วิศวกรรมซอฟต์แวร์

Software Engineering

สมเกียรติ วังศิริพิทักษ์

somkiat.wa@kmitl.ac.th

ห้อง 518 หรือ ห้อง 506 (MIV Lab)

PART II

Quality Management

Quality Management

- You'll learn about the principles, concepts, and techniques that are applied to manage and control software quality.
- These questions are addressed in the remaining lectures:
  - What are the generic characteristics of high-quality software?
  - How do we review quality and how are effective reviews conducted?
  - What is software quality assurance?
  - What strategies are applicable for software testing?

Quality Management

- What methods are used to design effective test cases?
- Are there realistic methods that will ensure that software is correct?
- How can we manage and control changes that always occur as software is built?
- What measures and metrics can be used to assess the quality of requirements and design models, source code, and test cases?
- How do we ensure the security concerns for a product are being addressed during the software life cycle?

PART II

Quality Management

Quality Concepts

Quality Concepts

- Software became increasingly integrated in every facet of our lives.
- By the 1990s, major corporations recognized that billions of dollars each year were being wasted on software that didn't deliver the features and functionality that were promised.

- Increasingly concerned that a major software fault might cripple important infrastructure, costing tens of billions more.

- Poor software quality caused by a rush to release products without adequate testing continues to plague the software industry.

Quality Concepts

What is it?

- The answer isn't as easy as you might think.
- You know quality when you see it, and yet, it can be an elusive thing to define.
- But for computer software, quality is something that we must define, and that's what we'll do in this lecture.

Quality Concepts

Who does it?

- Everyone—software engineers, managers, all stakeholders—involved in the software process is responsible for quality.

Why is it important?

- You can do it right, or you can do it over again.
- If a software team stresses quality in all software engineering activities, it reduces the amount of rework that it must do.
- That results in lower costs, and more importantly, improved time to market.

Quality Concepts

What are the steps?

- To achieve high-quality software, **four activities** must occur:
  - proven software engineering process and practice,
  - solid project management,
  - comprehensive quality control, and
  - the presence of a quality assurance infrastructure.

What is the work product?

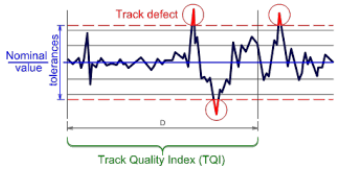
- Software that meets its customer's needs, performs accurately and reliably, and provides value to all who use it.

Quality Concepts

How do I ensure that I've done it right?

- Track quality
  by examining the results of all quality control activities.

  AND



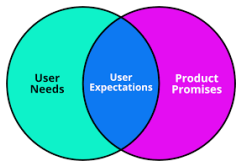
- Measure quality
  by examining errors before delivery and defects released to the field.

What Is Quality?

- At a somewhat more pragmatic level, **David Garvin** of the Harvard Business School suggests that
  "quality is a complex and multifaceted concept"
  that can be described from five different points of view.

❶ • The transcendental view argues that quality is something you immediately recognize but cannot explicitly define.

❷ • The user view sees quality in terms of an end user's specific goals.
  - If a product meets those goals, it exhibits quality.



What Is Quality?

❸ • The manufacturer's view defines quality in terms of the original specification of the product.
  - If the product conforms to the spec, it exhibits quality.

❹ • The product view suggests that quality can be tied to inherent characteristics (e.g., functions and features) of a product.

❺ • Finally, the value-based view measures quality based on how much a customer is willing to pay for a product.

- In reality, quality encompasses all these views and more.

What Is Quality?

- **Robert Glass** contends that quality is important, but if the **user** isn't satisfied, nothing else really matters.

  user satisfaction = compliant product + good quality
  + delivery within budget and schedule

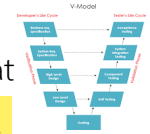
Software Quality

- In the most **general** sense, software quality can be defined as: An **effective software process** applied in a manner that creates a **useful product** that provides **measurable value** for **those who produce it** and **those who use it**.
- Three important points.

