**What is Software?**

To the Hardware: A collection of instructions to be executed by a computer/processor

To the Developer: A collection of human-readable statements (in a language) that can be converted to a collection of instructions to be executed by a computer/processor

**The Organization of Software**

Program: Any piece of software that can "run on its own" (including utilities, tools, scripts, apps, etc...)

Sub-Program: A collection of instructions/statements that implement an algorithm for accomplishing a specific task or tasks

Library: A group of related sub-programs for accomplishing a specific collection of (usually related) tasks

**From Software to Software Product:**

What People (Other than Us) Care About:

Having everything they need to solve one or more problems or achieve one or more goals (i.e., a complete means to one or more ends)

Definition:

A software product is one or more programs, sub-programs, or libraries, along with the data and supporting materials and services, that a client can use to solve problems or achieve goals

**Kinds of Software Products -**

Bespoke: Software products that are developed (usually under contract) for a specific client

Generic: Software products that are developed (usually speculatively) and then sold to market clients (either a mass market or a niche market)

**Software Engineering:**

Defined: The application of (scientific) theories, methods and tools to the specification, design, creation, verification/validation, deployment, operation, and maintenance of software products

Scope: From specification to maintenance, involves technical and managerial concerns, involves theories/methods from psychology, mathematics/statistics, computer science, and management (of people and resources), and consists of science and art

**Software Engineering vs. Computer Science/Engineering:**

Computer Engineering:

The application of theories (often from physics) to the creation of computational devices

Usually thought of as a subset of electrical and electronic engineering

Computer Science:

The theories and methods that underlie computation and the use of computational devices

Systems

System: A set of entities, their attributes, and the relationships between them.

Entities: The components of the system that are known/observed.

Attributes: The properties of the entities (i.e., the external manifestations of the way the objects are

known or observed).

Relationships: The bonds that link entities and attributes

**Abstraction:**

Abstract (verb): To extract what is important

Two Observations:

An object/concept can be conceptualized in an enormous number of ways

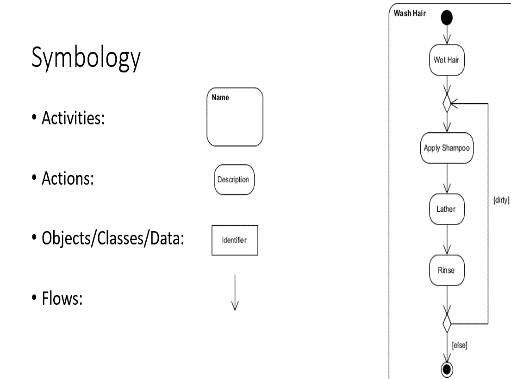
The level of detail (which is the inverse of the level of abstraction) is a defining feature of the system

Example: A Car

Low Detail - A source of emissions

Medium Detail - A contributor to traffic congestion at an intersection

High Detail - A mechanical motive device

System State: 

An unvarying (over space, and/or time, and/or some other dimension) description or summary of that system

Specification: All the attributes (at an appropriate level of abstraction) of the entities that comprise the system

**Modeling**

A model is a description of a system (that is created to help understand the system)

Modeling Methods

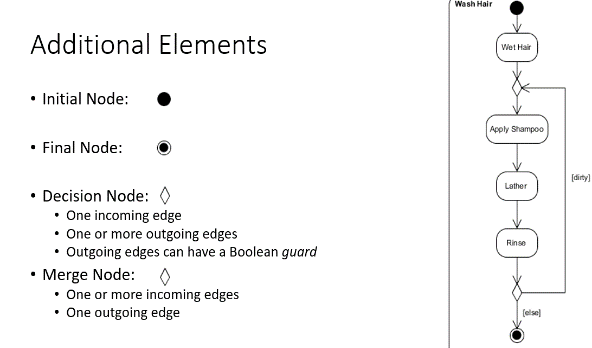
Physical Models: Paper, clay, plastic, wood, metal, etc...

Conceptual Models: Mathematical, computational/numerical, diagrammatic/visual/graphical, textual, etc...

