RESPONSI UAS

COMP6100001 - SOFTWARE ENGINEERING

UNIVERSITAS **BINA NUSANTARA**

SUBJECT MATTER EXPERT

People
Innovation
Excellence

BINUS
UNIVERSITY

UAS SOFTWARE ENGINEERING

LEARNING OBJECTIVES

1. System Analysis & Risk Identification (20%)
2. Testing & Quality Engineering (25%)
3. DevOps, SCM, and Deployment (20%)
4. Reliability, Security, and Maintenance (20%)
5. Metrics, Ethics, and Improvement Strategy (15%)

PART 1
SYSTEM ANALYSIS & RISK IDENTIFICATION

UNIVERSITAS **BINA NUSANTARA**

SUBJECT MATTER EXPERT

People
Innovation
Excellence

BINUS
UNIVERSITY

RISK DEFINITION

TWO KEY COMPONENTS

Risk is defined by two key components:

- **The probability (or likelihood) of failing to achieve a particular outcome**
- **The consequences (or impact) of failing to achieve that outcomes**

Internal vs. External Risk

- Internal: Risks that we **can** control
- External: Risks that we **cannot** control

People
Innovation
Excellence

BINUS
UNIVERSITY

REACTIVE VERSUS PROACTIVE RISK STRATEGIES

REACTIVE RISK MANAGEMENT

- project team reacts to risks when they occur
- mitigation—plan for additional resources in anticipation of fire fighting
- fix on failure—resource are found and applied when the risk strikes
- crisis management—failure does not respond to applied resources and project is in jeopardy

People
Innovation
Excellence

BINUS
UNIVERSITY

SOFTWARE RISK

SEVEN PRINCIPLES

- **Maintain a global perspective**—view software risks within the context of system and the business problem
- **Take a forward-looking view**—think about the risks that may arise in the future; establish contingency plans
- **Encourage open communication**—if someone states a potential risk, don't discount it.
- **Integrate**—a consideration of risk must be integrated into the software process
- **Emphasize a continuous process**—the team must be vigilant throughout the software process, modifying identified risks as more information is known and adding new ones as better insight is achieved.
- **Develop a shared product vision**—if all stakeholders share the same vision of the software, it likely that better risk identification and assessment will occur.
- **Encourage teamwork**—the talents, skills and knowledge of all stakeholder should be pooled

People
Innovation
Excellence

BINUS
UNIVERSITY

SOFTWARE RISK

RISK IDENTIFICATION

- **Product size** —risks associated with the overall size of the software to be built or modified.
- **Business impact** —risks associated with constraints imposed by management or the marketplace.
- **Customer characteristics** —risks associated with the sophistication of the customer and the developer's ability to communicate with the customer in a timely manner.
- **Process definition** —risks associated with the degree to which the software process has been defined and is followed by the development organization.
- **Development environment** —risks associated with the availability and quality of the tools to be used to build the product.
- **Technology to be built** —risks associated with the complexity of the system to be built and the "newness" of the technology that is packaged by the system.
- **Staff size and experience** —risks associated with the overall technical and project experience of the software engineers who will do the work.

People
Innovation
Excellence

BINUS
UNIVERSITY

SOFTWARE RISK

LEVELS OF RISK MANAGEMENT

1. **Crisis management**: Fire fighting; address risks only after they have become problems.
2. **Fix on failure**: Detect and react to risks quickly, but only after they have occurred.
3. **Risk mitigation**: Plan ahead of time to provide resources to cover risks if they occur, but do nothing to eliminate them in the first place.
4. **Prevention**: Implement and execute a plan as part of the software project to identify risks and prevent them from becoming problems.
5. **Elimination of root causes**: Identify and eliminate factors that make it possible for risks to exist at all.

SOFTWARE RISK

RISK COMPONENTS

- **performance risk**—the degree of uncertainty that the product will meet its requirements and be fit for its intended use.
- **cost risk**—the degree of uncertainty that the project budget will be maintained.
- **support risk**—the degree of uncertainty that the resultant software will be easy to correct, adapt, and enhance.
- **schedule risk**—the degree of uncertainty that the project schedule will be maintained and that the product will be delivered on time.

SOFTWARE RISK

RISK PROJECTION

- Risk projection, also called risk estimation, attempts to rate each risk in two ways
  - the likelihood or probability that the risk is real
  - the consequences of the problems associated with the risk, should it occur.
- The are four risk projection steps:
  - establish a scale that reflects the perceived likelihood of a risk
  - delineate the consequences of the risk
  - estimate the impact of the risk on the project and the product,
  - note the overall accuracy of the risk projection so that there will be no misunderstandings.

BINUS UNIVERSITY

People
Innovation
Excellence

SOFTWARE RISK

IMPACT ASSESSMENT

Category \ Components		Performance	Support	Cost	Schedule
Catastrophic	1	Failure to meet the requirement would result in mission failure		Failure results in increased costs and schedule delays with expected values in excess of $500K	
	2	Significant degradation to nonachievement of technical performance	Nonresponsive or unsupportable software	Significant financial shortages, budget overrun likely	Unachievable IOC
Critical	1	Failure to meet the requirement would degrade system performance to a point where mission success is questionable		Failure results in operational delays and/or increased costs with expected value of $100K to $500K	
	2	Some reduction in technical performance	Minor delays in software modifications	Some shortage of financial resources, possible overruns	Possible slippage in IOC
Marginal	1	Failure to meet the requirement would result in degradation of secondary mission		Costs, impacts, and/or recoverable schedule slips with expected value of $1K to $100K	
	2	Minimal to small reduction in technical performance	Responsive software support	Sufficient financial resources	Realistic, achievable schedule
Negligible	1	Failure to meet the requirement would create inconvenience or nonoperational impact		Error results in minor cost and/or schedule impact with expected value of less than $1K	
	2	No reduction in technical performance	Easily supportable software	Possible budget underrun	Early achievable IOC

Note: [1] The potential consequence of undetected software errors or faults.
[2] The potential consequence if the desired outcome is not achieved.

SOFTWARE RISK

BUILDING THE RISK TABLE

- Estimate the probability of occurrence
- Estimate the impact on the project on a scale of 1 to 4, where
  - 1 = catastrophic
  - 2 = critical
  - 3 = marginal
  - 4 = negligible
- sort the table by probability and impact

People
Innovation
Excellence

BINUS
UNIVERSITY

SOFTWARE RISK

SAMPLE RISK TABLE PRIOR TO SORTING

Risks	Category	Probability	Impact	RMMM
Size estimate may be significantly low	PS	60%	2	
Larger number of users than planned	PS	30%	3	
Less reuse than planned	PS	70%	2	
End users resist system	BU	40%	3	
Delivery deadline will be tightened	BU	50%	2	
Funding will be lost	CU	40%	1	
Customer will change requirements	PS	80%	2	
Technology will not meet expectations	TE	30%	1	
Lack of training on tools	DE	80%	3	
Staff inexperienced	ST	30%	2	
Staff turnover will be high	ST	60%	2	
ΣΣΣ				

Impact values:
1—catastrophic
2—critical
3—marginal
4—negligible

SOFTWARE RISK

RISK EXPOSURE (IMPACT)

The overall risk exposure, RE, is determined using the following relationship [Hal98]:

$$RE = P \times C$$

where

P is the probability of occurrence for a risk, and

C is the cost to the project should the risk occur.

SOFTWARE RISK

RISK EXPOSURE EXAMPLE

- **Risk identification.** Only 70 percent of the software components scheduled for reuse will, in fact, be integrated into the application. The remaining functionality will have to be custom developed.
- **Risk probability.** 80% (likely).
- **Risk impact.** 60 reusable software components were planned. If only 70 percent can be used, 18 components would have to be developed from scratch (in addition to other custom software that has been scheduled for development). Since the average component is 100 LOC and local data indicate that the software engineering cost for each LOC is $14.00, the overall cost (impact) to develop the components would be 18 x 100 x 14 = $25,200.
- **Risk exposure.** RE = 0.80 x 25,200 ~ $20,200.

SOFTWARE RISK

RISK MITIGATION, MONITORING, AND MANAGEMENT

- Mitigation —how can we avoid the risk?
- Monitoring —what factors can we track that will enable us to determine if the risk is becoming more or less likely?
- Management —what contingency plans do we have if the risk becomes a reality?

PART 2
TESTING & QUALITY ENGINEERING

UNIVERSITAS **BINA NUSANTARA**

SUBJECT MATTER EXPERT

People
Innovation
Excellence

BINUS
UNIVERSITY

SOFTWARE QUALITY

INTERNAL VS EXTERNAL QUALITY

Internal Quality

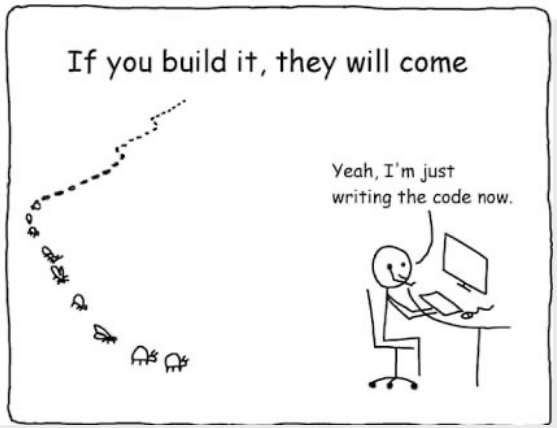
- Is the code well structured?
- Is the code understandable?
- How well documented?