Software Quality

1

- An effective software process establishes the infrastructure that supports any effort at building a high-quality software product.



  - The management aspects of process create the **checks and balances** that help avoid project chaos—a key contributor to poor quality.

  - Software engineering practices allow the developer to **analyze the problem** and **design a solid solution**—both critical to building high-quality software.

  - Finally, umbrella **activities** such as *change management* and *technical reviews* have as much to do with quality as any other part of software engineering practice.

Software Quality

2

- A useful product delivers the content, functions, and features that the end user desires, but as important, it delivers these assets in a reliable, error-free way.

  - A useful product always satisfies those requirements that have been explicitly **stated by stakeholders**.
  - In addition, it satisfies a set of **implicit requirements** (e.g., *ease of use*) that are expected of all high-quality software.

Software Quality

- By adding value for both the producer and user of a software product, high quality software provides **benefits** for the software organization and the end user community.

  - The **SW organization** gains added value because high-quality SW requires **less maintenance effort**, **fewer bug fixes**, and **reduced customer support**.
    - This enables software engineers to spend more time creating new applications and **less on rework**.

  - The **user** community gains added value because the application provides a useful capability in a way that **expedites some business process**.
  - The **end result** is (1) greater software product **revenue**, (2) better **profitability** when an application supports a business process, and/or (3) **improved availability of information** that is crucial for the business.

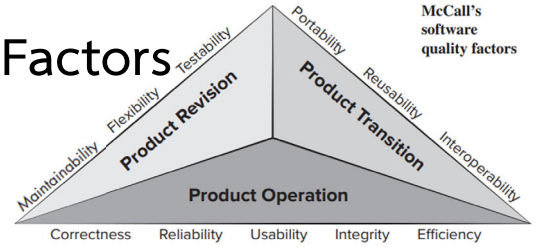
Quality Factors

McCall & Walters are researcher na

- **McCall** and **Walters** proposed a useful way to think about and organize **factors** affecting software quality.
- Their software quality factors focus on **three** software product aspects:

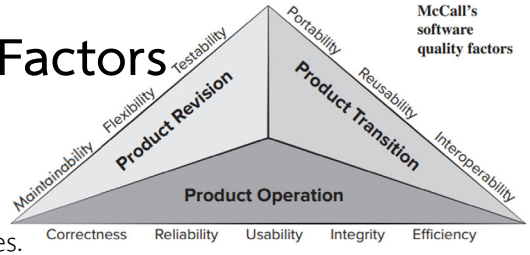
  - its operation characteristics,
  - its ability to undergo change, and
  - its adaptability to new environments.

Quality Factors



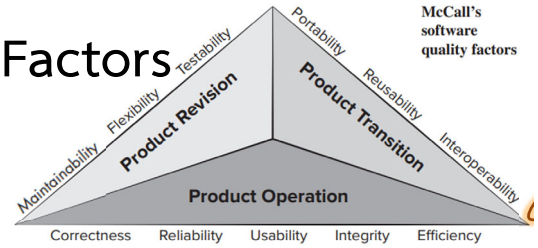
- McCall's quality factors provide a **basis** for engineering software that provides high levels of user satisfaction by **focusing on** the **overall user experience** delivered by the software product.
- This cannot be done unless developers ensure that the requirements specification is correct and that defects are removed early in the software development process.

Quality Factors



- **Correctness**. The extent to which a program satisfies its specification and fulfills the customer's mission objectives.
- **Reliability**. The extent to which a program can be expected to perform its intended function with required precision. [It should be noted that other, more complete definitions of reliability have been proposed.]
- **Efficiency**. The amount of computing resources and code required by a program to perform its function.
- **Integrity**. Extent to which access to software or data by unauthorized persons can be controlled.
- **Usability**. Effort required to *learn*, *operate*, *prepare input* for, and *interpret output* of a program.