**Model Types**

Static: Models of the aspects of the system that do not change (sometimes called structural)

Dynamic: Models of how the system changes over time (sometimes called behavioral)

**Importance of Modeling**

Complex systems are hard to understand, models can be used to make them easier to understand

An Example:

You can probably build a doghouse without a model, but a house would be very difficult to build without

a model and a skyscraper would be impossible to build without a model

**Modeling and Software**

A Common Problem:

Start with a doghouse

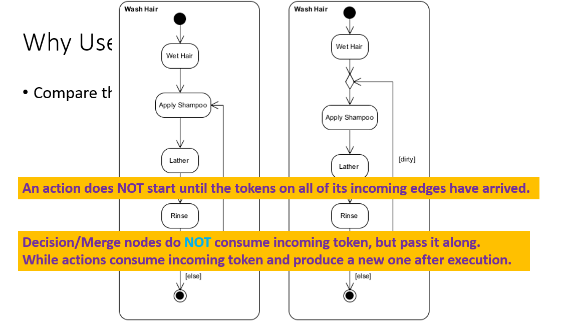
Keep adding on until you have a skyscraper

What Models Provide:

An understanding of a system (as it is or should be)

A specification of a system; A template for construction

A record of the decisions that were made

**UML Activity Diagrams**

Purpose: Describe a task/procedure/process

Components:

Activity - a non-atomic (i.e., decomposable) task

Action - an atomic (i.e., not decomposable) task

Data - inputs to or outputs from actions (and may include state information in square brackets)

Flow - flow of control and flow of data

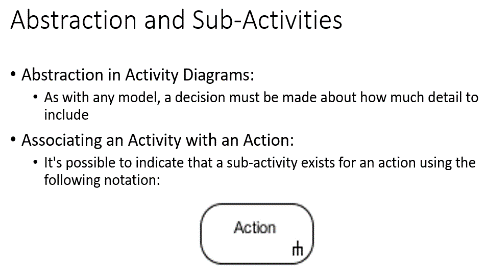
**Representation and Interpretation**

Directed Graph:

Nodes/Vertices - represent actions or objects

Edges/Arcs/Links - represent control flows and/or data flows

Tokens:

Produced and consumed by nodes (though not represented on the diagram)

Flow instantaneously along edges

Execution:

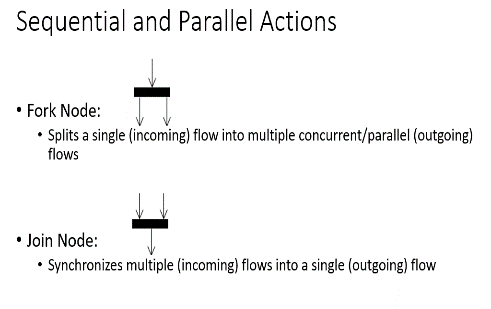
An action node begins executing when tokens are available on all incoming edges

When an action node begins execution, it consumes all incoming tokens

While/after an action node executes it produces outgoing tokens

**Still More Elements**

Pin: A terminator for data flows into (input pins) or out of (output pins) an action node

Activity Parameter: 

An object node on the boundary of an activity

Contains the name of a particular object and the name of its type

Syntax: name:type

1. Actions and activities are the same. (F)

2. There can be several activity final nodes in a diagram. (T)

3. There can be several initial nodes in a diagram. (T)

4. A decision node always has two branches. (F)

5. In general, there are as many merge nodes as decision nodes in a diagram. (T)

**Software Process**

* A process (a.k.a. activity) is a collection of related tasks (a.k.a. actions) that transforms a set of inputs into a set of outputs.
* Processes and Algorithms:
  + The set of algorithms is a subset of the set of processes (which, includes, for example heuristics)
* Describing Processes:
  + Specify the inputs to and outputs from the process
  + Specify the tasks
  + Specify the inputs to and outputs from each task (a.k.a. data flows)
  + Specify the order of tasks and the conditions under which they occur (a.k.a. control flows)

Definition of a Software Process:

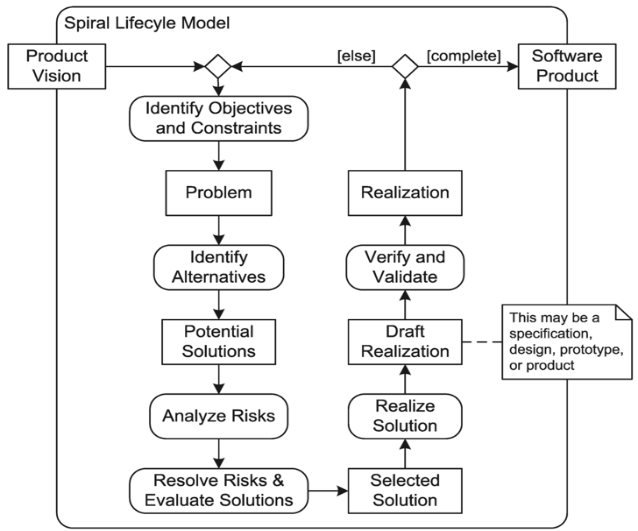
* A set of actions/tasks (and corresponding inputs and outputs) that results in the specification, development, validation, and/or evolution of a software product

An Observation:

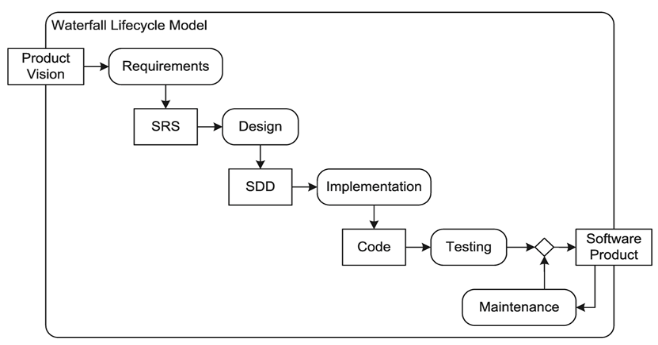
* Software processes are very complex and, hence, our discussion of them will necessarily involve abstraction (i.e., we will really be discussing process models)



**Common Actions/Tasks in Software Processes**

* Project Planning:
  + Financial/economic analysis
  + Scheduling
  + Resource allocation
  + Cost estimation
  + Risk management
* Product Design:
  + Identification of needs and desires
  + Specification of requirements
  + Prototyping
* Engineering Design:
  + Creation of static models
  + Creation of dynamic models
  + Consideration of architectural styles and design patterns
* Implementation:
  + Development
  + Code/documentation management
  + Debugging
  + Verification and validation
* Deployment
* Support and Maintenance

**Problem Solving/Design Actions/Tasks in Software Processes**

* An Observation: Many of the actions/tasks in software processes are said to involve "problem solving" and/or "design"
* For Our Purposes: Problem Solving and design involve the same (or at least the same kind of) steps
* The Tasks:
  + Identification of goals, objectives and constraints
  + Generation of alternatives
  + Evaluation of alternatives
  + Selection of an alternative
* Categorizing the Actions:
* Analysis (to understand the problem)
* Resolution (to solve the problem)

**Models of Software Processes**

* As Always: Since they are models, they are simplifications (and some are very abstract)
* Types of Models of Software Processes:
  + Descriptive - what actually happens
  + Prescriptive - what should (i.e., is supposed to) happen

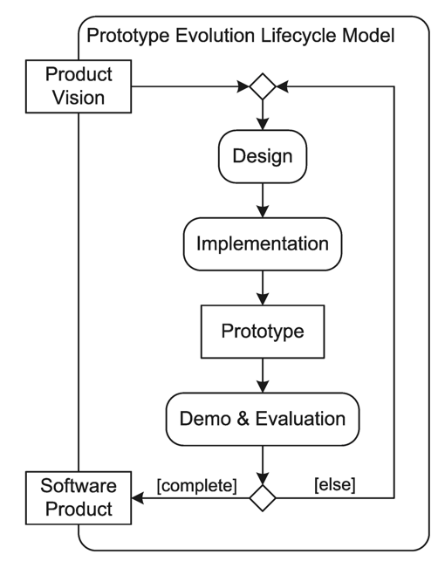
The Waterfall Model

Pros: Predictable, Easy to estimate, Easy to fix, Documentation, Distributable

Cons: Requiring stable Req., Requiring complete & correct Req., Documentation, Communication Overhead, Management, Taking too long

SRS: Software Requirements Specification

SDD: Software Design Document



Prototyping

* A prototype is a working model of some or all of a finished product.
* Throwaway prototypes
* Evolutionary prototypes

Pros

* Easy to handle changes, getting a working product quickly
* less documentation or management required, Lightweight

Cons

* Hard to predict (time & cost), Product may be unmaintainable,
* Little discipline -> bad quality control -> buggy product

Risk: An occurrence with negative consequences.

