Tactics to Achieve Software Qualities

March 2014

Ying SHEN

SSE, Tongji University

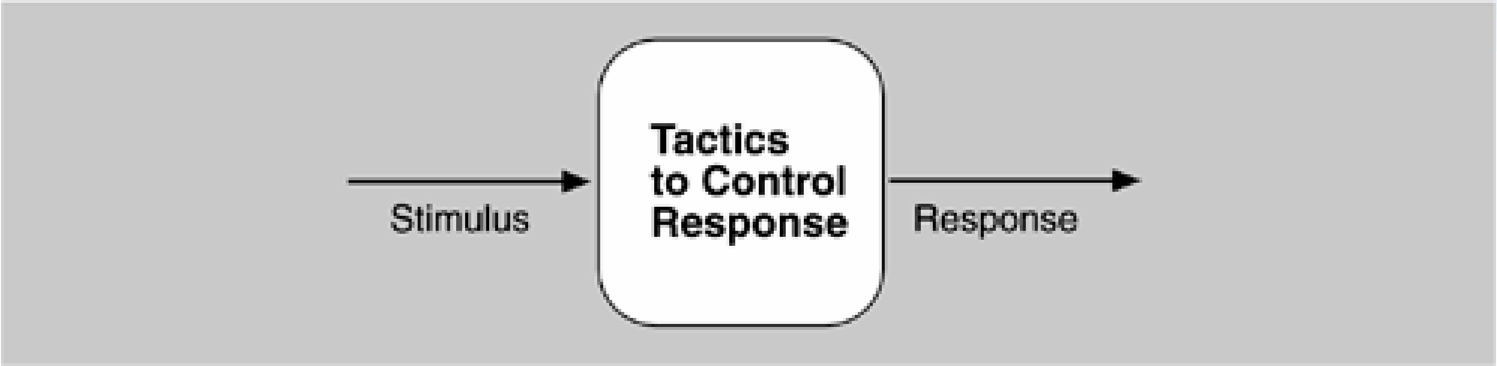
Lecture objectives

This lecture will enable students to

- draw concrete scenario of software quality attributes;
- be familiar with tactics for achieving two types of software qualities.

Achieving QAs through tactics

A tactic is a design decision that influences the control of a quality attribute response.



The focus of a tactic is on a single quality attribute response.

Tactics differ from architectural patterns.

- Tradeoffs are built into the architectural patterns.

Achieving QAs through tactics

Tactics are techniques that an architect can use to *achieve* the required quality attributes.

Qualities are achieved via design decisions/tactics.

A system design consists of a collection of decisions.

- Some help control the quality attribute responses.
- Others ensure achievement of functionality.

Achieving QAs through tactics

Tactics to achieve two types of quality attributes:

- Availability
- Modifiability
- …

Achieving QAs through tactics

Tactics to achieve two types of quality attributes:

- Availability
- Modifiability
- …

What is availability?

Availability

- The software is there
- The software is ready to carry out its task
  - when one needs it

