Quality Attributes

Parinya Ekparinya

Parinya.Ek@kmitl.ac.th

Let’s have some fun exercises!!

```
int a[1817];int main(z,p,q,r){for(p=80;q+p-80;p-
=2*a[p])for(z=9;z--;)q=3&(r=time(0)+r*57)/7,q=q?q-1?q-
2?1-p%79?-1:0:p%79-77?1:0:p<1659?79:0:p>158?-
79:0,q?!a[p+q*2]?a[p+=a[p+=q]=q]=q:0:0;for(;q++-
1817;)printf(q%79?"%c":"%c\n","#"[!a[q-1]]);}
```

Exercise 1: Please argue that this is a bad program!!

```
int a[1817];int main(z,p,q,r){for(p=80;q+p-80;p-
=2*a[p])for(z=9;z--;)q=3&(r=time(0)+r*57)/7,q=q?q-1?q-
2?1-p%79?-1:0:p%79-77?1:0:p<1659?79:0:p>158?-
79:0,q?!a[p+q*2]?a[p+=a[p+=q]=q]=q:0:0;for(;q++-
1817;)printf(q%79?"%c":"%c\n","#"[!a[q-1]]);}
```

Exercise 2: Please argue that this is a good program!!

รันได้

Let’s run the code!!

No matter the source, all requirements encompass the following categories:

1. Functional requirements ระบบทำได้
2. Quality attribute requirements เหมาะสม
3. Constraints

Functionality

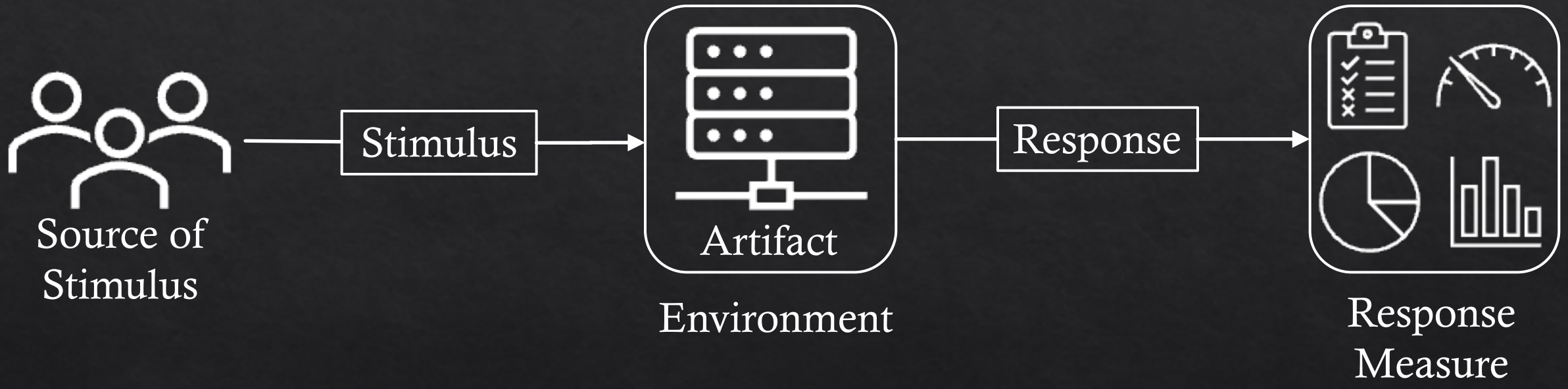
- Functionality is the ability of the system to do the work for which it was intended.
- Functionality does not determine architecture.
- Functionality is achieved by assigning responsibilities to architectural elements, resulting in one of the most basic of architectural structures.

Functionality is objective.
But good/bad is subjective!!

