SWEN 6301 Software Construction

Lecture 2: Laying the Foundation

Ahmed Tamrawi


Software Construction
a very complicated task

Laying the Foundation

Variables & Statements

Concurrency

Software Craftsmanship

Creating High-Quality Code

Code Improvements

System Considerations

Laying the Foundation

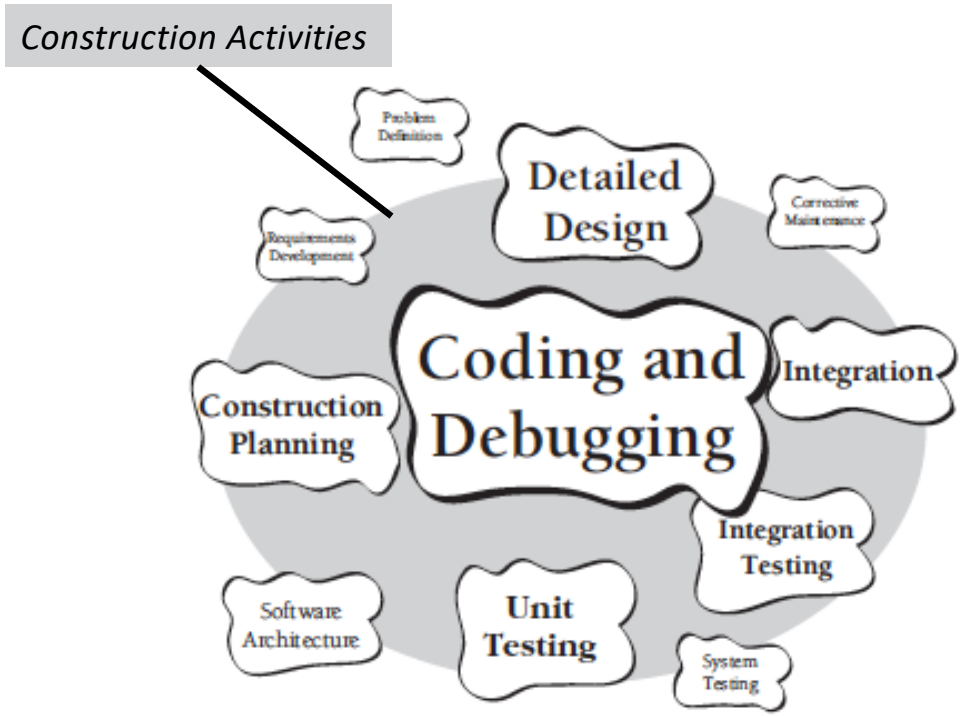
Software Construction	Metaphors for a Richer Understanding of Software Development	Measure Twice, Cut Once: Upstream Prerequisites	Key Construction Decisions

Software Construction

SWEN 6301 Software Construction Definition

Software construction is the process of **creating** and **evolving** software source code that results on *extensible*, *maintainable*, *robust*, and *secure* software

What Is Software Construction?



Coding implies the mechanical translation of a preexisting design into a computer language; **construction** is not at all mechanical and involves substantial creativity and judgment.

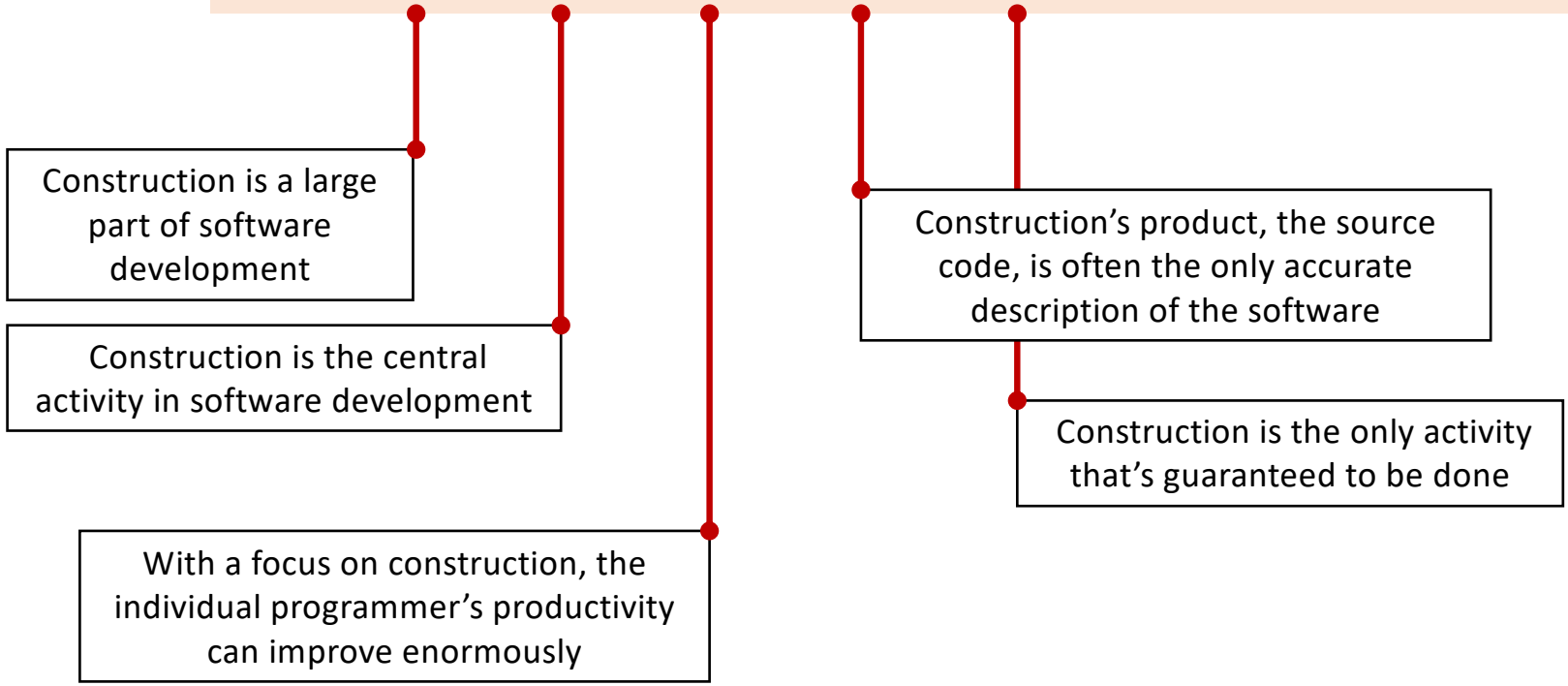
Construction focuses on **coding and debugging** but also includes detailed design, unit testing, integration testing, and other activities.

High-level View of Construction Activities

Verifying that the groundwork has been laid so that construction can proceed successfully
Determining how your code will be tested
Designing and writing classes and routines
Creating and naming variables and named constants
Selecting control structures and organizing blocks of statements
Unit testing, integration testing, and debugging your own code
Reviewing other team members' low-level designs and code and having them review yours
Polishing code by carefully formatting and commenting it
Integrating software components that were created separately
Tuning code to make it faster and use fewer resources

Specific Tasks of Construction Activities

Why is Software Construction Important?



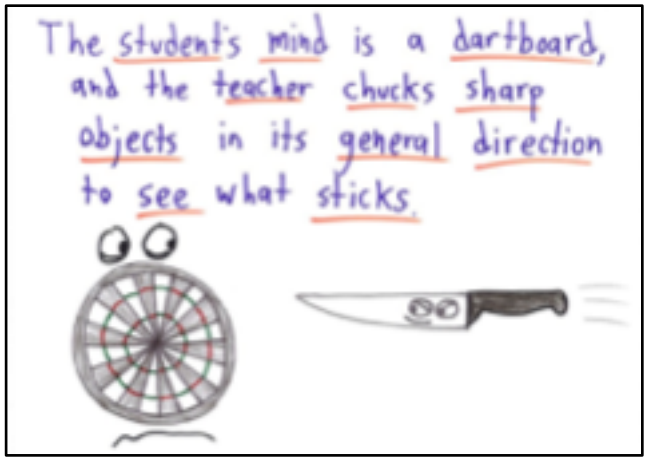
SUMMARY

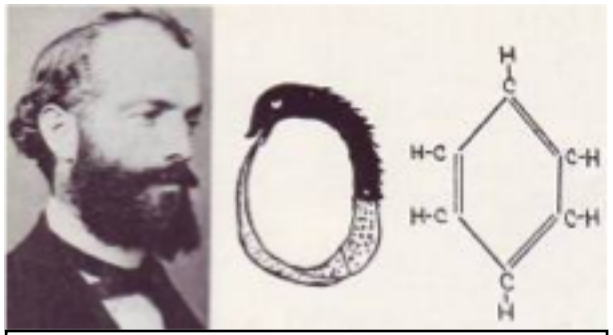
- Software construction is the central activity in software development; construction is the only activity that's guaranteed to happen on every project.
- The main activities in construction are detailed design, coding, debugging, integration, and developer testing (unit testing and integration testing).
- Other common terms for construction are "coding" and "programming."
- The quality of the construction substantially affects the quality of the software.
- Your understanding of how to do construction determines how good a programmer you are.

Metaphors for a Richer Understanding of Software Development

The Importance of Metaphors

Important developments often arise out of analogies. By comparing a topic you understand poorly to something similar you understand better, you can come up with insights that result in a better understanding of the less-familiar topic. This use of metaphor is called “modeling.”





The chemist Kekulé perceived the structure of the benzene ring after dreaming of a snake biting its own tail

The kinetic theory of gases was based on a “billiard-ball” model. Gas molecules have mass and collide elastically

A good metaphor is **simple**, relates well to other **relevant** metaphors, and explains much of the **experimental evidence** and other **observed phenomena**

How to Use Software Metaphors

To give you insight into your programming problems and processes

To help you think about your programming activities

To help you imagine better ways of doing things.

Algorithm

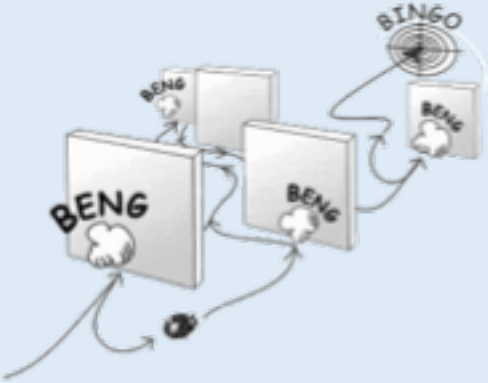
set of well-defined instructions for carrying out a particular task

predictable, deterministic, and not subject to chance

tells you how to go from point A to point B with no detours, no side trips to points D, E, and F, and no stopping to smell the roses or have a cup of joe

An algorithm gives you the instructions directly

Heuristic



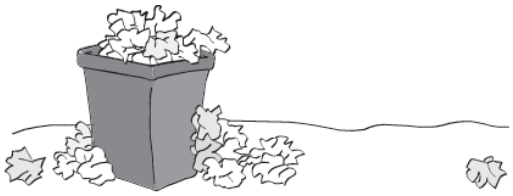
helps you look for an answer

subject to chance as a heuristic tells you only how to look, not what to find

doesn't tell you how to get directly from point A to point B; it might not even know where point A and point B are

A heuristic tells you how to discover the instructions, or where to look for them

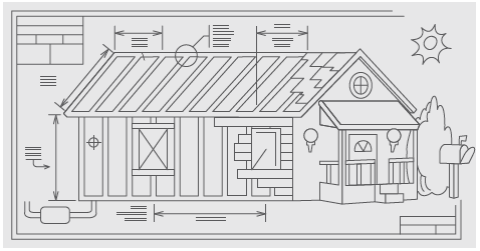
Common Software Metaphors



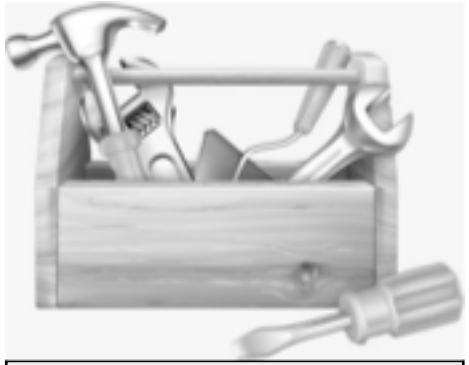
Software Penmanship: Writing Code

Software Farming: Growing a System

Software Oyster Farming System Accretion



Software Construction: Building Software



Applying Software Techniques: The Intellectual Toolbox

SUMMARY

- Metaphors are heuristics, not algorithms. As such, they tend to be a little sloppy.
- Metaphors help you understand the software-development process by relating it to other activities you already know about.
- Some metaphors are better than others
- Treating software construction as similar to building construction suggests that careful preparation is needed and illuminates the difference between large and small projects.
- Thinking of software-development practices as tools in an intellectual toolbox suggests further that every programmer has many tools and that no single tool is right for every job.
- Metaphors are not mutually exclusive. Use the combination of metaphors that works best for you.

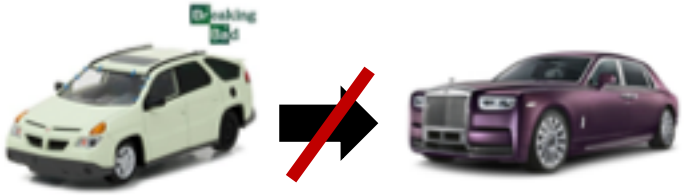
Measure Twice, Cut Once: Upstream Prerequisites

Software development which amounts to 65% of project cost; Doing the most expensive part of the project twice is as bad an idea in software as it is in any other line of work.

Importance of Prerequisites

A common denominator of programmers who build high quality software is their use of high-quality practices. Such practices emphasize quality at the beginning, middle, and end of a project.