External Quality

- Does the software crash?
- Does it meet the requirements?
- Is the UI well designed?

OBJECTIVES AND IMPORTANCE OF SOFTWARE TESTING

PRINCIPLES OF TESTING #1

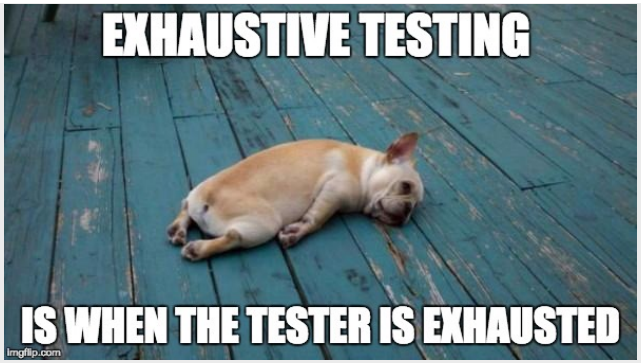


Avoid the *absence of defects* fallacy

- Testing shows the presence of defects
- Testing does not show the absence of defects
- “No test team can achieve 100% defect detection effectiveness”

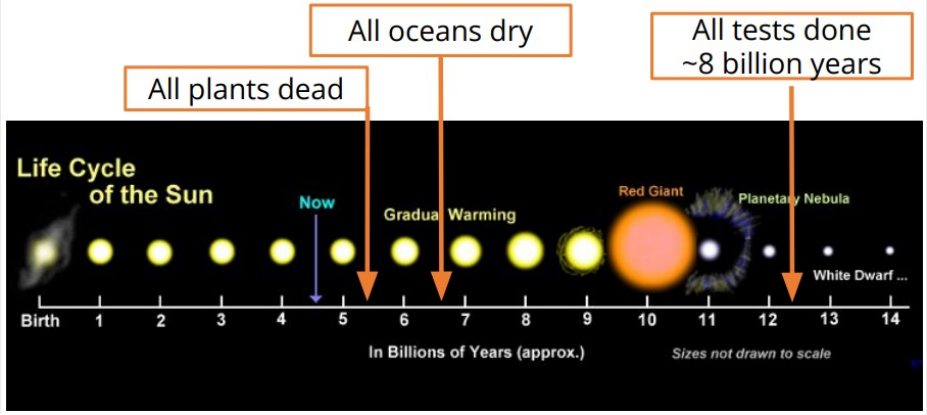
OBJECTIVES AND IMPORTANCE OF SOFTWARE TESTING

PRINCIPLES OF TESTING #2



Exhaustive testing is impossible

- A simple function, 1 input, string, max. 26 lowercase characters + symbols (@,.,_,-)
- Assume we can use 1 zettaFLOPS: 10^{21} tests per second



OBJECTIVES AND IMPORTANCE OF SOFTWARE TESTING

PRINCIPLES OF TESTING #3

Start testing early

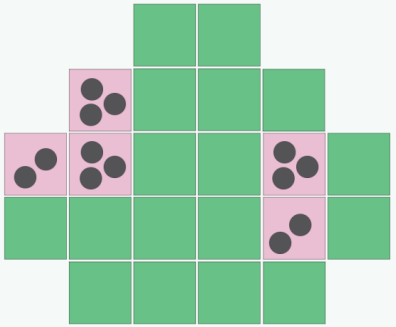
- To let tests guide design
- To get feedback as early as possible
- To find bugs when they are cheapest to fix
- To find bugs when have caused least damage

OBJECTIVES AND IMPORTANCE OF SOFTWARE TESTING

PRINCIPLES OF TESTING #4

Defects are usually clustered

- "Hot" components requiring frequent change, bad habits, poor developers, tricky logic, business uncertainty, innovative, size, ...
- Use as heuristic to focus test effort



Clustered Defects

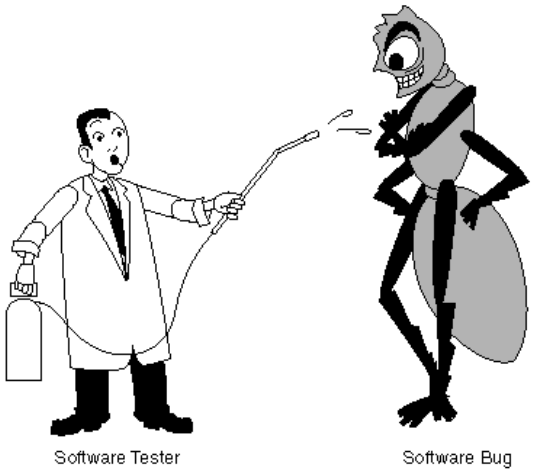
OBJECTIVES AND IMPORTANCE OF SOFTWARE TESTING

PRINCIPLES OF TESTING #5

The pesticide paradox

"Every method you use to prevent or find bugs leaves a residue of subtler bugs against which those methods are ineffectual."

- Re-running the same test suite again and again on a changing program gives a false sense of security
- Variation in testing

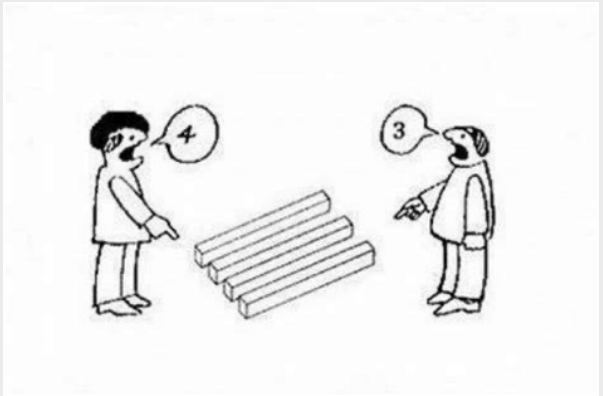


OBJECTIVES AND IMPORTANCE OF SOFTWARE TESTING

PRINCIPLES OF TESTING #6

Testing is context-dependent

- Depends on what kind of application that will be developed



Perspective is everything.

OBJECTIVES AND IMPORTANCE OF SOFTWARE TESTING

PRINCIPLES OF TESTING #7

Verification is not validation

- **Verification**
  - Does the software system meet the requirements specifications?
  - Are we building the **software right**?
- **Validation**
  - Does the software system meet the user's real needs?
  - Are we building the **right software**?

People
Innovation
Excellence

BINUS
UNIVERSITY

LEVELS OF TESTING

UNIT TESTING

- Focuses on individual components or functions.
- Ensures that each component performs as expected in isolation.
- **Example**: Testing a login function independently of other components.

People
Innovation
Excellence

BINUS
UNIVERSITY

LEVELS OF TESTING

INTEGRATION TESTING

- Tests interactions between integrated components.
- Ensures modules work together as expected.
- **Example**: Testing that the login feature successfully retrieves data from a database.

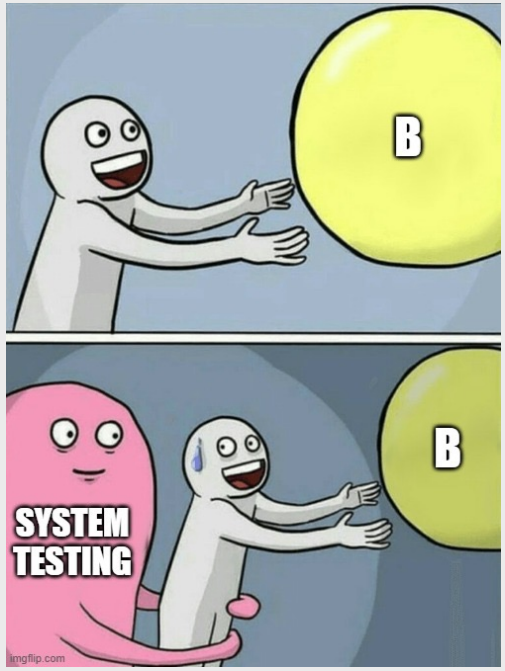
People
Innovation
Excellence

BINUS
UNIVERSITY

LEVELS OF TESTING

SYSTEM TESTING

- Examines the complete, integrated system.
- Validates that the software meets specified requirements.
- **Example**: Testing all modules of an e-commerce app to ensure smooth purchasing flow.



People
Innovation
Excellence

BINUS
UNIVERSITY

LEVELS OF TESTING

ACCEPTANCE TESTING

- Conducted from the end-user perspective to ensure the software meets their needs.
- Often involves both alpha (internal) and beta (external) testing.
- **Example**: Users test a social media app's features in a real-world setting.