**Risk Management**

* The practice of identifying, analyzing, and controlling or mitigating risks
* Can (and should) be incorporated into any life cycle processes. For example, prototype evolution model.

Contributions:

* Explicit incorporation of increases in fidelity/detail
* Explicit incorporation of risk
* One of the early iterative processes

Cons:

* Not very many people are trained in and good at-risk management
* Very general and adaptable, demands expertise in tailoring software processes

Incremental Processes

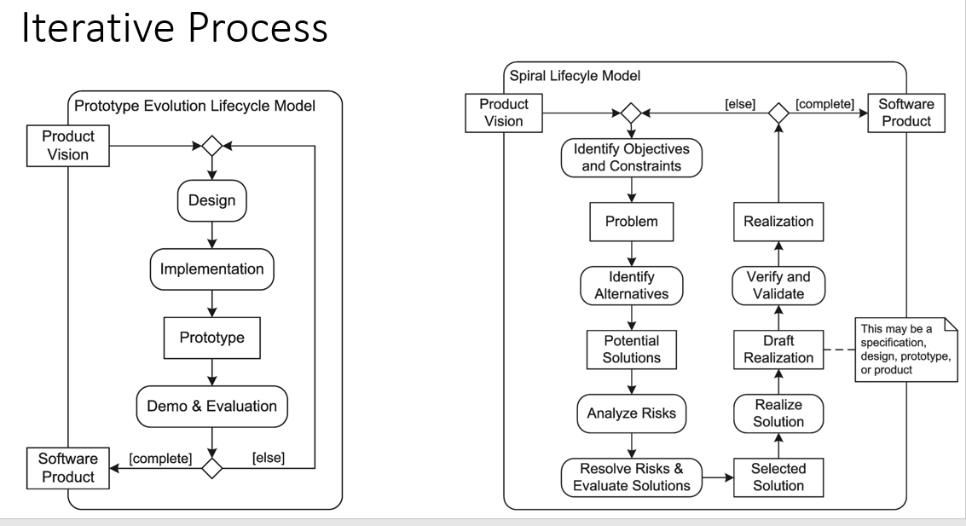
* Definition: An incremental process is one that produces its output in phases and the output of each phase is a working version of the software product

Example:

* When a customer orders the breakfast special, the chef can first cook and deliver some of the eggs, then cook and deliver the rest of the eggs and the bacon (an incremental process) or cook all of the eggs and bacon and deliver them at the same time (not an incremental process)

Iterative vs. Incremental Software Processes

* Iterative Processes: Any software process can be made iterative and most are naturally iterative
* Incremental Processes: Software processes are sharply distinguished by whether they are incremental (e.g., the waterfall process is not -- the intermediate outputs are not versions of the product)



**The Rational Unified Process (RUP) Stages**

Inception:

* Develop use cases
* Estimate the schedule and resource requirements
* Estimate risks
* Create a business case

Elaboration:

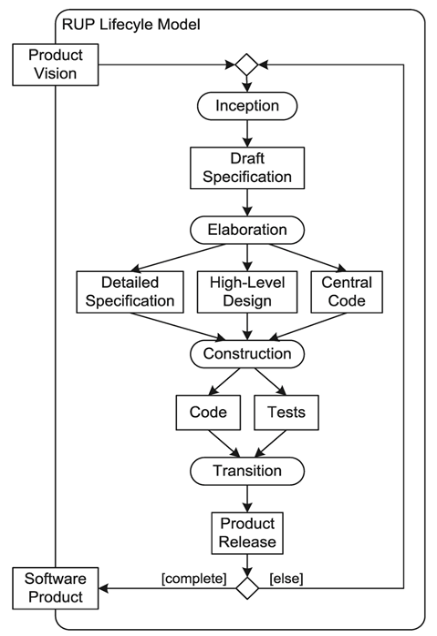
* Understand the problem domain
* Create baseline design
* Create test plan
* Decide upon metrics
* Organize resources