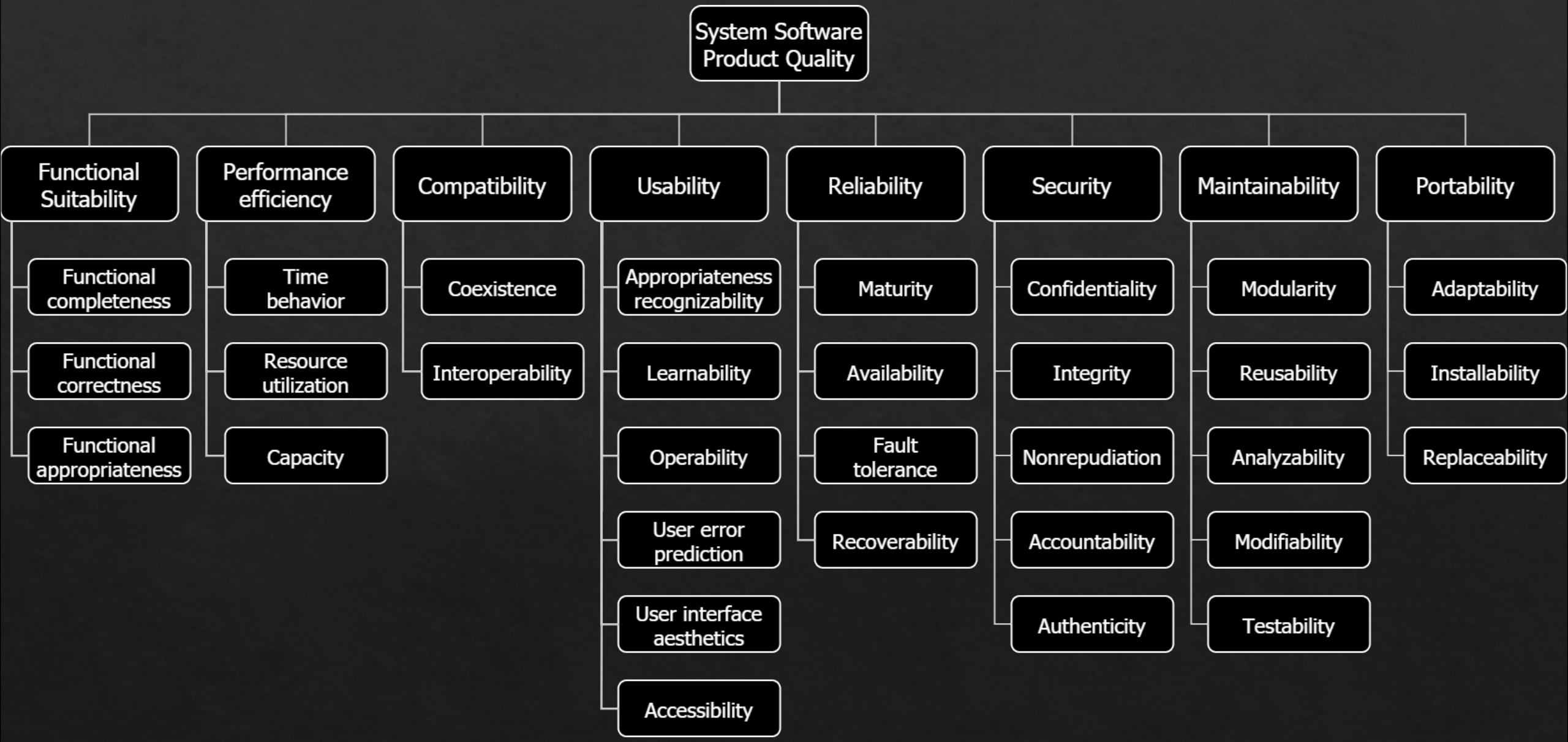
Quality Attributes

- A quality attribute is a measurable or testable property of a system that is used to indicate how well the system satisfies the needs of its stakeholders.
- While functionality describes what the system does, quality describes how well the system does its function.
- Quality attribute scenarios is used to characterizing quality attributes.

Quality Attribute Scenario



ISO/IEC FCD 25010 Product Quality Standard



Availability

Availability

- Availability refers to a property of software—namely, that it is there and ready to carry out its task when you need it to be.
- A failure is the deviation of the system from its specification, where the deviation is externally visible.
- A failure's cause is called a fault.
- A fault can be either internal or external to the system under consideration.
- Faults can be prevented, tolerated, removed, or forecast. Through these actions, a system becomes "resilient" to faults.
- Availability refers to the ability of a system to mask or repair faults such that the cumulative service outage period does not exceed a required value over a specified time interval.

Availability General Scenario (1)

Source of stimulus	Internal/external: people, hardware, software, physical infrastructure, physical environment
Stimulus	Fault: omission, crash, incorrect timing, incorrect response
Artifacts	Processors, communication channels, persistent storage, processes
Environment	Normal operation, startup, shutdown, repair mode, degraded operation, overloaded operation

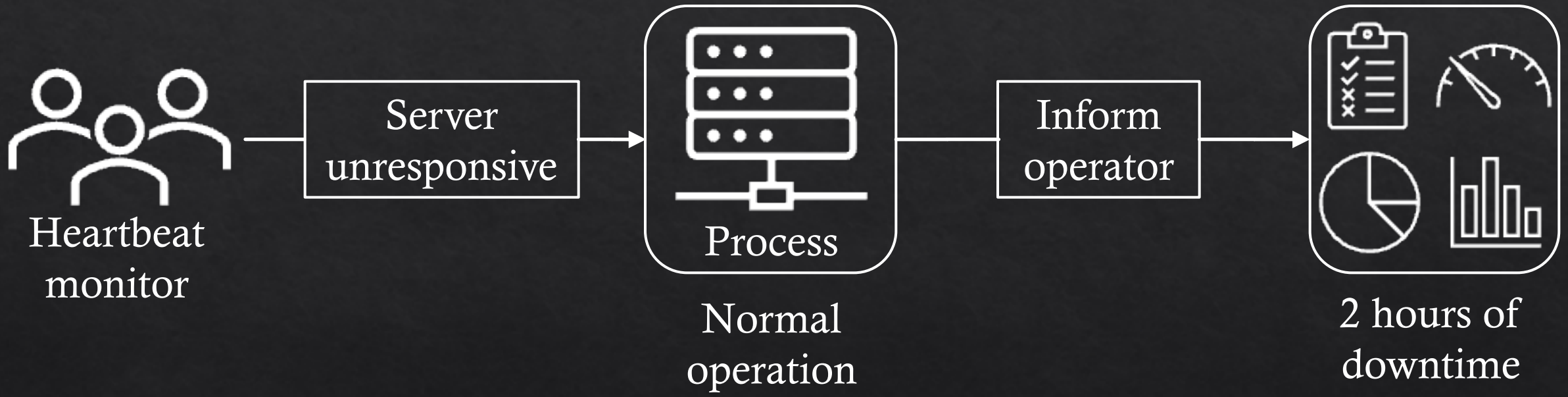
Availability General Scenario (2)