People
Innovation
Excellence

BINUS
UNIVERSITY

TESTING TECHNIQUES

BLACK-BOX, WHITE-BOX, GREY-BOX

- **Black-Box Testing**:
  - Tests based on requirements and user expectations without knowledge of internal code.
  - Useful for functional and usability testing.
  - **Example**: Testing if form validation works without examining code.
- **White-Box Testing**:
  - Requires knowledge of the internal code structure, focusing on logic and data flow.
  - Useful for unit and integration testing.
  - **Example**: Testing specific functions within a login script.
- **Grey-Box Testing**:
  - Combines elements of black-box and white-box testing.
  - Useful for testers with partial knowledge of code, often used for integration.
  - **Example**: Testing with some knowledge of system interactions, such as APIs.

DEFECT IDENTIFICATION

WHAT IS STATIC ANALYSIS?

- **Systematic** examination of an **abstraction** of program **state space**.
- Does not execute code! (like code review)
- **Abstraction**: produce a representation of a program that is simpler to analyze.
- Results in fewer states to explore; makes difficult problems tractable.
- Check if a **particular property** holds over the entire state space:
- Liveness: “something good eventually happens.”
- Safety: “this bad thing can’t ever happen.”
- Compliance with mechanical design rules.

People
Innovation
Excellence

BINUS
UNIVERSITY

DEFECT IDENTIFICATION

DYNAMIC ANALYSIS REASONS ABOUT PROGRAM EXECUTIONS

- Tells you properties of the program that were definitely observed
  - Code coverage
  - Performance profiling
  - Type profiling
  - Testing
- In practice, implemented by program instrumentation
  - Think “Automated logging”
  - Slows down execution speed by a small amount

People
Innovation
Excellence

BINUS
UNIVERSITY

DEFECT IDENTIFICATION

STATIC ANALYSIS VS DYNAMIC ANALYSIS

Static Analysis

- Requires only source code
- Conservatively reasons about all possible inputs and program paths
- Reported warnings may contain false positives
- Can report all warnings of a particular class of problems
- Advanced techniques like verification can prove certain complex properties, but rarely run in CI due to cost

Dynamic Analysis

- Requires successful build + test inputs
- Observes individual executions
- Reported problems are real, as observed by a witness input
- Can only report problems that are seen. Highly dependent on test inputs. Subject to false negatives
- Advanced techniques like symbolic execution can prove certain complex properties, but rarely run in CI due to cost

People
Innovation
Excellence

BINUS
UNIVERSITY

DEFINITION AND IMPORTANCE OF SOFTWARE RELIABILITY

- **Definition of Software Reliability**:
  - Software reliability is the probability of failure-free software operation over a specified period under specified conditions.
- **Importance of Reliability**:
  - Reliable software minimizes unexpected failures, ensuring that critical systems (e.g., healthcare, finance, transportation) operate safely and consistently.
  - Reliability is closely linked to user satisfaction and safety; failures in systems like aircraft control or medical devices can have severe consequences.
- **Example**: Importance of reliability in banking applications, where downtime impacts user trust and financial integrity.

RELIABILITY MODELS AND METRICS

COMMON RELIABILITY MODELS

- **Failure Rate Model**: Calculates the probability of failure per unit time.
- **Mean Time Between Failures (MTBF)**: Average time between system failures; the higher the MTBF, the more reliable the system.
- **Mean Time to Repair (MTTR)**: Average time taken to repair a system after a failure.
- A simple measure of reliability is mean-time-between-failure (MTBF), where

$$MTBF = MTTF + MTTR$$

- The acronyms MTTF and MTTR are mean-time-to-failure and mean-time-to-repair, respectively.

FAULT TOLERANCE TECHNIQUES

REDUNDANCY

- Involves having duplicate systems or components to take over in case of failure.
- **Example**: A load-balanced server setup where traffic is distributed across multiple servers; if one fails, others continue to serve users.

People
Innovation
Excellence

BINUS
UNIVERSITY

FAULT TOLERANCE TECHNIQUES

GRACEFUL DEGRADATION

- Ensures that even if a system component fails, the system continues to operate at a reduced capacity.

- **Example**: A video streaming service might reduce video quality temporarily if high-definition servers fail, maintaining the service at a lower quality.

People
Innovation
Excellence

BINUS
UNIVERSITY

FAULT TOLERANCE TECHNIQUES

RECOVERY MECHANISMS

- **Checkpointing and Rollback**: Regularly saves system states to enable rollbacks after failures.
- **Failover Systems**: Automatically switches to a backup system when a failure occurs.
- **Self-Healing Mechanisms**: Systems detect, isolate, and correct errors without human intervention.
- **Example**: In databases, checkpoints allow recovery to the last stable state after a crash, preserving data integrity.

DESIGNING FOR RELIABILITY IN SOFTWARE ARCHITECTURE

PRINCIPLES OF RELIABILITY IN ARCHITECTURE

- **Modularity**: Designing the system as independent modules isolates faults.
- **Redundancy and Replication**: Adding extra components or copies of services to prevent complete failure.
- **Error Detection and Correction**: Embedding mechanisms that detect and automatically correct errors.
- **Failover Architecture**: Setting up backup systems that take over instantly in case of failure.

People
Innovation
Excellence

BINUS
UNIVERSITY

DESIGNING FOR RELIABILITY IN SOFTWARE ARCHITECTURE

RELIABILITY PATTERNS

- **Circuit Breaker Pattern**: Prevents repeated attempts to access a failed service.
- **Bulkhead Pattern**: Isolates components to prevent a failure in one from impacting others.
- **Event-Driven Architecture**: Uses decoupled services to handle high availability and reliability.
- **Example**: A microservices architecture for an e-commerce site uses circuit breakers to isolate failing payment services, allowing other services to continue operating.

People
Innovation
Excellence

BINUS
UNIVERSITY

MEASURING AND IMPROVING SOFTWARE RELIABILITY

MEASUREMENT TECHNIQUES

- **Failure Tracking**: Recording failure occurrences and analyzing patterns to target improvements.
- **User Feedback**: Tracking customer complaints and reported issues to identify reliability gaps.

BINUS UNIVERSITY

PART 3
DEVOPS, SCM, AND DEPLOYMENT

UNIVERSITAS **BINA NUSANTARA**

SUBJECT MATTER EXPERT

People
Innovation
Excellence

BINUS
UNIVERSITY

INTRODUCTION TO SOFTWARE CONFIGURATION MANAGEMENT

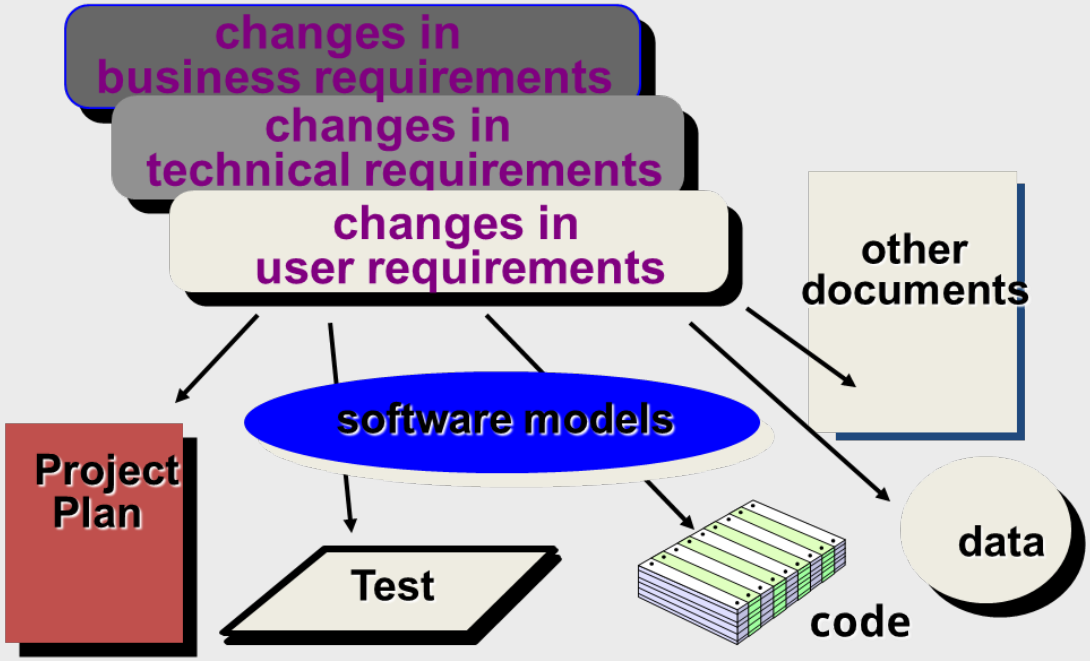
DEFINITION AND IMPORTANCE OF SCM

- **Software Configuration Management (SCM)** is a discipline within software engineering that focuses on controlling and tracking changes in software, ensuring integrity and traceability throughout the software development life cycle.

