

TPO – Colloquium I.

80/20 Rule

The Pareto principle (also known as the 80–20 rule, the law of the vital few, and the principle of factor sparsity) states that, for many events, roughly 80% of the effects come from 20% of the causes. For example, [Microsoft](#) noted that by fixing the top 20% of the most reported bugs, 80% of the errors and crashes would be eliminated (it applies to bugs too, not just features).

Cohesion (of Modules)

In [computer programming](#), cohesion refers to the *degree to which the elements of a [module](#) belong together*.^[1] Thus, it is a measure of how strongly-related each piece of functionality expressed by the source code of a software [module](#) is. Cohesion is a measure of how strongly-related or focused the responsibilities of a single module are.

Types of Cohesion

Coincidental cohesion (worst), Logical, Temporal, Procedural, Communicational, Sequential, Functional cohesion (best).

Coupling (of Modules)

In [software engineering](#), coupling or dependency is the degree to which each [program module](#) relies on each one of the other modules. Coupling is usually contrasted with [cohesion](#). Low coupling often correlates with high cohesion, and vice versa.

Cohesion vs. Coupling

Together they talk about the quality a module should have. Coupling talks about the interdependencies between the various modules while cohesion describes how related functions within a module are. Low cohesion implies that a given module performs tasks which are not very related to each other and hence can create problems as the module becomes large.

Alpha Testing

Alpha testing is simulated or actual operational testing by potential users/customers or an independent test team at the developers' site. Alpha testing is often employed for off-the-shelf software as a form of internal acceptance testing, before the software goes to beta testing.

Beta Testing

Beta testing comes after alpha testing and can be considered a form of external [user acceptance testing](#). Versions of the software, known as [beta versions](#), are released to a limited audience outside of the programming team. The software is released to groups of people so that further testing can ensure the product has few faults or [bugs](#). Sometimes, beta versions are made available to the open public to increase the [feedback](#) field to a maximal number of future users.

Reference Model

A reference model in [systems](#), [enterprise](#), and [software engineering](#) is an abstract framework or domain-specific ontology consisting of an interlinked set of clearly defined concepts produced by an expert or body of experts in order to encourage clear communication. A reference model can represent the component parts of any consistent idea, from business functions to system components, as long as it represents a complete set. This frame of reference can then be used to communicate ideas clearly among members of the same community.

Reference models are often illustrated as a set of concepts with some indication of the relationships between the concepts.

White-box Testing

White-box testing (also known as clear box testing, glass box testing, transparent box testing, and structural testing) is a method of testing [software](#) that tests internal structures or workings of an application, as opposed to its functionality (i.e. [black-box testing](#)). In white-box testing an internal perspective of the system, as well as programming skills, are used to design test cases. The tester chooses inputs to exercise paths through the code and determine the appropriate outputs. This is analogous to testing nodes in a circuit, e.g. [in-circuit testing](#) (ICT).

While white-box testing can be applied at the [unit](#), [integration](#) and [system](#) levels of the [software testing](#) process, it is usually done at the unit level.

Black-box Testing

Black-box testing is a method of [software testing](#) that tests the functionality of an application as opposed to its internal structures or workings. This method of test can be applied to all levels of software testing: [unit](#), [integration](#), [system](#) and [acceptance](#). It typically comprises most if not all testing at higher levels, but can also dominate [unit testing](#) as well.

Top-down and Bottom-up Design

Top-down and bottom-up are both strategies of [information processing](#) and knowledge ordering, used in a variety of fields including software, humanistic and scientific theories (see [systemics](#)), and management and organization. In practice, they can be seen as a style of thinking and teaching.

Top-down Design

A top-down approach (also known as stepwise design or [deductive reasoning](#),^[1] and in many cases used as a synonym of [analysis](#) or [decomposition](#)) is essentially the breaking down of a system to gain insight into its compositional sub-systems. In a top-down approach an overview of the system is formulated, specifying but not detailing any first-level subsystems. Each subsystem is then refined in yet greater detail, sometimes in many additional subsystem levels, until the entire specification is reduced to base elements. A top-down model is often specified with the assistance of "black boxes", these make it easier to manipulate. However, black boxes may fail to elucidate elementary mechanisms or be detailed enough to realistically validate the model. Top down approach starts with the big picture. It breaks down from there into smaller segments.