The earlier parts of the project have already laid some of the groundwork for success/failure



Construction

Requirements/Design

If you emphasize quality at the beginning, you plan for, require, and design a high-quality product.

Testing

Testing ≠ Quality Assurance

The overarching goal of preparation is **Risk Reduction**

a good project planner clears major risks out of the way as early as possible so that the bulk of the project can proceed as smoothly as possible

most common project risks in software development

Preparation for construction is not an **exact science**, and the specific approach to risk reduction must be decided project by project.

Causes of Incomplete Preparation

MANAGERS

Managers are notoriously unsympathetic to programmers who spend time on construction prerequisites

WISCA or WIMP syndrome: Why Isn't Sam Coding Anything? or Why Isn't Mary Programming?

DEVELOPERS

do not have the expertise to carry out upstream activities

The skills needed to plan a project, create a compelling business case, develop comprehensive and accurate requirements, and create high-quality architectures are far from trivial

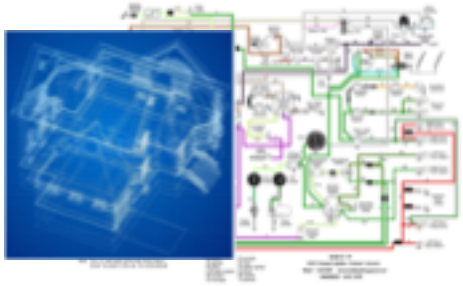
The recommendation to "do more upstream work" sounds like nonsense: If the work isn't being done well in the first place, doing more of it will not be useful!

Some programmers do know how to perform upstream activities, but they don't prepare because they can't resist the urge to begin coding as soon as possible.

Utterly Compelling and Foolproof Argument for Doing Prerequisites Before Construction

Appeal to Logic

Appeal to Analogy



Appeal to Data

Boss-Readiness Test

Appeal to Data

The cost to fix a defect **rises dramatically** as the time from when it's introduced to when it's detected increases

Researchers have found that purging an error by the beginning of construction allows rework to be done 10 to 100 times less expensively than when it's done in the last part of the process, during system test or after release

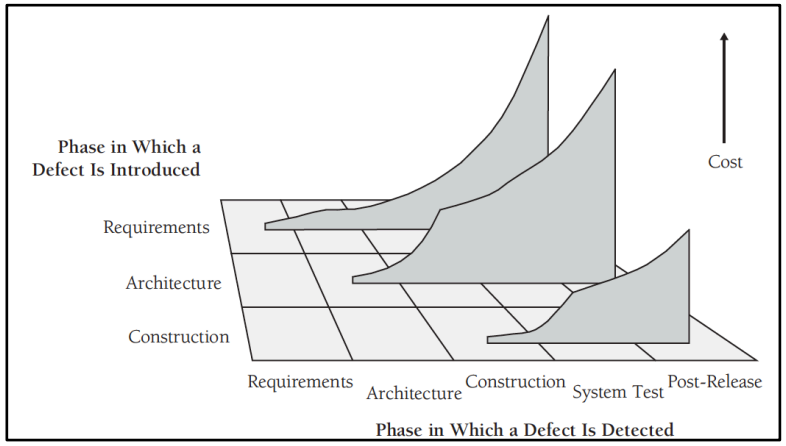


Table 3-1 Average Cost of Fixing Defects Based on When They're Introduced and Detected

	Time Detected				
Time Introduced	**Requirements**	**Architecture**	**Construction**	**System Test**	**Post-Release**
Requirements	1	3	5–10	10	10–100
Architecture	—	1	10	15	25–100
Construction	—	—	1	10	10–25

Source: Adapted from "Design and Code Inspections to Reduce Errors in Program Development" (Fagan 1976), *Software Defect Removal* (Dunn 1984), "Software Process Improvement at Hughes Aircraft" (Humphrey, Snyder, and Willis 1991), "Calculating the Return on Investment from More Effective Requirements Management" (Leffingwell 1997), "Hughes Aircraft's Widespread Deployment of a Continuously Improving Software Process" (Willis et al. 1998), "An Economic Release Decision Model: Insights into Software Project Management" (Grady 1999), "What We Have Learned About Fighting Defects" (Shull et al. 2002), and *Balancing Agility and Discipline: A Guide for the Perplexed* (Boehm and Turner 2004).

Boss-Readiness Test

Which of these statements are self-fulfilling prophecies?

1. *We'd better start coding right away because we're going to have a lot of debugging to do.*
2. *We haven't planned much time for testing because we're not going to find many defects.*
3. *We've investigated requirements and design so much that I can't think of any major problems we'll run into during coding or debugging.*

Determine the Kind of Software You're Working On

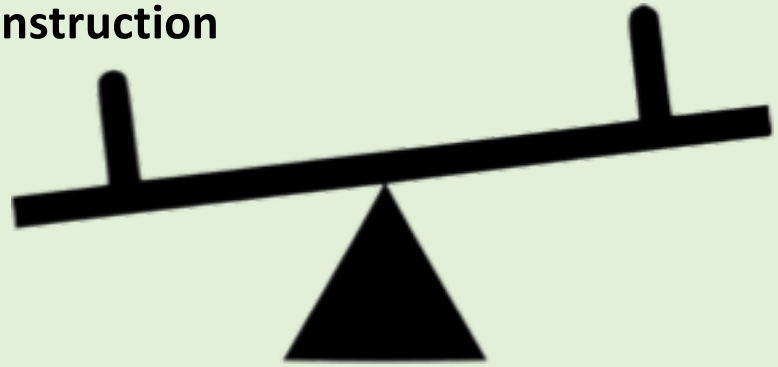
Capers Jones

Vice president and Chief Technology Officer of Namcook Analytics LLC

*He summarized 20 years of software research by pointing out that he and his colleagues have seen **40 different methods for gathering requirements**, **50 variations in working on software designs**, and **30 kinds of testing** applied to projects in more than **700 different programming languages**.*

Determine the Kind of Software You're Working On

Different kinds of software projects call for different balances between **preparation** and **construction**

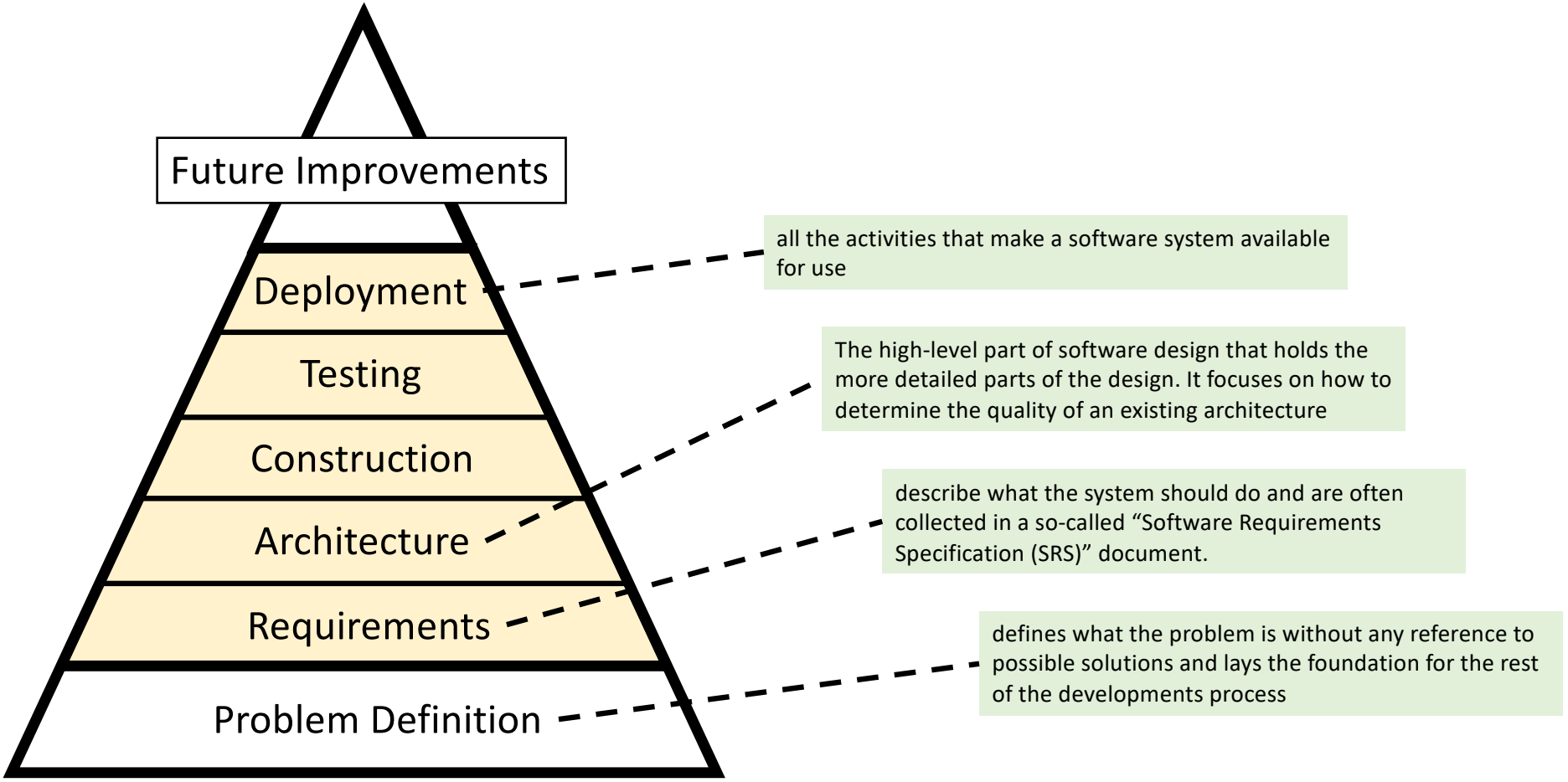


Every project is **unique**, but projects do tend to fall into **general development styles**

Typical Good Practices for Three Common Kinds of Software Projects

	Kind of Software		
	Business Systems	**Mission-Critical Systems**	**Embedded Life-Critical Systems**
Typical applications	Internet site Intranet site Inventory management Games Management information systems Payroll system	Embedded software Games Internet site Packaged software Software tools Web services	Avionics software Embedded software Medical devices Operating systems Packaged software
Life-cycle models	Agile development (Extreme Programming, Scrum, timebox development, and so on) Evolutionary prototyping	Staged delivery Evolutionary delivery Spiral development	Staged delivery Spiral development Evolutionary delivery
Planning and management	Incremental project planning As-needed test and QA planning Informal change control	Basic up-front planning Basic test planning As-needed QA planning Formal change control	Extensive up-front planning Extensive test planning Extensive QA planning Rigorous change control
Requirements	Informal requirements specification	Semiformal requirements specification As-needed requirements reviews	Formal requirements specification Formal requirements inspections
Design	Design and coding are combined	Architectural design Informal detailed design As-needed design reviews	Architectural design Formal architecture inspections Formal detailed design Formal detailed design inspections
Construction	Pair programming or individual coding Informal check-in procedure or no check-in procedure	Pair programming or individual coding Informal check-in procedure As-needed code reviews	Pair programming or individual coding Formal check-in procedure Formal code inspections
Testing and QA	Developers test their own code Test-first development Little or no testing by a separate test group	Developers test their own code Test-first development Separate testing group	Developers test their own code Test-first development Separate testing group Separate QA group
Deployment	Informal deployment procedure	Formal deployment procedure	Formal deployment procedure

Software Development Phases

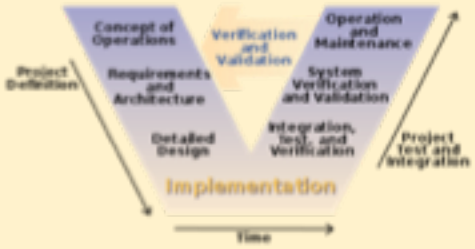


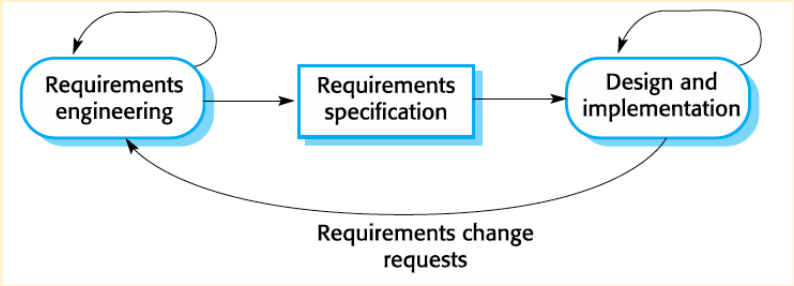
Software Development Process

PLAN-DRIVEN METHODS

Waterfall Model

V-Model





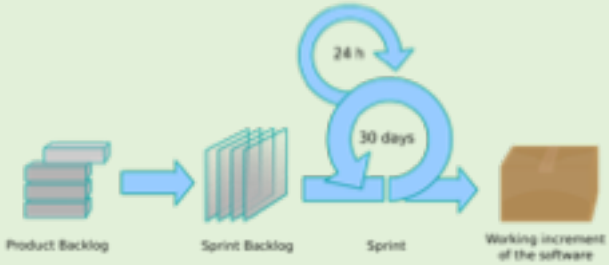
AGILE METHODS

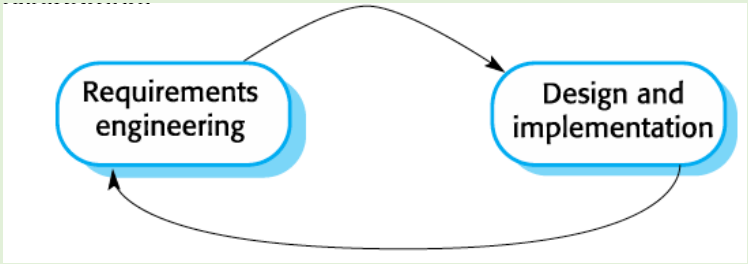
eXtreme Programming (XP)