Quality Factors



- **Maintainability**. Effort required to locate and fix an error in a program. [This is a very limited definition.]
- **Flexibility**. Effort required to modify an operational program.
- **Testability**. Effort required to test a program to ensure that it performs its intended function.
- **Portability**. Effort required to transfer the program from one hardware and/or software system environment to another.
- **Reusability**. Extent to which a program [or parts of a program] can be reused in other applications—related to the packaging and scope of the functions that the program performs.
- **Interoperability**. Effort required to couple one system to another.

Quality Factors

ISO 25010 quality model

- The ISO 25010 quality model is the newest standard (created in 2011 and revised in 2017 and 2023) (https://www.iso.org/standard/78176.html)
- This standard defines two quality models.
  - The *quality in use* model describes **five characteristics** that are appropriate when considering using the product in a particular context (e.g., using the product on a specific platform by a human).
  - The *product quality* model describes **eight characteristics** that focus on both the static and dynamic nature of computer systems.

Quality Factors

ISO 25010 quality model

- Quality in Use Model
  - Effectiveness.
    Accuracy and completeness with which users achieve goals
  - Efficiency.
    Resources expended to achieve user goals completely with desired accuracy
  - Satisfaction. Usefulness, trust, pleasure, comfort
  - Freedom from risk.
    Mitigation of economic, health, safety, and environmental risks
  - Context coverage.
    Completeness, flexibility

Quality Factors

ISO 25010 quality model

- Product Quality Model
  - Functional suitability. Complete, correct, appropriate
  - **Performance efficiency**. Timing, resource utilization, capacity
  - **Compatibility**. Coexistence, interoperability
  - **Usability**. Appropriateness, learnability, operability, error protection, aesthetics, *accessibility*
  - **Reliability**. Maturity, availability, fault tolerance, recoverability
  - **Security**. Confidentiality, integrity, accountability, authenticity
  - **Maintainability**. Modularity, reusability, modifiability, testability
  - **Portability**. Adaptability, installability, replaceability

Quality Factors

ISO 25010 quality model

- The addition of the **quality in use model** helps to **emphasize** the importance of **customer satisfaction** in the assessment of software quality.
- The **product quality model** points out the importance of assessing both the **functional** and **nonfunctional requirements** for the software product.

SW0

Qualitative Quality Assessment

padlet

- The quality dimensions and **factors** discussed previously focus on the complete software product and can be used as a generic indication of the quality of an application.
- Your software team can develop a set of quality characteristics and associated questions that would probe the degree to which each factor has been satisfied.

- How would you evaluate a user interface and assess its **usability** (which is an important quality factor in ISO 25010) ?

Slide 26

SW0 Padlet & Attendance Check

Somkiat Wangsiripitak, 2024-03-04T15:40:36.086

Qualitative Quality Assessment

Qualitative Quality Assessment

Qualitative Quality Assessment

Qualitative Quality Assessment

- As the interface design is developed, the software team would review the design prototype and **ask** the **questions** noted.
  - If the answer to most of these questions is **yes**, it is likely that the user interface exhibits high quality.
- A collection of questions similar to these would be developed for **each quality factor** to be **assessed**.
  - In the case of usability, it is always important to observe representative users interact with the system.
  - For some other quality factors it may be important to test the software in the wild (or at least in the production environment).

Quantitative Quality Assessment

- The software engineering community strives to develop precise measures for software quality and is sometimes frustrated by the subjective nature of the activity.
- Several software design defects can be detected using software metrics.

  - The process consists of finding code fragments that suggest the presence of things like high coupling or unnecessary levels of complexity.

  - Any time software metric values computed for a code fragment fall outside the range of acceptable values, it signals the existence of a quality problem that should be investigated.

Quantitative Quality Assessment

- Later in this course, we'll present a set of **software metrics** that can be applied to the quantitative assessment of software quality.
- In all cases, the metrics represent indirect measures; that is, we never really measure ***quality*** but rather some manifestation of quality.

The Software Quality Dilemma

- **Bertrand Meyer** discusses what we call the **quality dilemma**:
  - If you produce a **software** system that has terrible quality, you lose because no one will want to buy it.