- **Importance**:
  - **Control of Change**: Manages and documents changes to prevent errors and inconsistencies.
  - **Team Collaboration**: Facilitates coordination among team members, especially in large projects.
  - **Quality Assurance**: Ensures that the software maintains its integrity over time.
  - **Compliance and Traceability**: Provides an audit trail for changes, important for regulatory compliance.

INTRODUCTION TO SOFTWARE CONFIGURATION MANAGEMENT

WHAT ARE THESE CHANGES?



KEY CONCEPTS IN SCM

SOFTWARE CONFIGURATION ITEMS (SCI)

- **SCI** are the artifacts that are placed under configuration management. This includes code files, documents, requirements, test cases, and more.
- **Examples**:
  - Source code files
  - Design documents
  - User manuals
  - Test scripts
  - Build scripts
  - Requirements specifications

KEY CONCEPTS IN SCM

BASELINES

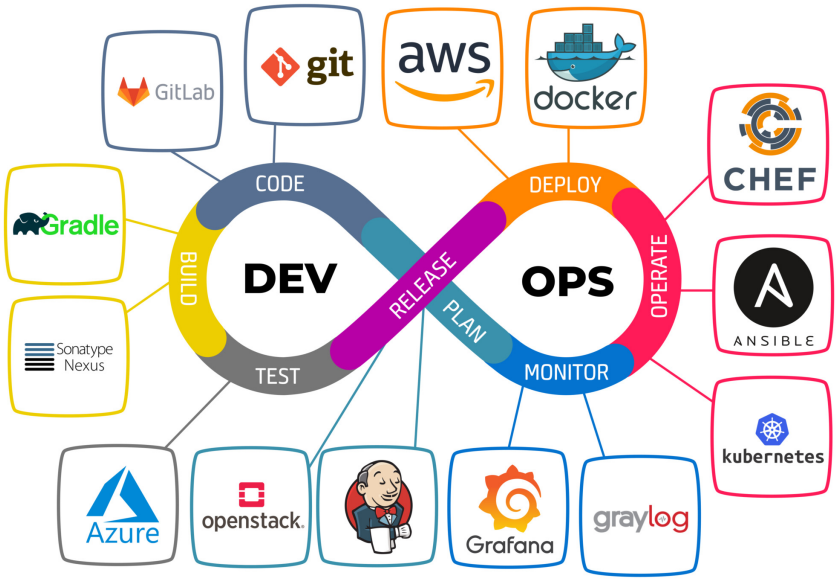
- A **baseline** is a formally reviewed and agreed-upon version of a configuration item, which serves as a starting point for further development and can be changed only through formal change control procedures.

- **Types of Baselines**:
  - **Functional Baseline**: Initial specifications established during requirements analysis.
  - **Design Baseline**: Established after the design phase.
  - **Product Baseline**: The final product, including all components and documentation.

DEFINITION AND GOALS OF DEVOPS

DEFINITION OF DEVOPS

- DevOps is a set of practices, principles, and cultural philosophies aimed at unifying software development (Dev) and IT operations (Ops).
- Focuses on collaboration, automation, and feedback loops to deliver high-quality software at speed.



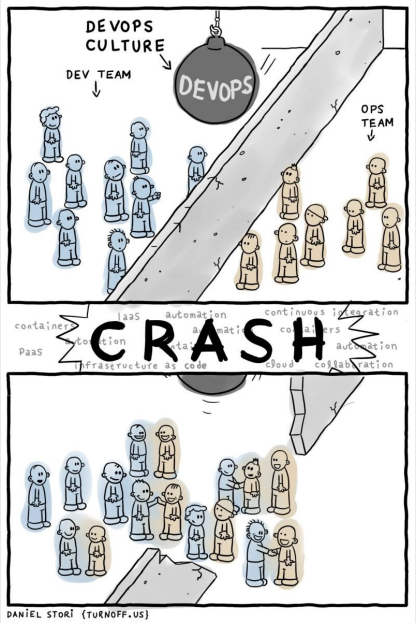
People
Innovation
Excellence

BINUS
UNIVERSITY

CULTURAL SHIFT: COLLABORATION BETWEEN DEVELOPMENT AND OPERATIONS

TRADITIONAL VS DEVOPS CULTURE

- In traditional models, development and operations work separately, leading to miscommunication and delays.
- DevOps advocates for shared responsibilities and joint efforts throughout the software lifecycle.



People
Innovation
Excellence

BINUS
UNIVERSITY

DEVOPS PRINCIPLES

AUTOMATION

- Automates repetitive tasks, reducing human error and speeding up processes.
- **Example**: Using scripts to automate testing, deployment, and infrastructure setup.

DEVOPS PRINCIPLES

CONTINUOUS INTEGRATION (CI)

- Developers frequently merge code changes into a shared repository where automated tests validate the changes.
- **Example**: GitHub Actions or Jenkins running automated tests on every code commit.

DEVOPS PRINCIPLES

CONTINUOUS DELIVERY (CD)

- Extends CI by ensuring the code is always ready to be deployed to production.
- **Example**: Staging environment where validated builds are automatically deployed and await approval for production.

People
Innovation
Excellence

BINUS
UNIVERSITY

OVERVIEW OF DEVOPS TOOLCHAINS

DEFINITION AND PURPOSE OF DEVOPS TOOLCHAINS

- **Definition of a DevOps Toolchain**:
  - A DevOps toolchain is a set of tools that automate and streamline the development, testing, integration, and deployment processes. It covers the software lifecycle from coding to monitoring in production.
- **Purpose of a DevOps Toolchain**:
  - Enhances collaboration between teams, reduces errors, accelerates deployment, and supports continuous integration and deployment.

People
Innovation
Excellence

BINUS
UNIVERSITY

OVERVIEW OF DEVOPS TOOLCHAINS

COMPONENTS OF A TYPICAL TOOLCHAIN

- **Planning Tools**: Track requirements and manage workflows (e.g., Jira).
- **Version Control Tools**: Manage code versions (e.g., Git).
- **Build and CI/CD Tools**: Automate testing, integration, and deployment (e.g., Jenkins).
- **Monitoring and Logging Tools**: Observe system health and analyze logs (e.g., Prometheus).

Example: A retail company's toolchain includes Git for version control, Jenkins for CI/CD, Docker for containerization, and Grafana for monitoring performance.

People
Innovation
Excellence

BINUS
UNIVERSITY

CONTINUOUS INTEGRATION TOOLS

JENKINS AND TRAVIS CI

- CI automates the process of integrating code changes frequently and testing them to catch issues early.

Jenkins:

- **Features**: Open-source, extensive plugin library, customizable pipelines.
- **Use Case**: Automating test runs on code commits to ensure stability.
- **Example**: Running unit tests and code quality checks on each commit to GitHub.

Travis CI:

- **Features**: Cloud-based CI tool with GitHub integration, easy configuration via .travis.yml.
- **Use Case**: Automating build and test processes for open-source projects.
- **Example**: Automatically testing and deploying a Node.js application to a staging environment.

People
Innovation
Excellence

BINUS
UNIVERSITY

CONTINUOUS DEPLOYMENT TOOLS

SPINNAKER AND GOCD

- CD automates the deployment process, pushing validated code to production without manual intervention.

Spinnaker:

- **Features**: Multi-cloud deployment support, pipelines for release management.
- **Use Case**: Deploying applications across cloud platforms like AWS and Google Cloud.
- **Example**: A microservices architecture deploying independently to AWS and Kubernetes clusters.

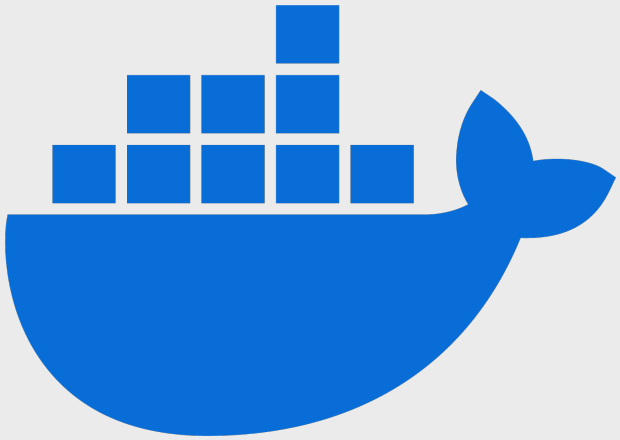
GoCD:

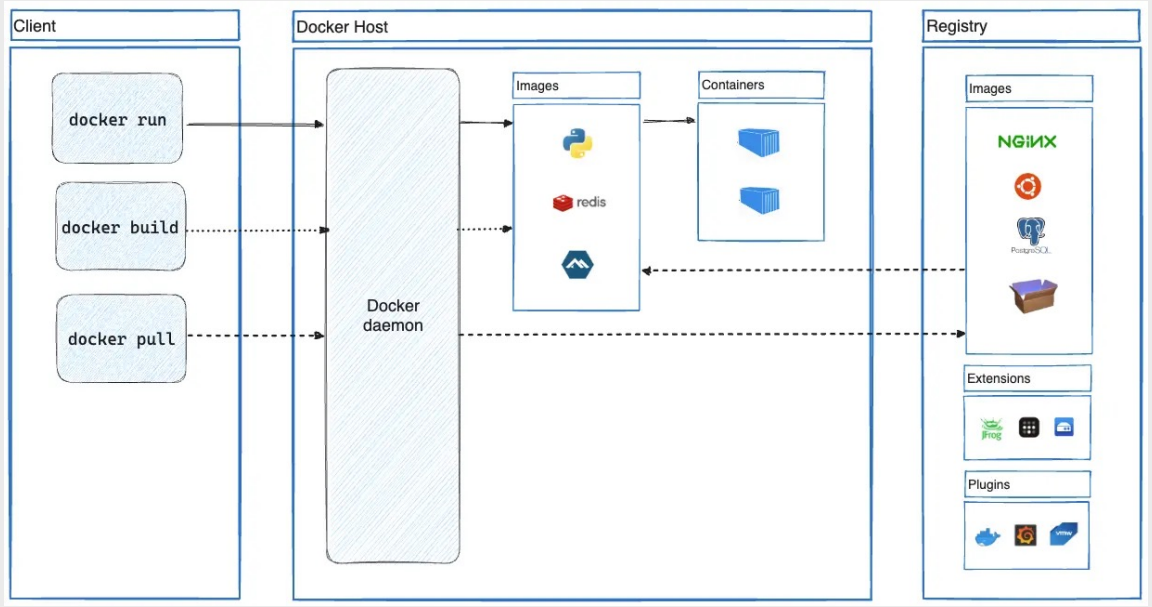
- **Features**: Visual pipeline modeling, supports multiple deployment environments.
- **Use Case**: Setting up CD pipelines with stages for testing, staging, and production.
- **Example**: A Java application deploying from a development environment to production through GoCD pipelines.

CONTAINERIZATION AND ORCHESTRATION

CONTAINERIZATION WITH DOCKER

- Docker packages applications into containers, enabling consistency across environments.
- **Benefits**: Environment consistency, easy scaling, and efficient resource use.
- **Example**: Packaging a Python application in a Docker container to run identically on different servers.

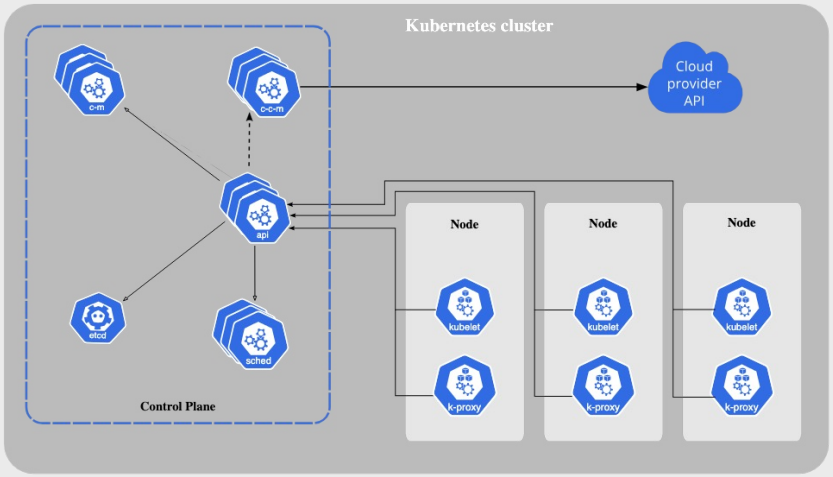




CONTAINERIZATION AND ORCHESTRATION

ORCHESTRATION WITH KUBERNETES

- Kubernetes automates the deployment, scaling, and management of containerized applications.
- **Key Concepts**: Pods, clusters, nodes, and services.
- **Use Case**: Managing microservices by deploying and scaling containers across multiple nodes.
- **Example**: Deploying a web application with an auto-scaling feature to handle varying traffic loads.



People
Innovation
Excellence

BINUS
UNIVERSITY

INFRASTRUCTURE AS CODE (IAC) TOOLS

TERRAFORM AND ANSIBLE

- IaC involves managing and provisioning infrastructure through code instead of manual processes.

Terraform:

- **Features**: Supports multiple cloud providers, declarative syntax.
- **Use Case**: Defining and managing cloud infrastructure with reusable code templates.
- **Example**: Using Terraform to set up and configure AWS resources like EC2 instances, databases, and networks.

Ansible:

- **Features**: Agentless configuration management, focuses on automation of setup and updates.
- **Use Case**: Automating server configuration and software installation across multiple machines.
- **Example**: Ansible playbooks to install, configure, and maintain applications across servers.

People
Innovation
Excellence

BINUS
UNIVERSITY

CONTINUOUS MONITORING AND LOGGING

PROMETHEUS, GRAFANA, AND ELK STACK

- Continuous monitoring provides insights into application performance, availability, and reliability in real-time.

Prometheus:

- **Features**: Open-source, time-series database, powerful querying language.
- **Use Case**: Tracking metrics for CPU usage, memory consumption, and network activity.
- **Example**: Monitoring a microservices application's health metrics in real-time.

Grafana:

- **Features**: Data visualization, supports multiple data sources.
- **Use Case**: Creating dashboards for monitoring performance metrics from Prometheus.
- **Example**: A dashboard displaying metrics of response times and error rates for a web service.

CONTINUOUS MONITORING AND LOGGING

PROMETHEUS, GRAFANA, AND ELK STACK

ELK Stack (Elasticsearch, Logstash, Kibana):

- **Features**: Full logging solution with search, storage, and visualization.
- **Use Case**: Collecting and analyzing logs to troubleshoot application issues.
- **Example**: Monitoring and analyzing logs for error patterns to optimize application performance.

SOFTWARE DEPLOYMENT

SOFTWARE DEPLOYMENT STRATEGIES

- **Basic Deployment:** A basic deployment involves updating every node in the target environment simultaneously to introduce a new version
- **Rolling Deployment:** A rolling deployment involves updating a subset of application instances (instead of the full update of a basic deployment)
- **Multi-Service Deployment:** multi-service deployment simultaneously updates every node in the target environment with multiple services
- **Blue/Green Deployment:**A blue/green deployment involves deploying two versions of the application simultaneously
- **Canary Deployment:** A canary deployment involves releasing a service or application in increments
- **A/B Testing:** A/B testing involves routing a small subset of users to the new functionality to collect statistics and help inform business decisions.
- **Shadow Deployment:** A shadow deployment involves releasing two parallel versions of the software, forking the incoming requests to the current version, and sending them to the new version.

PART 4
RELIABILITY, SECURITY, AND MAINTENANCE

UNIVERSITAS **BINA NUSANTARA**

SUBJECT MATTER EXPERT

People
Innovation
Excellence

BINUS
UNIVERSITY

TYPES OF MAINTENANCE

- **Corrective Maintenance**:
  - Involves fixing bugs and errors found after software deployment.
  - **Example**: Correcting a bug that causes the application to crash under specific conditions.
- **Adaptive Maintenance**:
  - Involves updating software to remain compatible with new environments (e.g., OS updates, hardware changes).
  - **Example**: Modifying a mobile app to work with a new version of iOS.
- **Perfective Maintenance**:
  - Enhances or refines existing functionalities based on user feedback or to improve performance.
  - **Example**: Adding a new search filter to an e-commerce application to improve user experience.
- **Preventive Maintenance**:
  - Aims to detect and prevent future problems by addressing potential vulnerabilities and optimizing code.
  - **Example**: Refactoring code to reduce technical debt and improve maintainability.

People
Innovation
Excellence

BINUS
UNIVERSITY

CHALLENGES IN SOFTWARE MAINTENANCE

COMMON CHALLENGES

- **High Complexity**: Legacy code and dependencies make changes difficult.
- **Inadequate Documentation**: Lack of documentation can hinder understanding and increase time spent on changes.
- **Resource Constraints**: Limited time, budget, and skilled personnel often impact maintenance quality.
- **Managing Technical Debt**: Accumulated shortcuts or outdated code require attention but are often deferred.
- **Changing Requirements**: Business or regulatory changes add complexity to ongoing maintenance.

Examples and Discussion:

- **Legacy Systems**: Maintaining older systems with outdated technology is challenging, especially with limited compatibility with modern tools.
- **Technical Debt**: Discuss how technical debt accumulates when quick fixes are prioritized over quality.

People
Innovation
Excellence

BINUS
UNIVERSITY

CHALLENGES IN SOFTWARE MAINTENANCE

COMMON CHALLENGES

- **High Complexity**: Legacy code and dependencies make changes difficult.
- **Inadequate Documentation**: Lack of documentation can hinder understanding and increase time spent on changes.
- **Resource Constraints**: Limited time, budget, and skilled personnel often impact maintenance quality.
- **Managing Technical Debt**: Accumulated shortcuts or outdated code require attention but are often deferred.
- **Changing Requirements**: Business or regulatory changes add complexity to ongoing maintenance.

