2. Motivation and Life Cycles

Last Mod: 2024-04-29

Section Topics:

- To provide “engineering process” motivation, by examining the challenges of software development.
- Many life cycle phases of a software system.
  - More to it than ‘code it and throw it to the customers.
- Different models that can be used to plan your system development process.
- Mention some software process and quality standards.

- Finally, a list of some of the agreed:
  - worst practices to avoid and
  - best advice to embrace

  during software development.

NOTE:

To prepare for the first assignment we must start covering Requirements gathering and analysis very soon.

- So, some of this section's slides will not be covered in class.
- Instead, they are 'required readings' located in four appendices of these lecture notes.

Table of Contents

2.1 Required Readings

These are located near the end of this document.

- Appendix A – Productivity.
- Appendix B – Saving Money.
- Appendix C – Software Development Standards
- Appendix D – Other Lists of Good Practices.

2.2 Optional Readings

- "Capability Maturity Model, Version 1.1" by Mark C. Paulk et al, IEEE Software Magazine, July, 1993.
- "Process Improvement and the Corporate Balance Sheet" by Raymond Dion, IEEE Software Magazine, July 1993.

2.3 Motivation

- In the 1970's, software became a major portion of the effort and expense in developing and maintaining automated systems for
  - factories,
  - air traffic control systems,
  - aircraft avionics,
  - financial systems,
  - accounting and inventory systems, etc.
- Previously, the major expense was the computer hardware (few were under $1M!).

- As computer hardware came down in price, and software systems got huge and intricate, the majority of cost became the software. E.g.
  - Microsoft spent $150,000,000 developing Windows NT OS for a $2000 computer. Though shrink wrap, high volume software is cheap ($100-$5000), developing it, or any custom software is not!
- And, developed software is usually buggy, late, and over-budget.
  - Windows NT used 200 developers and testers, initially comprised 4.3 million lines of code, and was 6 months late!

- Surprisingly, even though custom software is developed for a specific purpose, it is not always well suited to its intended purpose!
  - Difficult getting customers to carefully express their exact needs.
  - They don't appreciate or understand how intricate the software will need to be.
  - And, because you (the developer) probably don't know the application area ('domain') very well, nor can you read their minds!

At a NATO conference in the ‘60s the terms ‘Software Crisis’ and ‘Software Engineering’ were coined to describe this overall situation.

- And address it: Apply some engineering development principles and practices to software development.

The Software Crisis:

- Advances in hardware sophistication were outpacing our ability to develop software which could employ the hardware's potential.
  - E.g. Before even Microsoft fully exploited the features that the Intel 80286 offered, the 80386 was already out! This is partly due to:

- And our ability to enhance and maintain software was also threatened by:
  - poor inflexible designs,
  - poor documentation,
  - poor software development tools, and
  - lack of resources (including money, time, good software engineers).

Our programming productivity can't cope with either:

- demand for volume of programming needed,
- nor the complexity of the systems to be developed (humans not good with complexity).

The result was:

- late schedule
- over budget
- low productivity (thus high cost)
- poor quality.

2.4 Developing Software 'Products'

If you develop a software product for yourself, it's likely suitable as it is written:

- by its user
- for its user, and
- is easily subsequently maintained/tailored by 'original' author.

In contrast, a software ‘product’ is:

- written by programmers often un-knowledgeable about the ‘application domain’,
- to be used by many diverse users,
- who themselves cannot fix or improve the product when necessary.

Software products must be maintainable by other than just the original author.

- Original authors likely already working on next project, perhaps even in other cities.
- Or have left the company (15% attrition is not uncommon in Vancouver).

To develop a real software 'product', proper:

- requirements analysis,
- design and coding,
- documentation,
- and quality assurance (QA) testing

is needed.

How are we defining proper?
Are we following a specific set of guidelines?
Is proper not dependent on the project you are working with?

How do we know that three times the amount of effort is required?

This requires three times more work (per 1000 lines of code) than developing a program for yourself!

Even if you are working on something for yourself, it seems that requirement analysis and testing will always be involved. Does this mean that the design and documentation is taking up the bulk of the additional effort?

[Brooks95] suggests that when developing system programs (like program editors, compilers, and operating systems), a **further three times** increase in work is required! I.e. 3 x 3 = 9 times!

2.5 Maintenance Difficulties

Maintenance is the triple combination of:

- fixing problems.
- enhancing any aspect of a product.
- migrating or porting the product to run on a different computer, operating system (OS), network, or database system.

