Software Engineering Practice

Software Engineering Code of Ethics

- The *Software Engineering Code of Ethics and Professional Practice* (SE Code) was developed by the ACM/IEEE-CS Joint Task Force on Software Engineering Ethics and Professional Practices and jointly approved by the ACM and the IEEE-CS as the standard for teaching and practicing software engineering. (http://www.acm.org/serving/se/code.htm)

- The preamble to the SE Code (short version) states:
  - Software engineers shall commit themselves to making the analysis, specification, design, development, testing and maintenance of software a beneficial and respected profession. In accordance with their commitment to the health, safety and welfare of the public, software engineers shall adhere to the following Eight Principles:

Software Engineering Code of Ethics - Eight Principles

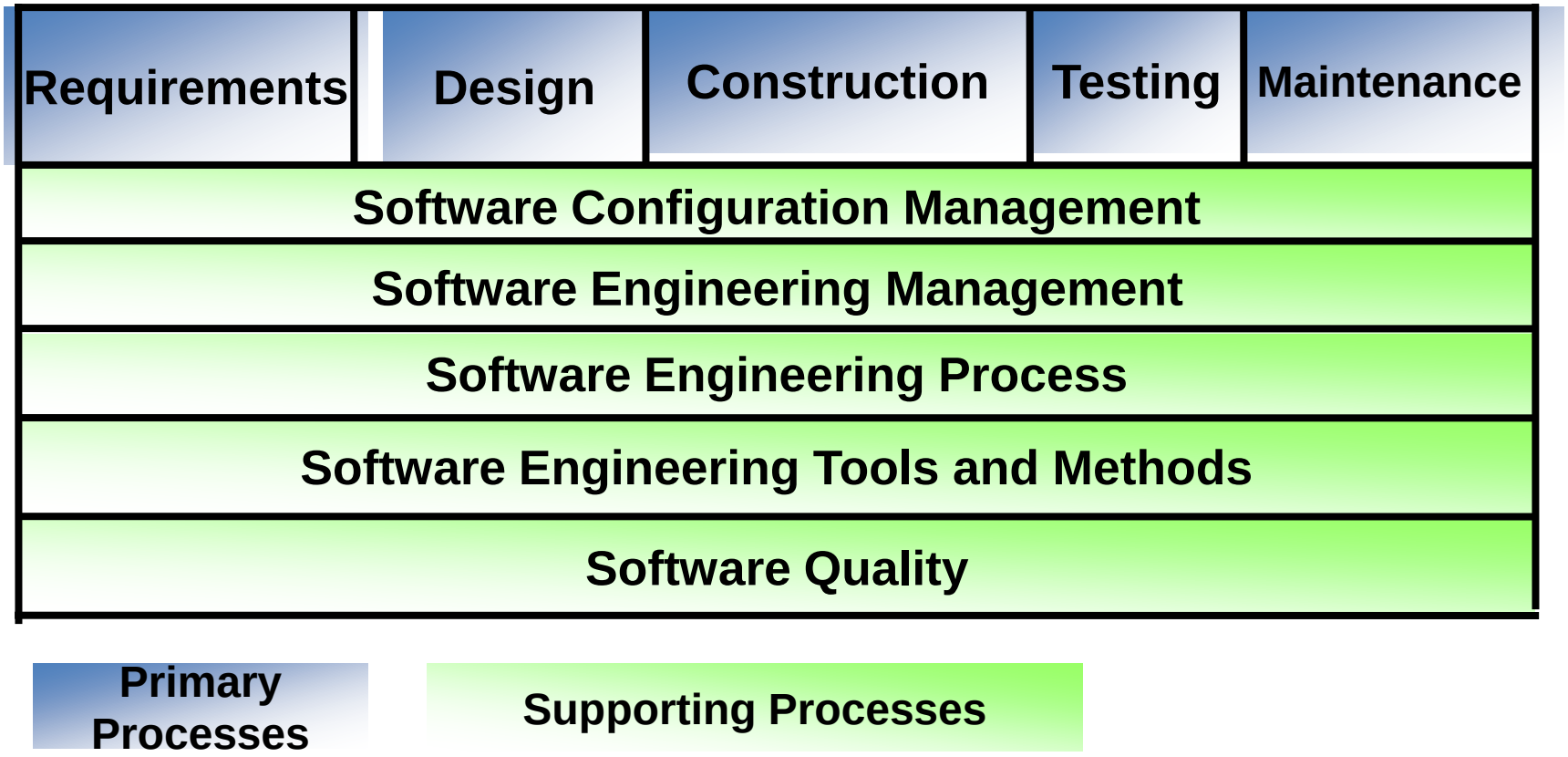
- PUBLIC - Software engineers shall act consistently with the public interest.
- CLIENT AND EMPLOYER - Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest.
- PRODUCT - Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.
- JUDGMENT - Software engineers shall maintain integrity and independence in their professional judgment.
- MANAGEMENT - Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.
- PROFESSION - Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.
- COLLEAGUES - Software engineers shall be fair to and supportive of their colleagues.
- SELF - Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

www.acm.org

www.swebok.org

(Software Engineering Book of Knowledge by IEEE)