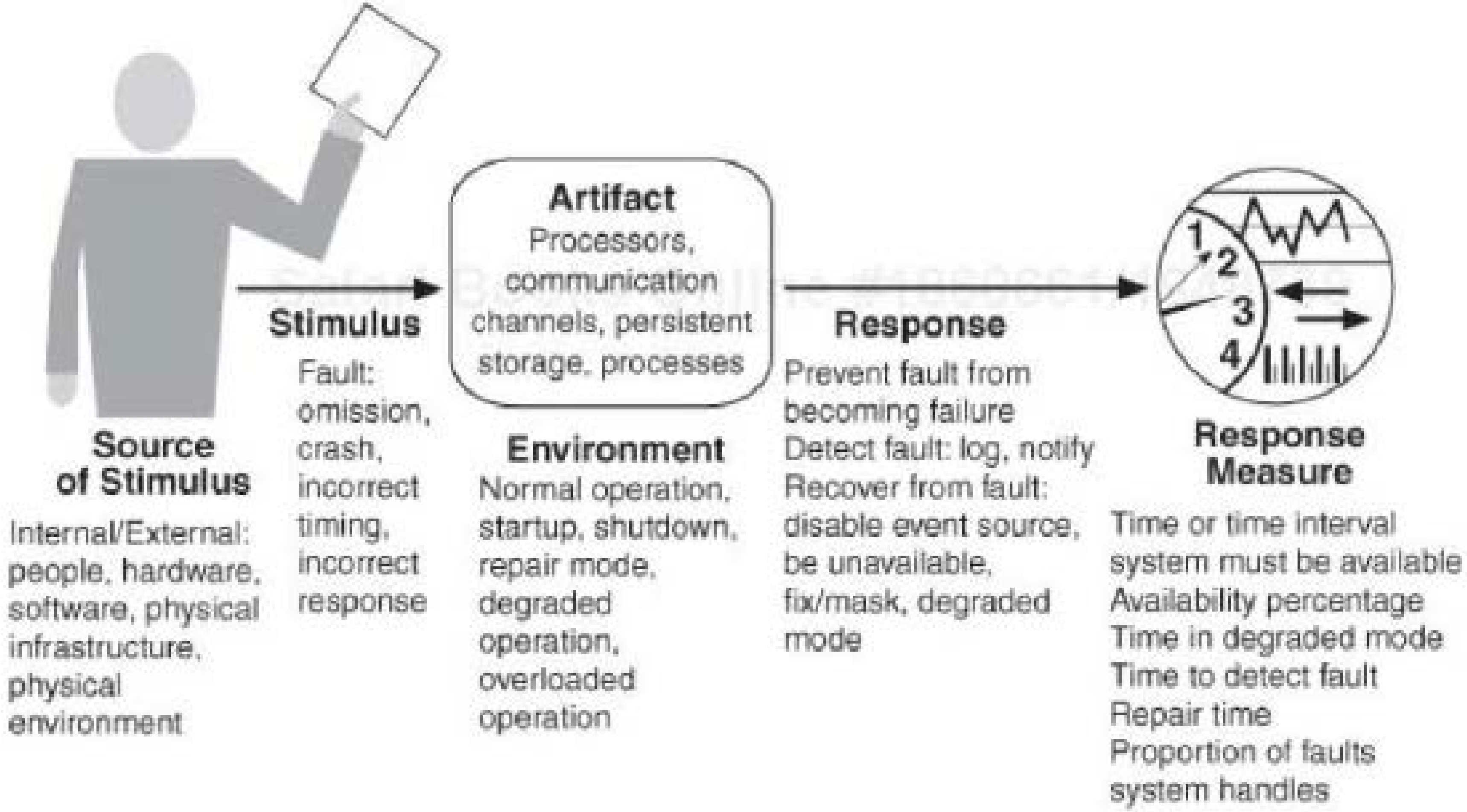
Failure

- Deviation from intended functional behavior (spec)
- Observable by system users

Failure vs fault

- Fault: event which may cause a failure

Availability general scenario



Source of Stimulus

Internal/External: people, hardware, software, physical infrastructure, physical environment

Stimulus

Fault: omission, crash, incorrect timing, incorrect response

Artifact

Processors, communication channels, persistent storage, processes

Environment

Normal operation, startup, shutdown, repair mode, degraded operation, overloaded operation