Feature Driven Development (FDD)

Lean Software Development

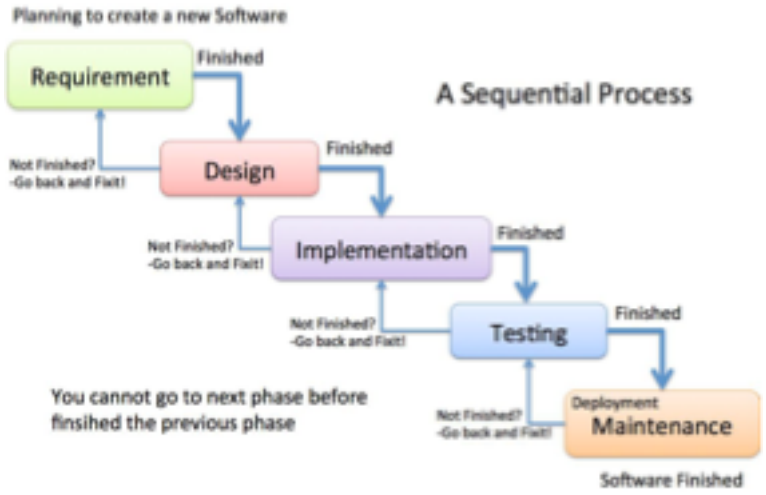
SCRUM





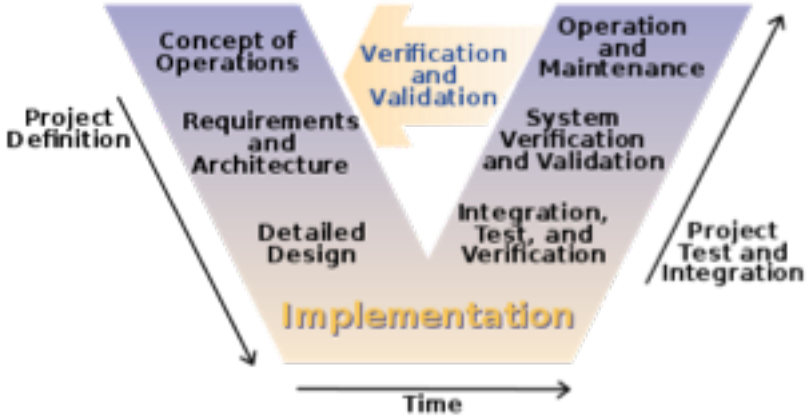
Plan-Driven Methods

Waterfall Model



*In practice, it is **impossible to create perfect requirements and design** before you start implementing the code, so it is common to go back and update these **phases iteratively**.*

V-Model



Reflects a project management view of software development and fits the needs of project managers, accountants and lawyers rather than software developers or users.

Agile Software Development

A group of software development methods based on iterative and incremental development

Important Agile features and principles

Customer Involvement	Test Driven Development (TDD)	Less Documentation. Only what is necessary	Continuously Integrate Changes **Refactoring**
Communication	Work in Iterations **Incremental**	Working Software at all times	
Pair Programming			

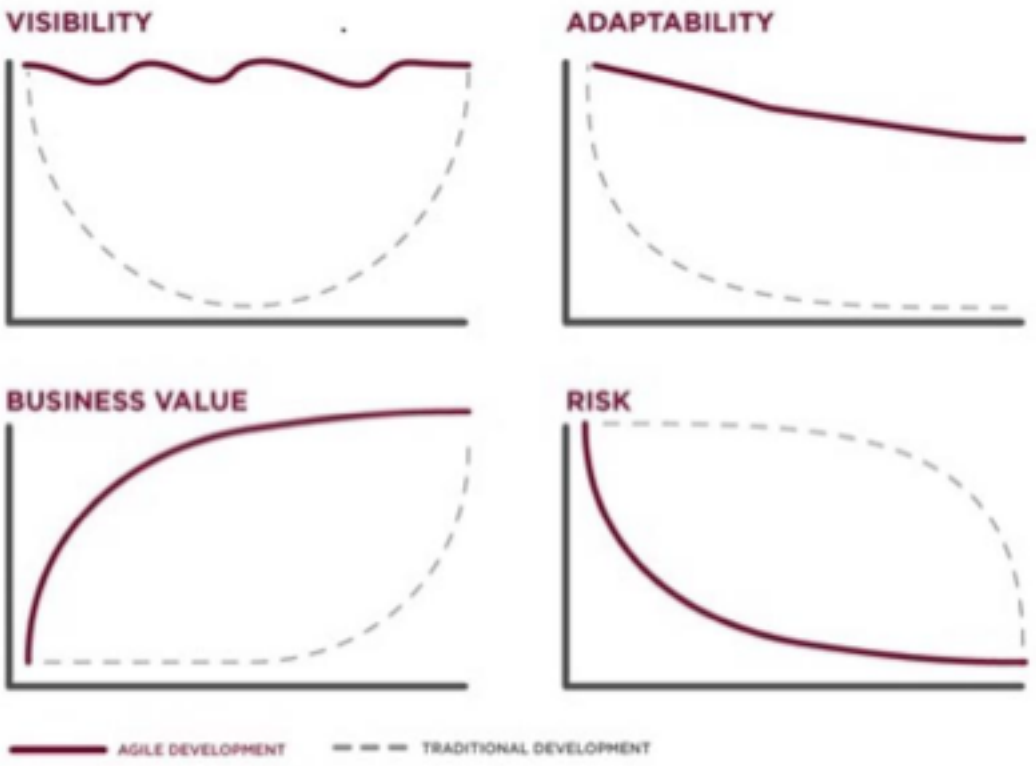
Popular Agile methods

SCRUM

eXtreme Programming


Manifesto for Agile Software Development
We are uncovering better ways of developing
software by doing it and helping others do it.
Through this work we have come to value:
Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan
That is, while there is value in the items on
the right, we value the items on the left more.
Kent Beck
Mike Beedle
Arie van Bennekum
Alistair Cockburn
Ward Cunningham
Martin Fowler
James Grenning
Jim Highsmith
Andrew Hunt
Ron Jeffries
Jon Kern
Brian Marick
Robert C. Martin
Steve Mellor
Ken Schwaber
Jeff Sutherland
Dave Thomas
http://agilemanifesto.org/iso/en/manifesto.html

Agile Development vs. Traditional Development



Agile Software Development: eXtreme Programming

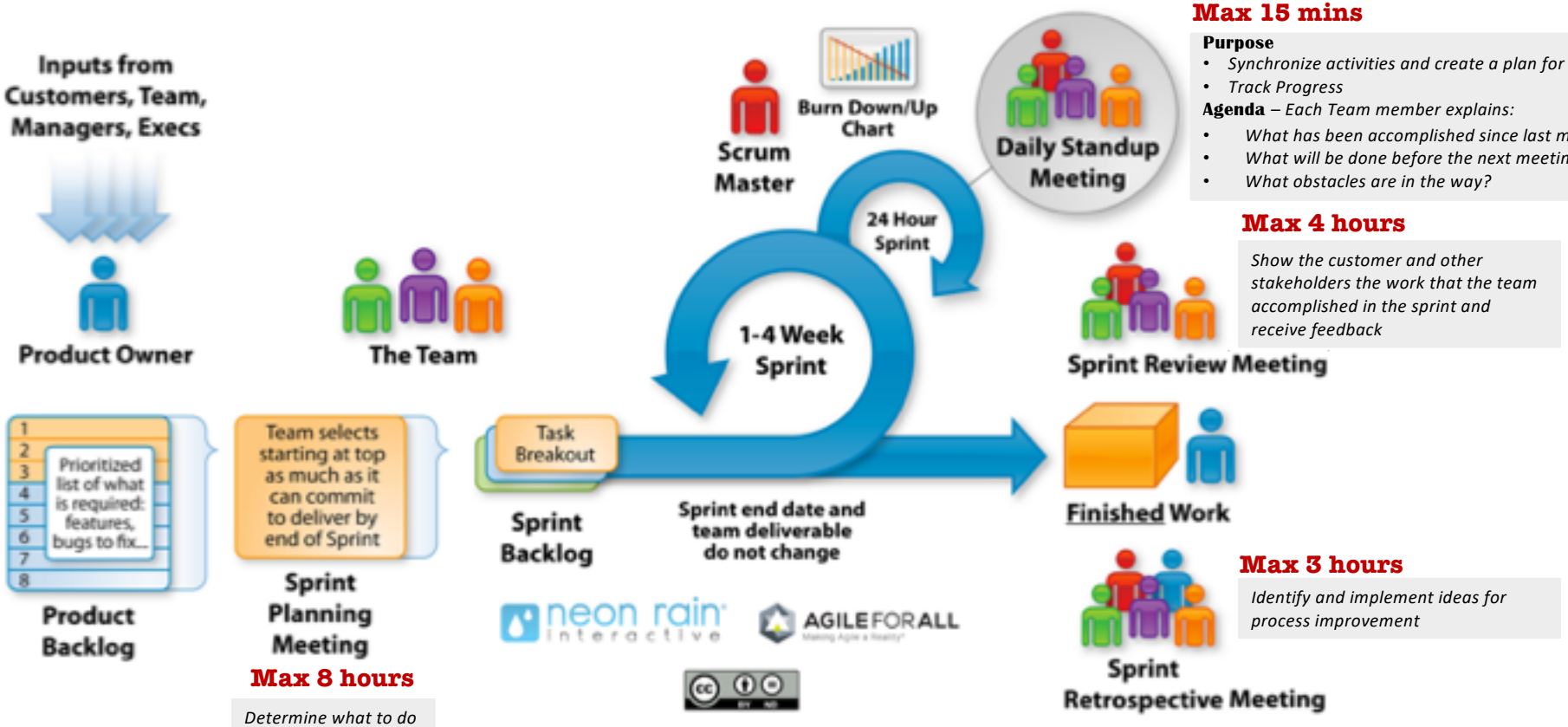
it advocates frequent "releases" in short development cycles, which is intended to improve productivity and introduce checkpoints at which new customer requirements can be adopted

Elements of extreme programming include:

- Pair Programming
- Extensive Code Review
- Unit testing of all code (TDD)
- Avoiding programming of features until they are needed
- Flat management structure
- Expecting changes in the customer's requirements
- Frequent communication with the customer and among programmers

Agile Software Development: SCRUM

A framework for managing work with an emphasis on software development. It is designed for teams of developers (3 to 9) who break their work into actions that can be completed within timeboxed iterations, called sprints (30 days or less, most commonly two weeks) and track progress and re-plan in 15-minute stand-up meetings, called daily scrums.



Problem Definition

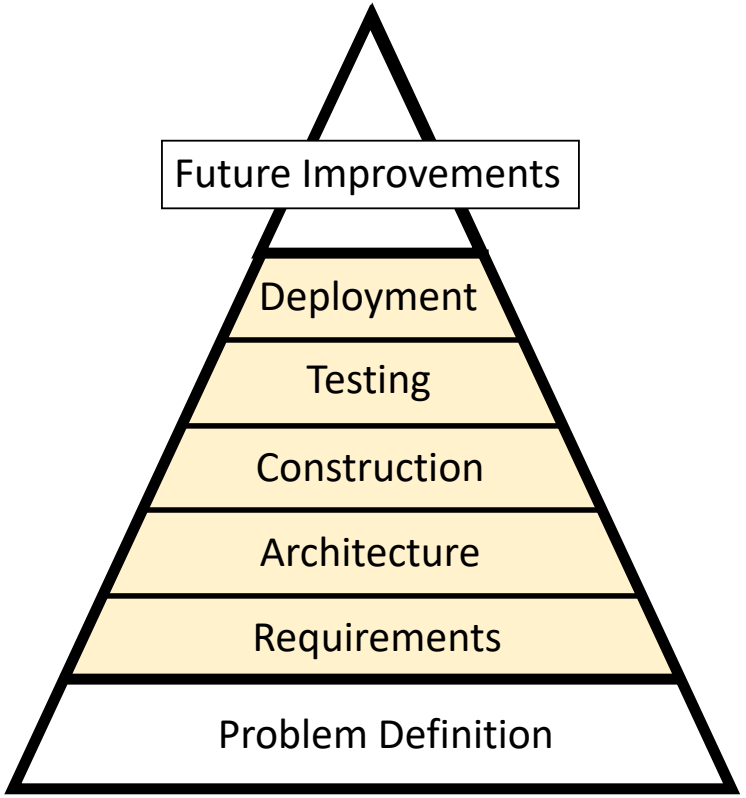
*The **problem definition** lays the foundation for the rest of the programming process.*

A problem definition defines what the problem is without any reference to possible solutions

- A simple statement (one or two pages).
- It should be in user language and described from a user's point of view.
- Technical computer terms are better to be avoided
- a.k.a. product vision, mission statement and product definition

EXAMPLE

- "We can't keep up with orders for the Gigatron" ☺
- "We need to optimize our automated data-entry system to keep up with orders" 😐



Problem Definition

Be sure a software or a computer program is required to address a given problem, maybe the best solution is not a software

EXAMPLE

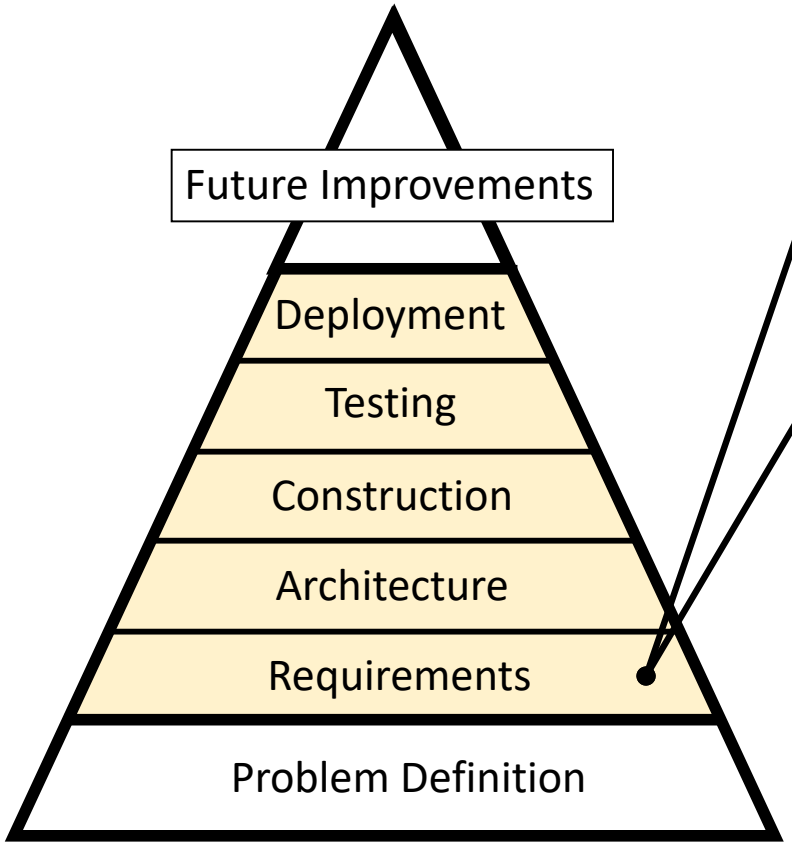
There is a computer system producing reports of quarterly profits. What should we do for yearly profits? A new computer program?
Just add up the quarterly profits to calculate the yearly profit.