You are responsible for memorizing this triple.

Even maintenance is not done well.

Nor is it easy.

- Design architectures are brittle, and
- difficult to change without breaking something else.
- On average, for every 2 bugs you fix in a (big) system, you introduce one new one!
  - This is especially true if there is more than 50 lines of code changed.

Where is this number coming from?

2.6 Wild Software Projects

The software industry is:

- extremely competitive,
- fast changing in market and technology,
- and full of seemingly fickle, unhelpful customers who don't appreciate the detailed information needed to design software products.
  - They're busy doing their own jobs.

And your S/W dev company is always having to trade off schedule, budget, quality, product features, and people and technology resources.

It can get crazy.

Steve McConnell has written a book called "Rapid Development: Taming Wild Software Schedules" [McConnell96].

This indicates some work environments are not always that pleasant.

In [Rot89] a survey of 600 companies found that 35% had at least one development project totally out of control.

- Not just somewhat late or over budget, but 50-200% over schedule or budget (with 100% being common).
- Do you think the programmers on these projects were having fun?

I imagine not.

- Do you think they were getting weekends and evenings off?

2.7 Definition of Software Engineering

This leads me to a definition of software engineering that **you are responsible to memorize**:

"Software Engineering is the application of sound engineering and project management practices with the goal of

- increasing productivity,
- increasing development predictability, and
- and increasing software quality,

as measured over the entire lifecycle of a software system."

How do we measure quality?
How do we measure productivity? Is it dependent on the number of user stories a team can complete?

Engineers have for many decades worried about:

- whether different versions of complementary pieces would fit together (configuration management),
- whether designs need to be reviewed by other team members for quality (as single people are quite fallible),
- whether consumers could understand how to operate developed devices, etc.

Managing the development of software is a cross between development engineering and manufacturing engineering.

And project managers have developed ways of:

- estimating development duration and budget, and
- quantitatively monitoring projects as they progress (how many square meters of a building are complete so far).

What programmers must realize, is that software development and project management encompasses far more than just programming.

- And, if you want to get promoted and paid well, you'll need to master these skills.
  - E.g. Microsoft Project
  - Various "agile" project management software tools.

- And even to just ‘work’ in software shop, you’ll have to understand these issues.
  - E.g. You can’t even start modifying or writing code in a software company until you know how to check out source files from the source code configuration/revision/version management system!

2.8 Software Development Life Cycle (SDLC)

The Software Development Life Cycle includes these phases:

- Requirements gathering, analysis, and specification.
- Design,
- Implementation (Coding),
- Testing,
- Deployment, and
- Maintenance.

Why is testing not a part of the implementation phase? You are making many assumptions that the code which you are building off of is correct, without any proof of its success.

How much confidence can you have in your code when you have not tested it? Would it not be better to do incremental development by testing the code you have added?

For larger projects, is this manageable?

From my limited understanding of TDD, coding and testing is done together.

Defining the SDLC emphasizes that you can save money if using a bit more effort in the beginning phases will save more money later.

2.9 Development Models

There's a number of ways to manage the process of software development.