Response การตอบสนอง	Prevent the fault from becoming a failure ป้องกัน ความผิดพลาด ไม่ให้กลายเป็น ความล้มเหลว การตรวจจับ ความผิดพลาด Detect the fault: • Log the fault บันทึกข้อผิดพลาด • Notify appropriate entities (people or systems) แจ้งหน่วยงานที่เหมาะสม Recover from the fault: กู้คืน ความผิดพลาด • Disable source of events causing the fault ปิดการใช้งาน แหล่งที่มาของสาเหตุที่ทำให้ fault • Be temporarily unavailable while repair is being affected ปิดการใช้งานชั่วคราวในขณะที่ • Fix or mask the fault/failure or contain the damage it causes แก้ไข บดบัง ความผิดพลาด หรือ ความเสียหาย ที่เป็นสาเหตุ ความเสียหาย ซ่อมแซม การที่มีงานจะ เสื่อมโทรม ในขณะที่การซ่อมแซม ก็ไม่เข้ามาสัมผัส • Operate in a degraded mode while repair is being affected

Availability General Scenario (3)

Response measure	• Time or time interval when the system must be available • Availability percentage (e.g., 99.999%) • Time to detect the fault • Time to repair the fault • Time or time interval in which system can be in degraded mode • Proportion (e.g., 99%) or rate (e.g., up to 100 per second) of a certain class of faults that the system prevents, or handles without failing

Sample Availability Scenario



Availability Calculation

- Typically, the availability of a system can be measured as the proportion of time it has provided the specified services within required bounds over a specified time interval.
- There is a well-known expression used to derive steady-state availability:

$$\frac{\text{เวลาที่ทำงานได้จริง}}{\text{เวลาที่ควรทำได้ทั้งหมด}} \left\{ \frac{MTBF}{(MTBF + MTTR)} \right.$$

เวลาเฉลี่ย (เวลาที่ทำงานได้)

↳ เวลาที่ทำงานได้ ↳ เวลาเฉลี่ยในการซ่อม

- MTBF refers to the mean time between failures
- MTTR refers to the mean time to repair

System Availability Requirements

Availability	Downtime/90 days	Downtime/365 days
90%	9 days	36 days, 12 hours
95%	4 days, 12 hours	18 days, 6 hours
99%	21 hours, 36 minutes	3 days, 15 hours, 36 minutes
99.9%	2 hours, 9 minutes, 36 seconds	8 hours, 45 minutes, 36 seconds
99.99%	12 minutes, 58 seconds	52 minutes, 34 seconds
99.999%	1 minutes, 18 seconds	5 minutes, 15 seconds
99.9999%	~8 seconds	~32 seconds

Tactics for Availability

Detect faults	Recover from faults		Prevent faults
	Preparation and Repair	Reintroduction	
• Monitor • Ping/Echo • Heartbeat • Timestamp • Sanity Checking • Condition Monitoring • Voting • Exception Detection • Self-Test	• Active redundancy (hot spare) • Passive redundancy (warm spare) • Spare (cold spare) • Rollback • Exception handling • Software upgrade • Retry • Ignore fault behavior • Graceful degradation • Reconfiguration	• Shadow • State re-synchronization • Escalating restart • Non-stop forwarding (NSF)	• Removal from service • Transactions • Predictive model • Exception prevention • Increase competence set