SWEBOK Guide = 10 Knowledge Areas

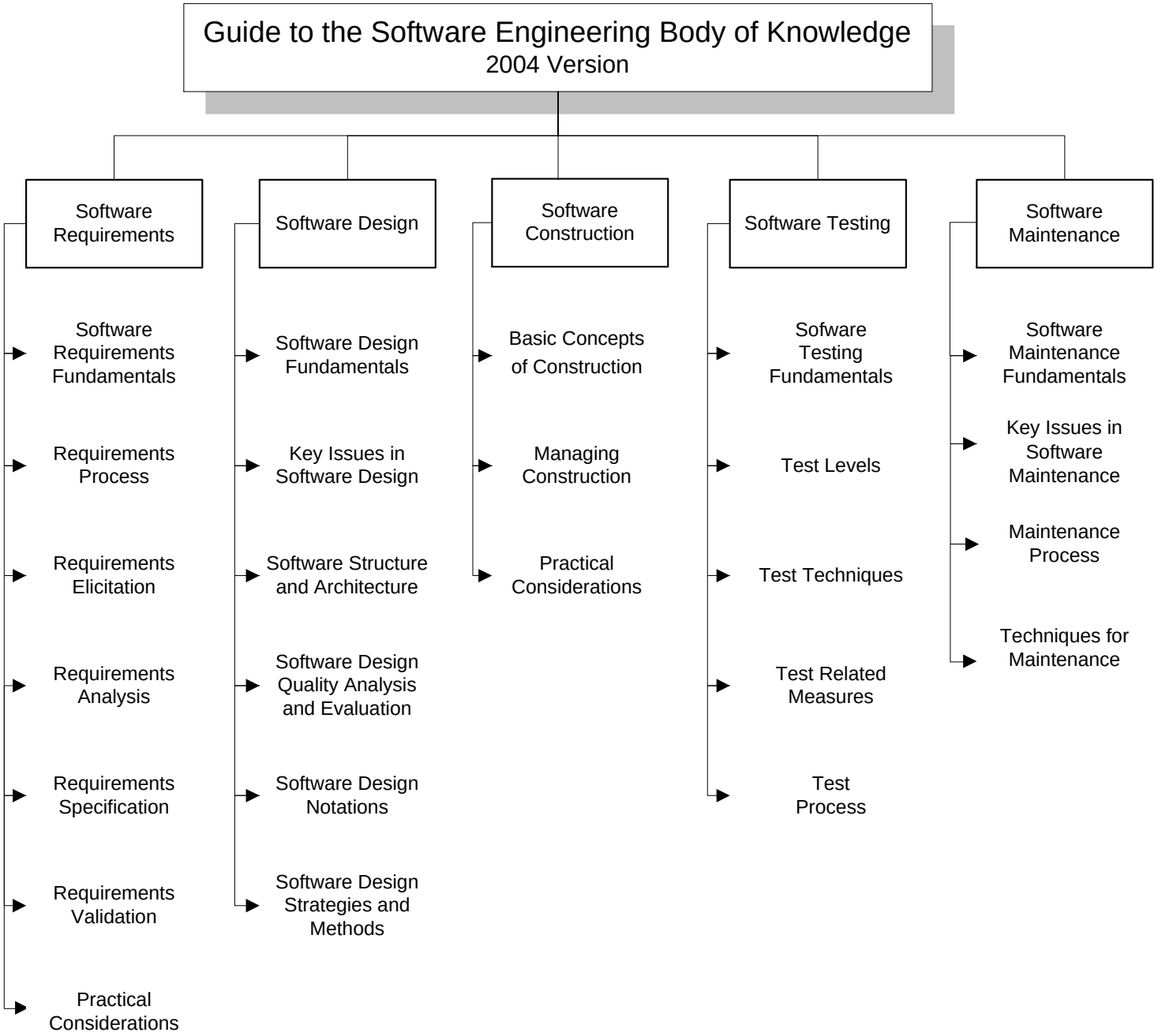


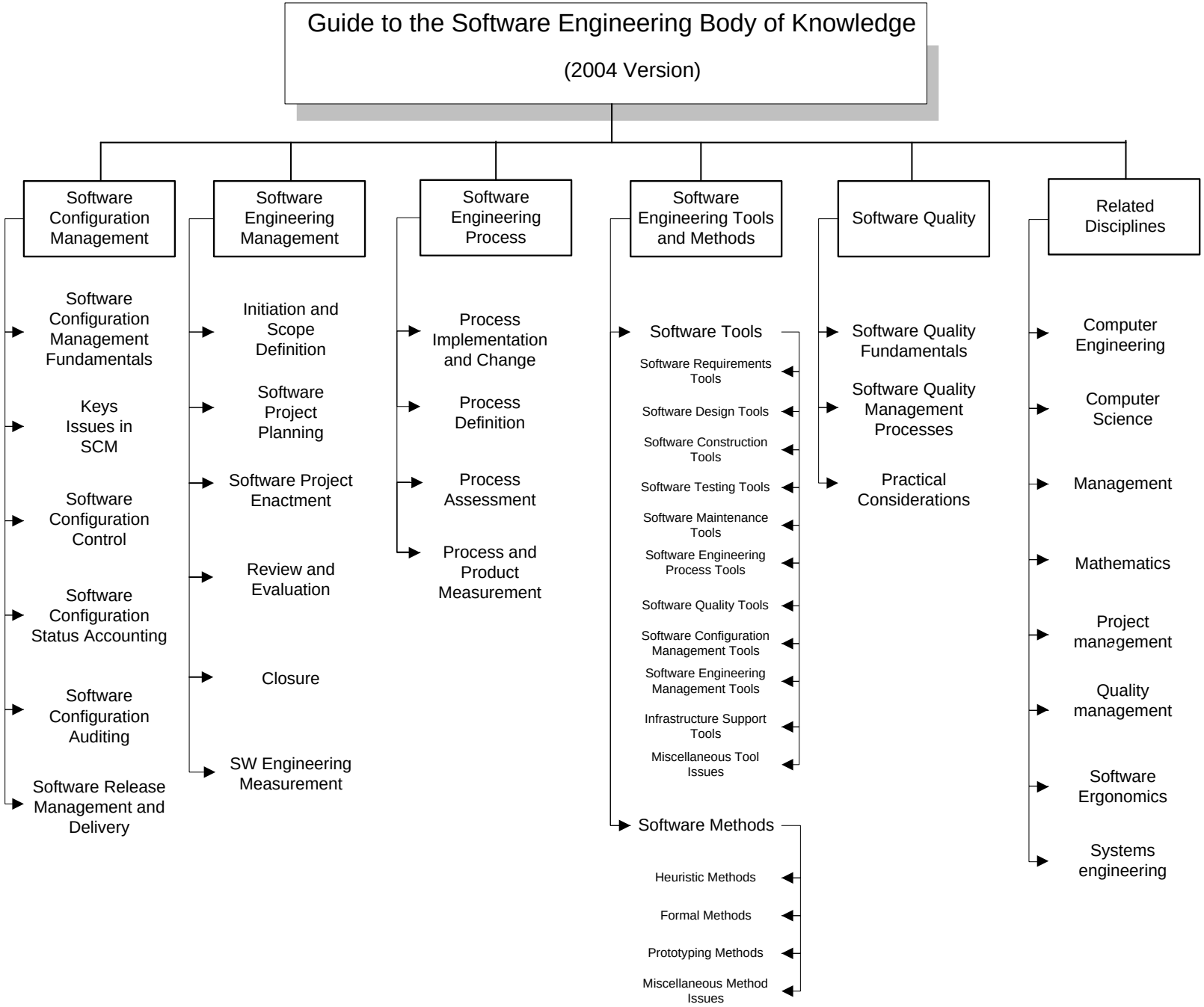
Knowledge Areas and Related Disciplines

- Software Requirements
- Software Design
- Software Construction
- Software Testing
- Software Maintenance
- Software Configuration Management
- Software Eng. Management
- Software Eng. Tools & Methods
- Software Engineering Process
- Software Quality

Related Disciplines

- Computer Engineering
- Computer Science
- Mathematics
- Project Management
- Management
- Quality Management
- Software Ergonomics
- Systems Engineering


Guide to the Software Engineering Body of Knowledge
2004 Version
Software Requirements
Software Requirements Fundamentals
Requirements Process
Requirements Elicitation
Requirements Analysis
Requirements Specification
Requirements Validation
Practical Considerations
Software Design
Software Design Fundamentals
Key Issues in Software Design
Software Structure and Architecture
Software Design Quality Analysis and Evaluation
Software Design Notations
Software Design Strategies and Methods
Software Construction
Basic Concepts of Construction
Managing Construction
Practical Considerations
Software Testing
Sofware Testing Fundamentals
Test Levels
Test Techniques
Test Related Measures
Test Process
Software Maintenance
Software Maintenance Fundamentals
Key Issues in Software Maintenance
Maintenance Process
Techniques for Maintenance