  - If on the other hand you spend infinite time, extremely large effort, and huge sums of money *to build the absolutely perfect piece of software*, then it's going to take so long to complete and it will be so expensive to produce that you'll be out of business anyway.

  - People in industry try to get to that magical middle ground where the product is good enough not to be rejected right away, such as during evaluation, but also not the object of so much perfectionism and so much work that it would take too long or cost too much to complete.

"Good Enough" Software

- Software companies do it every day, creating software with known bugs and deliver it to a broad population of end users.
- They recognize that some of the functions and features delivered in version 1.0 may not be of the highest quality and **plan for improvements** in version 2.0.
- They do this knowing that some customers will complain, **but** they recognize that **time to market** may **trump** *better quality* as long as the delivered product is "good enough."

"Good Enough" Software

- Exactly what is "good enough"?
- It is true that "good enough" may work in some application domains and for a few major software companies.
  - If a **company** has a **large** marketing budget and can convince enough people to buy version 1.0, it has succeeded in locking them in.

  - It can argue that it will improve quality in subsequent versions.
  - By delivering a good enough version 1.0, it has cornered the market.

“Good Enough” Software

- If you work for a **small company**, be wary of this philosophy.
  - When you deliver a good enough (buggy) product, you risk permanent damage to your company’s reputation.
  - You may **never** get a chance to deliver version 2.0 because bad buzz may cause your sales to plummet and your company to fold.

“Good Enough” Software

- If you work in certain application domains (e.g., ***real-time*** *embedded software*) or build application software that is integrated with hardware (e.g., ***automotive*** software, ***telecommunications*** software) …
  - Delivering software with known bugs can be negligent and open your company to expensive litigation.
  - In some cases, it can even be criminal.
  - No one wants good enough aircraft avionics software!

proceed with caution

The Cost of Quality

- There is no question that quality has a cost.
- But lack of quality also has a cost—
  - **not only to end users** who must live with buggy software,
  - but **also to the software organization** that has built and must maintain it.
- The **cost of quality** can be divided into costs associated with prevention, appraisal, and failure +external failure cost

The Cost of Quality

- Prevention costs include …
  - (1) the cost of **management activities** required to plan and coordinate all **quality control** and **quality assurance** activities,
  - (2) the cost of added **technical activities** to develop complete requirements and design models,
  - (3) **test planning** costs, and
  - (4) the cost of all **training** associated with these activities.
- Investment in significant prevention costs will provide an excellent return.

The Cost of Quality

- Appraisal costs include activities to gain insight into product condition the "**first time through**" each process.
- Examples of appraisal costs include:
  (1) the cost of conducting **technical reviews** for software engineering work products,
  (2) the cost of **data collection** and **metrics evaluation**, and
  (3) the cost of **testing** and **debugging**.

The Cost of Quality

- Failure costs are those that would disappear if no errors appeared before shipping a product to customers.
  - Failure costs may be subdivided into
    internal failure costs and **external failure costs**.
- Internal failure costs are incurred when you detect an error in a product prior to shipment. They include:
  (1) the cost required to perform rework (repair) to correct an error,
  (2) the cost that occurs when rework inadvertently generates side effects that must be mitigated, and
  (3) the costs associated with the collection of quality metrics that allow an organization to assess the modes of failure.

The Cost of Quality

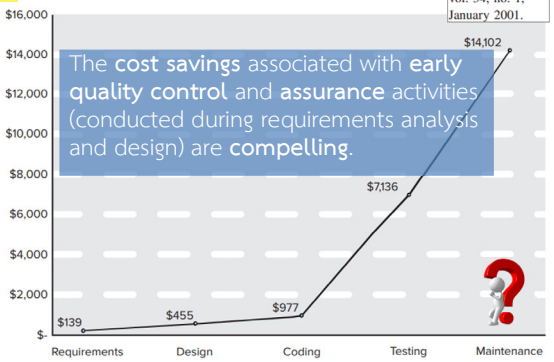
- External failure costs are associated with defects found after the product has been shipped to the customer.
  - Examples of external failure costs are complaint resolution, product return and replacement, help line support, and labor costs associated with warranty work.
  - A poor reputation and the resulting loss of business is another external failure cost that is difficult to quantify but nonetheless very real.