Examples and Discussion:

- **Legacy Systems**: Maintaining older systems with outdated technology is challenging, especially with limited compatibility with modern tools.
- **Technical Debt**: Discuss how technical debt accumulates when quick fixes are prioritized over quality.

People
Innovation
Excellence

BINUS
UNIVERSITY

CHALLENGES IN SOFTWARE MAINTENANCE

COMMON CHALLENGES

- **High Complexity**: Legacy code and dependencies make changes difficult.
- **Inadequate Documentation**: Lack of documentation can hinder understanding and increase time spent on changes.
- **Resource Constraints**: Limited time, budget, and skilled personnel often impact maintenance quality.
- **Managing Technical Debt**: Accumulated shortcuts or outdated code require attention but are often deferred.
- **Changing Requirements**: Business or regulatory changes add complexity to ongoing maintenance.

Examples and Discussion:

- **Legacy Systems**: Maintaining older systems with outdated technology is challenging, especially with limited compatibility with modern tools.
- **Technical Debt**: Discuss how technical debt accumulates when quick fixes are prioritized over quality.

People
Innovation
Excellence

BINUS
UNIVERSITY

CHALLENGES IN SOFTWARE MAINTENANCE

COMMON CHALLENGES

- **High Complexity**: Legacy code and dependencies make changes difficult.
- **Inadequate Documentation**: Lack of documentation can hinder understanding and increase time spent on changes.
- **Resource Constraints**: Limited time, budget, and skilled personnel often impact maintenance quality.
- **Managing Technical Debt**: Accumulated shortcuts or outdated code require attention but are often deferred.
- **Changing Requirements**: Business or regulatory changes add complexity to ongoing maintenance.

Examples and Discussion:

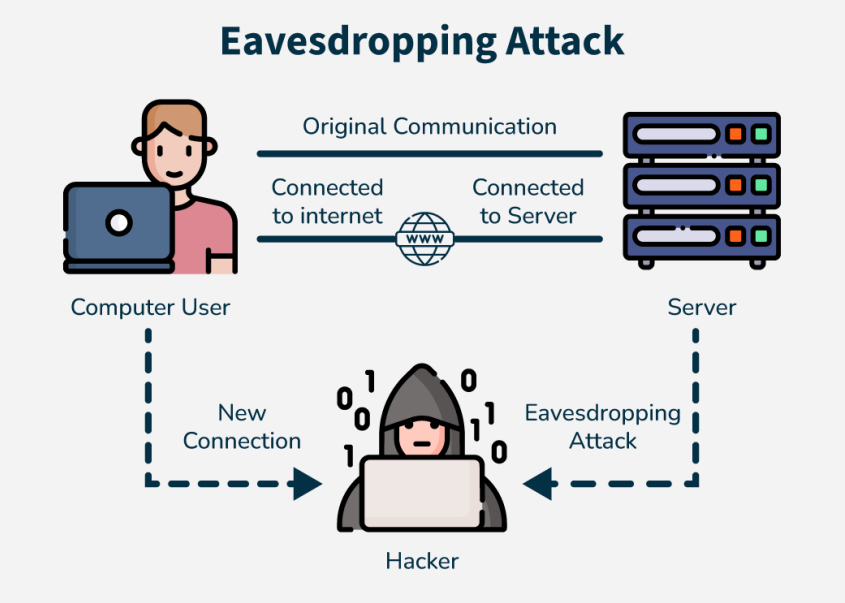
- **Legacy Systems**: Maintaining older systems with outdated technology is challenging, especially with limited compatibility with modern tools.
- **Technical Debt**: Discuss how technical debt accumulates when quick fixes are prioritized over quality.

COMMON SECURITY VULNERABILITIES AND THREATS

ATTACKING THE NETWORK: SNIFFING, EAVESDROPPING, ETC.

- You can listen to the traffic going by on the net

- This is typically traffic on your subnet
  - Still it can be most interesting
  - If you can plug in to the backbone…



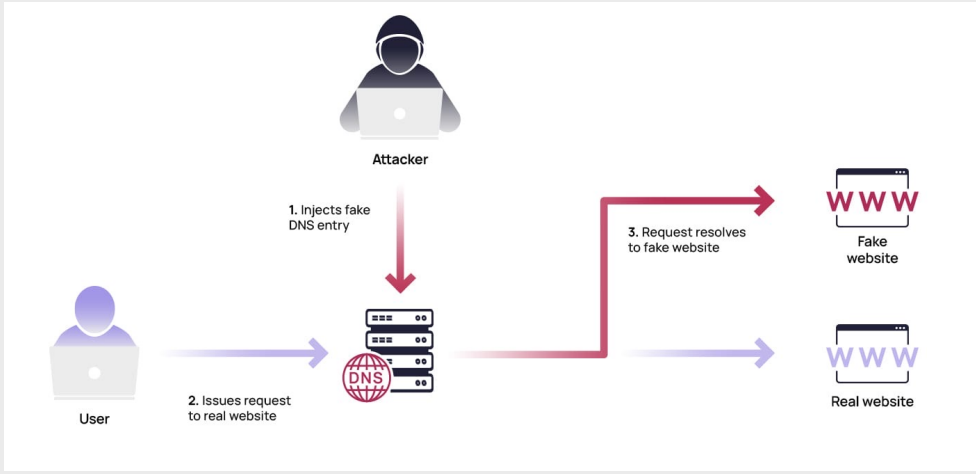
People
Innovation
Excellence

BINUS
UNIVERSITY

COMMON SECURITY VULNERABILITIES AND THREATS

ATTACKING THE NETWORK: SPOOFING

- Pretending to be someone you're not
- IP spoofing
  - Pretending to a "client" you're not (with a specific IP address)
- E-mail Spoofing
- DNS spoofing
  - Pretending to be a server that you're not
  - Fool a DNS server to give out incorrect IP addresses for DNS Names
- Note: also be careful of typos or similar characters attacks:
  - http://mytimes.com
  - http://paypa1.com



People
Innovation
Excellence

BINUS
UNIVERSITY

COMMON SECURITY VULNERABILITIES AND THREATS

OTHER VULNERABILITIES AND THREATS

- **Injection Attacks**: Malicious code is injected, often via user input fields.
- **Broken Authentication**: Weak authentication allows attackers to access unauthorized areas.
- **Cross-Site Scripting (XSS)**: Attackers inject scripts into web pages viewed by others.
- **Insecure Deserialization**: Insecure handling of serialized data that can be exploited.
- **Insufficient Logging and Monitoring**: Lacks detection of and response to breaches.
- **Insider Threats**: Authorized users intentionally or unintentionally cause harm.
- **Social Engineering**: Manipulating users to gain unauthorized access.
- **Advanced Persistent Threats (APTs)**: Continuous hacking processes targeting sensitive data.

SECURITY BEST PRACTICES AND CODING STANDARDS

THE ”BIG THREE” CONCEPTS IN NETWORK SECURITY

- **Authentication**
  - Knowing with whom you are communicating
    - User knowing the server and/or server knowing the user
      - Which is more important??
- **Authorization**
  - User having privilege to perform an operation on server
- **Confidentiality**
  - Communicating without others knowing what’s been said
  - Intermediaries cannot change what was said
  - Typically includes protection from replay attack
    - (Typically does *not* provide secrecy of communication. Others can know communication occurred)

IMPLEMENTING SECURITY TESTING IN THE DEVELOPMENT PROCESS

TYPES OF SECURITY TESTING

- **Static Analysis**: Examines code for security flaws without executing it (e.g., SonarQube).
- **Dynamic Analysis**: Tests running applications to identify vulnerabilities (e.g., Burp Suite).
- **Penetration Testing**: Simulates real-world attacks to identify weak points.
- **Fuzz Testing**: Sends random data to the application to find security loopholes.

TESTING PERFORMANCE

PERFORMANCE TESTING

- Goal: Identify *performance bugs*. What are these?
  - Unexpected bad performance on some subset of inputs
  - Performance degradation over time
  - Difference in performance across versions or platforms
- Not as easy as functional testing. What's the oracle?
  - Fast = good, slow = bad // but what's the threshold?
  - How to get reliable measurements?
  - How to debug where the issue lies?