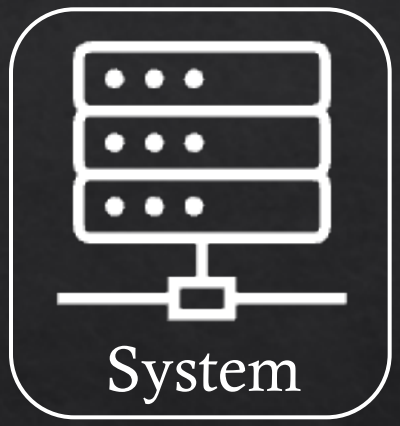
ตรวจจับความผิดพลาด	กู้คืนจากความผิดพลาด		ป้องกันความผิดพลาด
	การเตรียมการและการซ่อมแซม	การแนะนำตัวอีกครั้ง	
• เฝ้าสังเกต • ปิง/เอคโค่ • การเต้นของหัวใจ • การประทับเวลา • ตรวจสภาพ • ร่างกาย การตรวจสอบ • โหวต • การตรวจจับข้อยกเว้น • แบบทดสอบตัวเอง	• Active redundancy (อะไหล่สำรอง) • ความซ้ำซ้อนแบบพาสซีฟ (อะไหล่สำรอง) • อะไหล่ (อะไหล่เย็น) • ย้อนกลับ • การจัดการข้อยกเว้น • การอัพเกรดซอฟต์แวร์ • ลองอีกครั้ง • ละเว้นพฤติกรรมความผิดพลาด • ย่อยสลายอย่างสง่างาม • การกำหนดค่าใหม่	• เงา • การชิงโครไนซ์สถานะใหม่ • การเริ่มต้นใหม่ที่เพิ่มขึ้น • การส่งต่อแบบไม่หยดนิ่ง (NSF)	• การถอดออกจาก บริการ • รปแบบการ • ทำนายธรรมรม • การป้องกัน ข้อยกเว้น • เพิ่ม ชดความสามารถ


Detect Faults

Monitor

เทสว่ามีฟังก์ชันปกติ

- This component is used to monitor the state of health of various other parts of the system: processors, processes, I/O, memory, and so forth.
- It orchestrates software using other tactics in this category to detect malfunctioning components.
- For example, the system monitor can initiate self-tests, or be the component that detects faulty timestamps or missed heartbeats.

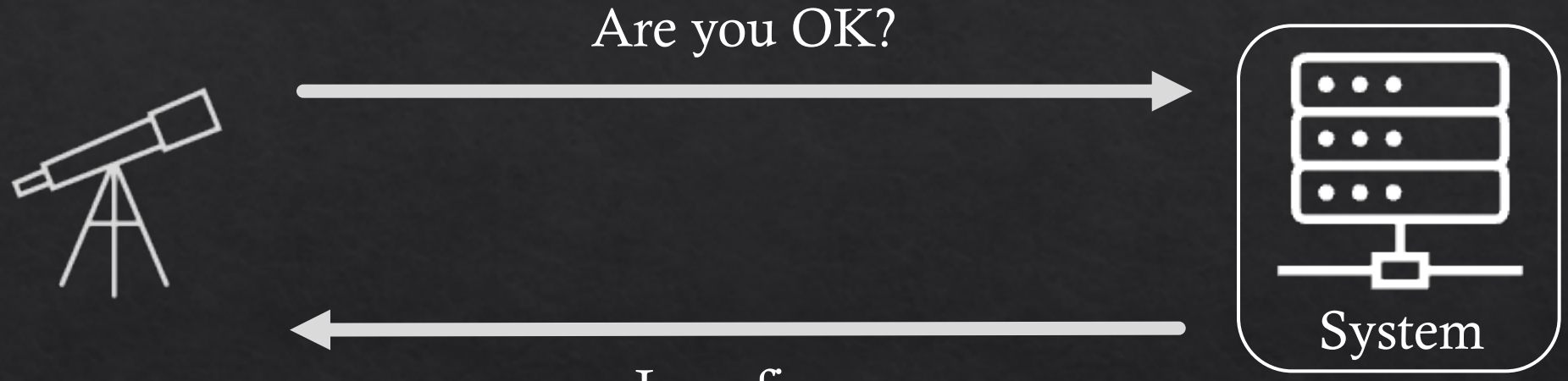

System

Ping/Echo

monitor ถาม

มีการเช็คโดยการถาม → ตอบกลับมาปกติมั้ย

- In this tactic, an asynchronous request/response message pair is exchanged between nodes.
- It is used to determine reachability and the round-trip delay through the associated network path.
- The echo also indicates that the pinged component is alive.
- The ping is often sent by a system monitor. ถามบ่อย มีใส่ใส่ใส่
- Ping/echo requires a time threshold to be set; this threshold tells the pinging component how long to wait for the echo before considering the pinged component to have failed ("timed out").
- Standard implementations of ping/echo are available for nodes interconnected via Internet Protocol (IP).

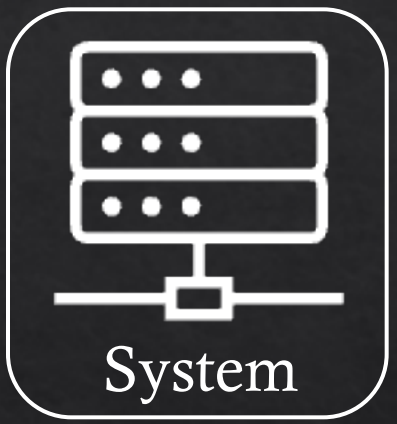

Are you OK?
System
I am fine.