Guide to the Software Engineering Body of Knowledge
(2004 Version)
Software Configuration Management
Software Configuration Management Fundamentals
Keys Issues in SCM
Software Configuration Control
Software Configuration Status Accounting
Software Configuration Auditing
Software Release Management and Delivery
Software Engineering Management
Initiation and Scope Definition
Software Project Planning
Software Project Enactment
Review and Evaluation
Closure
SW Engineering Measurement
Software Engineering Process
Process Implementation and Change
Process Definition
Process Assessment
Process and Product Measurement
Software Engineering Tools and Methods
Software Tools
Software Requirements Tools
Software Design Tools
Software Construction Tools
Software Testing Tools
Software Maintenance Tools
Software Engineering Process Tools
Software Quality Tools
Software Configuration Management Tools
Software Engineering Management Tools
Infrastructure Support Tools
Miscellaneous Tool Issues
Software Methods
Heuristic Methods
Formal Methods
Prototyping Methods
Miscellaneous Method Issues
Software Quality
Software Quality Fundamentals
Software Quality Management Processes
Practical Considerations
Related Disciplines
Computer Engineering
Computer Science
Management
Mathematics
Project management
Quality management
Software Ergonomics
Systems engineering

Software Engineering Knowledge

You often hear people say that software development knowledge has a 3-year half-life: half of what you need to know today will be obsolete within 3 years. In the domain of technology-related knowledge, that's probably about right. But there is another kind of software development knowledge—a kind that I think of as "software engineering principles"—that does not have a three-year half-life. These software engineering principles are likely to serve a professional programmer throughout his or her career.

Steve McConnell

The Essence of Problem Solving

(as Per George Polya, A Hungarian Mathematician – How to Solve It(1945))

1) Understand the problem
   - What are the unknowns?
   - Can the problem be compartmentalized?
   - Can the problem be represented graphically?
2) Plan a solution
   - Have you seen similar problems like this before?
   - Has a similar problem been solved and is the solution reusable?
   - Can subproblems be defined and solved?
3) Carry out the plan
   - Is the solution as per plan?
   - Is each component of the solution correct?
4) Examine the results for accuracy
   - Is it possible to test each component of the solution?

The Essence of Problem Solving

(in Software Engineering)

1) Understand the problem (communication and analysis)

- Who has a stake in the solution to the problem?
- What are the unknowns (data, function, behavior)?
- Can the problem be compartmentalized?
- Can the problem be represented graphically?

2) Plan a solution (planning, modeling and software design)

- Have you seen similar problems like this before?
- Has a similar problem been solved and is the solution reusable?
- Can subproblems be defined and solved?

3) Carry out the plan (construction; code generation)

- Is the solution as per plan? Is the source code traceable to the design?
- Is each component of the solution correct? Has the design and code been reviewed?

4) Examine the results for accuracy (testing and quality assurance)

- Is it possible to test each component of the solution?
- Does the solution produce results that conform to the data, function, and behavior that are required?

Seven Core Principles for Software Engineering

(David Hooker – 1996)

1) Remember the reason that the software exists
   - The software should provide value to its users and satisfy the requirements
2) Keep it simple, stupid (KISS)
   - All design and implementation should be as simple as possible
3) Maintain the vision of the project
   - A clear vision is essential to the project's success
4) Others will consume what you produce
   - Always specify, design, and implement knowing that someone else will later have to understand and modify what you did
5) Be open to the future
   - Never design yourself into a corner; build software that can be easily changed and adapted
6) Plan ahead for software reuse
   - Reuse of software reduces the long-term cost and increases the value of the program and the reusable components
7) Think, then act
   - Placing clear, complete thought before action will almost always produce better results

Principles that Guide Process - I

Principle #1. ***Be agile.***

Whether the process model you choose is prescriptive or agile, the basic tenets of agile development should govern your approach.

- Economy of action
- Simple approach
- Concise work products
- Local decisions

Principle #2. ***Focus on quality at every step.***

The exit condition for every process activity, action, and task should focus on the quality of the work product that has been produced.

Principle #3. ***Be ready to adapt.***

Process is not a religious experience and dogma has no place in it. When necessary, adapt your approach to constraints imposed by the problem, the people, and the project itself.

Principle #4. ***Build an effective team.***

Software engineering process and practice are important, but the bottom line is people. Build a self-organizing team that has mutual trust and respect.

Principles that Guide Process - II

Principle #5. ***Establish mechanisms for communication and coordination.***

Projects fail because important information falls into the cracks and/or stakeholders fail to coordinate their efforts to create a successful end product.

Principle #6. ***Manage change.***

The approach may be either formal or informal, but mechanisms must be established to manage the way changes are requested, assessed, approved and implemented.

Principle #7. ***Assess risk.***

Lots of things can go wrong as software is being developed. It's essential that you establish contingency plans.

Principle #8. ***Create work products that provide value for others.***

Create only those work products that provide value for other process activities, actions or tasks.

Principles that Guide Practice - I

Principle #1. ***Divide and conquer.***

Stated in a more technical manner, analysis and design should always emphasize *separation of concerns (SoC).*

Principle #2. ***Understand the use of abstraction.***

At it core, an abstraction is a simplification of some complex element of a system used to communication meaning in a single phrase.

Principle #3. Strive for consistency.

A familiar context makes software easier to use.