People
Innovation
Excellence

BINUS
UNIVERSITY

TESTING PERFORMANCE

PERFORMANCE REGRESSION TESTING

- Measure execution time of critical components
- Log execution times and compare over time

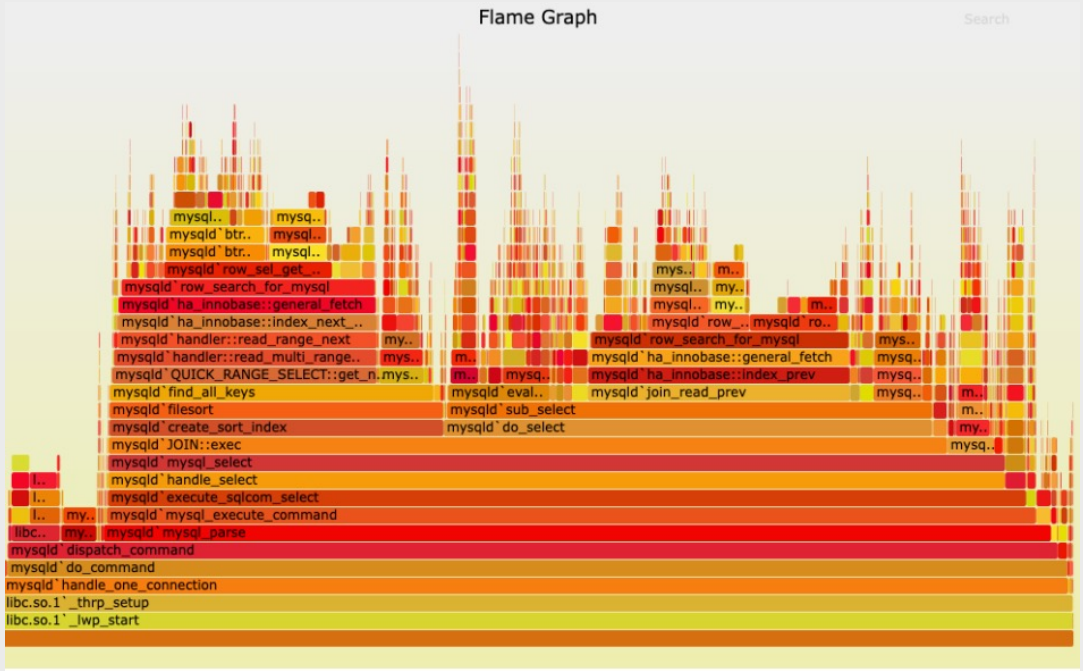
People
Innovation
Excellence

BINUS
UNIVERSITY

TESTING PERFORMANCE

PROFILING

- Finding bottlenecks in execution time and memory
- Flame graphs are a popular visualization of resource consumption by call stack.

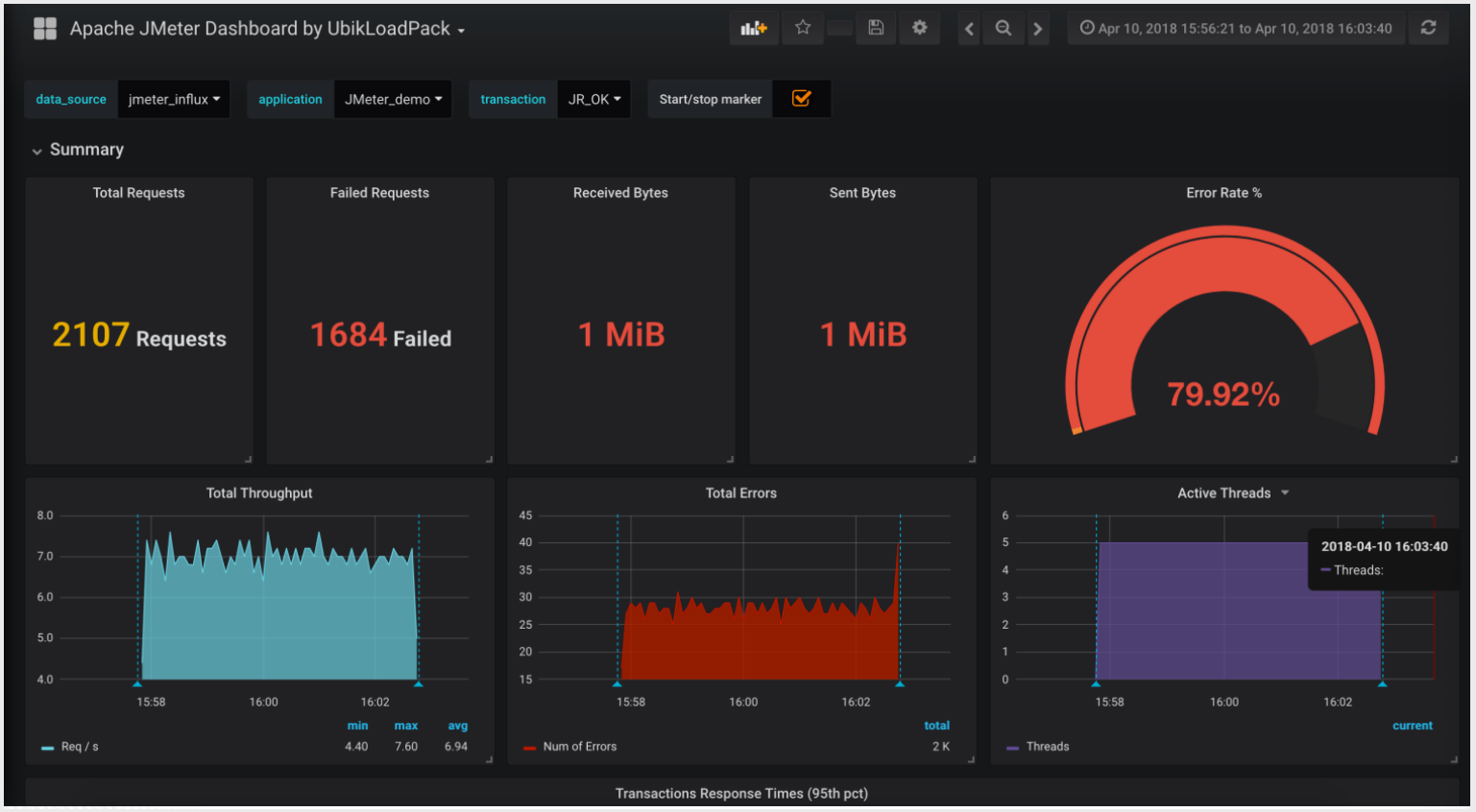


People
Innovation
Excellence

BINUS
UNIVERSITY

TESTING PERFORMANCE

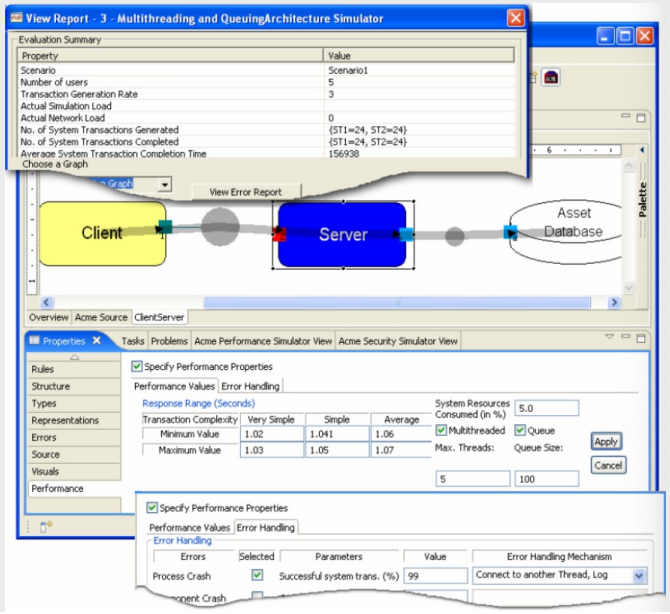
DOMAIN-SPECIFIC PERF TESTING (E.G. JMETER)



TESTING PERFORMANCE

PERFORMANCE-DRIVEN DESIGN

- Modeling and simulation
  - e.g. queuing theory
- Specify load distributions and derive or test configurations



People
Innovation
Excellence

BINUS
UNIVERSITY

TESTING PERFORMANCE

STRESS TESTING

- Robustness testing technique: test beyond the limits of normal operation.
- Can apply at any level of system granularity.
- Stress tests commonly put a greater emphasis on robustness, availability, and error handling under a heavy load, than on what would be considered “correct” behavior under normal circumstances.

TESTING PERFORMANCE

SOAK TESTING

- **Problem**: A system may behave exactly as expected under artificially limited execution conditions.
  - E.g., Memory leaks may take longer to lead to failure (also motivates static/dynamic analysis, but we'll talk about that later).
- **Soak testing**: testing a system with a significant load over a significant period of time (positive).
- Used to check reaction of a subject under test under a possible simulated environment for a given duration and for a given threshold.

PART 5
METRICS, ETHICS, AND IMPROVEMENT STRATEGY

UNIVERSITAS **BINA NUSANTARA**

SUBJECT MATTER EXPERT

People
Innovation
Excellence

BINUS
UNIVERSITY

INTRODUCTION TO SOFTWARE PROCESS IMPROVEMENT (SPI)

OVERVIEW AND IMPORTANCE

- **Software Process Improvement (SPI)** refers to the practice of assessing and enhancing software development processes to improve quality, productivity, and predictability. SPI aims to identify weaknesses in current processes and implement changes that lead to better performance and outcomes.

Importance of SPI:

- **Quality Enhancement**: Improves the quality of software products.
- **Productivity Increase**: Streamlines processes to reduce time and cost.
- **Risk Reduction**: Identifies and mitigates potential issues early.
- **Customer Satisfaction**: Leads to higher satisfaction through reliable delivery.
- **Competitive Advantage**: Positions organizations better in the market.