Heartbeat

- This fault detection mechanism employs a periodic message exchange between a system monitor and a process being monitored.
- A special case of heartbeat is when the process being monitored periodically resets the watchdog timer in its monitor to prevent it from expiring and thus signaling a fault.
- For systems where scalability is a concern, transport and processing overhead can be reduced by piggybacking heartbeat messages onto other control messages being exchanged.
- The difference between heartbeat and ping/echo lies in who holds the responsibility for initiating the health check the monitor or the component itself.

* Detect ผิดปกติ

I am fine.



รู้ว่าเหตุการณ์เกิดตอนไหน

ตายยิ่ง

Timestamp

- This tactic is used to detect incorrect sequences of events, primarily in distributed message-passing systems.
- A timestamp of an event can be established by assigning the state of a local clock to the event immediately after the event occurs.
- Sequence numbers can also be used for this purpose, since timestamps in a distributed system may be inconsistent across different processors.

นอกเหนือลำดับเหตุการณ์

Condition Monitoring

- This tactic involves checking conditions in a process or device, or validating assumptions made during the design.
- By monitoring conditions, this tactic prevents a system from producing faulty behavior. The computation of checksums is a common example of this tactic.
- The monitor must itself be simple (and, ideally, provably correct) to ensure that it does not introduce new software errors.

Sanity Checking

- This tactic checks the validity or reasonableness of specific operations or outputs of a component.
- It is typically based on a knowledge of the internal design, the state of the system, or the nature of the information under scrutiny.
- It is most often employed at interfaces, to examine a specific information flow.

สีหมาย monitor เอาไว้เช็คความผิดพลาดเอง

Voting

- Voting involves comparing computational results from multiple sources that should be producing the same results and, if they are not, deciding which results to use.
- This tactic depends critically on the voting logic, which is usually realized as a simple, rigorously reviewed and tested singleton so that the probability of error is low.
- Voting also depends critically on having multiple sources to evaluate.
- Typical multi-source schemes include:
  - Replication เหมือนกันหมด
  - Functional redundancy
  - Analytic redundancy

Multiple Sources

In / out ไม่ต้องเหมือน

Voting: Replication

- Replication is the simplest form of voting; here, the components are exact clones of each other.
- Having multiple copies of identical components can be effective in protecting against random failures of hardware.
- But it cannot protect against design or implementation errors, in hardware or software, since there is no form of diversity embedded in this tactic.

Voting: Replication

ใม่เหมือนเพื่อน
↑ = อาจจะผิดปกติ

4.2 kg

4.2 kg

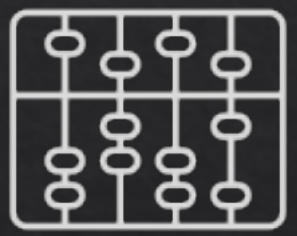
3.7 kg

Voting: Functional Redundancy

- Functional redundancy is intended to address the issue of common-mode failures (where replicas exhibit the same fault at the same time because they share the same implementation) in hardware or software components, by implementing design diversity.
- This tactic attempts to deal with the systematic nature of design faults by adding diversity to redundancy.
- The outputs of functionally redundant components should be the same given the same input.
- The functional redundancy tactic is still vulnerable to specification errors—and, of course, functional replicas will be more expensive to develop and verify.

Voting: Functional Redundancy

ต่างวิธีการ (implement function เดียวกัน แต่ต่างวิธี)



42

42

42

มีความหลากหลาย in /out

Voting: Analytic Redundancy

- Analytic redundancy permits not only diversity among components' private sides, but also diversity among the components' inputs and outputs.
- This tactic is intended to tolerate specification errors by using separate requirement specifications.

แต่ละตัวมี spec ของตัวเอง

- For example, avionics programs have multiple ways to compute aircraft altitude, such as using barometric pressure, with the radar altimeter, and geometrically using the straight-line distance and lookdown angle of a point ahead on the ground.

หา แอมพลิจูด

- The voter mechanism used with analytic redundancy needs to be more sophisticated than just letting majority rule or computing a simple average. It may be asked to produce a higher-fidelity value than any individual component can, by blending and smoothing individual values over time.

เลือกเชื่อ อันใด อันหนึ่ง

Voting: Analytic Redundancy

- Multiple ways to compute aircraft altitude

Barometric Pressure

Radar Altimeter

Geometric Calculation

Set on 9th Code

Exception Detection

- This tactic focuses on the detection of a system condition that alters the normal flow of execution. It can be further refined:
  - System exceptions
  - parameter fence
  - Parameter typing
  - Timeout

Self-test

- Components (or, more likely, whole subsystems) can run procedures to test themselves for correct operation.
- Self-test procedures can be initiated by the component itself or invoked from time to time by a system monitor.
- These may involve employing some of the techniques found in condition monitoring such as checksums.