Bad things happen when low-quality software is produced.

The Cost of Quality

- The relative costs to find and repair an error or defect **increase** dramatically as we go from **prevention** to **detection** to **internal failure** to **external failure** costs.
- The industry average cost to correct a defect during **code generation** is approximately **$977 per error**.
- The industry average cost to correct the same error if it is discovered during **system testing** is **$7,136 per error**.

Relative cost of correcting errors and defects
Source: Boehm, Barry and Basili, Victor R., "Software Defect Reduction Top 10 List," IEEE Computer, vol. 34, no. 1, January 2001.



Risks

The Story

- The downside of poorly designed and implemented applications does not always stop with **dollars** and **time**. An extreme example :
- Throughout the month of Nov. 2000 at a hospital in Panama, 28 patients received massive overdoses of gamma rays during treatment for a variety of cancers.
- In the months that followed, 5 of these patients died from radiation poisoning and 15 others developed serious complications.
- What caused this tragedy?
- A software package, developed by a U.S. company, was modified by hospital technicians to compute modified doses of radiation for each patient.
- The three Panamanian medical physicists, who tweaked the software to provide additional capability, were charged with second-degree murder.
- The U.S. company was faced with serious litigation in two countries.

This is a warning for any creator of computer programs: **poorly deployed code can kill**.

Negligence and Liability

The Story

- A governmental or **corporate** entity **hires** a major software **developer** or consulting company to analyze requirements and then design and construct a software-based "system" to support some major activity.
- The system might support a major corporate function (e.g., pension management) or some governmental function (e.g., health care administration or homeland security).
- Work begins with the best of intentions on both sides, but by the time the system is delivered, things have gone bad.
- The system is late, fails to deliver desired features and functions, is error-prone, and does not meet with customer approval. → Litigation ensues.

Negligence and Liability

The Story

- In most cases, the **customer claims** that the developer has been **negligent** (in the manner in which it has applied software practices) and is therefore not entitled to payment.
- The **developer** often **claims** that the customer has repeatedly changed its requirements and has subverted the development partnership in other ways.
- In every case, the quality of the delivered system comes into question.

Quality and Security

- As the criticality of Web-based and mobile systems grows, application **security** has become increasingly important.
- Software that does not exhibit high quality is easier to hack,
  - Low-quality software can indirectly increase the security risk with all its attendant costs and problems.
- There are two kinds of software problems.
  - One is bugs, which are implementation problems.
  - The other is software flaws—architectural problems in the design.
- People pay too much attention to bugs and not enough on flaws.

Achieving Software Quality

- Software quality doesn't just appear.
- It is the result of good project management and solid software engineering practice.
- Management and practice are applied within the context of four broad activities that help a software team achieve high software quality:
  - software engineering methods,
  - project management techniques,
  - quality control actions, and
  - software quality assurance.

+
Machine Learning and Defect Prediction

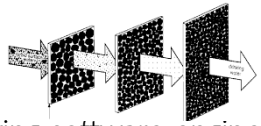
PART II

Quality Management

Software Reviews

Software Reviews

- Software **reviews** are a "filter" for the software process work flow.
  - Too few, and the flow is "dirty."
  - Too many, and the flow slows to a trickle.



- Reviews are applied at various points during software engineering and serve to uncover errors and defects.

  - Software reviews "purify" software engineering work products, including requirements and design models, code, and testing data.
- Using **metrics**, you can **determine** which reviews work and emphasize them and at the same time remove ineffective reviews from the flow to accelerate the process.

Software Reviews

What is it?

- You'll make mistakes as you develop software engineering work products.
- **Technical reviews** are the most effective mechanism for finding mistakes early in the software process.

Who does it?