Principle #4. ***Focus on the transfer of information.***

Pay special attention to the analysis, design, construction, and testing of interfaces.

Principles that Guide Practice - II

Principle #5. ***Build software that exhibits effective modularity.***

Separation of concerns (Principle #1) establishes a philosophy for software. Modularity provides a mechanism for realizing the philosophy.

Principle #6. ***Look for patterns.***

Brad Appleton [App00] suggests that: "The goal of patterns within the software community is to create a body of literature to help software developers resolve recurring problems encountered throughout all of software development."

Principle #7. ***When possible, represent the problem and its solution from a number of different perspectives.***

Principle #8. ***Remember that someone will maintain the software.***

Communication Principles

Principle #1. ***Listen.***

Try to focus on the speaker's words, rather than formulating your response to those words.

Principle # 2. ***Prepare before you communicate.***

Spend the time to understand the problem before you meet with others.

Principle # 3. ***Someone should facilitate the activity.***

Every communication meeting should have a leader (a facilitator)

(1) to keep the conversation moving in a productive direction;
(2) to mediate any conflict that does occur, and
(3) to ensure than other principles are followed.

Principle #4. ***Face-to-face communication is best.***

But it usually works better when some other representation of the relevant information is present.

Principle # 5. ***Take notes and document decisions.***

Someone participating in the communication should serve as a "recorder" and write down all important points and decisions.

Communication Principles

Principle # 6. ***Strive for collaboration.***

Collaboration and consensus occur when the collective knowledge of members of the team is combined …

Principle # 7. ***Stay focused, modularize your discussion.***

The more people involved in any communication, the more likely that discussion will bounce from one topic to the next.

Principle # 8. ***If something is unclear, draw a picture.***

Principle # 9. ***Move on to the next topic***

(a) Once you agree to something, move on; (b) If you can't agree to something, move on; (c) If a feature or function is unclear and cannot be clarified at the moment, move on.

Principle # 10. ***Negotiation is not a contest or a game. It works best when both parties win.***

Planning Principles

Principle #1. ***Understand the scope of the project.***

It's impossible to use a roadmap if you don't know where you're going. Scope provides the software team with a destination.

Principle #2. ***Involve the customer in the planning activity.***

The customer defines priorities and establishes project constraints.

Principle #3. ***Recognize that planning is iterative.***

A project plan is never engraved in stone. As work begins, it very likely that things will change.

Principle #4. ***Estimate based on what you know.***

The intent of estimation is to provide an indication of effort, cost, and task duration, based on the team's current understanding of the work to be done.

Planning Principles

Principle #5. ***Consider risk as you define the plan***.
If you have identified risks that have high impact and high probability, contingency planning is necessary.

Principle #6. ***Be realistic.***
People don't work 100 percent of every day.

Principle #7. ***Adjust granularity as you define the plan.***
Granularity refers to the level of detail that is introduced as a project plan is developed.

Principle #8. ***Define how you intend to ensure quality.***
The plan should identify how the software team intends to ensure quality.

Principle #9. ***Describe how you intend to accommodate change.***
Even the best planning can be obviated by uncontrolled change.

Principle #10. ***Track the plan frequently and make adjustments as required.***
Software projects fall behind schedule one day at a time.

Modeling Principles

In software engineering work, two classes of models can be created:

Requirements models (also called analysis models) represent the customer requirements by depicting the software in three different domains: the information domain, the functional domain, and the behavioural domain.

Design models represent characteristics of the software that help practitioners to construct it effectively: the architecture, the user interface, and component-level detail.

Requirements Modeling Principles

Principle #1. ***The information domain of a problem must be represented and understood.***

Principle #2. ***The functions that the software performs must be defined.***

Principle #3. ***The behavior of the software (as a consequence of external events) must be represented.***

Principle #4. ***The models that depict information, function, and behavior must be partitioned in a manner that uncovers detail in a layered (or hierarchical) fashion.***

Principle #5. ***The analysis task should move from essential information toward implementation detail.***

Design Modeling Principles

Principle #1. Design should be traceable to the requirements model.

Principle #2. Always consider the architecture of the system to be built.

Principle #3. Design of data is as important as design of processing functions.

Principle #4. Interfaces (both internal and external) must be designed with care

Principle #5. User interface design should be tuned to the needs of the end-user. However, in every case, it should stress ease of use.

Principle #6. Component-level design should be functionally independent.

Principle #7. Components should be loosely coupled to one another and to the external environment.

Principle #8. Design representations (models) should be easily understandable.

Principle #9. The design should be developed iteratively. With each iteration, the designer should strive for greater simplicity.