Recover from Faults (Preparation & Repair)

Redundant Spare

- Active Redundancy (hot spare): all nodes in a *protection group* receive and process identical inputs in parallel, allowing redundant spare(s) to maintain synchronous state with the active node(s).
  - A protection group is a group of nodes where one or more nodes are "active," with the remainder serving as redundant spares.
- Passive Redundancy (warm spare): only the active members of the protection group process input traffic; one of their duties is to provide the redundant spare(s) with periodic state updates.
- Spare (cold spare): redundant spares of a protection group remain out of service until a fail-over occurs, at which point a power-on-reset procedure is initiated on the redundant spare prior to its being placed in service.

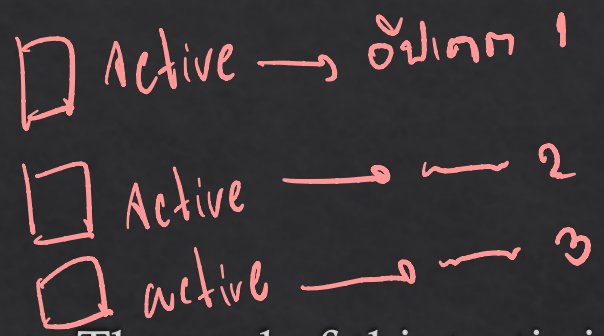
Rollback ไปสถานะ (State) ก่อนหน้านั้น

- A rollback reverts to a previous known good state, referred to as the "rollback line".
- Once the good state is reached, then execution can continue. This tactic is often combined with the transactions tactic and the redundant spare tactic so that after a rollback has occurred, a standby version

ระบุวิธีการ Handling

Exception Handling

- Once an exception has been detected, the system will handle it in some fashion.
- The easiest thing it can do is simply to crash but that's a terrible idea from the point of availability.
- The mechanism employed for exception handling depends largely on the programming environment employed, ranging from simple function return codes (error codes) to the use of exception classes that contain information helpful in fault correlation.



Software Upgrade

service อัปเกรด

- The goal of this tactic is to achieve in-service upgrades to executable code images in a non-service-affecting manner. Strategies include the following:
  - Function patch is used in procedural programming. The new version of the software function will employ the entry and exit points of the deprecated function.
  - Class patch is applicable for targets executing object-oriented code, where the class definitions include a backdoor mechanism that enables the runtime addition of member data and functions.
  - Hitless in-service software upgrade (ISSU) leverages the redundant spare tactic to achieve non-service-affecting upgrades to software and associated schema.

ทั้งคน/ ทั้งระบบ

Retry

ทำอีกรอบ

- The retry tactic assumes that the fault that caused a failure is transient, and that retrying the operation may lead to success.
- It is used in networks and in server farms where failures are expected and common.
- A limit should be placed on the number of retries that are attempted before a permanent failure is declared.

Ignore Faulty Behavior

- This tactic calls for ignoring messages sent from a particular source when we determine that those messages are spurious.
- For example, we would like to ignore the messages emanating from the live failure of a sensor.

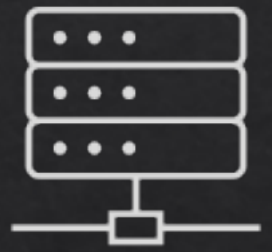
Graceful Degradation

- This tactic maintains the most critical system functions in the presence of component failures, while dropping less critical functions.
- This is done in circumstances where individual component failures gracefully reduce system functionality rather than causing a complete system failure.

Reconfiguration

เครื่องล่มไหว = ย้ายได้

- Reconfiguration attempts to recover from failures by reassigning responsibilities to the (potentially restricted) resources or components left functioning, while maintaining as much functionality as possible.

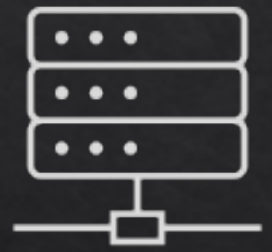


Function 1

&

Function 2

Function 2



Function 3

Recover from Faults (Reintroduction)

Shadow

คงก่อน ลองใช้ก่อน ค่อยปล่อยจริง

- This tactic refers to operating a previously failed or in-service upgraded component in a "shadow mode" (also known as "dark launch" by Google) for a predefined duration of time prior to reverting the component back to an active role.
- During this duration, its behavior can be monitored for correctness and it can repopulate its state incrementally.

State Resynchronization

- คือ checkpoint

- This reintroduction tactic is a partner to the redundant spare tactic.
- When used with active redundancy, the state resynchronization occurs organically, since the active and standby components each receive and process identical inputs in parallel.
- When used alongside the passive redundancy version of the redundant spare tactic, state resynchronization is based solely on periodic state information transmitted from the active component(s) to the standby component(s), typically via checkpointing.