People
Innovation
Excellence

BINUS
UNIVERSITY

SOFTWARE PROCESS IMPROVEMENT MODELS

CAPABILITY MATURITY MODEL (CMM)

- **Definition**: The Capability Maturity Model (CMM) is a framework that describes the key elements of an effective software process.
- **Purpose**: Provides organizations with guidance on how to gain control of their processes for developing and maintaining software.

People
Innovation
Excellence

BINUS
UNIVERSITY

SOFTWARE PROCESS IMPROVEMENT MODELS

CAPABILITY MATURITY MODEL (CMM): MATURITY LEVELS

- **Initial (Level 1)**: Processes are unpredictable, poorly controlled, and reactive.
- **Repeatable (Level 2)**: Basic project management processes are established to track cost, schedule, and functionality.
- **Defined (Level 3)**: Processes are documented, standardized, and integrated into a standard process for the organization.
- **Managed (Level 4)**: Detailed measures of the software process and product quality are collected and controlled.
- **Optimizing (Level 5)**: Continuous process improvement is enabled by quantitative feedback and from piloting innovative ideas and technologies.

SOFTWARE PROCESS IMPROVEMENT MODELS

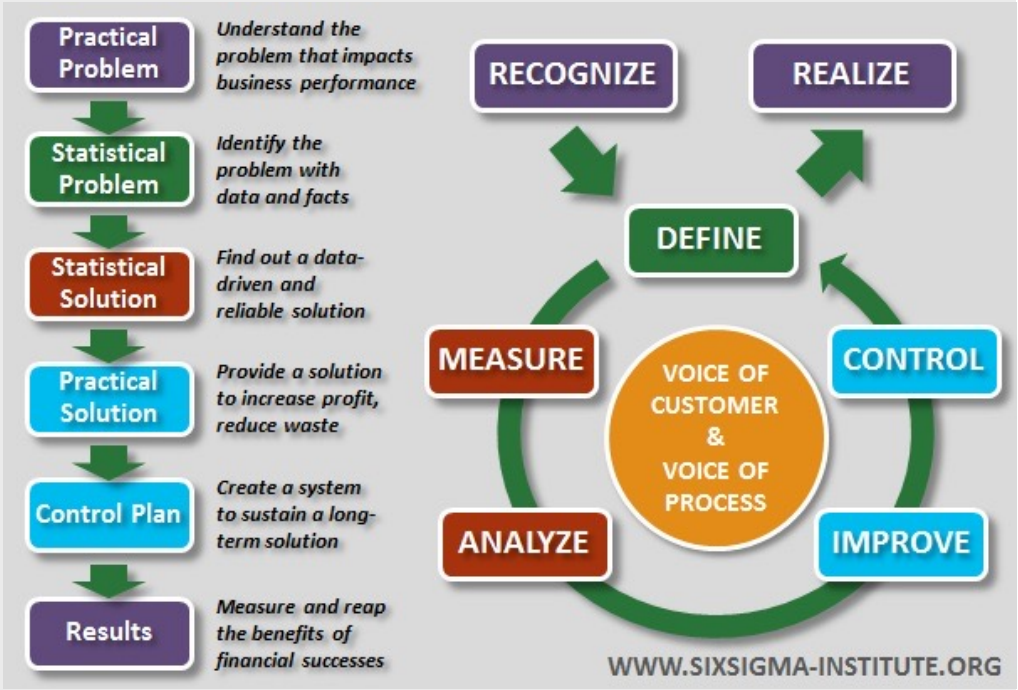
OTHER SPI MODELS: ISO/IEC 15504 (SPICE)

- **Definition**: An international standard for software process assessment.
- **Purpose**: Provides a framework for assessing the maturity of software development processes.
- **Process Capability Levels**: Incomplete, Performed, Managed, Established, Predictable, Optimizing.

SOFTWARE PROCESS IMPROVEMENT MODELS

OTHER SPI MODELS: SIX SIGMA

- **Definition**: A data-driven approach for eliminating defects and improving quality in any process.
- **Purpose**: Aims for near perfection in process performance.
- **Key Concepts**: DMAIC methodology (Define, Measure, Analyze, Improve, Control).



STRATEGIES FOR ENHANCING SOFTWARE PROCESSES

1. PROCESS ASSESSMENT

- **Purpose**:
  - Evaluate current processes to identify strengths and weaknesses.
  - Establish a baseline for improvement.
- **Methods**:
  - **Appraisals and Audits**: Formal evaluations against standards like CMM.
  - **Surveys and Interviews**: Gathering input from team members and stakeholders.

STRATEGIES FOR ENHANCING SOFTWARE PROCESSES

2. PROCESS MODELING AND ANALYSIS

- **Purpose**:
  - Understand and visualize existing processes.
  - Identify inefficiencies and bottlenecks.
- **Techniques**:
  - **Flowcharts**: Visual representations of process steps.
  - **Unified Modeling Language (UML)**: Standardized modeling language for specifying, visualizing, and documenting processes.

STRATEGIES FOR ENHANCING SOFTWARE PROCESSES

3. CHANGE MANAGEMENT

- **Importance**:
  - Effective SPI requires managing the human and organizational aspects of change.
  - Resistance to change can impede process improvement efforts.
- **Strategies**:
  - **Communication**: Clearly explain the need and benefits of changes.
  - **Training**: Equip team members with necessary skills and knowledge.
  - **Involvement**: Engage stakeholders in the change process.

STRATEGIES FOR ENHANCING SOFTWARE PROCESSES

4. CONTINUOUS IMPROVEMENT PRACTICES

- **Kaizen**:
  - **Definition**: A Japanese term meaning "change for better" or continuous improvement.
  - **Application**: Implement small, incremental changes regularly.
- **Plan-Do-Check-Act (PDCA) Cycle**:
  - **Plan**: Identify an opportunity and plan for change.
  - **Do**: Implement the change on a small scale.
  - **Check**: Analyze the results of the change.
  - **Act**: If successful, implement it on a larger scale.

OVERVIEW OF LEGAL ASPECTS IN SOFTWARE ENGINEERING

LEGAL CONSIDERATIONS IN SOFTWARE DEVELOPMENT

Legal Considerations:

- Software engineers must consider legal constraints throughout the development process to avoid violations and protect both users and the company.
- Legal aspects often overlap with issues of intellectual property, data security, and compliance.

Relevance in Software Development:

- Compliance with laws prevents potential lawsuits, financial loss, and reputational damage.

People
Innovation
Excellence

BINUS
UNIVERSITY

INTELLECTUAL PROPERTY RIGHTS

COPYRIGHTS, PATENTS, TRADEMARKS

- **Copyrights**: Protects the software code and documentation as original works.
  - **Example**: Source code is protected, making unauthorized copying or distribution illegal.
- **Patents**: Protects new, non-obvious inventions and unique processes in software.
  - **Example**: Google's patent on their page-ranking algorithm.
- **Trademarks**: Protects names, symbols, or logos associated with software or services.
  - **Example**: Microsoft's trademark of the Windows logo.
- **Importance**:
  - Intellectual property rights encourage innovation by allowing creators to benefit from their inventions while protecting against misuse.

LICENSING MODELS FOR SOFTWARE

SOFTWARE LICENSES

- **Types of Software Licenses**:
  - **Proprietary Licenses**: Restrict users from modifying, sharing, or redistributing the software.
  - **Open Source Licenses**: Allow users to access, modify, and share the source code (e.g., MIT, GPL).
  - **Freeware and Shareware**: Software distributed for free or for a limited trial period before payment.
- **Licensing Importance**:
  - Defines how software can be used, distributed, and modified, impacting how developers can engage with the software.

People
Innovation
Excellence

BINUS
UNIVERSITY

PRIVACY LAWS AND DATA PROTECTION

GDPR AND CCPA

- **General Data Protection Regulation (GDPR)**:
  - European regulation focusing on user data privacy and security, including the right to be forgotten and consent requirements.
- **California Consumer Privacy Act (CCPA)**:
  - Regulates data collection and sharing practices, especially for companies handling personal data of California residents.
- **Key Principles of Privacy Laws**:
  - **Data Minimization**: Only necessary data should be collected.
  - **Purpose Limitation**: Data collected should only be used for specified purposes.
  - **Data Security**: Ensuring data protection from breaches and misuse.

UNDERSTANDING PROFESSIONAL ETHICS

WHAT IS HUMAN FLOURISHING?

According to Harvard's Human flourishing program:

Human flourishing is composed of five central domains: **happiness and life satisfaction**, **mental and physical health**, **meaning and purpose**, **character and virtue**, and **close social relationships**.

ETHICAL RESPONSIBILITIES OF SOFTWARE PROFESSIONALS

RESPONSIBILITIES TO USERS AND SOCIETY

Responsibilities to Users:

- Ensure software safety, security, and accessibility.

Responsibilities to Society:

- Consider the broader societal impact, such as environmental effects and social welfare.