Problem Definition

Be sure you know what you're aiming at before you shoot

The penalty for failing to define the problem is that you can waste a lot of time solving the wrong problem. This is a double barreled penalty because you also don't solve the right problem.

Requirements



Requirements describe in detail what a software system is supposed to do, and they are the first step toward a solution

a.k.a "requirements development," "requirements analysis," "analysis," "requirements definition," "software requirements," "specification," "functional spec," and "spec."

Requirements

Requirements

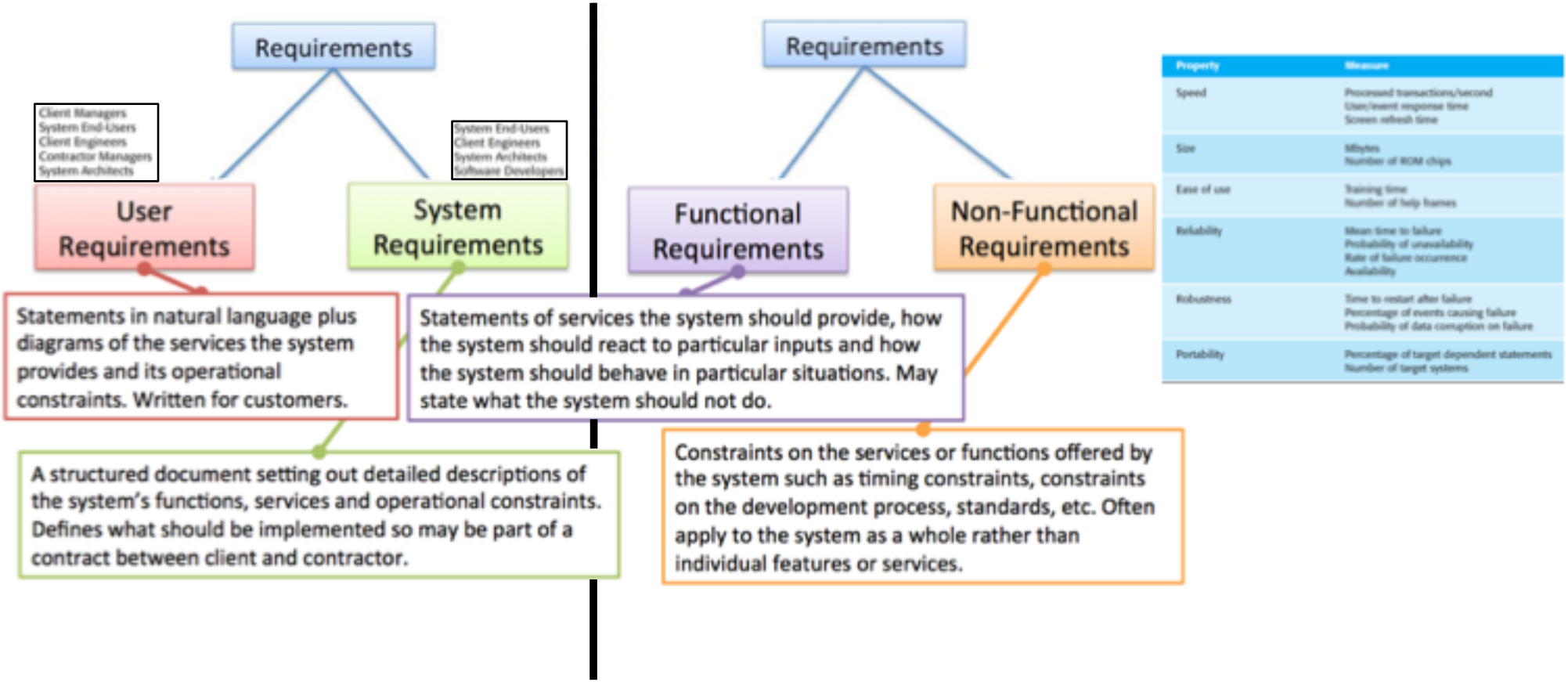
Requirements Engineering *(RE) refers to the process of formulating, documenting and maintaining software requirements.*

Software Requirement Specification Document

The requirements are in some cases created by the **customer**, at least the overall requirements (it defines "What" the customers want), while more details are normally created by **architects and developers** in the software company that is going to develop the actual software.

The **main challenge in Requirements Engineering** is that the customers most often don't know what they want or are not qualified to know what they need.

Requirements



EXAMPLE

Mental health care patient management system (MHC-PMS)

User Requirement Definition

1. The MHC-PMS shall generate monthly management reports showing the cost of drugs prescribed by each clinic during that month.

System Requirements Specification

1.1 On the last working day of each month, a summary of the drugs prescribed, their cost, and the prescribing clinics shall be generated.

1.2 The system shall automatically generate the report for printing after 17.30 on the last working day of the month.

1.3 A report shall be created for each clinic and shall list the individual drug names, the total number of prescriptions, the number of doses prescribed, and the total cost of the prescribed drugs.

1.4 If drugs are available in different dose units (e.g., 10 mg, 20 mg) separate reports shall be created for each dose unit.

1.5 Access to all cost reports shall be restricted to authorized users listed on a management access control list.

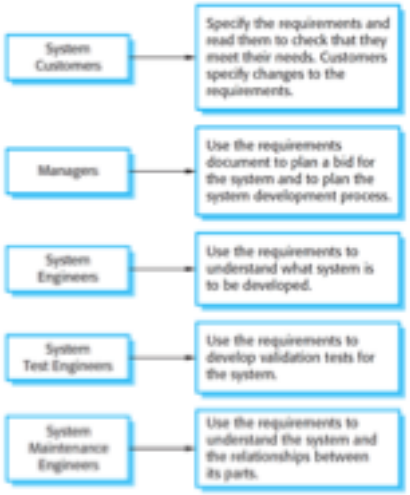
Requirements

Software Requirement Specification Document

- The software requirements document is the **official statement of what is required of the system developers**.
- Should include both a definition of **user requirements** and a specification of the **system requirements**.
- It is **NOT a design document**. As far as possible, it should set of **WHAT** the system should do rather than **HOW** it should do.

Chapter	Description
Preface	This should define the expected readership of the document and describe its version history, including a rationale for the creation of a new version and a summary of the changes made in each version.
Introduction	This should describe the need for the system. It should briefly describe the system's functions and explain how it will work with other systems. It should also describe how the system fits into the overall business or strategic objectives of the organization commissioning the software.
Glossary	This should define the technical terms used in the document. You should not make assumptions about the experience or expertise of the reader.
User requirements definition	Here, you describe the services provided for the user. The non-functional system requirements should also be described in this section. This description may use natural language, diagrams, or other notations that are understandable to customers. Product and process standards that must be followed should be specified.
System architecture	This chapter should present a high-level overview of the anticipated system architecture, showing the distribution of functions across system modules. Architectural components that are reused should be highlighted.
System requirements specification	This should describe the functional and non-functional requirements in more detail. If necessary, further detail may also be added to the non-functional requirements. Interfaces to other systems may be defined.
System models	This might include graphical system models showing the relationships between the system components, the system, and its environment. Examples of possible models are object models, data-flow models, or semantic data models.
System evolution	This should describe the fundamental assumptions on which the system is based, and any anticipated changes due to hardware evolution, changing user needs, and so on. This section is useful for system designers as it may help them avoid design decisions that would constrain likely future changes to the system.
Appendices	These should provide detailed, specific information that is related to the application being developed; for example, hardware and database descriptions. Hardware requirements define the minimal and optimal configurations for the system. Database requirements define the logical organization of the data used by the system and the relationships between data.
Index	Several indexes to the document may be included. As well as a normal alphabetic index, there may be an index of diagrams, an index of functions, and so on.

Sample Table of Contents of SRS



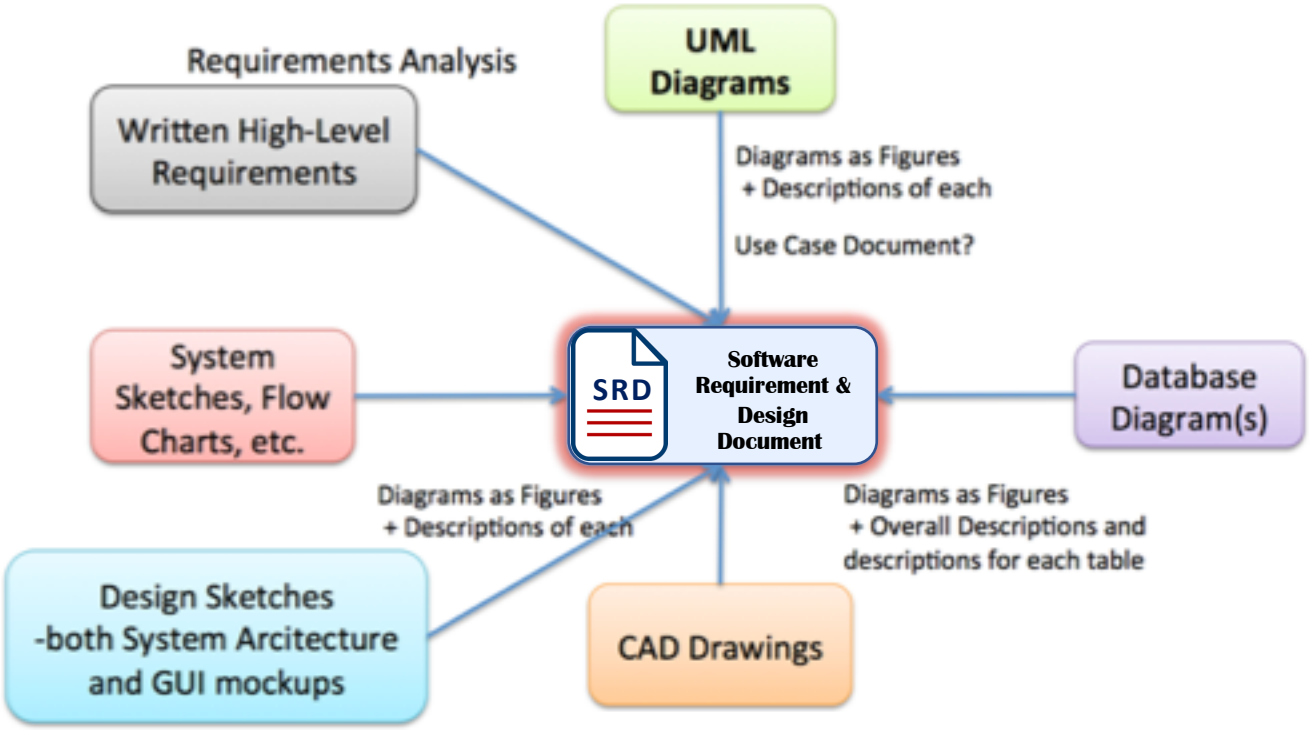
Users of SRS

IEEE Recommended Practice for Software Requirements Specifications

IEEE Computer Society

https://ieeexplore.ieee.org/document/720574/

In practice, requirements and design are inseparable. Many don't separate SRS and SDD (Software Design Document) documents, but include everything in a document called "Software Requirements and Design Document" (SRD).



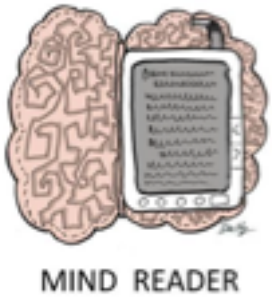
Why Have Official Requirements?