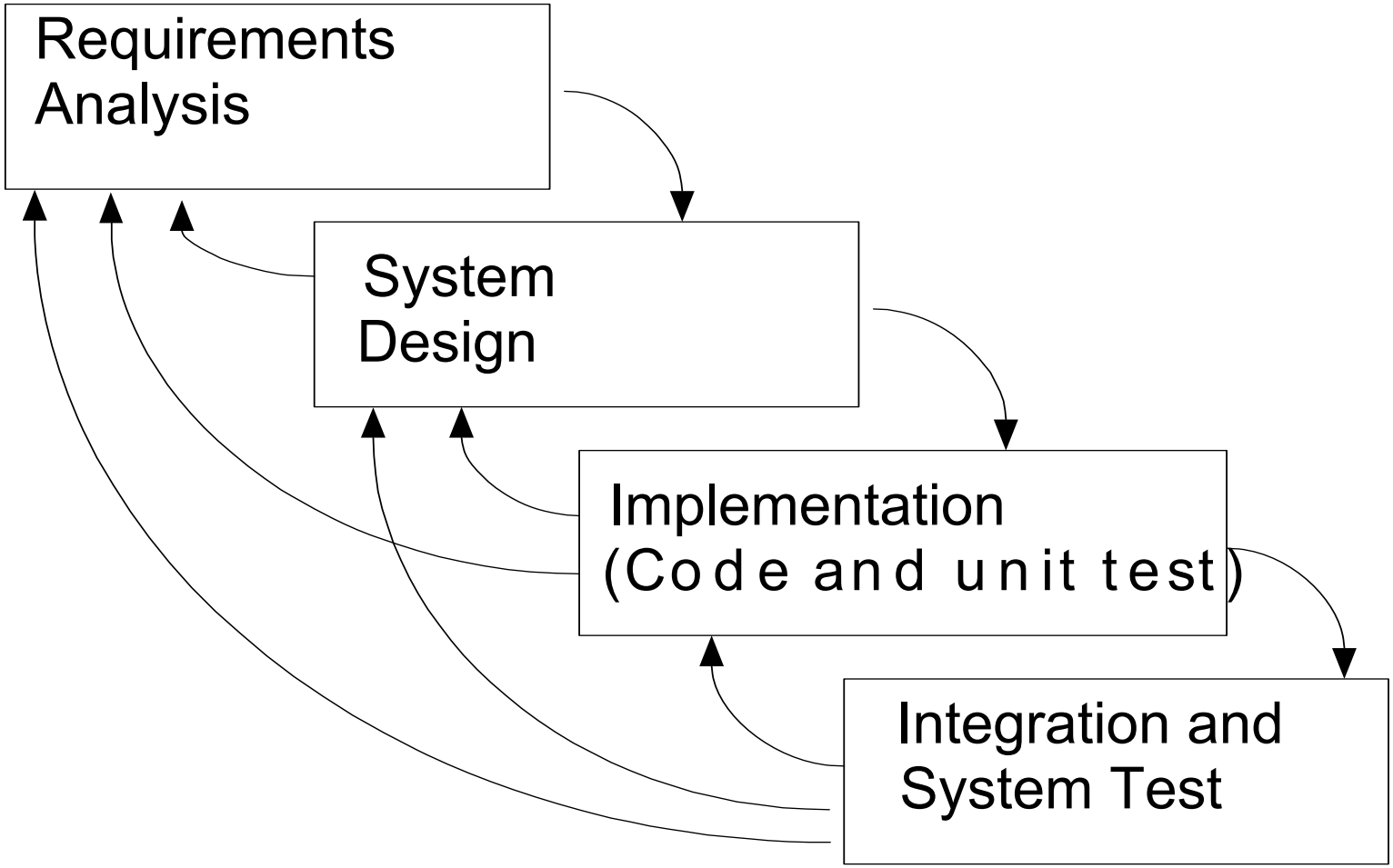
- Waterfall
- Rapid Prototype
- Iterative
- Spiral.

2.9.1 Waterfall Development Model

Waterfall model is used on many projects that start from scratch, possibly of an application for which there was not any previously automated system.

It is widely described as the ‘traditional’ waterfall model.

“Waterfall” name is from this diagram layout:



This diagram shows the 4 main phases of linear, waterfall development.

- Each phase feeds into the next.
- Next stage may find problems (inconsistencies, errors, missing information). This is *fed back* to a previous stage for revision.
- So don't waste time on a design, before finding a problem that may cause them to completely change their design
  - Peer reviews should be conducted at the end of every major phase.
  - Humans are poor at seeing problems in our own work, because we're too 'close' to it.
  - Just running the software may not find all the bugs. Who would you rather find your errors, your peers or your customers?

2.9.2 *Waterfall Sub-Phases and Deliverables.*

Requirements Gathering and Analysis:

- Deliverable = Requirements Specification Document

Project Planning:

- Deliverable = Project Plan describing project breakdown, schedule, effort, resources, budget, quality and risk factors and how they will be managed.

How do we measure effort?
Does this include a proof of concept?
What happens if there is no data which allows us to estimate the schedule?
Would we then apply forecasting?

Design in 4 parts:

- External (or User Interface) Design
  - Deliverable = draft user manual
  - Alternately called detailed requirements, or functional specification, with a suitably named deliverable.
- Architectural Design
  - Deliverable = Architectural Design Document
- Module Interface Design
  - Deliverable = module and exported function declarations
- Detailed Design
  - Deliverable = data structures and algorithms

Implementation (i.e. coding and unit testing)

- Deliverable = individually tested modules

Integration/System/Acceptance Testing and Fixing

- Deliverables =
  - test case reports,
  - remaining bug list,
  - Acceptance Tests signed by customer.