Bottom-up Design

A bottom-up approach (also known as [inductive reasoning](#),^[1] and in many cases used as a synonym of [synthesis](#)) is the piecing together of systems to give rise to grander systems, thus making the original systems sub-systems of the emergent system. Bottom-up processing is a type of [information processing](#) based on incoming data from the environment to form a [perception](#). Information enters the eyes in one direction (input), and is then turned into an image by the brain that can be interpreted and recognized as a perception (output). In a bottom-up approach the individual base elements of the system are first specified in great detail. These elements are then linked together to form larger subsystems, which then in turn are linked, sometimes in many levels, until a complete top-level system is formed. This strategy

often resembles a "seed" model, whereby the beginnings are small but eventually grow in complexity and completeness. However, "organic strategies" may result in a tangle of elements and subsystems, developed in isolation and subject to local optimization as opposed to meeting a global purpose.

P2P Architecture (Style, Model)

A pure P2P network does not have the notion of [clients](#) or servers but only equal [peer](#) nodes that simultaneously function as both "clients" and "servers" to the other nodes on the network. This model of network arrangement differs from the [client-server](#) model where communication is usually to and from a central server. A typical example of a file transfer that does not use the P2P model is the [File Transfer Protocol](#) (FTP) service in which the client and server programs are distinct: the clients initiate the transfer, and the servers satisfy these requests.

Client-server Architecture (Style, Model)

The client/server model is a [computing](#) model that acts as a [distributed application](#) which partitions tasks or workloads between the providers of a resource or service, called [servers](#), and service requesters, called [clients](#).^[1] Often clients and servers communicate over a [computer network](#) on separate hardware, but both client and server may reside in the same system. A server machine is a host that is running one or more server programs which share their resources with clients. A client does not share any of its resources, but requests a server's content or service function. Clients therefore initiate communication sessions with servers which await incoming requests.

Multitier (N-tier) Architecture (Style, Model)

In [software engineering](#), multi-tier architecture (often referred to as n-tier architecture) is a [client-server architecture](#) in which presentation, application processing, and data management functions are logically separated. For example, an application that uses [middleware](#) to service data requests between a user and a [database](#) employs multi-tier architecture. The most widespread use of multi-tier architecture is the three-tier architecture.

N-tier application architecture provides a model by which developers can create flexible and reusable applications. By segregating an application into tiers, developers acquire the option of modifying or adding a specific layer, instead of reworking the entire application. Three-tier architectures typically comprise a *presentation* tier, a *business* or *data access* tier, and a *data* tier.

Model-View-Controller (MVC) Architecture (Style, Model)

Model–View–Controller (MVC) is an architecture that separates the representation of information from the user's interaction with it.^{[1][2]} The *model* consists of application data and business rules, and the *controller* mediates input, converting it to commands for the model or view.^[3] A *view* can be any output representation of data, such as a chart or a diagram. Multiple views of the same data are possible, such as a pie chart for management and a tabular view for accountants. The central idea behind MVC is [code reusability](#) and [separation of concerns](#).^[4]

Pipeline Architecture (Style, Model)

In [software engineering](#), a pipeline consists of a chain of processing elements ([processes](#), [threads](#), [coroutines](#), etc.), arranged so that the output of each element is the input of the next. Usually some amount of [buffering](#) is provided between consecutive elements. The information that flows in these pipelines is often a [stream](#) of [records](#), [bytes](#) or [bits](#).

Repository Architecture (Style, Model)

The repository style fits situations where the main issue is to manage a richly structured body of information. In the library example in the section on Requirements Engineering, the data concerns such things as the stock of available books and the collection of members of the library. The data is persistent and it is important that it always reflects the true state of affairs. A natural approach to this problem is to devise database schemas for the various types of data in the application and store the data in one or more databases. The functionality of the system is incorporated in a number of, relatively independent, computational elements. The result is a repository architectural style.