Response

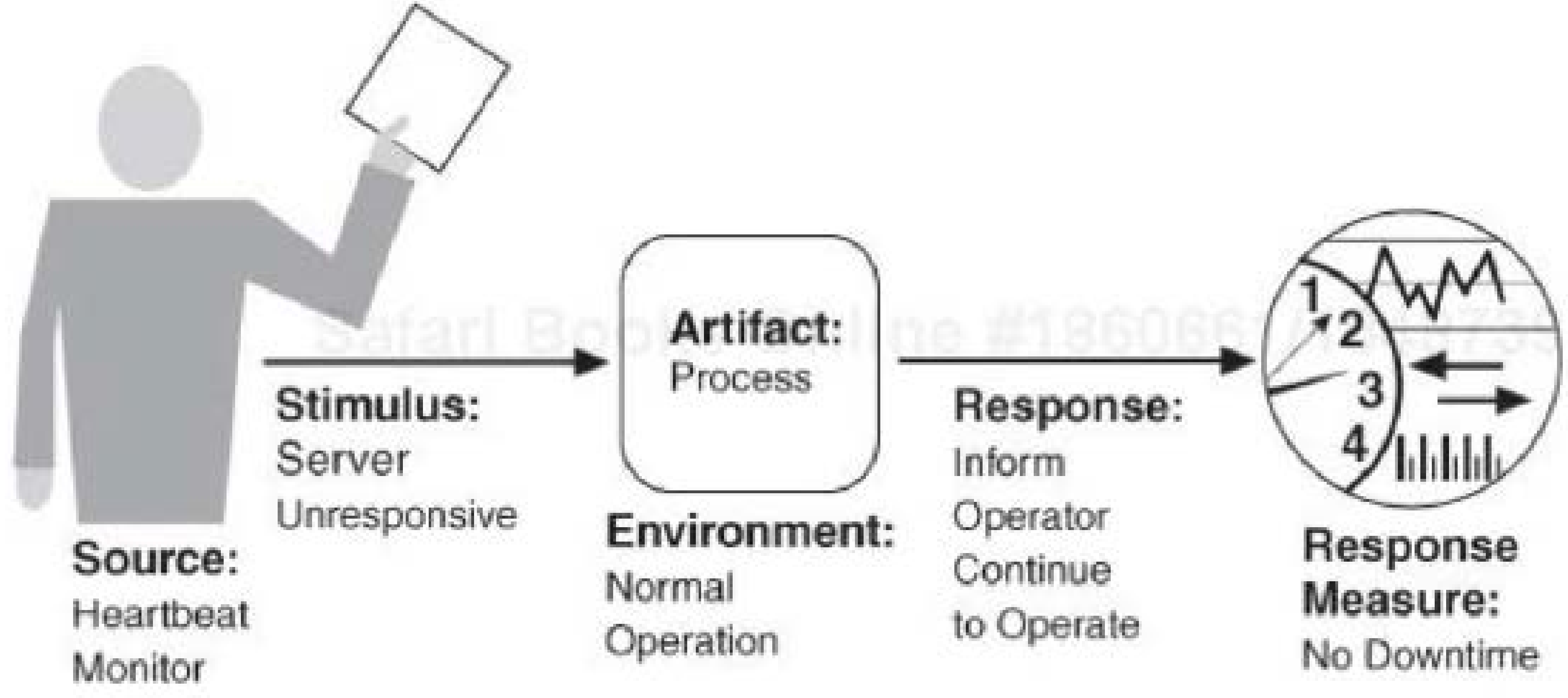
Prevent fault from becoming failure
Detect fault: log, notify
Recover from fault: disable event source, be unavailable, fix/mask, degraded mode

Response Measure

Time or time interval system must be available
Availability percentage
Time in degraded mode
Time to detect fault
Repair time
Proportion of faults system handles

Sample concrete availability scenario

The heartbeat monitor determines that the server is nonresponsive during normal operations. The system informs the operator and continues to operate with no downtime.