An **explicit set of requirements** is important

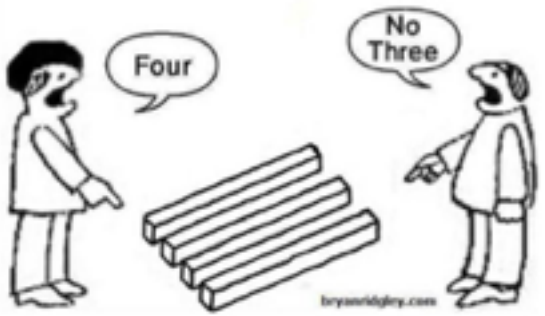
ensure that the user rather than the programmer drives the system's functionality

If they're not, the programmer usually ends up making requirements decisions during programming

keep you from guessing what the user wants



help you to avoid arguments



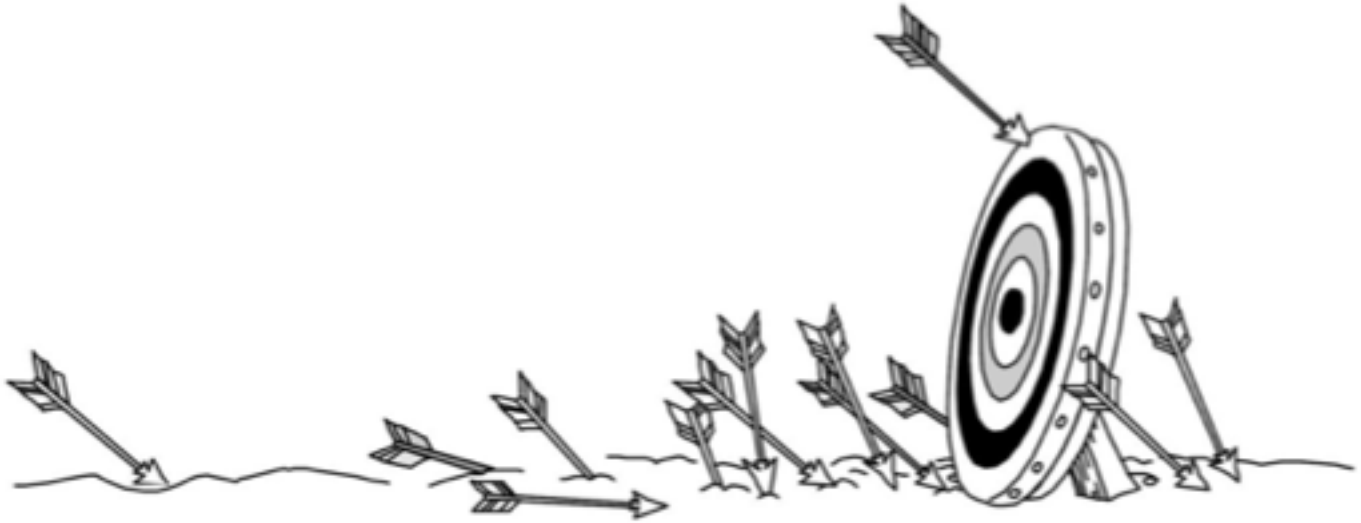
Paying attention to requirements helps to minimize changes to a system after development begins.

Coding error during coding: *you may change a few lines of code and work goes on.*

Requirements error during coding: *you may alter the design, throw away part of the old design, and accommodate code that's already written. Also, you may have to discard code and test cases affected by the requirement change and write new code and test cases. Even code that's unaffected must be retested.*

Why Have Official Requirements?

Specifying requirements adequately is a key to project success, perhaps even more important than effective construction techniques



Without good requirements, you can have the right general problem but miss the mark on specific aspects of the problem

Why Have Official Requirements?

On large projects an error in requirements detected during the architecture stage is typically 3 times as **expensive**

If detected during coding, it's 5–10 times; during system test, 10 times; and post-release, 10–100 times as **expensive**

The Myth of Stable Requirements

IN SOFTWARE HEAVEN

With stable requirements, a project can proceed from architecture to design to coding to testing in a way that's orderly, predictable, and calm.

Customers can't reliably describe what is needed before the code is written*. The problem isn't that the customers are a lower life-form. Just* ***as the more you work with the project, the better you understand it, the more they work with it, the better they understand it.***

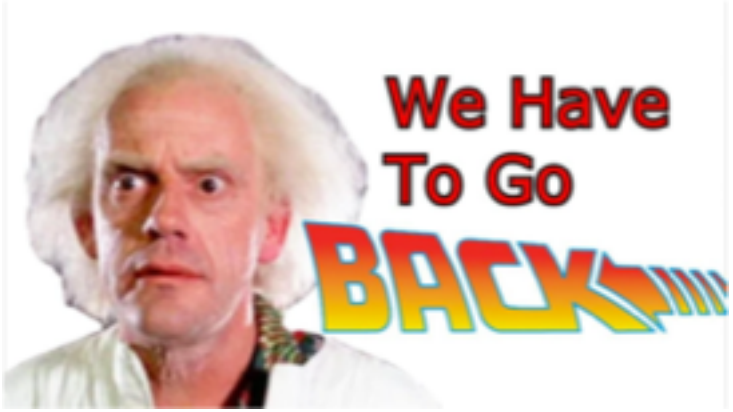
How much change is typical?

Studies at IBM and other companies have found that the average project experiences about a 25% change in requirements during development which accounts for 70 to 85% of the rework on a typical project.

Handling Requirements Changes During Construction

Use the requirements checklist to assess the quality of your requirements

If your requirements aren't good enough, stop work, back up, and make them right before proceed



Checklist: Requirements

The requirements checklist contains a list of questions to ask yourself about your project's requirements. This book doesn't tell you how to do good requirements development, and the list won't tell you how to do one either. Use the list as a sanity check at construction time to determine how solid the ground that you're standing on is–where you are on the requirements Richter scale.

Not all of the checklist questions will apply to your project. If you're working on an informal project, you'll find some that you don't even need to think about. You'll find others that you need to think about but don't need to answer formally. If you're working on a large, formal project, however, you may need to consider every one.

Specific Functional Requirements

- ❑ Are all the inputs to the system specified, including their source, accuracy, range of values, and frequency?
- ❑ Are all the outputs from the system specified, including their destination, accuracy, range of values, frequency, and format?
- ❑ Are all output formats specified for Web pages, reports, and so on?
- ❑ Are all the external hardware and software interfaces specified?
- ❑ Are all the external communication interfaces specified, including handshaking, error-checking, and communication protocols?
- ❑ Are all the tasks the user wants to perform specified?
- ❑ Is the data used in each task and the data resulting from each task specified?

Specific Nonfunctional (Quality) Requirements

- ❑ Is the expected response time, from the user's point of view, specified for all necessary operations?
- ❑ Are other timing considerations specified, such as processing time, data-transfer rate, and system throughput?
- ❑ Is the level of security specified?
- ❑ Is the reliability specified, including the consequences of software failure, the vital information that needs to be protected from failure, and the strategy for error detection and recovery?
- ❑ Are minimum machine memory and free disk space specified?
- ❑ Is the maintainability of the system specified, including its ability to adapt to changes in specific functionality, changes in the operating environment, and changes in its interfaces with other software?
- ❑ Is the definition of success included? Of failure?

Requirements Quality

- ❑ Are the requirements written in the user's language? Do the users think so?
- ❑ Does each requirement avoid conflicts with other requirements?
- ❑ Are acceptable tradeoffs between competing attributes specified–for example, between robustness and correctness?
- ❑ Do the requirements avoid specifying the design?
- ❑ Are the requirements at a fairly consistent level of detail? Should any requirement be specified in more detail? Should any requirement be specified in less detail?
- ❑ Are the requirements clear enough to be turned over to an independent group for construction and still be understood? Do the developers think so?
- ❑ Is each item relevant to the problem and its solution? Can each item be traced to its origin in the problem environment?
- ❑ Is each requirement testable? Will it be possible for independent testing to determine whether each requirement has been satisfied?
- ❑ Are all possible changes to the requirements specified, including the likelihood of each change?

Requirements Completeness

- ❑ Where information isn't available before development begins, are the areas of incompleteness specified?
- ❑ Are the requirements complete in the sense that if the product satisfies every requirement, it will be acceptable?
- ❑ Are you comfortable with all the requirements? Have you eliminated requirements that are impossible to implement and included just to appease your customer or your boss?

Handling Requirements Changes During Construction

Make sure everyone knows the cost of requirements changes

*Clients get excited when they think of a new feature but don't forget to mention the **Schedule** and **Cost** !*

Set up a change-control procedure

If your client's excitement persists, consider establishing a formal change control board to review such proposed changes. It's all right for customers to change their minds and to realize that they need more capabilities

Handling Requirements Changes During Construction

Use development approaches that accommodate changes

You can build a little, get a little feedback from your users, adjust your design a little, make a few changes, and build a little more.

The key is using short development cycles so that you can respond to your users quickly

Dump the project

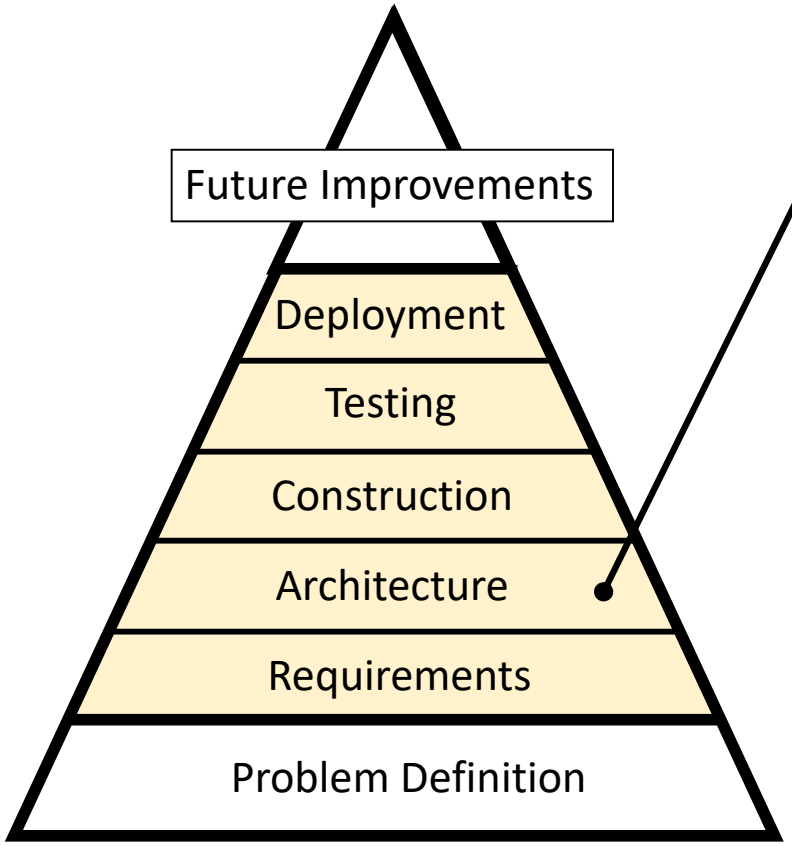
If the requirements are especially bad or volatile and none of the suggestions above are workable, cancel the project.

Handling Requirements Changes During Construction

Keep your eye on the business case for the project

Many requirements issues disappear before your eyes when you refer back to the business reason for doing the project

Architecture



Software architecture is the high-level part of software design, the frame that holds the more detailed parts of the design

a.k.a "system architecture," "high-level design," and "top-level design."

Typically, the architecture is described in a single document referred to as the "*architecture specification*" or "*top-level design*."

The quality of the architecture determines the conceptual integrity of the system. → determines the ultimate quality of the system

Architecture

Good architecture makes construction easy. Bad architecture makes construction almost impossible

A well-thought-out architecture

- *provides* the structure needed to maintain a system's conceptual integrity from the top levels down to the bottom.
- *provides* guidance to programmers—at a level of detail appropriate to the skills of the programmers and to the job at hand.
- *partitions* the work so that multiple developers or multiple development teams can work independently.

Without good software architecture, you may have the right problem but the wrong solution. It may be impossible to have successful construction.

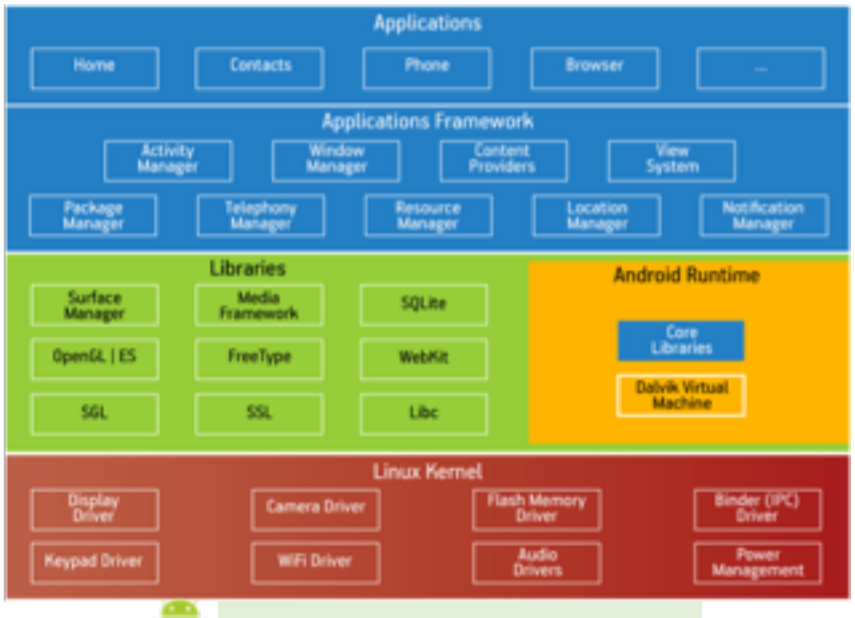
Architecture changes are like requirements changes in that seemingly **small changes** can be **far-reaching**

Typical Architectural Components: **Program Organization**

A system architecture first needs an overview that describes the system in broad terms

Without an overview, you won't understand how a class you're developing contributes to the system

12-Piece Jigsaw Puzzle



Android System Architecture

Typical Architectural Components: **Program Organization**

In the architecture, you should find evidence that **alternatives to the final organization** were considered and *find the reasons for choosing the final organization over its alternatives*.

Typical Architectural Components: **Program Organization**

The architecture should define the major building blocks.

Depending on the program size, each building block might be a single class or a subsystem consisting of many classes

What each building block is responsible for should be well defined.