Functional Requirements


In [software engineering](#), a functional requirement defines a function of a [software system](#) or its component. A function is described as a set of inputs, the behavior, and outputs (see also [software](#)). Functional requirements may be calculations, technical details, data manipulation and processing and other specific functionality that define *what* a system is supposed to accomplish. Behavioral requirements describing all the cases where the system uses the functional requirements are captured in [use cases](#). Functional requirements are supported by [non-functional requirements](#) (also known as *quality requirements*), which impose constraints on the design or implementation (such as performance requirements, security, or reliability). Generally, functional requirements are expressed in the form "system must do <requirement>", while non-functional requirements are "system shall be <requirement>". The

plan for implementing *functional* requirements is detailed in the system *design*. The plan for implementing *non-functional* requirements is detailed in the system *architecture*.

Non-functional Requirements

In [systems engineering](#) and [requirements engineering](#), a non-functional requirement is a [requirement](#) that specifies criteria that can be used to judge the operation of a system, rather than specific behaviors. This should be contrasted with [functional requirements](#) that define specific behavior or functions. The plan for implementing *functional* requirements is detailed in the [system design](#). The plan for implementing *non-functional* requirements is detailed in the [system architecture](#).

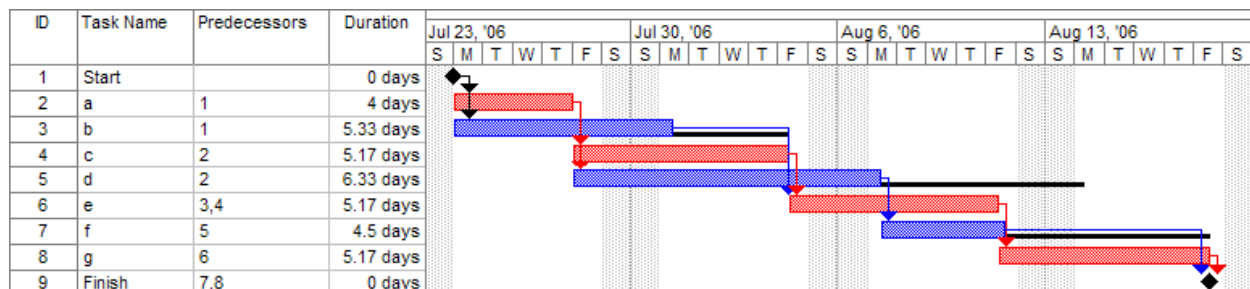
Delphi Method

The Delphi method ( [/'dɛlfai/ DEL-fy](#)) is a structured communication technique, originally developed as a systematic, interactive [forecasting](#) method which relies on a panel of experts.^[1]

In the standard version, the experts answer questionnaires in two or more rounds. After each round, a facilitator provides an anonymous summary of the experts' forecasts from the previous round as well as the reasons they provided for their judgments. Thus, experts are encouraged to revise their earlier answers in light of the replies of other members of their panel. It is believed that during this process the range of the answers will decrease and the group will converge towards the "correct" answer. Finally, the process is stopped after a pre-defined stop criterion (e.g. number of rounds, achievement of consensus, stability of results) and the [mean](#) or [median](#) scores of the final rounds determine the results.^[2]

Gantt Chart

A Gantt chart is a type of [bar chart](#), developed by [Henry Gantt](#), that illustrates a [project schedule](#). Gantt charts illustrate the start and finish dates of the terminal elements and summary elements of a [project](#). Terminal elements and summary elements comprise the [work breakdown structure](#) of the project.