Agile Modeling Principles

Principle #1. ***The primary goal of the software team is to build software, not create models.***

Principle #2. ***Travel light—don't create more models than you need.***

Principle #3. ***Strive to produce the simplest model that will describe the problem or the software.***

Principle #4. ***Build models in a way that makes them amenable to change.***

Principle #5. ***Be able to state an explicit purpose for each model that is created.***

Principle #6. ***Adapt the models you develop to the system at hand.***

Principle #7. ***Try to build useful models, but forget about building perfect models.***

Principle #8. ***Don't become dogmatic about the syntax of the model. If it communicates content successfully, representation is secondary.***

Principle #9. ***If your instincts tell you a model isn't right even though it seems okay on paper, you probably have reason to be concerned.***

Principle #10. ***Get feedback as soon as you can.***

Construction Principles

The construction activity encompasses a set of coding and testing tasks that lead to operational software that is ready for delivery to the customer or end-user.

Coding principles and concepts are closely aligned programming style, programming languages, and programming methods.

Testing principles and concepts lead to the design of tests that systematically uncover different classes of errors and to do so with a minimum amount of time and effort.

Preparation Principles

Before you write one line of code, be sure you:

- Understand of the problem you're trying to solve.
- Understand basic design principles and concepts.
- Pick a programming language that meets the needs of the software to be built and the environment in which it will operate.
- Select a programming environment that provides tools that will make your work easier.
- Create a set of unit tests that will be applied once the component you code is completed.

Coding Principles

As you begin writing code, be sure you:

- Constrain your algorithms by following structured programming practice.
- Consider the use of pair programming
- Select data structures that will meet the needs of the design.
- Understand the software architecture and create interfaces that are consistent with it.
- Keep conditional logic as simple as possible.
- Create nested loops in a way that makes them easily testable.
- Select meaningful variable names and follow other local coding standards.
- Write code that is self-documenting.
- Create a visual layout (e.g., indentation and blank lines) that aids understanding.

Validation Principles

After you've completed your first coding pass, be sure you:

- Conduct a code walkthrough when appropriate.
- Perform unit tests and correct errors you've uncovered.
- Refactor the code.

Testing Principles

Al Davis [Dav95] suggests the following:

Principle #1. ***All tests should be traceable to customer requirements.***

Principle #2. ***Tests should be planned long before testing begins.***

Principle #3. ***The Pareto principle applies to software testing.***

Principle #4. ***Testing should begin "in the small" and progress toward testing "in the large."***

Principle #5. ***Exhaustive testing is not possible.***

Deployment Principles

Principle #1. ***Customer expectations for the software must be managed.***

Too often, the customer expects more than the team has promised to deliver, and disappointment occurs immediately.

Principle #2. ***A complete delivery package should be assembled and tested.***

Principle #3. ***A support regime must be established before the software is delivered.***

An end-user expects responsiveness and accurate information when a question or problem arises.

Principle #4. ***Appropriate instructional materials must be provided to end-users.***

Principle #5. ***Buggy software should be fixed first, delivered later.***

The following slide set (8 slides) is taken from a scholarly presentation at System Design & Management Conference at MIT, USA in Oct, 2006.

Some slides removed from the presentation to facilitate walkthrough within the class

Fast & Flexible Software Development

Michael A. Cusumano

MIT Sloan School of Management

cusumano@mit.edu

Different Cultural Orientations

EUROPE: Software as a *Science*

–***Formal Methods, Object-Oriented Design***

JAPAN: Software as *Production*

–***Software Factories, Zero-Defects***

INDIA: Software as a *Service*

–***Infosys, Tata, Wipro, Satym, Cognizant, Patni***

The USA: Software as a *Business*

–***Windows, Office, Navigator, $$$$***

Different Process Philosophies

- **Waterfall-style (sequential, "Stage-gate")**

versus

- **Iterative-style (agile, incremental)**
  - Spiral
  - Rapid Prototyping
  - Synch-and-Stabilize (Microsoft, PC makers)
  - IBM Rational's Unified Process & Toolkit
  - HP's Evo Process (short cycles of mini-waterfalls)
  - Extreme Programming (XP), SCRUM, AGILE
  - Many other variations at companies

International Comparisons

(2003 IEEE Software, "Software Dev. Worldwide")

- **Survey:** Completed in 2002-2003, with Alan MacCormack (HBS), Chris Kemerer (Pittsburgh), and Bill Crandall (HP)
- **Objective:** Determine usage of iterative (Synch-&-Stabilize) versus Waterfall-ish techniques, with performance comparisons
  - 118 projects plus 30 from HP-Agilent for pilot survey
- **Participants**
  - **India:** Motorola MEI, Infosys, Tata, Patni
  - **Japan:** Hitachi, NEC, IBM Japan, NTT Data, SRA, Matsushita, Omron, Fuji Xerox, Olympus
  - **US:** IBM, HP, Agilent, Microsoft, Siebel, AT&T, Fidelity, Merrill Lynch, Lockheed Martin, TRW, Micron Tech
  - **Europe:** Siemens, Nokia, Business Objects