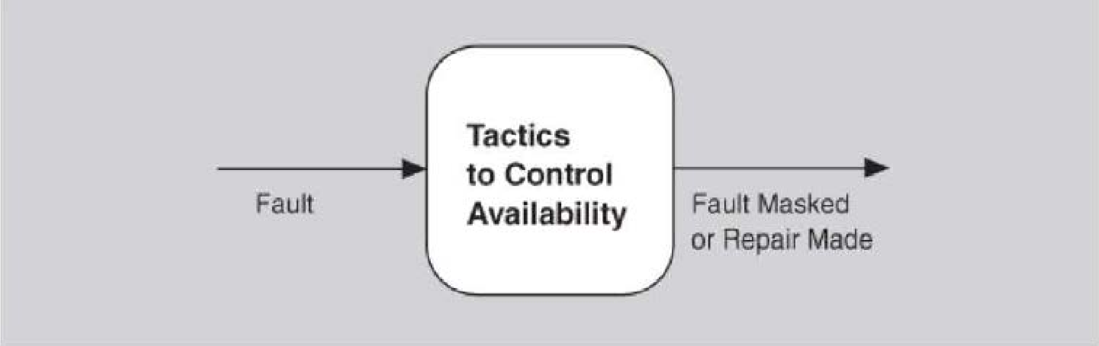


Tactics for availability

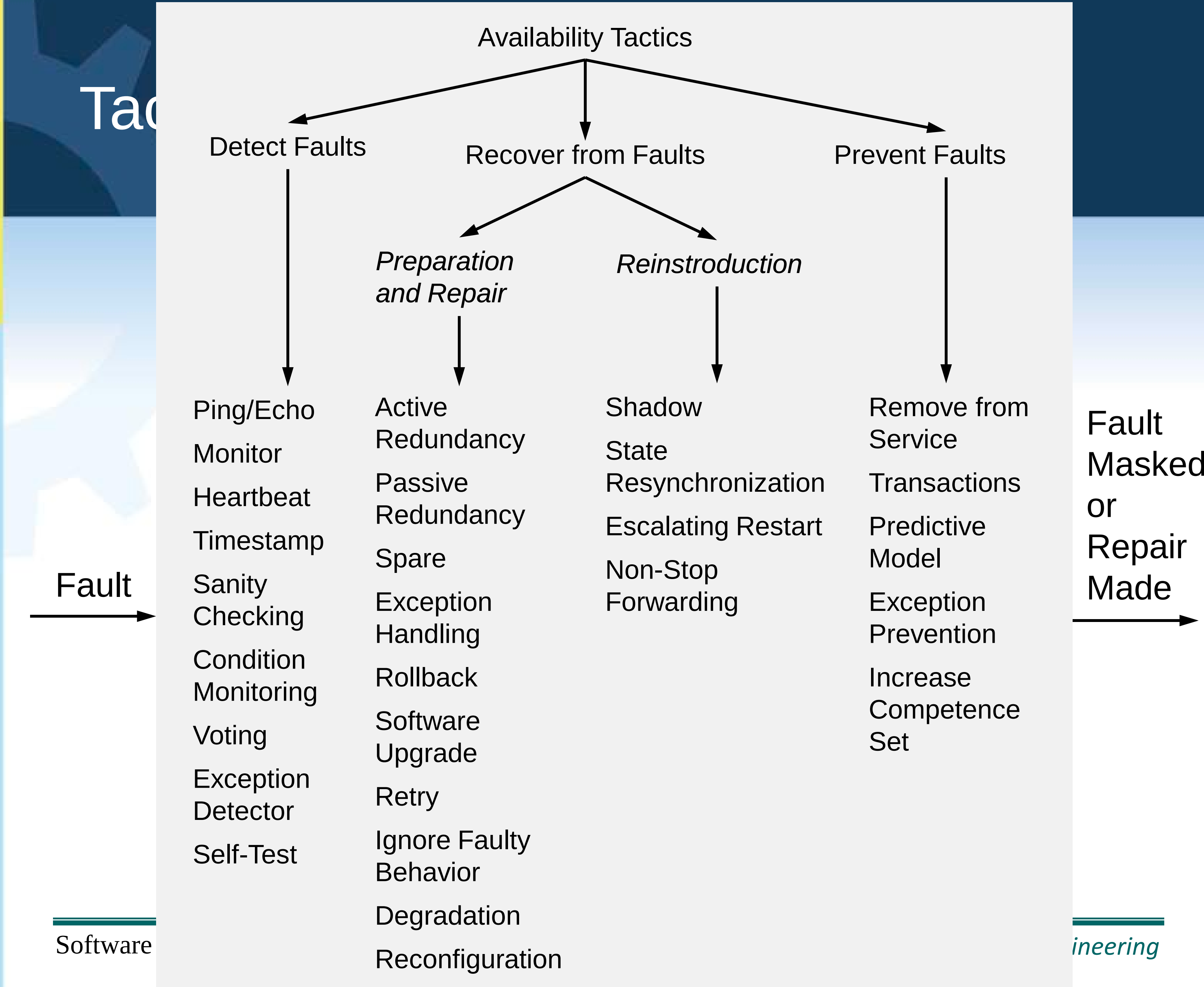
Availability tactics are designed to enable a system to endure system faults so that a service being delivered by the system remains compliant with its specification.