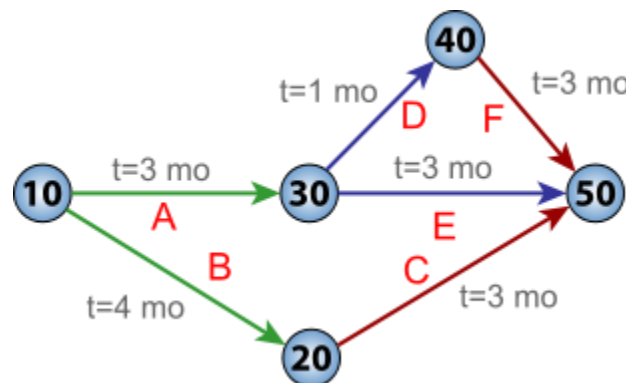


Critical Path (also PERT – Program Evaluation and Review Technique)

The critical path method (CPM) is an [algorithm](#) for scheduling a set of project activities.^[1] It is an important tool for effective [project management](#). The essential technique for using CPM ^[6] ^[7] is to construct a model of the project that includes the following:

1. A list of all activities required to complete the project (typically categorized within a [work breakdown structure](#)),
2. The time (duration) that each activity will take to completion, and
3. The [dependencies](#) between the activities.

Using these values, CPM calculates the longest path of planned activities to the end of the project, and the earliest and latest that each activity can start and finish without making the project longer. This process determines which activities are "critical" (i.e., on the longest path) and which have "total float" (i.e., can be delayed without making the project longer).



The Program (or Project) Evaluation and Review Technique, commonly abbreviated PERT, is a statistical tool, used in [project management](#), that is designed to analyze and represent the tasks involved in completing a given [project](#).

Least Slack Time Scheduling

Least Slack Time (LST) scheduling is a [scheduling algorithm](#). It assigns priority based on the *slack time* of a process. Slack time is the amount of time left after a job if the job was started now.

Project Plan (Definition)

A project plan, according to the [Project Management Body of Knowledge](#), is: "...a formal, approved document used to guide both *project execution* and *project control*. The primary uses of the project plan are to document planning assumptions and decisions, facilitate

communication among *stakeholders*, and document approved scope, cost, and schedule *baselines*. A project plan may be summarized or detailed."^[1]

Project Plan (Purpose)

The objective of a project plan is to define the approach to be used by the [Project team](#) to deliver the intended project management [scope](#) of the project.

At a minimum, a project plan answers basic questions about the project:

- **Why?** - What is the problem or value proposition addressed by the project? Why is it being sponsored?
- **What?** - What is the work that will be performed on the project? What are the major products/[deliverables](#)?
- **Who?** - Who will be involved and what will be their [responsibilities](#) within the project? How will they be organized?
- **When?** - What is the project timeline and when will particularly meaningful points, referred to as [milestones](#), be complete?

Project Planning

Project planning is part of [project management](#), which relates to the use of [schedules](#) such as [Gantt charts](#) to plan and subsequently report progress within the project environment.^[1]

Initially, the [project scope](#) is defined and the appropriate methods for completing the project are determined. Following this step, the [durations](#) for the various [tasks](#) necessary to complete the [work](#) are listed and grouped into a [work breakdown structure](#). The logical [dependencies](#) between tasks are defined using an [activity network diagram](#) that enables identification of the [critical path](#). [Float](#) or slack time in the schedule can be calculated using [project management software](#).^[2] Then the necessary [resources](#) can be [estimated](#) and [costs](#) for each activity can be allocated to each resource, giving the total project cost. At this stage, the [project plan](#) may be optimized to achieve the appropriate balance between [resource usage](#) and project duration to comply with the project objectives. Once established and agreed, the plan becomes what is known as the baseline. Progress will be measured against the baseline throughout the life of the project. Analyzing progress compared to the baseline is known as [earned value management](#).^[3]

The inputs of the project planning phase include the [project charter](#) and the concept proposal. The outputs of the project planning phase include the project requirements, the project schedule, and the [project management plan](#).^[4]

Total Slack

The total slack is equal to the allowable delay of an activity i without causing a violation of the project deadline and can be calculated as

$$LF_i - EF_i = LS_i - ES_i$$

with ES_i and EF_i the earliest start and latest finish of each activity i using forward calculations (i.e. obtained from the ESS). Similarly, the LS_i and LF_i are used to denote the latest start and latest finish of activity i using backward calculations (i.e. obtained from the LSS).

The total slack is given in table 2 for a project deadline of 15 and 19. Note that in case $D(ESS) = D(LSS)$, the activities with a total slack value equal to zero belong to the critical path.