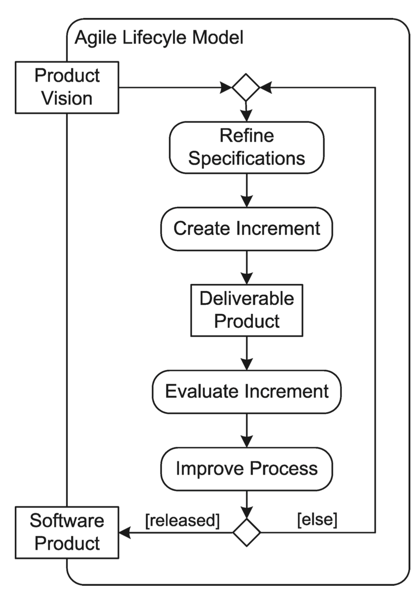
Construction:

* Complete the implementation
* Release code to α testers
* Identify remaining activities

Transition:

* Complete the final release
* Confirm user readiness and acceptance
* Confirm support readiness

Iteration:

* Can occur within a phase
* Can occur between phases

Core Process Workflows:

* Business modeling
* Requirements
* Analysis and design
* Implementation
* Testing
* Deployment
* Configuration and change management
* Project management

Pros

* Fully specified process, comes with templates
* Adaptable to projects of different sizes/needs
* Less risk
* Getting usable product quickly
* Iterative

Cons

* Requiring knowledgeable and experienced developers
* A lot of documentation and management effort
* Cannot handle requirements change very well

Agile Process

* Pros: Handle changes very well, Quick delivery of a working product to customer, can recognize bad projects early (then cancel them), Reduce waste and duplication of effort
* Cons: Customers and users have to commit time & effort, Incremental design may not be a good thing, Difficult to use on large projects, Hard to predict outcomes

Extreme Programming - XP (Beck; 2004)

* Basic Principles: Ensure rapid feedback, Embrace change, Collective ownership
* Practices: Small releases (every release should be as small as possible containing the most valuable business requirements), Have a metaphor for the project (that replaces the architecture), Unit tests for every production module and functional tests for features, Make a small initial investment (i.e., the simplest solution that works is the best) and refactor (i.e., improve) later, Pair programming (all production code is written with two people working at one computer; one person thinks about the simplest implementation the other things strategically), Continuous integration (code is integrated and tested every few hours), Strict coding standards
* Participants in the Planning Process:
  + Business
  + Development
* Planning Tools:
  + Story cards describe the necessary functionality (in the form of a task description) and an estimate of how long the functionality will take to implement
* The Planning Process:
  + Exploration - found out what new things the system could do
  + Commitment - Decide what subset of all possible requirements to pursue next (Business sorts by value; Development sorts by risk
  + Steer - Guide the development (involves iteration, recovery, new story construction, and re-estimation

**Scrum (Sutherland and Schwaber; 1995):**

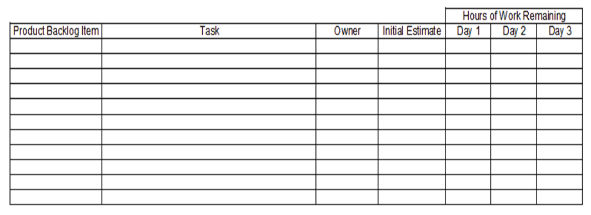
* Sprints: Fixed duration development cycles (usually 1-4 weeks) that end on a specific date (whether the work has been completed or not)
* Scrum Roles: Product Owner (responsible for achieving maximum business value), Team (multi-functional; 5-10 people), ScrumMaster (helps the Team be successful; protects the Team)
* Process: Start of the Sprint: There is a Sprint Planning Meeting at which the Team selects items from a prioritized list of requirements/features and commits to completing them; Each Day: Members of the Team report (at a standup meeting of 15 minutes or less) on progress
* Terminology: The prioritized list of requirements/features is called the Product Backlog, The tasks being completed are called the Sprint Backlog, The daily meeting is called the Daily Scrum, Members of the Team are sometimes called "Pigs" (because they are truly committed to making bacon and eggs) and everyone else is called a "Chicken" (because they are only involved in making bacon and eggs)
* Some Details about the Product Backlog: The Product Backlog includes a rough estimate of the "size", The Product Backlog also contains developer "to dos", exploratory work, and known bugs, The Product Backlog is updated by the Product Owner