A building block should have one area of responsibility, and it should know as little as possible about other building blocks' areas of responsibility

The communication rules for each building block should be well defined.

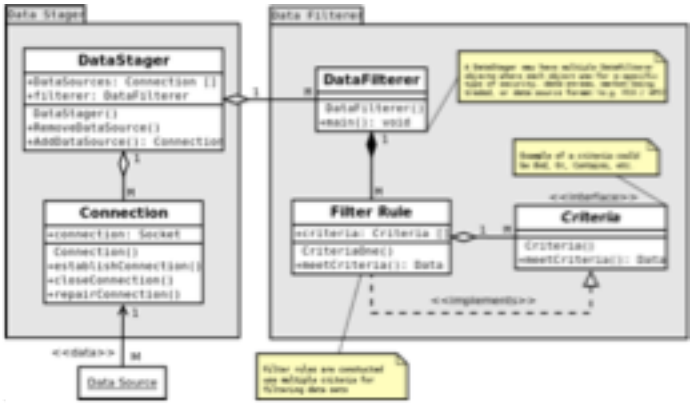
Typical Architectural Components: **Major Classes**

The architecture should:

- *specify* the major classes to be used.
- *identify* the responsibilities of each major class and how the class will interact with other classes.
- *include* descriptions of the class hierarchies, of state transitions, and of object persistence.
- If the system is large enough, *describe* how classes are organized into subsystems.

The architecture **doesn't need** to specify every class in the system. Aim for the 80/20 rule:

"Specify the 20 percent of the classes that make up 80 percent of the system's behavior"



Typical Architectural Components: **Data Design**

The architecture should *describe the major files and table designs to be used* & describe alternatives that were considered and justify the choices that were made. (e.g., database vs flat files)

Data should normally be accessed directly by only one subsystem or class, except through access classes or routines that allow access to the data in controlled and abstract ways.

Information Hiding

Typical Architectural Components: **Business Rules**

If the architecture depends on specific business rules, it should identify them and describe the impact the rules have on the system's design.

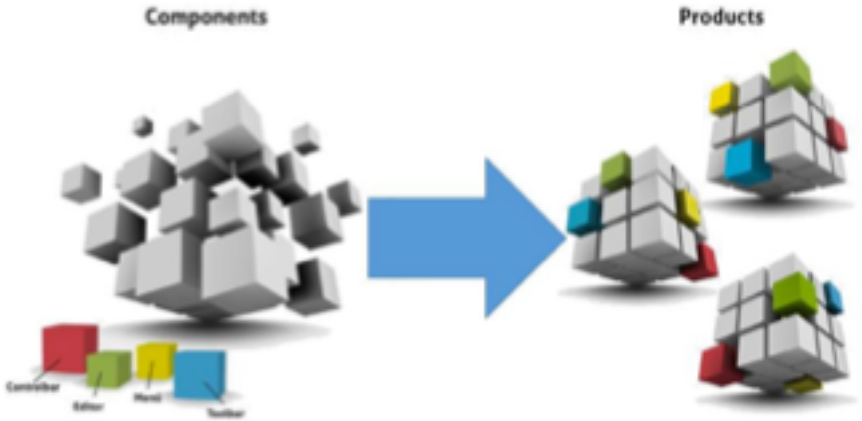
Suppose the system is required to follow a business rule that customer information should be no more than 30 seconds out of date

In that case, the impact that rule has on the architecture's approach to keeping customer information up to date and synchronized should be described.

Typical Architectural Components: **User Interface Design**

The user interface is often specified at requirements time. If it isn't, it should be specified in the **software architecture**. The architecture should specify major elements of Web page formats, GUIs, command line interfaces, and so on.

The architecture should be modularized so that a new user interface can be substituted without affecting the business rules and output parts of the program



Typical Architectural Components: **Resource Management**

The architecture should describe a plan for managing **scarce resources** such as database connections, threads, and handles.

Memory management is another important area for the architecture to treat in memory-constrained applications areas such as driver development and embedded systems.

Typical Architectural Components: **Security**

The architecture should describe the approach to design-level and code level security.

If a threat model has not previously been built, it should be built at architecture time.

Typical Architectural Components: **Performance**

If performance is a concern, performance goals should be specified in the requirements.

Performance goals can include resource usage, in which case the goals should also specify priorities among resources, including speed vs. memory vs. cost.

Typical Architectural Components: **Scalability**

The ability of a system to grow to meet future demands.

The architecture should describe how the system will address growth in number of users, servers, network nodes, database records, and size of database records, transaction volume etc.

If the system is not expected to grow and scalability is not an issue, the architecture should make that assumption explicit.

Typical Architectural Components: **Interoperability**

If the system is expected to share data or resources with other software or hardware, the architecture should describe how that will be accomplished.

Typical Architectural Components: **Internationalization/Localization**

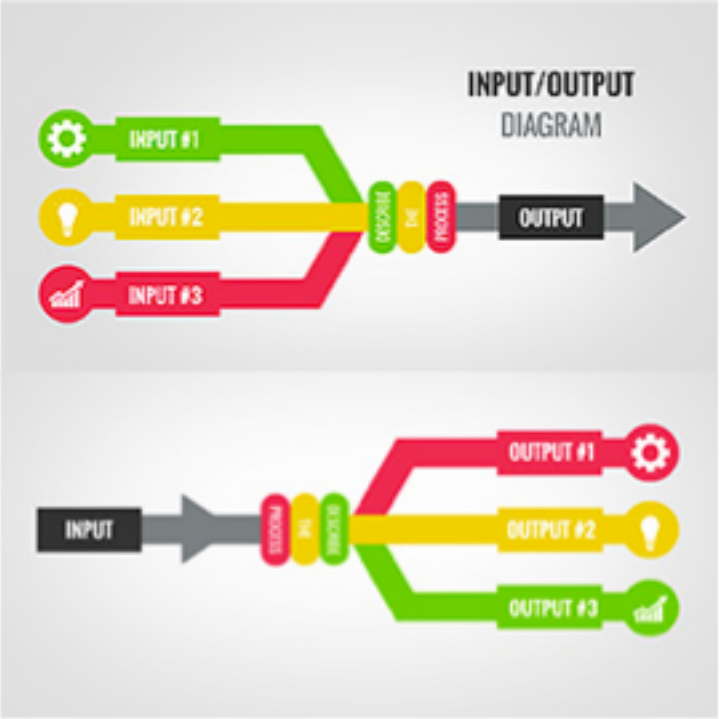
"**Internationalization**" is the technical activity of preparing a program to support multiple locales. Internationalization is often known as "I18n"

"**Localization**" (known as "L10n") is the activity of translating a program to support a specific local language.

The architecture can decide to use strings in line in the code where they're needed, keep the strings in a class and reference them through the class interface, or store the strings in a resource file. The architecture should explain which option was chosen and why.

Typical Architectural Components: **Input / Output**

The architecture should specify a look-ahead, look-behind, or just in-time reading scheme. And it should describe the level at which I/O errors are detected: at the field, record, stream, or file level.



Typical Architectural Components: **Error Processing**

Some people have estimated that as much as 90 percent of a program's code is written for exceptional, error processing cases or housekeeping, implying that only 10 percent is written for nominal cases

Error handling is often treated as a coding-convention-level issue, if it's treated at all. But because it has system wide implications, it is best treated at the architectural level.

Some Questions to Consider:

Is error processing corrective or merely detective?	Is error detection active or passive?	How does the program propagate errors?
What are the conventions for handling error messages?	How will exceptions be handled?	Inside the program, at what level are errors handled?
What is the level of responsibility of each class for validating its input data?	Do you want to use your environment's built-in exception-handling mechanism or build your own?	

Typical Architectural Components: **Fault Tolerance**

Fault tolerance is a collection of techniques that increase a **system's reliability** by detecting errors, recovering from them if possible, and containing their bad effects if not.

Typical Architectural Components: **Architectural Feasibility**

The designers might have concerns about a system's ability to meet its performance targets, work within resource limitations, or be adequately supported by the implementation environments.

The architecture should demonstrate that the system is technically feasible. If infeasibility in any area could render the project unworkable, the architecture should indicate how those issues have been investigated.

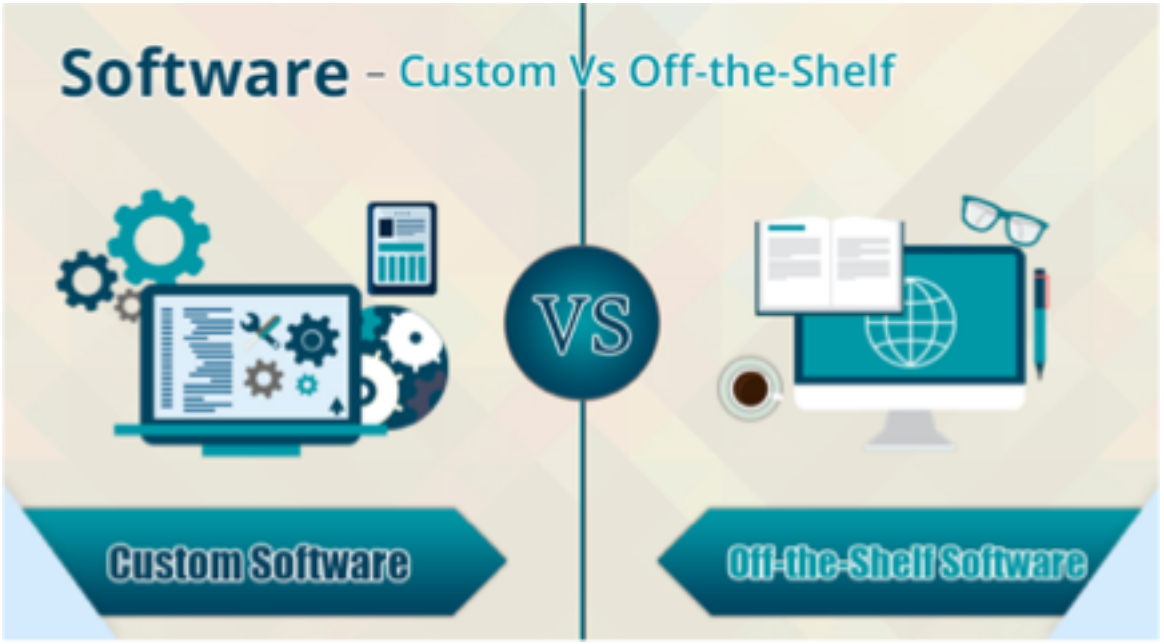
Typical Architectural Components: **Overengineering**

The architecture should clearly indicate whether programmers should **err** on the side of overengineering or on the side of doing the simplest thing that works

Typical Architectural Components: **Buy-vs.-Build Decisions**

The most radical solution to building software is not to build it at all—to buy it instead or to download open-source software for free.

If the architecture isn't using off-the-shelf components, it should explain the ways in which it expects custom-built components to surpass ready made libraries and components.



Typical Architectural Components: **Reuse Decisions**

If the plan calls for using preexisting software, test cases, data formats, or other materials, the architecture should explain how the reused software will be made to conform to the other architectural goals

Typical Architectural Components: **Change Strategy**

Because building a software product is a learning process for both the programmers and the users, the product is likely to change throughout its development.

One of the major challenges facing a software architect is making the architecture flexible enough to accommodate likely changes.

The architecture should clearly describe a strategy for handling changes.

Typical Architectural Components: **General Architectural Quality**

A good architecture should fit the problem. When you look at the architecture, you should be pleased by how natural and easy the solution seems. It shouldn't look as if the problem and the architecture have been forced together with duct tape

The Mythical Man-Month (Brooks 1995)

The architecture's objectives should be clearly stated.

The architecture should describe the motivations for all major decisions.

Good software architecture is largely machine- and language-independent.

The architecture should explicitly identify risky areas

The architecture should tread the line between underspecifying and overspecifying the system.

The architecture should contain multiple views.

Checklist: Architecture

Here's a list of issues that a good architecture should address. The list isn't intended to be a comprehensive guide to architecture but to be a pragmatic way of evaluating the nutritional content of what you get at the programmer's end of the software food chain. Use this checklist as a starting point for your own checklist. As with the requirements checklist, if you're working on an informal project, you'll find some items that you don't even need to think about. If you're working on a larger project, most of the items will be useful.

Specific Architectural Topics

- ❑ Is the overall organization of the program clear, including a good architectural overview and justification?
- ❑ Are major building blocks well defined, including their areas of responsibility and their interfaces to other building blocks?
- ❑ Are all the functions listed in the requirements covered sensibly, by neither too many nor too few building blocks?
- ❑ Are the most critical classes described and justified?
- ❑ Is the data design described and justified?
- ❑ Is the database organization and content specified?
- ❑ Are all key business rules identified and their impact on the system described?
- ❑ Is a strategy for the user interface design described?
- ❑ Is the user interface modularized so that changes in it won't affect the rest of the program?
- ❑ Is a strategy for handling I/O described and justified?
- ❑ Are resource-use estimates and a strategy for resource management described and justified for scarce resources like threads, database connections, handles, network bandwidth, and so on?
- ❑ Are the architecture's security requirements described?
- ❑ Does the architecture set space and speed budgets for each class, subsystem, or functionality area?
- ❑ Does the architecture describe how scalability will be achieved?
- ❑ Does the architecture address interoperability?
- ❑ Is a strategy for internationalization/localization described?
- ❑ Is a coherent error-handling strategy provided?
- ❑ Is the approach to fault tolerance defined (if any is needed)?
- ❑ Has technical feasibility of all parts of the system been established?
- ❑ Is an approach to overengineering specified?
- ❑ Are necessary buy-vs.-build decisions included?
- ❑ Does the architecture describe how reused code will be made to conform to other architectural objectives?
- ❑ Is the architecture designed to accommodate likely changes?

