between the releases of 2 subsequent iterations.

At this point, the system is released for production use.

Schedule-oriented terms in the UP

UP disciplines
(UP set of activities (and related artifacts))

- Business modeling: (the domain model artifact (the general term for any work product: code, web graphics, database schema, text documents, diagrams, models, and so on), to visualize noteworthy concepts in the application domain
- Requirements - (use case model and supplementary specification artifacts to capture functional and non-functional requirements)
- Design (the design model artifact, to design the S/W objects)
- implementation
- test
- deployment
- configuration & change management
- project management
- environment

* all UP activities and artifacts (models, diagrams, documents, ...) are optional. you can choose whichever is more relevant.

* Note that although an iteration includes work in most disciplines, the relative efforts and emphasis change over time (requirement and design → implementation)

UP Development Case: (an artifact in the environment discipline)
the choice of practices and UP artifacts for a project may be written up in a short document.
(short document)

* all UP activities and artifacts (models, diagrams, documents, ---) are optional

Inception := is not the requirement phase.

- envision the product scope, vision, and business case.
- Do the stakeholders have basic agreement on the vision of the project, and is it worth investing in serious investigation?
- most requirements analysis occurs during the elaboration phase, in parallel with early production-quality programming and testing.

* the intent of inception is to establish some initial common vision for the objectives of the project, determine if it is feasible, and decide if it is worth some serious investigation in elaboration.

Artifacts in Inception	Comments
Vision and Business Case	= describes high-level goals and constraints, business case, and provides executive summary.
Use-Case Model	= describes functional requirements. During inception, names of most use cases will be identified, and perhaps 10% of use cases will be analyzed in detail.
Supplementary Specification	= describes other requirements, mostly non-functional. During inception, it is useful to have some idea of the key non-functional requirements that have will have a major impact on the architecture.
Glossary	= Key domain terminology, and data dictionary.
Risk List & Risk Management Plan	= describes the risks (business, technical, resource, schedule) and ideas for their mitigation or response.
Prototypes and proof-of-concepts	= to clarify the vision, and validate technical ideas.
Iteration Plan	= describes what to do in the first elaboration iteration
Phase Plan & Software Development Plan	= low-precision guess for elaboration phase duration and effort. Tools, people, education, and other resources.
Development Case	= description of customized UP steps and artifacts for this project. In the UP, one always customizes it for the project.

* these artifacts are only partially completed in this phase. They will be iteratively refined in subsequent iterations. Name capitalization implies an officially named UP artifact.
* artifacts are optional (choose to create only those that really add value for the project)
* the greatest value of modeling is to improve understanding, rather than to document reliable specifications (Agile modeling perspective)
* artifacts from previous projects can be used partially on later ones.
* all UP projects should organize artifacts the same way, with the same names (Risk management plan, Development Case, ---) to simplify finding reusable artifacts from prior projects.

evolutionary requirements: (managed requirement = systematic approach to finding, documenting, organizing, and tracking the changing requirements of a system)

In UP, requirements are categorized according to the FURPS+ model,

- Functional = features, capabilities, security
- Usability = human factors, help, documentation
- Reliability = frequency of failure, recoverability, predictability
- Performance = response times, throughput, accuracy, availability, resource usage
- Supportability = adaptability, maintainability, internationalization, configurability

The "+" in FURPS+ indicates ancillary and sub-factors, such as:

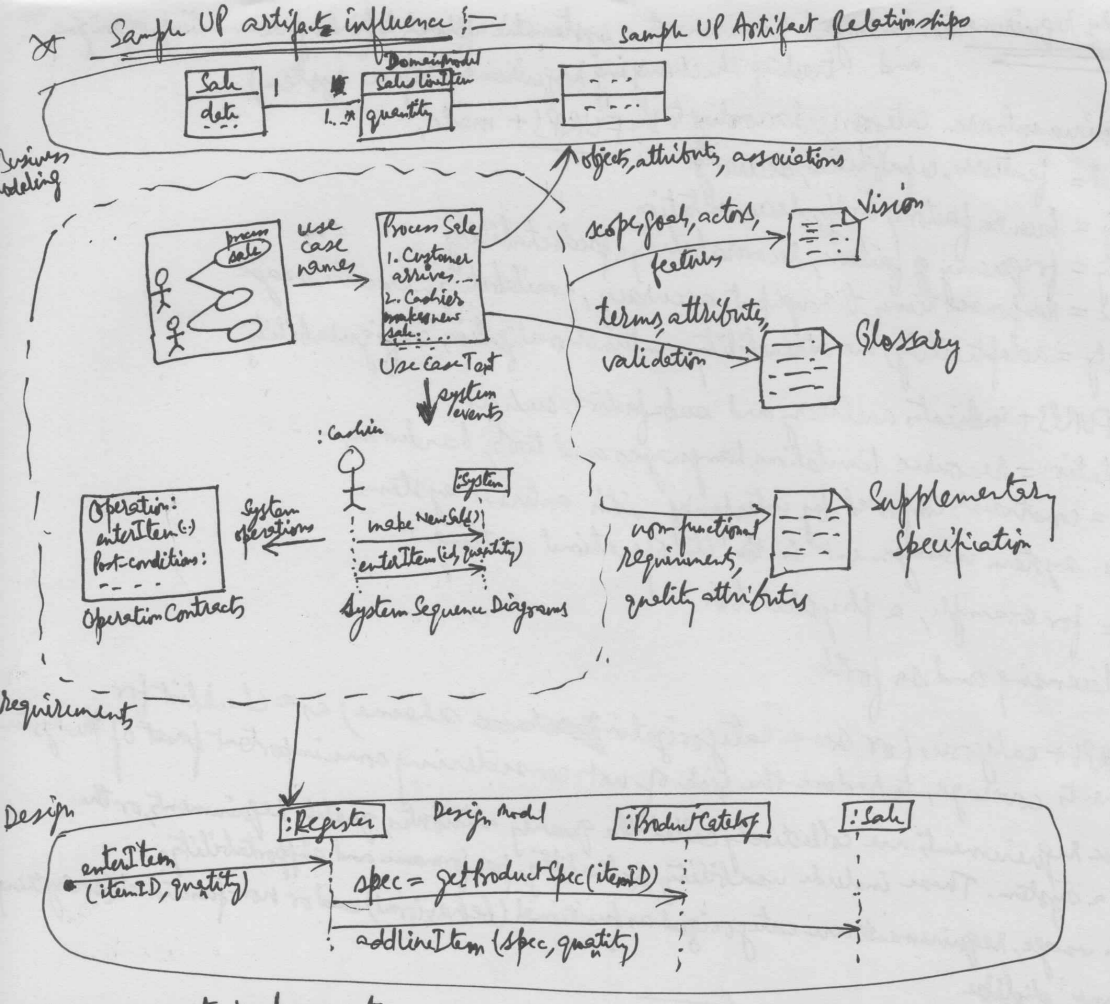
- Implementation = resource limitations, languages and tools, hardware, ...
- Interface = constraints imposed by interfacing with external systems.
- Operations = system management in its operational setting
- Packaging = for example, a physical box
- Legal = licensing and so forth

* use FURPS+ categories (or some categorization scheme) as a checklist for requirements coverage, to reduce the risk of not considering some important facet of the system.
* Some of these requirements are collectively called the quality attributes, quality requirements, or the "-ilities" of a system. These include usability, reliability, performance and supportability.
* In common usage, requirements are categorized as functional (behavioral) or non-functional (everything else);

---

requirements artifacts: → are (optional)

- use case model
- supplementary specification (all non-functional requirements, features)
- Glossary (noteworthy terms, data dictionary)
- Vision (high-level requirements summary, business case summary, project's big ideas short-document)
- Business rules / Domain rules (domain/business policies or requirements)



* use cases are text documents
* defer all conditional and branching statements to the Extensions section.

---

* In iterative development, we do not implement all the requirements at once
* **Elaboration phase** :- Build core architecture, resolve the high-risk elements, define most requirements, and estimate the overall schedule and resources.

key ideas and best practices in elaboration :-
- do short timeboxed risk-driven iterations
- start programming early
- adaptively design, implement, and test the core and risky parts of the architecture
- test early, often, realistically
- adapt based on feedback from tests, users, developers
- write most of the use cases and other requirements in detail, through a series of workshops, once per elaboration iteration

artifacts in elaboration :- domain model (business or real situation object or conceptual class), design model, S/W architecture document, data model, use-case storyboards, UI prototype

Domain model: = is a concept
— inspires the naming and definition of the software class

(use S/w class names in domain layer inspired from names in the domain model, with objects having domain-familiar information and responsibilities)

How to create a domain model:

1. to find conceptual classes: — reuse or modify existing models or
   — use a category list (making a list of candidate conceptual classes)
   — identify noun phrases
2. draw them as classes in UML class diagram
3. add associations and attributes

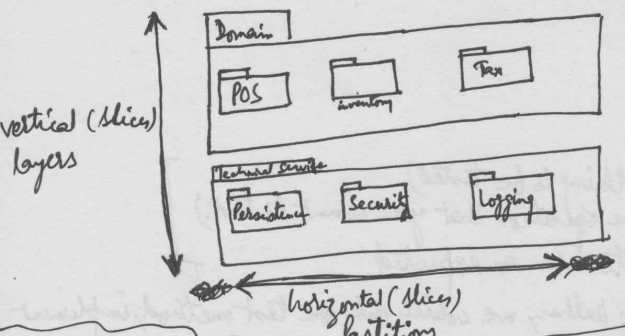
* if we don't think of some conceptual class X as a number or text in the real world, X is probably a conceptual class, not an attribute.

---

* logical architecture is the large-scale organization of software classes into packages (or namespaces), subsystems, and layers. (logical in the sense, that there is no decision about how these elements are deployed)
* layer is a very coarse-grained grouping of classes, packages, or subsystems that has cohesive responsibility for a major aspect of the system.
* Application logic and domain objects = S/w objects representing domain concepts (for example, a software class Sale) that fulfill application requirements,
* In information systems, relaxed layered architecture is used. (a higher layer calls upon several lower layers)

---

* organize the large-scale logical structures of a system into discrete layers of distinct, related responsibilities, with a clean, cohesive separation of concerns such that the "lower" layers are low-level and general services, and higher layers are more application specific.
* Collaboration and coupling is from higher to lower layers; (lower-to higher layer coupling is avoided)

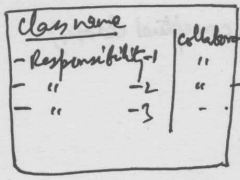


---

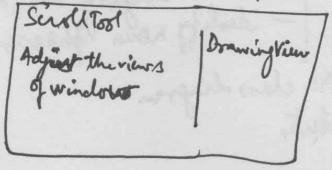
* Model-View Separation Principle = model (domain) objects should not have direct knowledge of views (UI) objects, at least as view objects.

Agile Modeling: - reduce drawing overhead
= model to understand and communicate
= model with others
= creating several models in parallel. (e.g. static and dynamic models, ...)

* **CRC Cards** = (Class Responsibility Collaboration) cards
- paper index cards on which one writes (class name, responsibilities, collaborators of class)



e.g.:



Low Coupling: low (dependency of one element to other elements / or connection / or knowledge)

↑ because a change in one element may affects other elements

* assign responsibilities for low coupling

High cohesion (highly related functional responsibility of element / elements/operations)

(cohesion = "measures how functionally related the operations of a software element are, and also measures how much work a software element is doing.)

* assign responsibilities for high cohesion

Test-Driven Development (Test-first development) = test code is written first (before) the class to be tested.

framework tools (NUnit → .Net / JUnit → Java)

* write unit testing method in a Test class

How to:
- create the fixture (the thing to be tested).
- do something to it (some operation that you want to test.)
- evaluate that the results are as expected

* we do not write all the unit tests for the class first; rather, we write only one test method, implement the solution in the class to make it pass, and then repeat.

Refactoring: - a structured, disciplined method to rewrite or restructure existing code without changing its external behavior, applying small transformation steps combined with re-executing tests each step.
(Behavior preserving transformations)

* Continuously refactoring code is XP practice, and applicable to all iterative methods
* Refactoring makes code: short, tight, clear, without duplication, make long methods shorter, removes code smell

Refactoring names (happy):=
① Extract Method :-
② Extract Content -
③ Introduce Explaining Variable
④ Replace Constructor call with Factory Method
...
..

How to plan an iteration:

① decide the length of iteration. (shorter is better)
② convene iteration planning meeting. (done at the end of current iteration) (repeatedly every)
③ list of potential goals (new features or use cases, defects, ...) from customer and Architect
{business goals, technical goals}
④ each member is asked for their individual resource budget for iteration
⑤ describe every goals in some detail and questions are resolved. Brainstorm for more detailed tasks for the goals with some vague estimates.
summation of all task estimates into a running total.

* in agile project management, (developers are actively involved in planning and estimating process as well).