Goal of availability tactics

- Keep faults from becoming failures or
- bound the effects of the fault and make repair possible.



Tac



Fault

Availability Tactics

Detect Faults

Ping/Echo
Monitor
Heartbeat
Timestamp
Sanity Checking
Condition Monitoring
Voting
Exception Detector
Self-Test

Recover from Faults

Preparation and Repair

Active Redundancy
Passive Redundancy
Spare
Exception Handling
Rollback
Software Upgrade
Retry
Ignore Faulty Behavior
Degradation
Reconfiguration

Reinstroduction

Shadow
State Resynchronization
Escalating Restart
Non-Stop Forwarding

Prevent Faults

Remove from Service
Transactions
Predictive Model
Exception Prevention
Increase Competence Set

Fault Masked or Repair Made

Fault detection: Ping/echo

Ping/echo: An asynchronous request/response message pair exchanged between nodes.

- Comp. 1 (often a system monitor) issues a “ping” to comp. 2
- Comp. 1 expects an “echo” from comp. 2
- Answer within predefined time threshold

Usable for a group of components

- Mutually responsible for one task

Usable for client/server

- Tests the server and the communication path

Standard implementations are available for nodes interconnected via IP.

Fault detection: Monitor

Monitor: A component that is used to monitor the state of health of various other parts of the system (processors, I/O, memory, etc)

- Detect failure or congestion in the network

Fault detection: Heartbeat

Heartbeat: A fault detection mechanism that employs a periodic message exchange between a system monitor and a process being monitored.

- Comp. 1 emits a “heartbeat” message periodically
- Comp. 2 listens for it
- If heartbeat fails
  - Comp. 1 assumed failed
  - Fault correcting comp. 3 is notified

Heartbeat can also carry data

Fault detection: Heartbeat

```
// Each process has its own ID.
var processId = "alpha";
var pubClient = require("redis").createClient();
setInterval(function () {
  pubClient.publish("heartbeat", processId);
}), 100);
```

Fault detection: Heartbeat

```
/* Heartbeat messages */
struct heartbeat {                  // Sent in both directions
        DWORD seqno;                // Sequence number of this Heartbeat
};
```

Fault detection: Voting

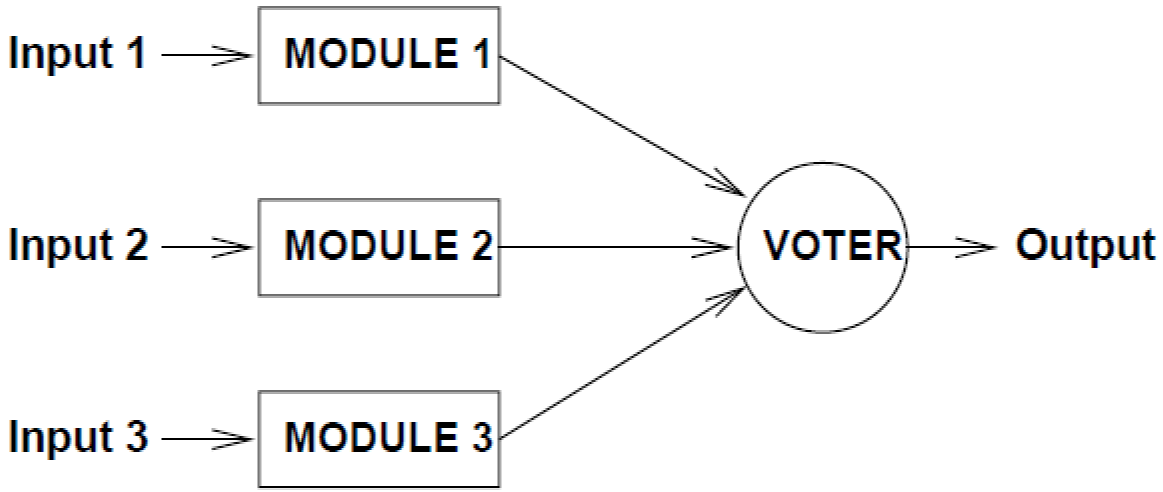
Triple modular redundancy (TMR) employs three components that do the same thing.