Scrum - Sprint Planning Meetings

* Team members report on their availability
* The top item in the product backlog is broken down into tasks that are recorded in the Sprint Backlog
* Team members volunteer for the tasks (i.e., take ownership of them) and provide time estimates
* The process continues until all available hours are accounted for (Note: Since commitments are being made these meetings can take hours)

Scrum - Sprints

* The team should strive to have a working product at the end of every sprint
* The Sprint Backlog contains, at a minimum, the tasks, the name of the task owner, the initial estimate, and the hours of work remaining
* Teams often post a Sprint Burndown Chart that graphically illustrates the numbers of hours remaining
* It is very important that the team have a definition of "Done" that can be used to assess when work is complete on the current increment

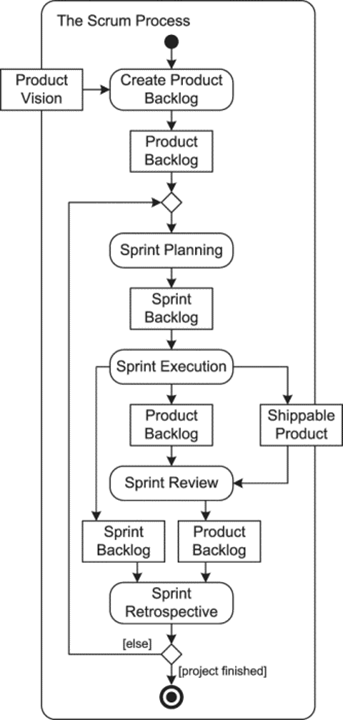
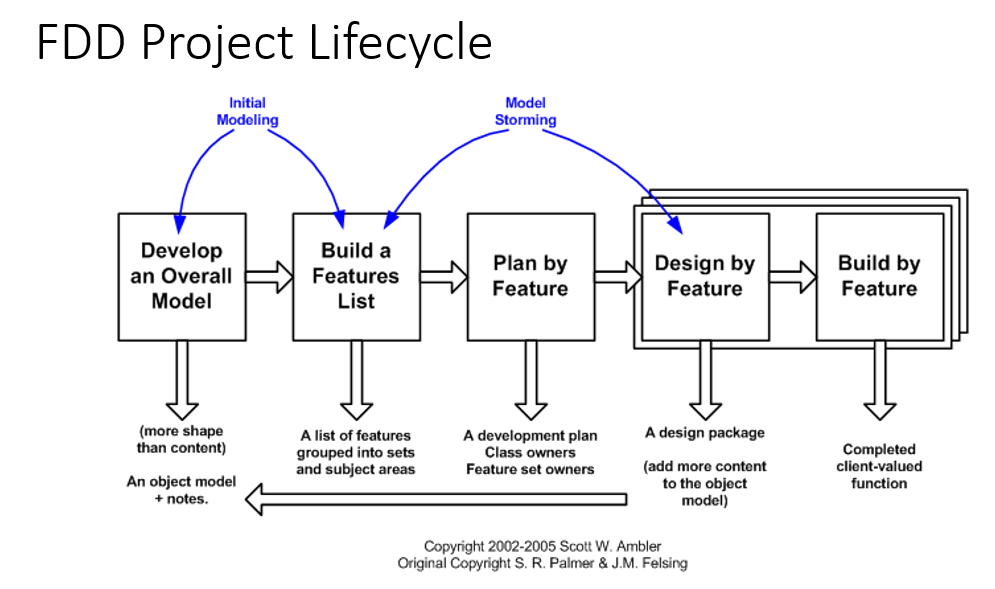


**Feature Driven Development - FDD (De Luca; 1977)**

* The Process: Develop an overall model, Build a feature list, Plan by feature, Design by feature,