“Conventional” Good Practices

		India	*Japan*	*USA*	*Europe etc*		*Total*
Number of Projects		*24*	*27*	*31*	*22*		*104*
Architectural Specs %		83%	70%	55%	73%		69%
Functional Specs %		96%	93%	74%	82%		86%
Detailed Design %		100%	85%	32%	68%		69%
Code Generators -- Yes		63%	41%	52%	55%		52%
Design Reviews -- Yes		100%	100%	77%	77%		88%
Code Reviews -- Yes		96%	74%	71%	82%		80%

“Newer” Iterative Practices

		India	*Japan*	*USA*	*Europe etc*		*Total*
No. of Projects		*24*	*27*	*31*	*22*		*104*
Subcycles -- Yes		79%	44%	55%	86%		64%
Beta tests -- Yes		67%	67%	77%	82%		73%
Pair Testing -- Yes		54%	44%	35%	32%		41%
Pair Programmer -- Yes		58%	22%	36%	27%		35%
Daily Builds at project start		17%	22%	36%	9%		22%
In the middle		13%	26%	29%	27%		24%
At the end		29%	37%	36%	41%		36%
Regression test each build		92%	96%	71%	77%		84%

“Crude” Output Comparisons

		India	**Japan**	**USA**	**Europe etc.**		***TOTAL***
Projects		24	27	31	22		***104***
LOC/ Month	median	209	469 cf. *389* in 1990	270 cf. *245* in 1990	436		***374***
Bugs/ 1000 LOC	median	.263	**.020** cf. .20 in 1990	.400 cf. .80 in 1990	.225		***.150***