Escalating Restart

- This reintroduction tactic allows the system to recover from faults by varying the granularity of the component(s) restarted and minimizing the level of service affectation.
- For example, consider a system that supports four levels of restart, numbered 0–3.
  - Level 0 kills and recreates all child threads of the faulty component.
  - Level 1 frees and reinitializes all unprotected memory; protected memory is untouched.
  - Level 2 frees and reinitializes all memory, both protected and unprotected.
  - Level 3 involves completely reloading and reinitializing the executable image and associated data segments.
- Support for the escalating restart tactic is particularly useful for the concept of graceful degradation, where a system is able to degrade the services it provides while maintaining support for mission-critical or safety-critical applications.

- This concept originated in router design, and assumes that functionality is split into two parts: the supervisory or control plane (which manages connectivity and routing information) and the data plane (which does the actual work of routing packets from sender to receiver).
- If a router experiences failure of an active supervisor, it can continue forwarding packets along known routes—with neighboring routers—while the routing protocol information is recovered and validated.
- When the control plane is restarted, it implements a "graceful restart," incrementally rebuilding its routing protocol database even as the data plane continues to operate.

Prevent Faults

Removal From Service

- This tactic refers to temporarily placing a system component in an out-of-service state for the purpose of mitigating potential system failures.
- For example, a component of a system might be taken out of service and reset to scrub latent faults (such as memory leaks, fragmentation, or soft errors in an unprotected cache) before the accumulation of faults reaches the service-affecting level, resulting in system failure.
- Other terms for this tactic are *software rejuvenation* and *therapeutic reboot*.
- If you reboot your computer every night, you are practicing removal from service.

Transactions

- Systems targeting high-availability services leverage transactional semantics to ensure that asynchronous messages exchanged between distributed components are atomic, consistent, isolated, and durable properties collectively referred to as the "ACID properties."
- The most common realization of the transactions tactic is the "two-phase commit" (2PC) protocol.
- This tactic prevents race conditions caused by two processes attempting to update the same data item at the same time.

Predictive Model

- A predictive model, when combined with a monitor, is employed to monitor the state of health of a system process to ensure that the system is operating within its nominal operating parameters, and to take corrective action when the system nears a critical threshold.
- The operational performance metrics monitored are used to predict the onset of faults; examples include:
  - the session establishment rate (in an HTTP server)
  - threshold crossing (monitoring high and low watermarks for some constrained, shared resource)
  - statistics on the process state (e.g., in-service, out-of-service, under maintenance, idle)
  - message queue length statistics

Exception Prevention

- This tactic refers to techniques employed for the purpose of preventing system exceptions from occurring.
- The use of exception classes, which allows a system to transparently recover from system exceptions, was discussed earlier.
- Other examples of exception prevention include:
  - error-correcting code (used in telecommunications)
  - abstract data types such as smart pointers
  - the use of wrappers to prevent faults such as dangling pointers or semaphore access violations

ศักยภาพ Case ที่เกิด ผิดพลาดได้มากขึ้น faults

Increase Competence Set

- Increasing a component's competence set means designing a component to handle more cases—faults—as part of its normal operation.
- For example, a component that assumes it has access to a shared resource might throw an exception if it discovers that access is blocked.
- Another component might simply wait for access or return immediately with an indication that it will complete its operation on its own the next time it does have access.
- In this example, the second component has a larger competence set than the first.

Integrability

Integrability เพิ่ม Component

- For practical software systems, software architects need to be concerned with the *costs* and *technical risks* of anticipated and (to varying degrees) unanticipated future integration tasks. These risks may be related to schedule, performance, or technology.
- A general, abstract representation of the integration problem is that a project needs to integrate a unit of software C, or a set of units C_1, C_2, … C_n, into a system S.
- S might be a platform, into which we integrate $\{C_i\}$, or it might be an existing system that already contains $\{C_1, C_2, \ldots, C_n\}$ and our task is to design for, and analyze the costs and technical risks of, integrating $\{C_{n+1}, \ldots C_m\}$.
- We assume we have control over S, but the $\{C_i\}$ may be outside our control.

Integrability General Scenario (1)

Source of stimulus	One or more of the following: • Mission/system stakeholder • Component marketplace • Component vendor
Stimulus	One of the following: • Add new component • Integrate new version of existing component • Integrate existing components together in a new way
Artifacts	One of the following: • Entire system • Specific set of components • Component metadata • Component configuration