What is contained in the test case reports? is it all the tests that passed? is it a description of what is being tested?
Why do the acceptance tests need to be signed by the customer?

User Training/Installation/Cut-Over

- Deliverables =
  - Tutorials and videos,
  - example data files,
  - installation manual,
  - working system,
  - happy customers,
  - ongoing customer support and
  - customer database.

What does working mean?
Does it signify that all the requirements of the system are met with minimal maintenance?
Does it mean that no observable errors have been reported yet?

Corrective/Enhancement/Porting Maintenance

- Deliverables = new release of many of the above.

The Cmpt 276 project will require some of these deliverables.

2.9.3 ***Rapid Prototype Model***

In the rapid prototype dev model you build a 'somewhat working' prototype of what is needed.

- *This is very desirable when the user requirements are very unclear and you need to put something in front of the customers to stir up some controversy over the functionality needed, in order to actually understand their needs.*

In the 80's some tools were developed to allow you to rapidly wip together some of a user interface (UI), even though there was no real functionality behind it.

- Or you could do some 'quick and dirty' programming to accomplish the same thing.
- When you got the input on this prototype that you needed, you then threw it away.
- You threw it away, mainly because it was junky anyway, and you wanted to start the project with a solid architectural underpinning.

- The rapid prototype model has both faded and re-emerged. We now have GUI interface

development tools such as Visual Basic, C++ Builder, and Java Builder.

- Can rapidly create a bare bones GUI user interface.
- Whether functional or not, we can easily change it until the customer likes it.
- And then we can fill in the functionality.

2.9.4 Iterative Development Model

The iterative model is simply to develop (possibly using the waterfall model) a subset of the system, with some operational characteristics. This has several advantages:

- It provides early, possibly essential functionality to the users.
- It provides feedback from users on further requirements.
- It may provide cash to the developer because of it's minor functionality.

This, and the next model, have recently morphed into a number of so-called “Agile” development strategies. Names like:

- Extreme programming
- Scrum
- Kanban

We'll cover some of these later. But generally they are iterative in nature.

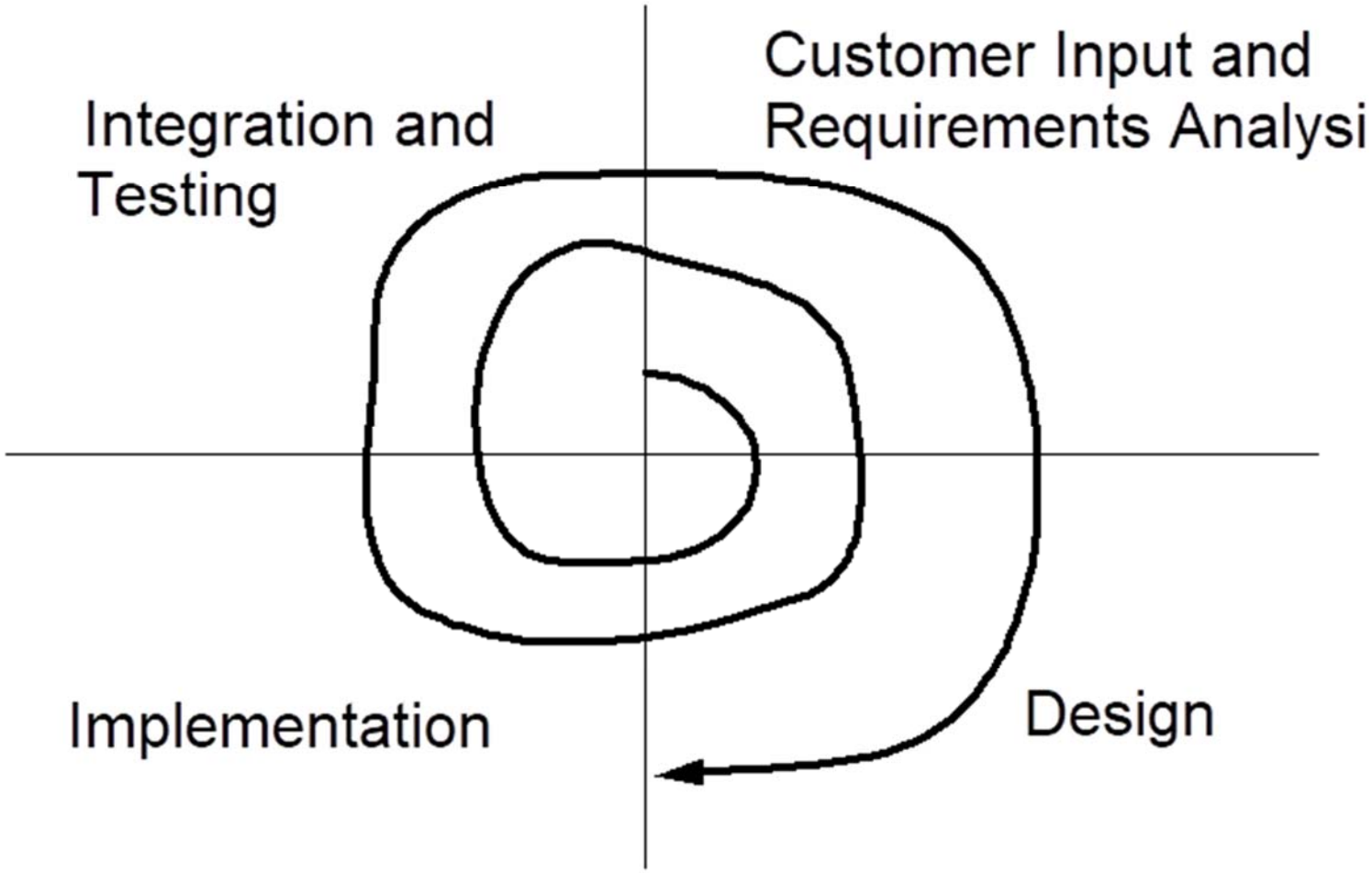
They're useful when:

- Difficult to establish/elicit requirements.
- Hard to estimate effort given:
  - Incomplete requirements mentioned above.
  - Unknown application domain for user.
  - Unknown application domain for developers.
  - Unknown effort for certain features or technologies.

- Useful when programmers only feel comfortable estimating effort in small steps.
  - Not useful if customer wants an upfront price quote.
  - However, nicely allows programmers to avoid huge promises of delivery effort and timeline.
    - And huge amounts of near-to-final-delivery overtime.

2.9.5 ***Spiral Model***

The spiral model is just a visual representation of the iterative model.



There are several variant models of the spiral, some with six sectors rather than four.

2.10 Other Best Practices and Principles

This section is composed of several short lists of valuable advice.

Except for this author's, the others are delegated to 'required readings' in an appendix.

2.10.1 Tront's 4 Principles

- KIS - Keep It Simple. Complexity management is major challenge in developing large software systems. Strive for:
  - Clean minimal interfaces between modules/procedures/data.
  - Minimize interfaces between people, otherwise there can be N(N-1)/2 lines of communication.

    Does this run the risk of increasing the probability of an error occuring?
  - Avoid fancy product functions and designs.
    - Especially those that can by synthesized from lower level primitives.
    - In [Brooks75] chapter called the Second System Effect, suggests (unrealistically) that managers avoid

developers who are working on their second major system.

- Such staff tend to put fancy features in due to confidence created by their first project.
- This happens even in Cmpt 276 if SFU coop students come back overconfident from coop!

What does it mean to not see what you are building?
Does this mean that the requirements are vague?

- Require Visibility. If you can't see what you are building, you:
  - can't measure progress and take management action if behind.
  - can't add new personnel without visible reading material for them.
  - can't review designs for appropriateness.
  - can't feed other departments (technical writing, graphics, purchasing) needed information until too late in the project.

Text

- Plan for Continuous Change
  - All manuals, designs, tests, source code should have revision numbers, dates, revision history comments, right margin change bars, etc.
  - New revisions should be approved before being made, and checked for quality and compliance after being made. Is this not the objective of a pull request?
  - Requirements will change whether your resist or not.
  - Variants for different customers, hardware, countries will be required.
    - Use a configuration management system and an automated build/make process.
  - Maintenance will be required.

- And keep track of which customers have which releases.

– Personnel will leave and be added. Document designs so the information doesn't walk out the door.

- Don't Under-Estimate:

What are person-months

  - Time - calendar time, effort in person-months, procurement time, overhead, meeting time, personnel training time, etc.
  - Effort - especially integration, testing, documentation, and maintenance needed.
  - Cost - of tools, management, customer and supplier interfacing, overhead, etc.
  - Amount of change.