General Architectural Quality

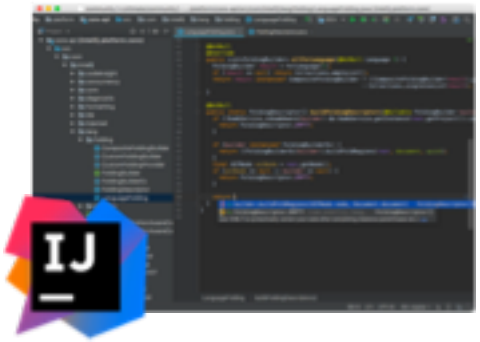
- ❑ Does the architecture account for all the requirements?
- ❑ Is any part overarchitected or underarchitected? Are expectations in this area set out explicitly?
- ❑ Does the whole architecture hang together conceptually?
- ❑ Is the top-level design independent of the machine and language that will be used to implement it?
- ❑ Are the motivations for all major decisions provided?
- ❑ Are you, as a programmer who will implement the system, comfortable with the architecture?

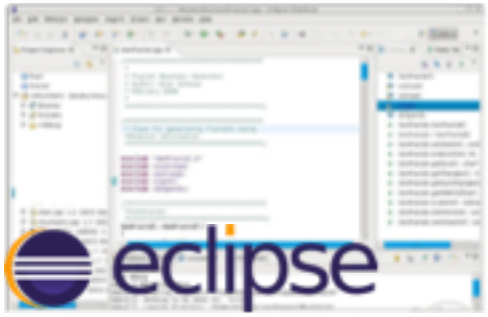
SUMMARY

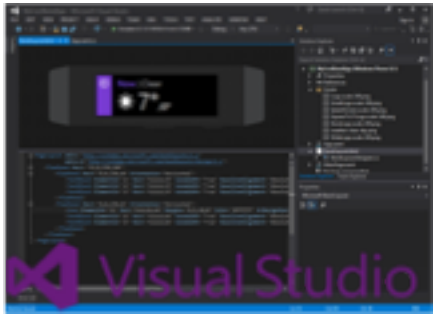
- The overarching goal of preparing for construction is risk reduction.
- If you want to develop high-quality software, attention to quality must be part of the software-development process from the beginning to the end
- Part of a programmer's job is to educate bosses and coworkers about the software- development process, including the importance of adequate preparation before programming begins.
- The kind of project you're working on significantly affects construction prerequisites
- If a good problem definition hasn't been specified, you might be solving the wrong problem during construction.
- If good requirements work hasn't been done, you might have missed important details of the problem
- If a good architectural design hasn't been done, you might be solving the right problem the wrong way during construction.
- Understand what approach has been taken to the construction prerequisites on your project, and choose your construction approach accordingly.

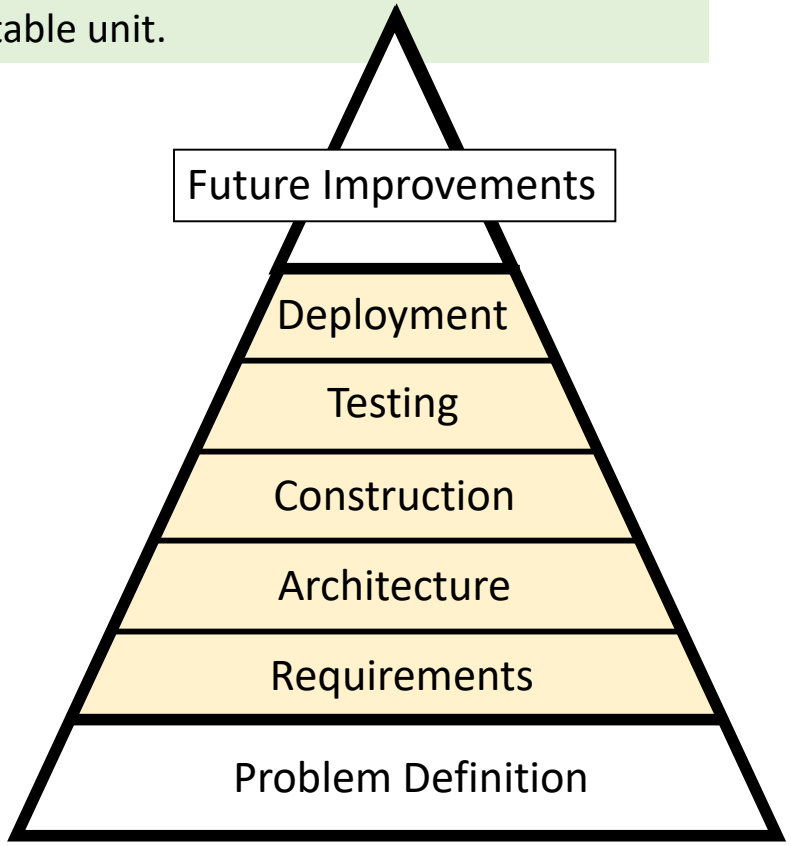
Construction

Software is usually designed and created (coded/written/programmed) in integrated development environments (IDE) like Eclipse, Xcode or Microsoft Visual Studio that can simplify the process and compile the program to an executable unit.









Key Construction Decisions

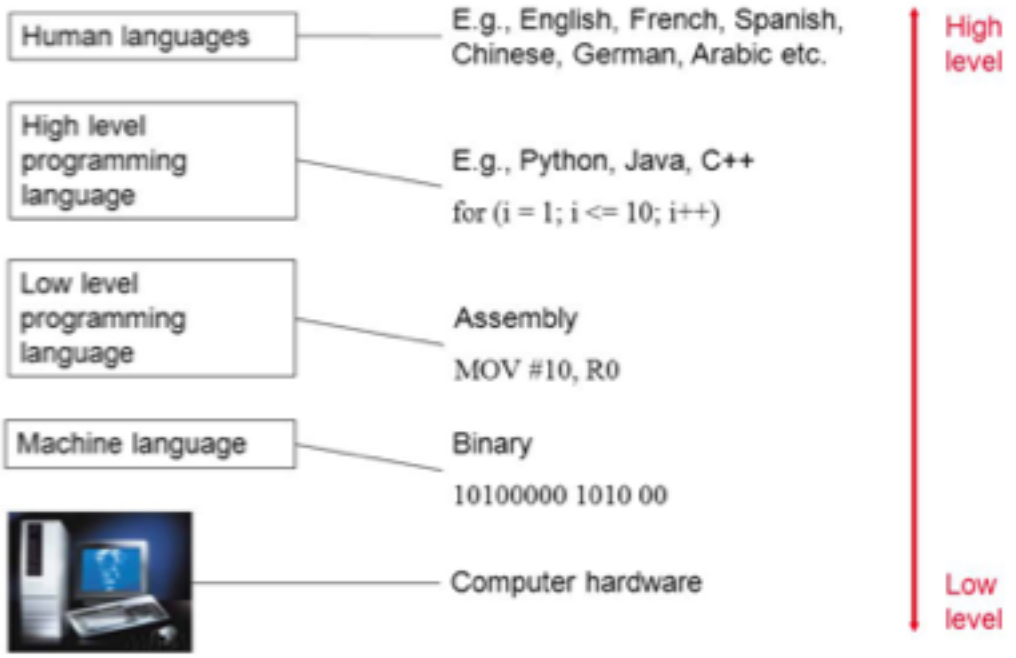
Choice of Programming Language	Programming Conventions	Your Location on the Technology Wave	Selection of Major Construction Practices

Choice of Programming Language

Studies have shown that the programming-language choice affects **productivity** and **code quality** in several ways

Programmers are **more productive** using a familiar language than an unfamiliar one

Programmers working with **high-level languages** achieve better productivity and quality than those working with *lower-level languages*.



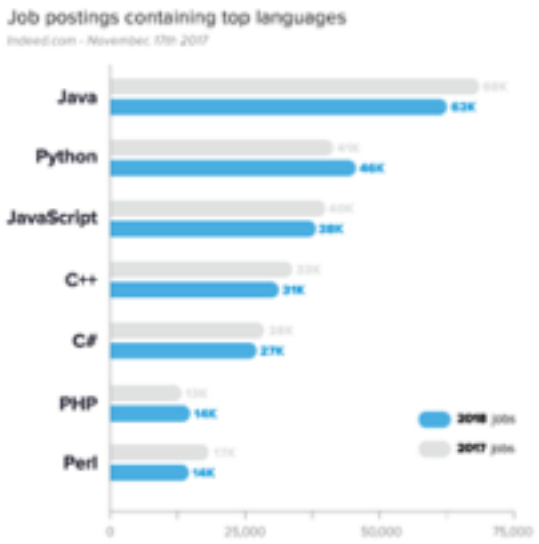
Choice of Programming Language

Ratio of High-Level-Language Statements to Equivalent C Code

Language	Level Relative to C
C	1
C++	2.5
Fortran 95	2
Java	2.5
Perl	6
Python	6
Smalltalk	6
Microsoft Visual Basic	4.5

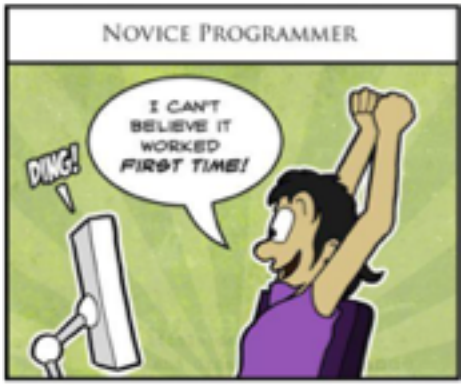
Source: Adapted from *Estimating Software Costs* (Jones 1998), *Software Cost Estimation with Cocomo II* (Boehm 2000), and "An Empirical Comparison of Seven Programming Languages" (Prechelt 2000).

Choice of Programming Language



Kind of Program	Best Languages	Worst Languages
Command-line processing	Cobol, Fortran, SQL	-
Cross-platform development	Java, Perl, Python	Assembler, C#, Visual Basic
Database manipulation	SQL, Visual Basic	Assembler, C
Direct memory manipulation	Assembler, C, C++	C#, Java, Visual Basic
Distributed system	C#, Java	-
Dynamic memory use	C, C++, Java	-
Easy-to-maintain program	C++, Java, Visual Basic	Assembler, Perl
Fast execution	Assembler, C, C++, Visual Basic	JavaScript, Perl, Python
For environments with limited memory	Assembler, C	C#, Java, Visual Basic
Mathematical calculation	Fortran	Assembler
Quick-and-dirty project	Perl, PHP, Python, Visual Basic	Assembler
Real-time program	C, C++, Assembler	C#, Java, Python, Perl, Visual Basic
Report writing	Cobol, Perl, Visual Basic	Assembler, Java
Secure program	C#, Java	C, C++
String manipulation	Perl, Python	C
Web development	C#, Java, JavaScript, PHP, Visual Basic	Assembler, C

Some languages simply don't support certain kinds of programs, and those have not been listed as "worst" languages. For example, Perl is not listed as a "worst language" for mathematical calculations.



Mother Tongues

Tracing the roots of computer languages through the ages

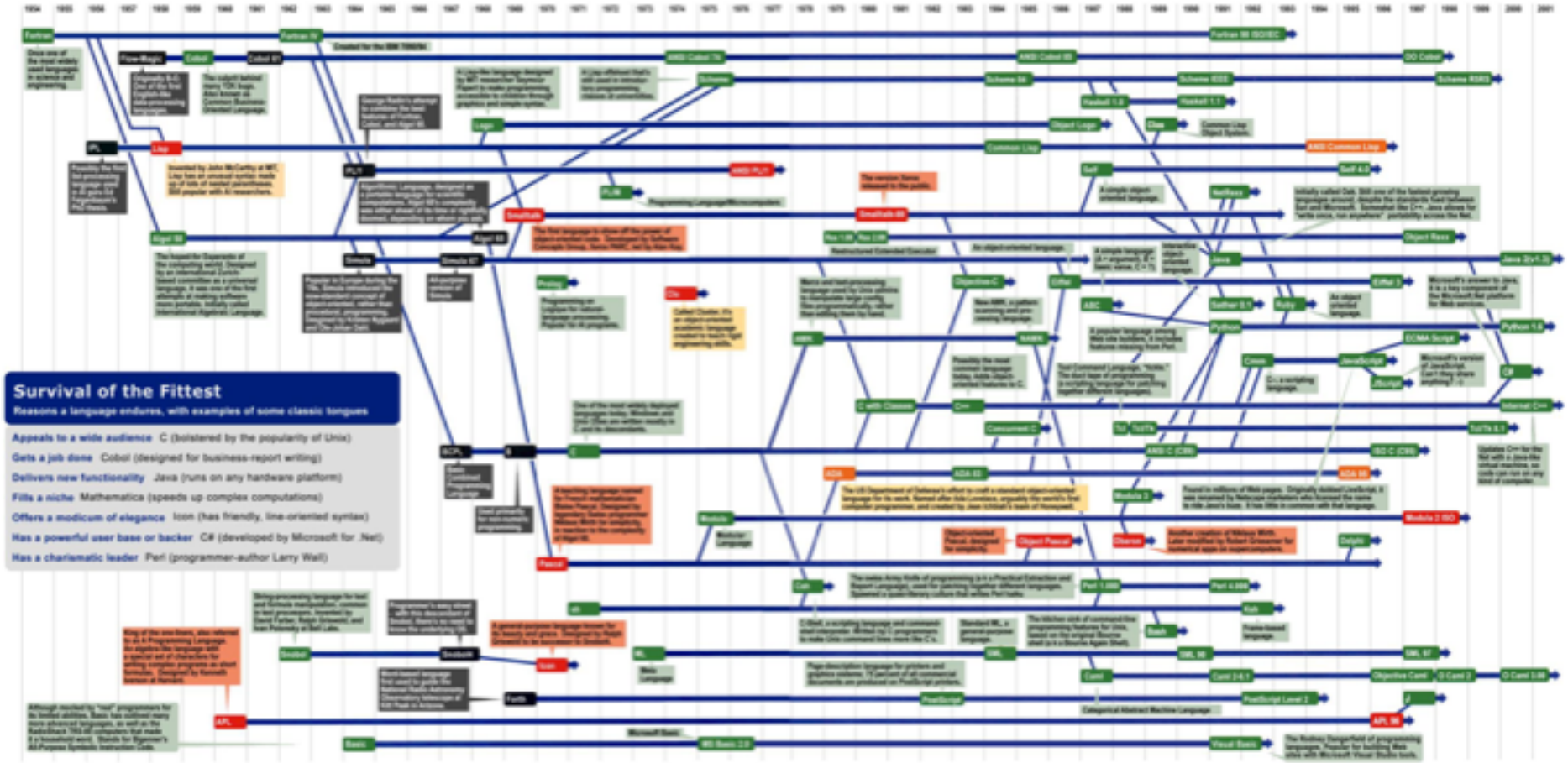
Just like half of the world's spoken tongues, most of the 2,300-plus computer programming languages are either endangered or extinct. As powerhouses C/C++, Visual Basic, Cobol, Java and other modern source codes dominate our systems, hundreds of older languages are running out of life.