- Each component receives identical inputs, and forwards their output to voting logic.

When an inconsistency occurs, the voter reports a fault.

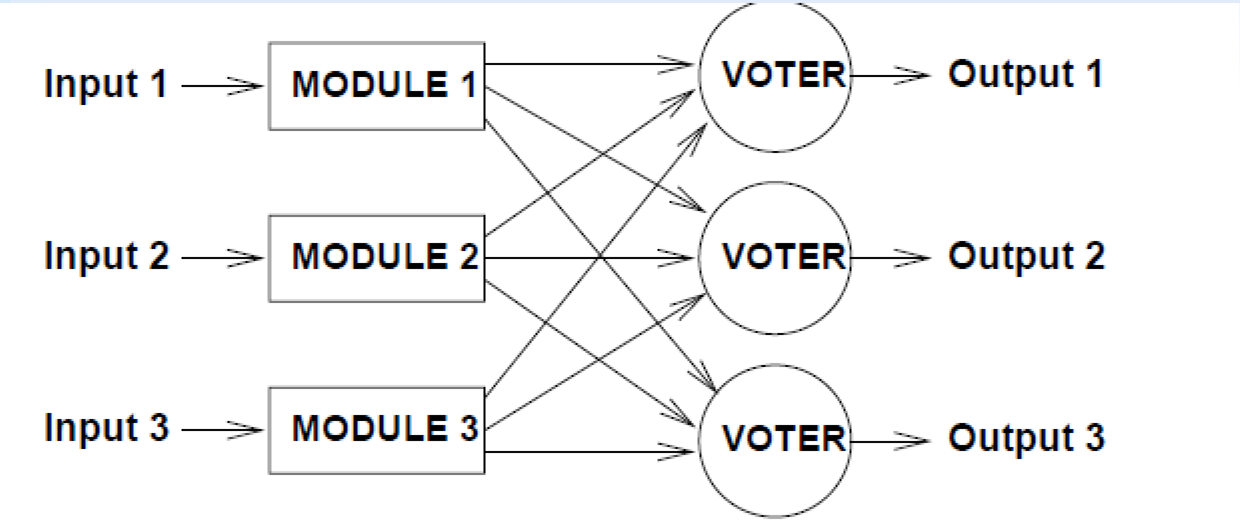
- It must also decide what output to use (majority voting or average of outputs).

Fault detection: Voting



Masks failure of a single component.
Voter is a SINGLE POINT OF FAILURE.

Fault detection: Voting



Fault detection: Voting

Replication

- Components are exact clones of each other.

Functional redundancy

- It is intended to address the issue of common-mode failures (design or implementation faults).
- Components are diversely designed and implemented with the same output given the same input.

Analytic redundancy

- It allows diversity among the components' inputs and outputs.
- It is intended to tolerate specification errors by using separate requirement specifications