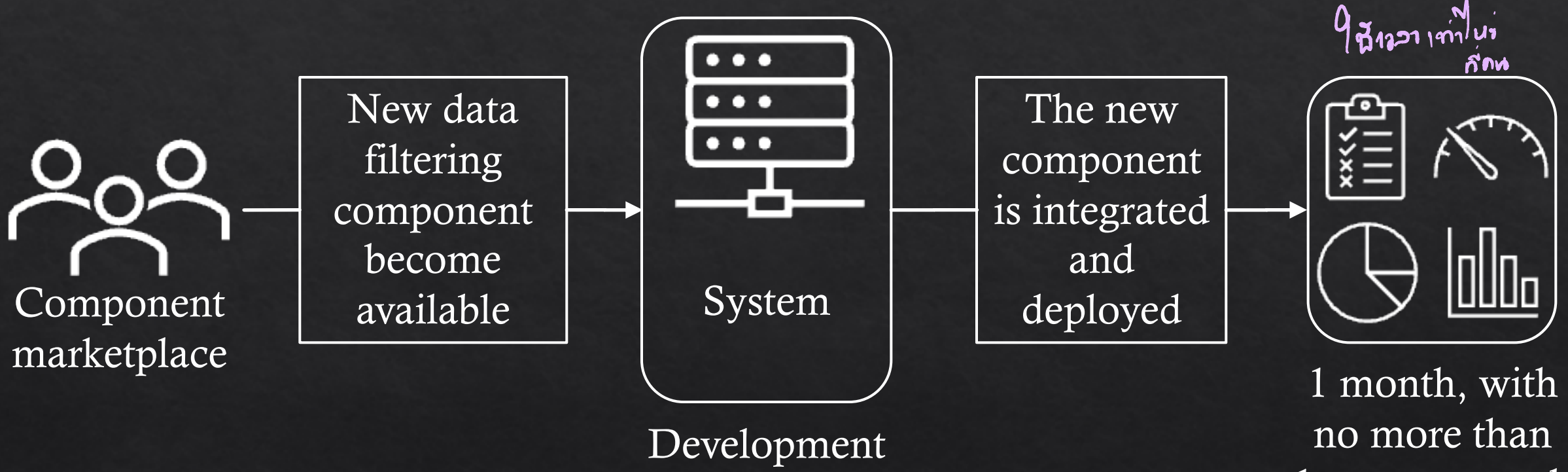
Integrability General Scenario (2)

Environment	One of the following: • Development • Integration • Deployment • Runtime
Response	One or more of the following: • Changes are {completed, integrated, tested, deployed} • Components in the new configuration are successfully and correctly (syntactically and semantically) exchanging information • Components in the new configuration are successfully collaborating • Components in the new configuration do not violate any resource limits

Integrability General Scenario (3)

Response measure	One or more of the following:
	
	• Cost, in terms of one or more of:
	  • Number of components changed
	  • Percentage of code changed % Code ที่ต้องแก้
	  • Lines of code changed
	  • Effort
	  • Money
	  • Calendar time
	
	• Effects on other quality attribute response measures (to capture allowable tradeoffs)

Sample Integrability Scenario



Tactics for Integrability

Limit Dependencies	Adapt	Coordinate
• Encapsulate • Use an intermediary • Restrict Communication Paths • Adhere to Standards • Abstract Common Services	• Discover • Tailor Interface • Configure Behavior	• Orchestrate • Manage Resources

Limit Dependencies

- Encapsulate: introducing an explicit interface to an element and ensures that all access to the element passes through this interface.
- Use an Intermediary: breaking dependencies between a set of components C_i and the system S.
- Restrict Communication Paths: restricting an element's visibility (when developers cannot see an interface, they cannot employ it) and by authorization (i.e., restricting access to only authorized elements).
- Adhere to Standards: restricting communication with a system S to require use of the standard often reduces the number of potential dependencies.
- Abstract Common Services: Where two elements provide services that are similar but not quite the same, it may be useful to hide both specific elements behind a common abstraction for a more general service.

Adapt

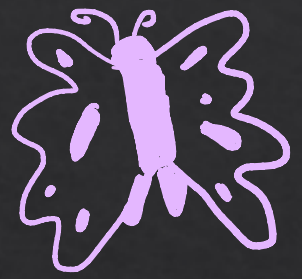
มีlab

- Discover: A discovery service is a catalog of relevant addresses. Entries in a discovery service are there because they were registered. This registration can happen statically, or it can happen dynamically when a service is instantiated. It is the mechanism by which applications and services locate each other.
- Tailor Interface: Tailoring an interface is a tactic that adds capabilities to, or hides capabilities in, an existing interface without changing the API or implementation. Capabilities such as translation, buffering, and data smoothing can be added to an interface without changing it.

Interface เดียวทั่วกัน ให้ทำหน้าที่เพิ่มได้ภายหลัง

- Configure Behavior: Behavior of a component can be configured during the build phase (recompile with a different flag), during system initialization (read a configuration file or fetch data from a database), or during runtime (specify a protocol version as part of your requests).

Coordinate



- Orchestrate: reducing the number of dependencies between a system S and new components $\{C_i\}$, and eliminating altogether the explicit dependencies among the components $\{C_i\}$, by centralizing those dependencies at the orchestration mechanism.
- Manage Resources: software components are not allowed to directly access some computing resources (e.g., threads or blocks of memory), but instead request those resources from a resource manager.
  Resource managers are typically responsible for allocating resource access across multiple components in a way that preserves some invariants (e.g., avoiding resource exhaustion or concurrent use), enforces some fair access policy, or both.