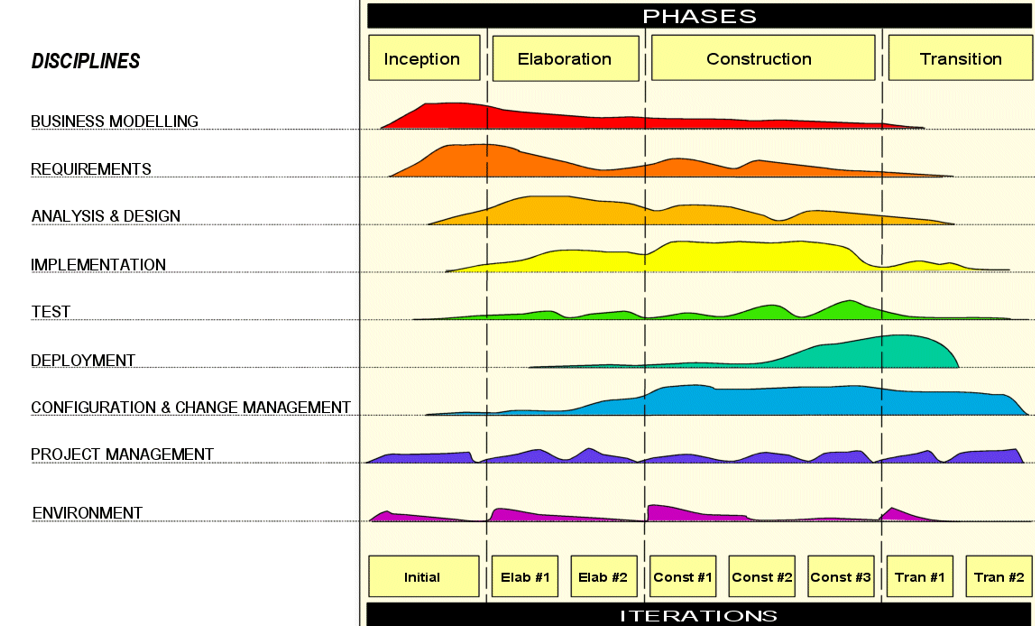
Build by feature

* Practices: Individual ownership of classes, Extensive use of code inspections, Regular builds



"Heavyweight" vs. "Agile"/"Lean" Methods

* A Common Categorization:
  + Heavyweight: Waterfall, Incremental Delivery, Spiral, RUP
  + Agile/Lean: XP, Scrum
* The Agile Manifesto:
  + Value individuals and interactions over processes and tools
  + Value working software over comprehensive documentation
  + Value customer collaboration over contract negotiation
  + Value responding to change over following a plan
* Another View:
  + The different process involves the same actions/tasks but vary dramatically in the cycle times

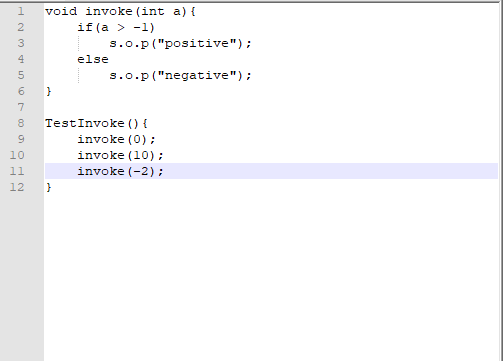


## **What is JUnit?**

JUnit is a unit testing framework for Java programming language. It plays a crucial role test-driven development and is a family of unit testing frameworks collectively known as JUnit. JUnit promotes the idea of "first testing then coding", which emphasizes on setting up the test data for a piece of code that can be tested first and then implemented. This approach is like "test a little, code a little, test a little, code a little." It increases the productivity of the programmer and the stability of program code, which in turn reduces the stress on the programmer and the time spent on debugging.

**Eclemma:**

Used for backend software testing by developers where they test a product/program using white box testing (method in which the internal structure/design/implementation of the item being tested is known to the tester). This tool provides coverage for all statements, branches, and paths within a program letting a tester or user know if all three of these things are being tested within the program with a percentage. In Eclipse, when using this tool to test the code, certain code is highlighted green (testing covered) and red (not covered).



Ex: Calculate the statement coverage by hand

For the code on the right.

When a = 0 -- >

Statement coverage: 1, 2, 3, and 6

Here the ratio is: 4:6

When a = -2 --->

Coverage: 1,2,4,5,6

Here ratio is: 5:6

At this point all statements have been covered.

Hence the test coverage should be 100% for

this code.