Safety Slack

The safety slack is equal to the allowable delay of an activity i when all predecessors finish as late as possible and can be calculated as

$$LS_i - \max_{\text{all predecessors } j}(LF_j)$$

with LF_j the latest finish time of an activity j that precedes activity i . The safety slack values are given in table 2 for the two project deadlines.

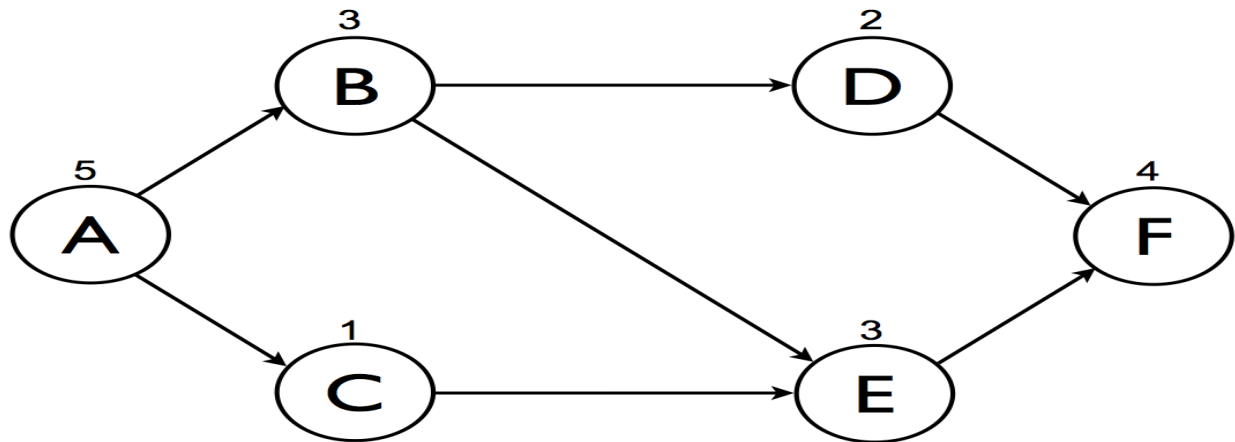
Free Slack

The free slack of an activity i is equal to the allowable delay of an activity that has no effect on the earliest start ES of a successor activity and can be calculated as

$$\min_{\text{all successors } j}(ES_j) - EF_i$$

with ES_j the earliest start time of an activity j that succeeds activity i . The free slack values are given in table 2 for the two project deadlines.

Project Network Diagram Sample



Software Development (Approach)

There are several different approaches to software development, much like the various views of political parties toward governing a country. Some take a more structured, engineering-based approach to developing business solutions, whereas others may take a more incremental approach, where software evolves as it is developed piece-by-piece. Most methodologies share some combination of the following stages of software development:

- Analysing the problem
- Market research
- Gathering requirements for the proposed business solution
- Devising a plan or design for the software-based solution
- Implementation (coding) of the software
- Testing the software
- Deployment
- Maintenance and bug fixing

Software Development (Models)

- Traditional models
 - I. Cascade or sequential model (Waterfall model)
 - II. V-Model
 - III. Incremental model
- Evolutionary models
 - I. Iterative model
 - II. Prototype model

- III. Spiral model
- Agile methods
 - I. Extreme programming (XP)
 - II. Scrum
- Combined models

Waterfall Model

The waterfall model is a [sequential design](#) process, often used in [software development processes](#), in which progress is seen as flowing steadily downwards (like a [waterfall](#)) through the phases of Conception, Initiation, [Analysis](#), [Design](#), Construction, [Testing](#), [Production/Implementation](#), and [Maintenance](#).

The waterfall development model originates in the [manufacturing](#) and [construction](#) industries; highly structured physical environments in which after-the-fact changes are prohibitively costly, if not impossible. Since no formal software development methodologies existed at the time, this hardware-oriented model was simply adapted for software development.^[1]

V-Model

The V-model^[2] represents a [software development process](#) which may be considered an extension of the [waterfall model](#). Instead of moving down in a linear way, the process steps are bent upwards after the [coding](#) phase, to form the typical V shape. The V-Model demonstrates the relationships between each phase of the development life cycle and its associated phase of [testing](#). The horizontal and vertical axes represents time or project completeness (left-to-right) and level of abstraction (coarsest-grain abstraction uppermost), respectively.