It seems that with enough data on all of these points for various projects, a suitable estimate could be given. In your experience, how often did you have access to these statistics so as to make an informed guess?

2.11 Section Wrap-Up

To begin Assignment #1, we can't spend any more time now on the history and problems of software engineering.

But we'll return to most of these topics as we progress through the course and the assignments.

Also, very unfortunately, we don't have time to cover and do project planning now.

- This is just a fact of life in a 1-semester software engineering course.
- At UBC, they have one semester of lectures, and then another semester to work on the project.
- Lastly, students in Cmpt 276 sometimes complain about the lack of Computer Aided Software Engineering (CASE) tools for the course.
  - However, you can't begin use these until you understand the principles, and at least sketch or word process simple things (like a Release Table).

– And when we do give students good, sophisticated CASE tools in Cmpt 370, they complain about how long it takes them to learn them.

2.12 Other References

- [Brooks95] "The Mythical Man-Month, Anniversary Edition" by Frederick Brooks Jr., Addison-Wesley, 1995.
- [Dion93] "Process Improvement and the Corporate Balance Sheet" by Raymond Dion, IEEE Software Magazine, July 1993.
- [Joch95] "How Software Doesn't Work" by Alan Joch, Byte magazine, December, 1995.
- [Jones91] "Applied Software Measurement" by Capers Jones, McGraw-Hill, 1991.
- [Kehoe95] "ISO 9000-3: A Tool for Software Product and Process Improvement" by

Raymond Kehoe and Alka Jarvis, Springer-Verlag, 1996.

- [McConnell96] “Rapid Development: Taming Wild Software Schedules” by Steve McConnell, Microsoft Press, 1996.
- [Paulk93] “Capability Maturity Model, Version 1.1” by Mark C. Paulk et al, IEEE Software Magazine, July, 1993.
- [Paulk95] “The Capability Maturity Model: Guidelines for Improving the Software Process” by the Carnegie Mellon University Software Engineering Institute and edited by Mark C. Paulk et al, Addison-Wesley, 1995.

- [Pulford96] “A Quantitative Approach to Software Management: The AMI Handbook” by Kevin Pulford et al, Addison-Wesley, 1996.
- [Rot89] “It’s Late, Costly, Incompetent -- But Try Firing a Computer System” by J. Rothfeder, in “Tutorial: Software Risk Management” edited by Barry Boehm, IEEE Computer Society, 1989, pages 63-64.
- [Wiegers96] “Creating a Software Engineering Culture” by Karl Wiegers, Dorset House, 1996.
- [Wiegers98] Course notes from “Software Technical Reviews” put on by the Software Productivity Centre in Vancouver, April, 1998. Chart reproduced with Wiegers permission.

- [Yourdon92] “The Decline and Fall of the American Programmer” by Ed Yourdon, Prentice-Hall, 1992.
- [Yourdon96] “The Rise and Resurrection of the American Programmer” by Ed Yourdon, Prentice-Hall, 1996.

2.13 Appendix A - Productivity

- Current productivities range from 300 to 1000 lines of code (LOC) per programmer month.
  - This is only 15-45 lines per day.
- Assuming programmers get about $60K salary, this costs a company about 2.5 x $60K = $150K per year including overhead (space, heat, light, computer, benefits, someone to calculate the benefits, to supervise, to answer the phone, etc.). So each single line of code costs $8-25.
- So some Vancouver products which contain 1M LOC, have a development cost of over $10 million dollars!

- And systems products like Window NT cost about 3 times as much: in Canadian dollars, $48/LOC.

2.13.1 Industry Statistics

- [Jones91] contains a study of 4000 varied projects in the US. The projects varied from information systems to military avionics, from 1K to 1M LOC, from 1 to 200 in staff size, and from 1 month to 5 years of development duration. Please look carefully at each diagram.
- So independent of coding language, results are expressed in a language-independent size metric called 'function points'. We'll study function points later. But it's a unit of program functionality requiring about 100

high-level language lines of code, or 300 lines of assembly code.

Of significant interest is Jones':

- Figure 3.9 showing schedule overrun. Note this is a log-log graph, so 100 FP (i.e. 10K LOC) projects on average take 50% longer than planned, and 10,000 FP (i.e. 1M LOC) projects on average take 100% longer than planned!
- Figure 3.12 illustrates how programmer productivity plummets by a factor of 10 as project size grows!
- Figure 3.14 shows how risk of project failure/cancellation increases to 50% in large projects! (Is this a crisis or what?) In addition,

I have read elsewhere that when projects are cancelled, they have usually already spent 75% of their budget (but usually do not have 75% of the functionality complete)!