An ad hoc collection of engineers-electronic lexicographers, if you will-aim to save, or at least document the lingo of classic software. They're combing the globe's 9 million developers in search of coders still fluent in these nearly forgotten lingua frangas. Among the most endangered are Ada, APL, B (the predecessor of C), Lisp, Oberon, Smalltalk, and Simula.

Code-raker Grady Booch, Rational Software's chief scientist, is working with the Computer History Museum in Silicon Valley to record and, in some cases, maintain languages by writing new compilers so our ever-changing hardware can grok the code. Why bother? "They tell us about the state of software practice, the minds of their inventors, and the technical, social, and economic forces that shaped history at the time," Booch explains. "They'll provide the raw material for software archaeologists, historians, and developers to learn what worked, what was brilliant, and what was an utter failure." Here's a peek at the strongest branches of programming's family tree. For a nearly exhaustive rundown, check out the Language List at *HTTP://www.informatik.uni-freiburg.de/Java/misc/lang_list.html*. - Michael Mendeno

Key

- 1954 Year introduced
- **Active:** thousands of users
- **Protected:** taught at universities; compilers available
- **Endangered:** usage dropping off
- **Extinct:** no known active users or up-to-date compilers
- Lineage continues



Sources: Paul Boutin; Brent Hailpern, associate director of computer science at IBM Research; The Retrocomputing Museum; Todd Proebsting, senior researcher at Microsoft; Gio Wiederhold, computer scientist, Stanford University

Programming Conventions

In high-quality software, you can see a relationship between the conceptual integrity of the architecture and its low-level implementation.

That's the point of construction guidelines for variable names, class names, routine names, formatting conventions, and commenting conventions.

Without a unifying discipline, your creation will be a jumble of sloppy variations in style. Such variations tax your brain—and only for the sake of understanding coding-style differences that are essentially arbitrary.

http://checkstyle.sourceforge.net/

http://checkstyle.org/eclipse-cs/

Before construction begins, spell out the programming conventions you'll use. Coding convention details are at such a level of precision that they're nearly impossible to retrofit into software after it's written.

Programming into a Language

Programmers who program **“in”** a language limit their thoughts to constructs that the language directly supports. If the language tools are primitive, the programmer’s thoughts will also be primitive.

Programmers who program **“into”** a language first decide what thoughts they want to express, and then they determine how to express those thoughts using the tools provided by their specific language.

Selection of Major Construction Practices

Checklist: Major Construction Practices

Coding

- ❑ Have you defined how much design will be done up front and how much will be done at the keyboard, while the code is being written?
- ❑ Have you defined coding conventions for names, comments, and layout?
- ❑ Have you defined specific coding practices that are implied by the architecture, such as how error conditions will be handled, how security will be addressed, what conventions will be used for class interfaces, what standards will apply to reused code, how much to consider performance while coding, and so on?
- ❑ Have you identified your location on the technology wave and adjusted your approach to match? If necessary, have you identified how you will program *into* the language rather than being limited by programming *in* it?

Teamwork

- ❑ Have you defined an integration procedure—that is, have you defined the specific steps a programmer must go through before checking code into the master sources?
- ❑ Will programmers program in pairs, or individually, or some combination of the two?

Quality Assurance

- ❑ Will programmers write test cases for their code before writing the code itself?
- ❑ Will programmers write unit tests for their code regardless of whether they write them first or last?
- ❑ Will programmers step through their code in the debugger before they check it in?
- ❑ Will programmers integration-test their code before they check it in?
- ❑ Will programmers review or inspect each other's code?

Tools

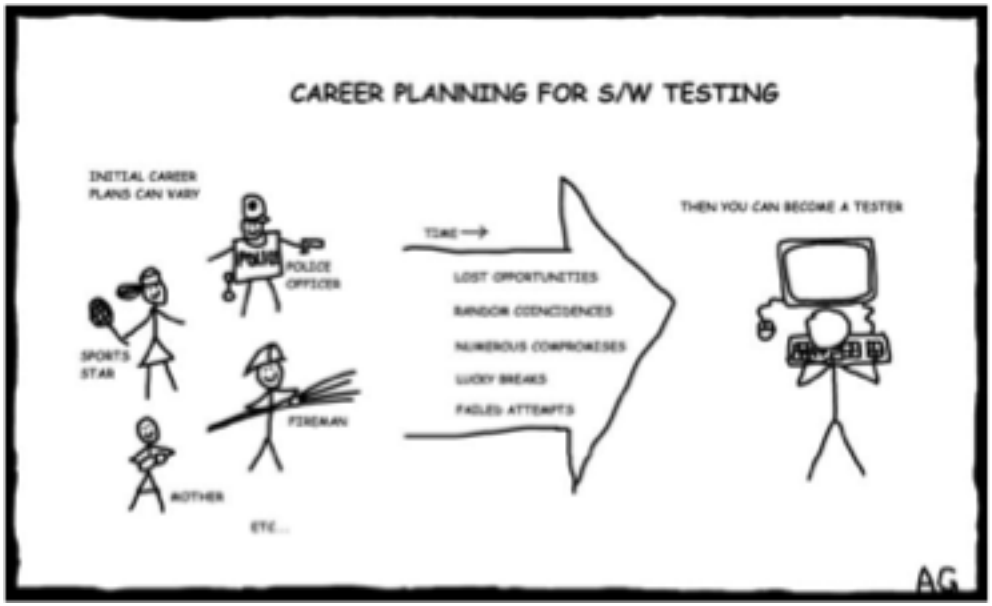
- ❑ Have you selected a revision control tool?
- ❑ Have you selected a language and language version or compiler version?
- ❑ Have you selected a framework such as J2EE or Microsoft .NET or explicitly decided not to use a framework?
- ❑ Have you decided whether to allow use of nonstandard language features?
- ❑ Have you identified and acquired other tools you'll be using—editor, refactoring tool, debugger, test framework, syntax checker, and so on?

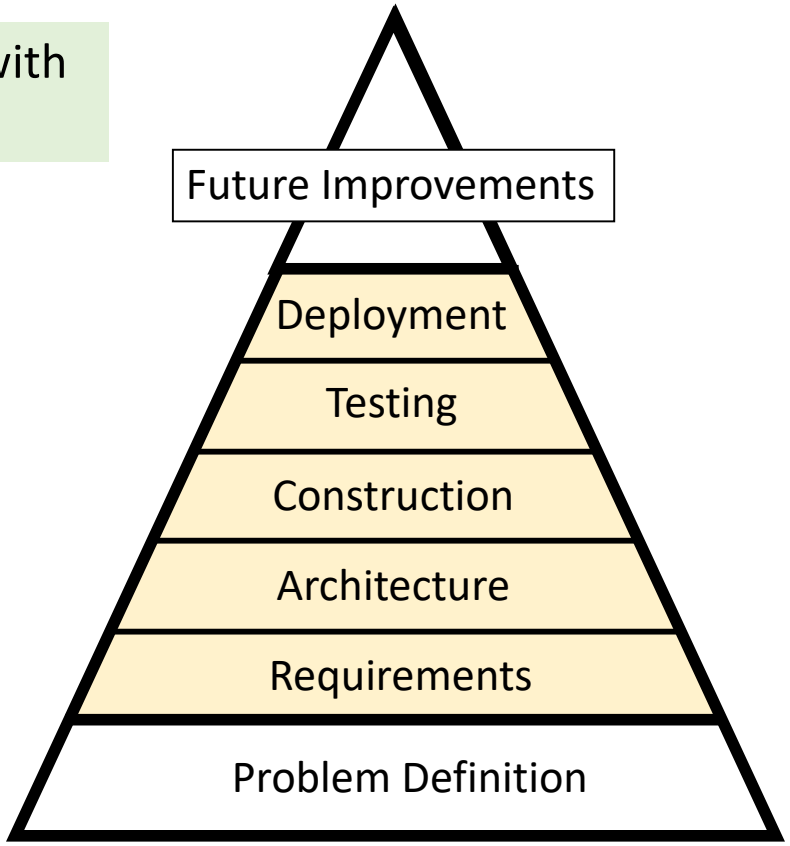
SUMMARY

- Every programming language has strengths and weaknesses. Be aware of the specific strengths and weaknesses of the language you're using.
- Establish programming conventions before you begin programming. It's nearly impossible to change code to match them later.
- More construction practices exist than you can use on any single project. Consciously choose the practices that are best suited to your project.
- Ask yourself whether the programming practices you're using are a response to the programming language you're using or controlled by it. Remember to program into the language, rather than programming in it.
- Your position on the technology wave determines what approaches will be effective— or even possible. Identify where you are on the technology wave, and adjust your plans and expectations accordingly

Testing

A process of executing a program or application with the intent of finding the software bugs





Testing

Consider different test levels:
Unit, Integration, System, Acceptance, and Regression

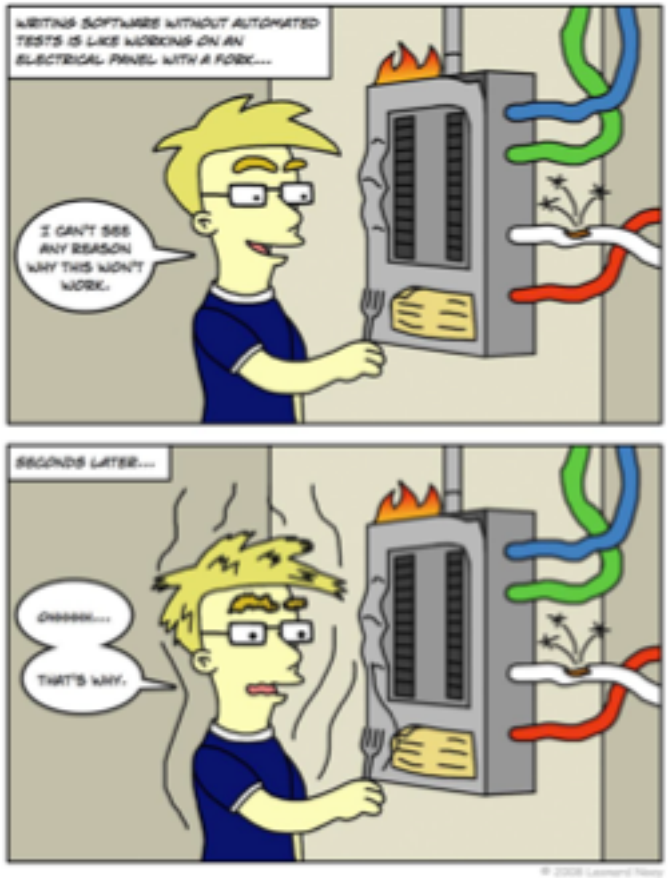
Testing of a single function, procedure, class.

Ensures that the whole system works properly

Checks that units tested in isolation work properly when put together

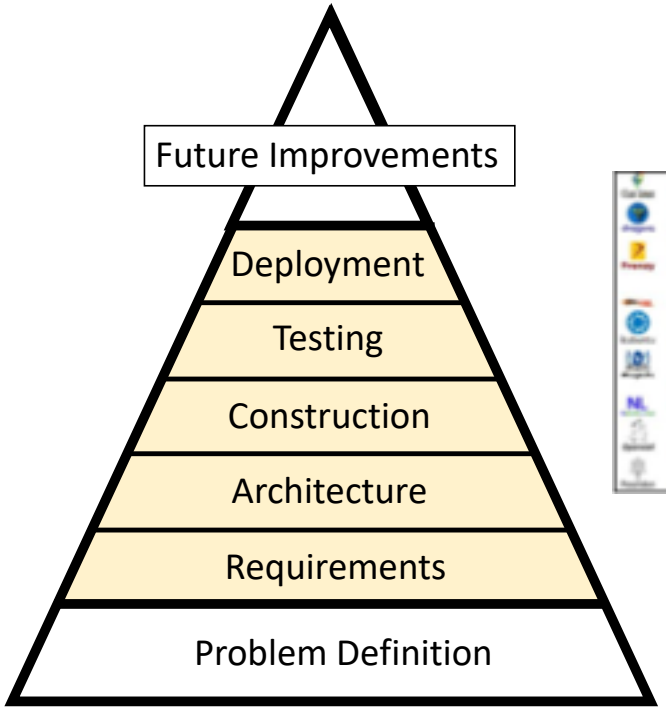
Checks if the user functionalities are delivered. It is often a contractual prerequisite for the user to accept/pay for the software

Checks that the system preserves its functionality after maintenance and/or evolution



Deployment

Software deployment is all the activities that make a software system available for use



Hundreds of OS based on Linux

つづく