Incremental Model

The incremental build model is a method of [software development](#) where the model is [designed](#), implemented and [tested](#) incrementally (a little more is added each time) until the product is finished. It involves both development and maintenance. The product is defined as finished when it satisfies all of its requirements. This model combines the elements of the [waterfall model](#) with the iterative philosophy of [prototyping](#).

Iterative Model

Iterative design is a [design](#) methodology based on a cyclic process of [prototyping](#), testing, analyzing, and refining a product or process. Based on the results of testing the most recent

iteration of a design, changes and refinements are made. This process is intended to ultimately improve the quality and functionality of a design. In iterative design, interaction with the designed system is used as a form of research for informing and evolving a project, as successive versions, or iterations of a design are implemented.

Prototype Model

Software prototyping, refers to the activity of creating [prototypes](#) of software applications, i.e., incomplete versions of the [software program](#) being developed. It is an activity that can occur in [software development](#) and is comparable to [prototyping](#) as known from other fields, such as [mechanical engineering](#) or [manufacturing](#).

A prototype typically simulates only a few aspects of, and may be completely different from, the final product.

Spiral Model

The spiral model is a [software development process](#) combining elements of both [design](#) and [prototyping](#)-in-stages, in an effort to combine advantages of [top-down and bottom-up](#) concepts. Also known as the spiral lifecycle model (or spiral development), it is a systems development method (SDM) used in [information technology](#) (IT). This model of development combines the features of the prototyping and the [waterfall model](#). The spiral model is intended for large, expensive and complicated projects.

Agile Software Development

Agile software development is a group of [software development methods](#) based on [iterative and incremental development](#), where requirements and solutions evolve through collaboration between [self-organizing, cross-functional teams](#). It promotes adaptive planning, evolutionary development and delivery, a [time-boxed](#) iterative approach, and encourages rapid and flexible response to change. It is a conceptual framework that promotes foreseen interactions throughout the development cycle.

4 Main Principles of Agile Development

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

Extreme Programming (XP)

Extreme Programming (XP) is a [software development methodology](#) which is intended to improve software quality and responsiveness to changing customer requirements. As a type of [agile software development](#),^{[1][2][3]} it advocates frequent "releases" in short development cycles ([timeboxing](#)), which is intended to improve productivity and introduce checkpoints where new customer requirements can be adopted.

Other elements of Extreme Programming include: programming [in pairs](#) or doing extensive [code review](#), [unit testing](#) of all code, avoiding programming of features until they are actually needed, a flat management structure, simplicity and clarity in code, expecting changes in the customer's requirements as time passes and the problem is better understood, and frequent communication with the customer and among programmers.^{[2][3][4]} The method takes its name from the idea that the beneficial elements of traditional software engineering practices are taken to "extreme" levels, on the theory that if a little is good, more is better

Scrum Development

Scrum is an iterative and incremental [agile software development](#) method for managing software projects and product or application development. Scrum has not only reinforced the interest in [project management](#),^[citation needed] but also challenged the conventional ideas about such management. Scrum focuses on project management institutions where it is difficult to plan ahead. Mechanisms of *empirical process control*, where [feedback loops](#) that constitute the core management technique are used as opposed to traditional [command-and-control](#) oriented management.^[citation needed] It represents a radically new approach for planning and managing projects, bringing [decision-making](#) authority to the level of operation properties and certainties.^[1]

Sprint (Scrum)

An iteration of work during which an increment of product functionality is implemented. By the book, an iteration lasts 30 days. This is longer than in other agile methods to take into account the fact that a functional increment of product must be produced each sprint.

Sprint Burndown Chart (Scrum)

A sprint burndown chart (or "sprint burndown graph") depicts the total task hours remaining per day. This shows you where your team stands regarding completing the tasks that comprise

the product backlog items that achieve the goals of the sprint. The X-axis represents days in the sprint, while the Y-axis is effort remaining (usually in ideal engineering hours).