- Figure 3.23 shows that number of bugs delivered to customers per 100 LOC (bug density) grows from 0.1 to 1.0 (i.e. by a factor of 10) as you move from small to large projects!
- Figure 3.10 shows large projects seem to have 40% more features added to them than originally planned.
  - This partly explains why they are over budget and schedule. But also thoroughly indicts poor requirements specification!

- Figure 3.17 shows that a 1000 FP (100K LOC) system gets 10% more functionality added to it per year for the rest of its life!
  - Companies find it hard to develop new products after they have developed their first few. All their effort is spent maintaining their old products!
- Figure 3.15 shows that the larger the product, the longer you are going to have to keep adding this 10% extra functionality per year. It shows average product lifetime grows from 3 years for 10K LOC, to 12 years for 1M LOC.
- Have I convinced you there is a 'Software Crisis' yet?

2.14 Appendix B - Saving Money

Since software now sometimes 80% of the cost of a mixed hardware+software project, we should not spend a lot of time trying to reduce the cost of the hardware. Here are three important observations:

- Greater than 50% of oversights and bugs introduced during the requirements and design phases, not during coding!

- Spending 30-40% extra effort on good analysis and design can result in:
  - 60% fewer bugs, and therefore
  - far less debugging and testing effort,
  - a more well architected system having less lines of code (!), and
  - less total development effort (even though added effort was spent on analysis and design!).

- In addition, the resulting code is easier to maintain (partly due to better design documentation and code commenting, and partly due to a better, more flexible design).

Some of these savings are less pronounced on small projects. Nonetheless, small projects can benefit from judicious use of good software engineering as appropriate.

So the question is: In your $100,000 company project, are you willing to spend $10,000 on analysis and design now in order to:

- save $10,000 later during development and
- save $10,000 during maintenance and
- in good customer relations.

What is the value being saved in the customer relations?

These kinds of gains are possible. Optionally read “Process Improvement and the Corporate Balance Sheet” by Raymond Dion, IEEE Software Magazine, July 1993.

Karl Wiegers is the author of a interesting book titled “Creating a Software Engineering Culture” [Wiegers96].

The following chart is reproduced with his permission from a more recent course [Weigers98] that this instructor attended.

- It summarizes the payback from adding requirements/design/code reviews to your software development process.

Originally published in "Creating a Software Engineering Culture" by Karl Wiegers, Dorset House, 1996

Some Real Benefits from Software Inspections

Company	Cost Impact	Productivity Impact	Quality Impact
Aetna Insurance Company		25% increase in coding productivity	inspections found 82% of errors
AT&T Bell Laboratories	cost of finding errors reduced 10X by inspections	14% increase	10X improvement
Bell Northern Research	avoided 33 hours of maintenance/defect found	inspections 2-4X faster than testing	found 80% of all defects by inspection
Bull HN Information Systems		late stage testing period shortened because defects removed earlier by inspection	inspections found 70-75% of all defects
Hewlett-Packard	save $21.5 million/year; 10X return on investment	reduced time to market by 1.8 months	
IBM	cost of rework of defects found by inspection is 10% that by testing		
Jet Propulsion Laboratory	$7.5 million savings from 300 inspections		
Litton	saved 63.4 staff hours per inspection		76% fewer defects in integration
Microsoft		cost to find and fix errors by testing is 4X that by code inspection	
Motorola	saved $2.5 million in one year on one project		

Verbally given permission to copy
by Karl 98/4/17

Taken From:
SPC
Software Technical Reviews
Course Notes

Karl also gave an interesting talk called “No New Models” in Vancouver in April 1998. His premise was that we didn’t need any new:

- development models,
- process models,
- object-oriented methodologies, etc.

As you will see throughout this course, there are already quite a few.

The problem is not that we need the perfect model for your company, but that your company could start taming wild software projects just by using some good software engineering models we already have!

2.15 Appendix C - Software Development Standards

There are several software process and quality standards.

And considerable discussion of “Software Process Improvement (SPI)”.

An example improvement is collecting metrics on what common kinds of bugs are found in your company’s testing and reviews,

- so as to incorporate checks for those into checklists used on requirements, design, and code reviews going forward.