- Software engineers perform technical reviews, also called **peer reviews**, with their colleagues.
- It is wise to include other stakeholders in these reviews.

Software Reviews

Why is it important?

- If you find an error early in the process, it is less expensive to correct.
- In addition, errors have a way of **amplifying** as the process proceeds.
  - So, a relatively minor error left untreated early in the process can be amplified into a major set of errors later in the project.

Defect Propagation

- Finally, reviews save time by reducing the amount of rework that will be required late in the project.

Software Reviews

What are the steps?

- Your approach to reviews will vary depending on the type of review you select.
- In general, **six steps** are employed, although not all are used for every type of review:
  - planning, preparation,
  - structuring the meeting,
  - noting errors,
  - making corrections (done outside the review), and
  - verifying that corrections have been performed properly.




Software Reviews

What is the work product?

- The output of a review is **a list of issues and/or errors** that have been uncovered.
- In addition, the technical status of the work product is also indicated.

How do I ensure that I've done it right?

- First, select the type of review that is appropriate for your development culture.
- Follow the guidelines that lead to successful reviews.
- If the reviews that you conduct **lead to higher-quality software**, you've done it right.

Software Reviews

- In this course, we focus on technical or peer reviews, exemplified by casual reviews, walkthroughs, and inspections.
- A **technical review** (TR) is the most effective filter from a quality control standpoint.
  - *Conducted by* software engineers and other stakeholders for all project team members, the TR is an effective means for uncovering errors and improving software quality.

Software Reviews

Errors and Defects

- Sometimes we make a clear **distinction** between …
  - an error (a quality problem found before the software is released to other stakeholders or end users) and
  - a **defect** (a quality problem found only after the software has been released to end users or other stakeholders).
- If software process improvement is considered, a quality **problem** that is **propagated from** one process framework activity (e.g., modeling) **to** another (e.g., construction) can also be called a "defect" because the problem should have been found before a work product (e.g., a design model) was "released" to the next activity.

The temporal distinction made between errors and defects here is **not mainstream** thinking.

Review Metrics and Their Use

- Because available project effort is **finite**, it is important for a SE organization to **understand** the **effectiveness of each action** by defining a set of **metrics** that can be used to assess their efficacy.
- The following **review metrics** can be collected for each review:
- **Preparation effort**, E_p.
  The effort (in person-hours) required to review a work product prior to the actual review meeting
- **Assessment effort**, E_a.
  The effort (in person-hours) that is expended during the actual review

Review Metrics and Their Use

- **Rework effort**, E_r.
  The effort (in person-hours) that is dedicated to the correction of those errors uncovered during the review
- **Review effort**, E_{review}.
  Represents the sum of effort measures for reviews: $E_{review} = E_p + E_a + E_r$
- **Work product size** (WPS).
  A measure of the size of the work product that has been reviewed (e.g., the number of UML models, the number of document pages, or the number of lines of code)

Review Metrics and Their Use

- **Minor errors found**, Err_{minor}.
  The number of errors found that can be categorized as minor (requiring less than some prespecified effort to correct)
- **Major errors found**, Err_{major}.
  The number of errors found that can be categorized as major (requiring more than some prespecified effort to correct)
- **Total errors found**, Err_{tot}.
  Represents the sum of the errors found: $Err_{tot} = Err_{minor} + Err_{major}$
- **Error density**.
  Represents the errors found per unit of work product reviewed:
  Error density = Err_{tot} / WPS

Review Metrics and Their Use

- **Example** of using these metrics.
- Consider a requirements model that is reviewed to uncover errors, inconsistencies, and omissions.
- It would be possible to compute the error density in several different ways.
- Assume the requirements model contains 18 UML diagrams as part of 32 overall pages of descriptive materials.
- The review uncovers 18 minor errors and 4 major errors.
- Therefore, Err_{tot} = 
- Error density is  errors per UML diagram, or  errors per requirements model page.