Velocity (Scrum)

In Scrum, velocity is how much product backlog effort a team can handle in one sprint. This can be estimated by viewing previous sprints, assuming the team composition and sprint duration are kept constant. It can also be established on a sprint-by-sprint basis, using commitment-based planning.

Product Backlog (Scrum)

The product backlog is the requirements for a system, expressed as a prioritized list of product backlog items. These included both functional and non-functional customer requirements, as well as technical team-generated requirements. While there are multiple inputs to the product backlog, it is the sole responsibility of the product owner to prioritize the product backlog.

Sprint Backlog (Scrum)

A prioritized list of tasks to be completed during the sprint.

User Story (Scrum)

A feature that is added to the backlog is commonly referred to as a story and has a specific suggested structure. The structure of a story is: "As a <user type> I want to <do some action> so that <desired result>" This is done so that the development team can identify the user, action and required result in a request and is a simple way of writing requests that anyone can understand. Example: As a wiki user I want a tools menu on the edit screen so that I can easily apply font formatting.

Planning Poker (Scrum)

In the Sprint Planning Meeting, the team sits down to estimate its effort for the stories in the backlog. The Product Owner needs these estimates, so that he or she is empowered to effectively prioritize items in the backlog and, as a result, forecast releases based on the team's velocity.^[21]

Feature-driven Development (FDD)

Feature-driven development (FDD) is an [iterative and incremental software development process](#). It is one of a number of [Agile methods](#) for developing [software](#) and forms part of the [Agile Alliance](#). FDD blends a number of industry-recognized [best practices](#) into a cohesive whole. These practices are all driven from a client-valued functionality ([feature](#)) perspective. Its main purpose is to deliver tangible, working software repeatedly in a timely manner.

Test-driven Development (TDD)

Test-driven development (TDD) is a [software development process](#) that relies on the repetition of a very short development cycle: first the developer writes an (initially failing) automated [test case](#) that defines a desired improvement or new function, then produces the minimum amount of code to pass that test and finally [refactors](#) the new code to acceptable standards.

Behaviour-driven Development (BDD)

In [software engineering](#), behavior-driven development (abbreviated BDD) is a [software development process](#) based on [test-driven development \(TDD\)](#).^{[1][2]} Behavior-driven development combines the general techniques and principles of TDD with ideas from [domain-driven design](#) and [object-oriented analysis and design](#) to provide [software developers](#) and [business analysts](#) with shared tools and a shared process to collaborate on software development.^{[1][3]}

Rapid Application Development (RAD)

Rapid application development (R.A.D) is a [software development methodology](#) that uses minimal planning in favor of rapid prototyping. The "planning" of software developed using RAD is interleaved with writing the software itself. The lack of extensive pre-planning generally allows software to be written much faster, and makes it easier to change requirements.

Structured Systems Analysis and Design Method (SSADM)

SSADM is a [waterfall method](#) for the analysis and design of [information systems](#). SSADM can be thought to represent a pinnacle of the rigorous document-led approach to system design, and contrasts with more contemporary [agile](#) methods such as [DSDM](#) or [Scrum](#).

The three most important techniques that are used in SSADM are:

Logical data modeling

This is the process of identifying, modeling and documenting the data requirements of the system being designed. The data are separated into entities (things about which a business needs to record information) and relationships (the associations between the entities).

Data Flow Modeling

This is the process of identifying, modeling and documenting how data moves around an information system. Data Flow Modeling examines processes (activities that transform data from one form to another), data stores (the holding areas for data), external entities (what sends data into a system or receives data from a system), and data flows (routes by which data can flow).

Entity Behavior Modeling

This is the process of identifying, modeling and documenting the events that affect each entity and the sequence in which these events occur.

Benefits of Software Methodologies

Higher quality final product

- better capture the customer's requirements
- better documentation
- better compatibility with other systems

Higher quality development process

- constant insight into the development process
- connection with models for quality assurance

Standardization of business

- common knowledge base
- easier scheduling people projects

Easier to work with staff, staff training

- easier integration of newcomers

Better impression on the client