2.15.1 Capability Maturity Model (CMM)

The Software Engineering Institute at Carnegie-Mellon University was funded to help improve the US software engineering industry.

- The U.S., though writing much of the world's software, was not doing it very well.
- More carefully designed and quality software was being written in Europe using much safer languages (e.g. Modula-2). The very good ADA language was not catching on in the U.S. Software could be developed in India less expensively by well-educated people who spoke better English than Americans.

Of relevance is a book called “The Decline and Fall of the American Programmer” by Ed Yourdon [Yourdon92].

- Interestingly, Yourdon, has now written “The Rise and Resurrection of the American Programmer” [Yourdon96], perhaps on recognizing the growing U.S. industry respect for software engineering process, and for institutions like the SEI and the work they are doing.

The Capability Maturity Model (CMM) is a 5-level model.

Software development companies can compare (even be audited) against the CMM, to determine their ability to predictably produce quality software on time and on budget.

In the late 1990s, the US software development industry was quite scared that companies might have to be at Level 3 to even bid on future US government software contracts. Though restriction never materialized.

The five level model and the key process areas that they must perform for each level is shown in the following table:

- Level 1: (no name)
  - chaos
- Legal 2: Repeatable
  - software configuration management
  - software quality assurance
  - software subcontract management
  - software project tracking and oversight
  - software project planning
  - requirements management (not just specification)

- Level 3: Defined
  - organizational process definition
  - organizational process focus
  - peer reviews
  - integrated software management
  - software product engineering
  - intergroup coordination
  - employee training programs
- Level 4: Managed
  - software quality management
  - quantitative process management

- Level 5: Optimizing
  - process change management
  - technology change management
  - defect prevention

Each major step predicated on the level below being in place.

- E.g. A level may depend on statistics gathered over several projects at the previous level.

It's not possible to jump to Level 5 (used on the space shuttle software) by spending extreme effort and money to write down the company development policy, and put the processes in place.

The CMM also includes an audit procedure, so that customers before they let a contract to a development company can ask the development company for the results of their CMM audit. Were they at Level 1, 2, or 3? A local auditor has told me that the laughable thing is that the problems that most companies have are so similar (or at least from the same set), that the auditor has pre-prepared sections for her report. She just pulls out the subset of the common ones (like need for configuration management) applicable to make up the audit report and recommended actions to comply for the particular recently-audited development company.

2.16 Appendix D - Other Lists of Good Practices

2.16.1 *Arlie Software Council Best and Worst*

U.S. DOD commissioned a council of highly regarded industry practitioners and consultants to recommended 'Best Practices'. More info in Section 6 of [Yourdon96].

The recommended practices are:

- Formal Risk Management
- User Manual as Specification
- Inspections, Reviews and Walkthroughs
- Metrics-Based Scheduling and Tracking
- Binary Quality Gates at the Inch-Pebble Level
- Program-Wide Visibility of Project Plan, and Progress vs. the Plan.
- Defect Tracking Against Quality Targets
- Separate Specification of Hardware and Software Functionality
- People-Aware Management Accountability

They also came up with a list of worst practices, also provided in Section 6 of [Yourdon96]:

- Don't use schedule compression to justify usage of new technology on any time critical project.
- Don't specify implementation technology in the Request For Proposal.
- Don't advocate use of unproven "silver-bullet" approaches.
- Don't expect to recover from any substantial schedule slip (10% or more) without making more than corresponding reductions in functionality to be delivered.

- Don’t put items that our not under the project’s control on the critical path.
- Don’t expect to achieve large positive improvements (10% or more) over past observed performance.
- On mixed hardware/software projects, don’t bury all project complexity in the software (as opposed to the hardware).
- Don’t conduct the critical system engineering tasks (particularly the hardware/software partitioning) without sufficient software expertise.

- Don’t believe that formal reviews provide an accurate picture of the project (even though formal reviews are a best practice).
  - Don’t make them widely-attended, politically-charged progress meetings.
    - Expect the usefulness of formal reviews to be inversely proportional to the number of people attending beyond five.

2.16.2 Joch's 9 Ways to Write More Reliable Software

Alan Joch in [Joch95] suggests:

- Fight for a stable design.
- Cleanly divide up tasks.
- Avoid shortcuts.
- Use assertions liberally.
- Use tools judiciously.
- Rely on fewer programmers.
- Diligently fight 'featuritis'.
- Use formal methods where appropriate.
- Begin testing once you write the first line of code.