Review Metrics and Their Use

- If reviews are conducted for a number of different types of work products (e.g., requirements model, design model, code, test cases) …
  - The **percentage** of errors uncovered for *each review* can be computed **against** the total number of errors found for *all reviews*.
  - In addition, the **error density** for each work product can be computed.

Review Metrics and Their Use

- Once data are collected for many reviews conducted across many projects, **average** values for **error density** enable you to estimate the number of errors to be found in a new document before it is reviewed.
- For example, if the average error density for a requirements model is 0.68 errors per page, and
  a new requirements model is 40 pages long, …
  a rough estimate suggests that your software team will find around  errors during the review of the document.
- If you find only 9 errors, you've either …
  - done an **extremely good job** in developing the requirements model **or**
  - your **review** approach was **not thorough enough**.

Review Metrics and Their Use

- It is **difficult to measure** the cost effectiveness of any technical review in **real time**.
  - A software engineering organization can assess the effectiveness of reviews and their cost benefit **only after reviews** have been **completed**, review metrics have been collected, average data have been computed, and then the downstream quality of the software is measured (via testing).

Review Metrics and Their Use

- Returning to the previous example, the average error density for requirements models was determined to be 0.68 per page.
- The effort required to correct a minor model error (immediately after the review) has been found to require 4 person-hours.
- The effort required for a major requirement error has been found to be 18 person-hours.
- Examining the review data collected, you find that minor errors occur about **six times** more frequently than major errors.
- Therefore, you can estimate that the **average effort** to find and correct a requirements error during review is about  person-hours. ($E_{reviews}$)

Review Metrics and Their Use

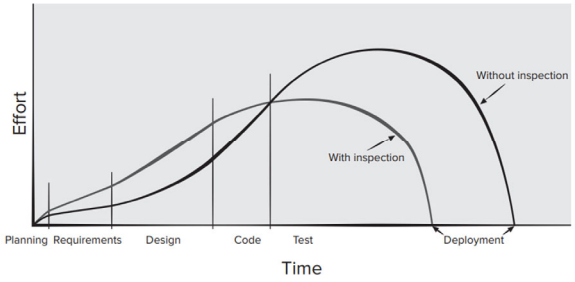
Slip through

- If Requirements-related errors **uncovered during testing** require an average of 45 person-hours ($E_{testing}$) to find and correct ... (no data are available on the relative severity of the error).
- Using the averages noted, we get: Effort saved per error = $E_{testing} - E_{reviews}$ =  person-hours/error
- Because 22 errors (Err_{tot}) were found during the review of the requirements model, a savings of about  person-hours of testing effort would be achieved.
- And that's just for requirements-related errors.
- Errors associated with design and code would add to the overall benefit.

Review Metrics and Their Use

- Effort saved leads to shorter delivery cycles and improved time to market.
- More than three decades of **Industry data for software reviews** (the figure).
- The effort expended when reviews are used does increase early in the development of a software increment, but this early investment for reviews pays dividends because testing and corrective effort is reduced.
- The deployment date for development with reviews is sooner than the deployment date without reviews.

Effort expended with and without reviews
Source: Fagan, Michael E., "Advances in Software Inspections," IEEE Transactions on Software Engineering, vol. SE-12, no. 7, July 1986, 744–751.



Postmortem Evaluations

- Many **lessons** can be **learned** if a software team takes the time to evaluate the results of a software project after the software has been delivered to end users.
- Use a postmortem evaluation (**PME**) as a mechanism to determine what went right and what went wrong when software engineering process and practice is applied in a specific project.
  - It is more like a Scrum retrospective.
- A PME examines the entire software project, focusing on both "excellences (that is, achievements and positive experiences) and challenges (problems and negative experiences)".

Postmortem Evaluations

- Often conducted in a **workshop format**, a PME is attended by members of the software team and stakeholders.
- The intent is to identify **excellences** and **challenges** and to extract **lessons learned** from both.
- The objective is to suggest **improvements** to both process and practice going forward.
- Many software engineers regard **PME documents** as some of the most **valuable documents** to save in the project archive.

More details of software reviews are explained in '**software verification and validation**' course.