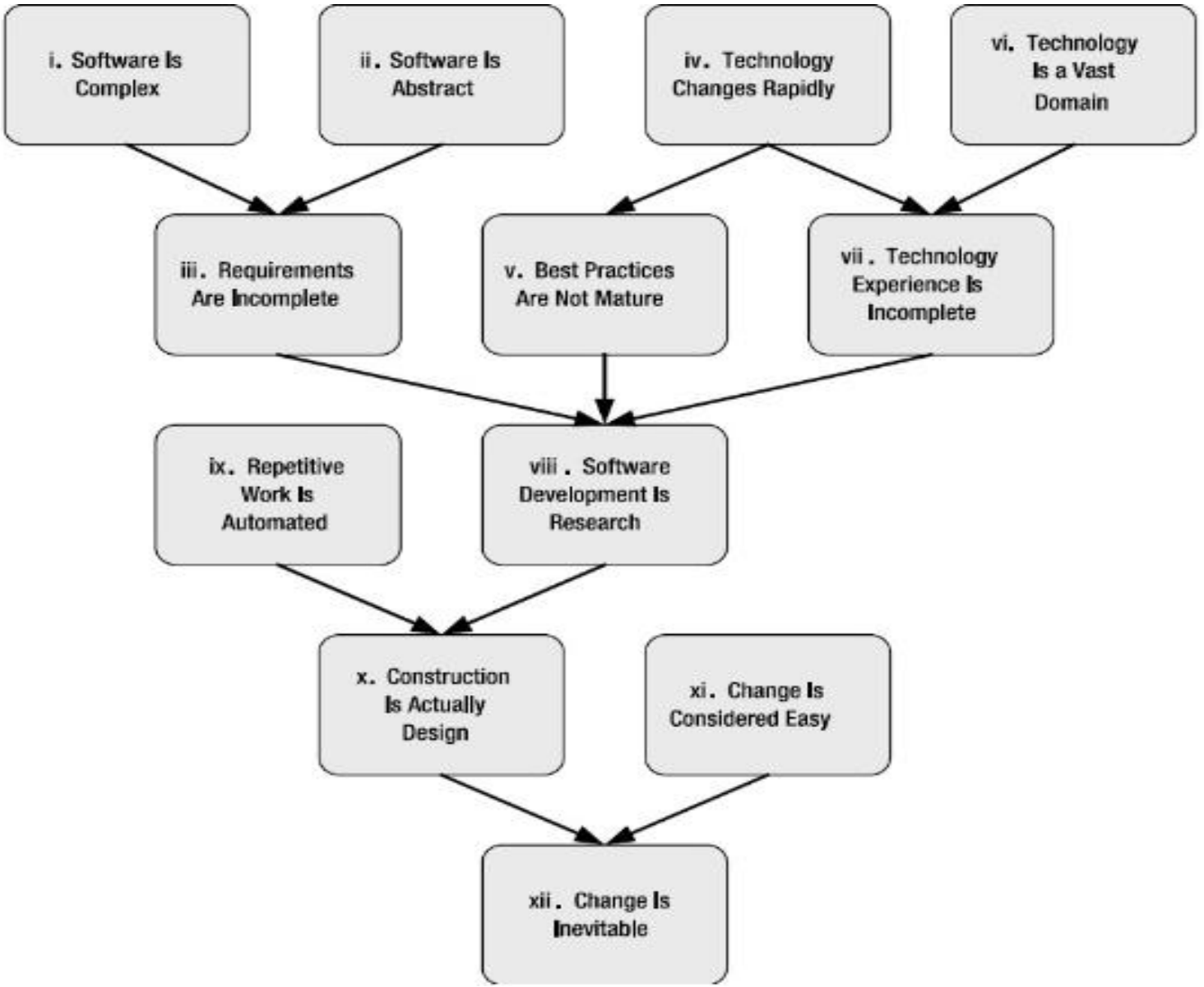
Observations from Global Survey

- **Most projects (64%) not pure waterfall**; **36% were!**
- **Mix of "conventional" and "iterative" common** -- use of functional specs, design & code reviews, but with subcycles, regression tests on frequent builds
- **Customer-reported defects improved** -- over past decade in US and Japan; LOC "productivity" may have improved a little

- **Japanese still report best quality & productivity** -- but what does this mean? Preoccupation with "zero defects"? Need more lines of code to write same functionality per day as US & Indian programmers?
- **Indian projects strong in process and quality** -- but not as strong as their dominance of CMM Level 5 suggests??

Why should we be
concerned about
Software Engineering Practice?

A Representation of Software Challenges



"Software Project Secrets : Why Software Projects Fail", George Stepanek, Apress ©2012

Top 10 Technology Disasters of 2013

1. Obamacare's Healthcare.gov website failed to work on launch in October. US citizens had to resort to phonelines and the post. A lack of testing was flagged as the key reason behind the failing.

2. Sabre's worldwide reservation system – deployed by over 300 airlines – went down, causing cancellations and delays for hundreds of thousands of passengers despite the system being offline for less than three hours.

3. Natwest system failure: The bank ran into problems earlier this year when system glitches caused chaos, denying ATM, chip and pin and internet banking facilities to customers. RBS also suffered a similar problem in recent weeks, which has led to some experts talking about the underinvestment in banking IT.

4. NHS: A major IT glitch with Scotland NHS Greater Glasgow and Clyde's servers meant doctors and nurses were unable to access vital patient information. Luckily no patient lives were endangered during the system failure.

5. Walmart's electronic bargains: In October, shoppers managed to buy computer monitors and projectors – valued at $500 – for as little as $8.99. The retailer blamed IT glitches and refused to honor these bargain deals.

Extracted from www.telegraph.co.uk

Top 10 Technology Disasters of 2013

6. Bank of England hardware failure: The BofE was unable to make its quantitative easing announcement due to a technical failure, causing disruption and confusion for the markets.

7. California's payroll failures: Californian state government is suing SAP over a failure to adequately develop and test a upgrade of its payroll system, causing problems for many of its 240,000 workforce.

8. M&S glitches at distribution center: Marks & Spencer encountered numerous issues at its brand new distribution center at Castle Donington in August. IT glitches meant that trading directors were concerned about letting stock orders be processed by the distribution center in case it affected availability in their stores.

9. Defective benefits systems: California, Massachusetts, Pennsylvania and Florida have all had major software problems with benefits systems implemented by Deloitte. In California, the defect-riddled system denied payments to over 300,000 unemployed people during Labor Day weekend.

10. Blackberry technical glitches: In September Blackberry users experienced a blackout, with problems starting in Europe and quickly spreading to the Americas and other continents.

Extracted from www.telegraph.co.uk

S/W in automobiles

- Average automobile has
  - 70 to 100 microprocessor-based electronic control units (ECUs), running
  - 100 million lines of software code
- Control software logic analyzes vehicle load, engine operations, battery parameters, temperatures, etc.
- Software development is the single most important consideration in new product development engineering
- 35-40% of the cost of a car is software and electronics (13-15% of that cost is software development)
- 50% of car warranty costs are related to electronics and embedded software
- Bugs:
  - 2005: Toyota recalled >160000 Prius hybrids due to S/W problem
  - May 2008: Chrysler recalled >20000 Jeep Commanders because of bug in automatic transmission S/W
  - June 2008: Volkswagen recalled ~4000 Passats and Tiguans for bug in engine-control-module S/W
  - November 2008: GM recalled >12000 Cadillacs that toggled air bag enable/disable bit

Extracted from Robert N. Charette, This Car Runs on Code, IEEE Spectrum, Feb. 2009

Another glitch

- “In 2008, in South Africa an anti-aircraft had a 'software glitch' during a training exercise. It was supposed to fire upwards into the sky, instead it lowered and it fired in a circle and killed nine soldiers, all because of a software glitch."

Source: http://www.cnn.com/2009/WORLD/americas/07/23/wus.warfare.remote.uav/index.html

Summary

The Software Engineering practice includes

- concepts,
- principles,
- methods,
- tools

used by software engineers to develop software.

Software engineers must be concerned both with

- technical details of doing things and
- things needed to build high-quality computer software

Software process offers roadmap to build quality products

Software Engineering practice provides the detail needed to travel the road.