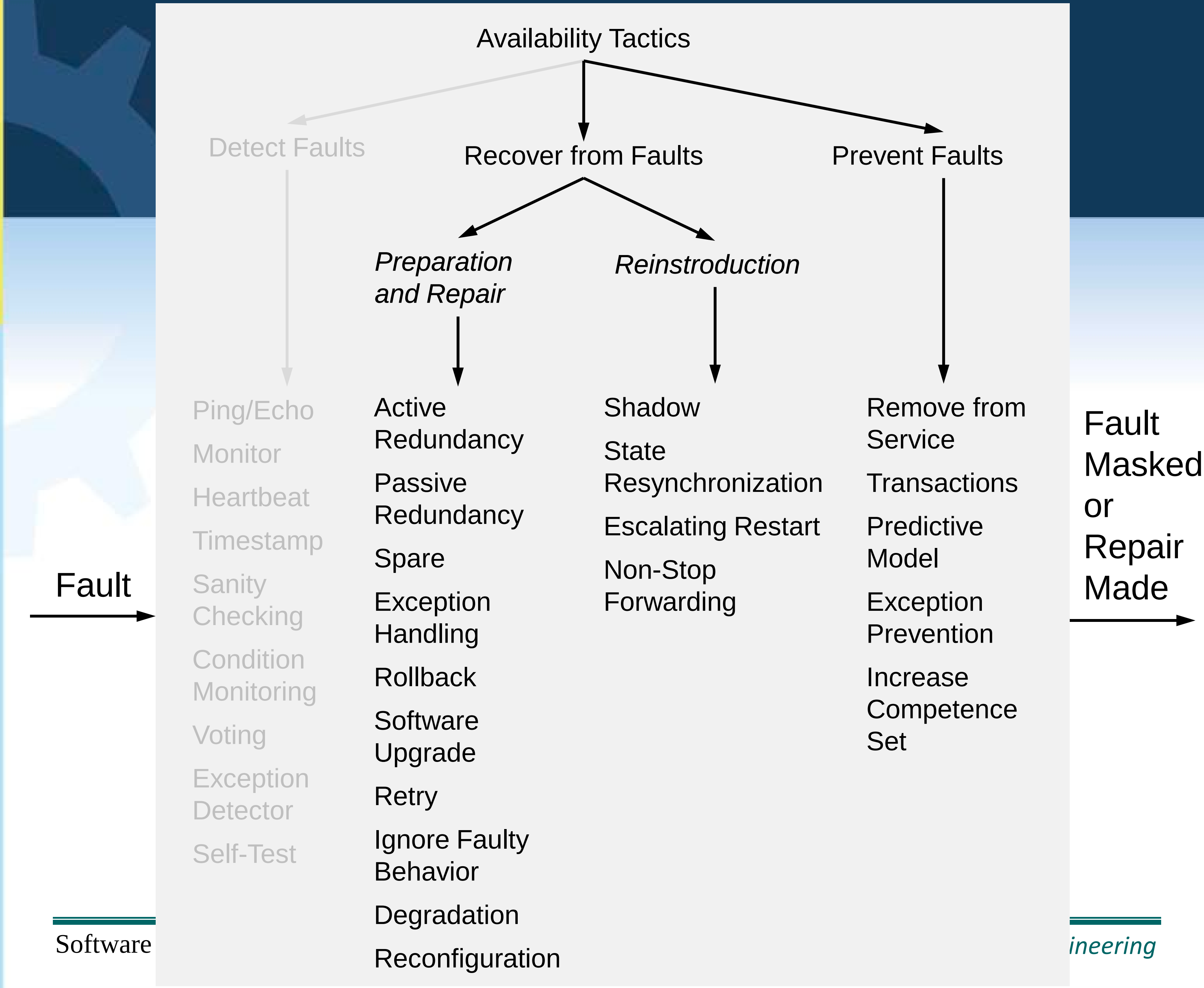
Fault detection: Others

Timestamp: used to detect incorrect sequences of events, primarily in distributed message-passing systems.

Sanity Checking: checks the validity or reasonableness of a component’s operations or outputs; typically based on a knowledge of the internal design, the state of the system, or the nature of the information under scrutiny.

Condition Monitoring: checking conditions in a process or device, or validating assumptions made during the design.

Self-test: procedure for a component to test itself for correct operation.



Availability Tactics

Detect Faults

- Ping/Echo
- Monitor
- Heartbeat
- Timestamp
- Sanity Checking
- Condition Monitoring
- Voting
- Exception Detector
- Self-Test

Recover from Faults

Preparation and Repair

- Active Redundancy
- Passive Redundancy
- Spare
- Exception Handling
- Rollback
- Software Upgrade
- Retry
- Ignore Faulty Behavior
- Degradation
- Reconfiguration

Reinstroduction

- Shadow
- State Resynchronization
- Escalating Restart
- Non-Stop Forwarding

Prevent Faults

- Remove from Service
- Transactions
- Predictive Model
- Exception Prevention
- Increase Competence Set

Fault

Fault Masked or Repair Made

Fault recovery

Preparation-and-repair tactics

- They are based on a variety of combinations of retrying a computation or introducing redundancy.

Reintroduction tactic

- It is where a failed component is reintroduced after it has been corrected.

Preparation-and-repair: Active redundancy

All redundant components respond to events in parallel.

- All in the same state.

Response from only one component is used.

Downtime: switching time to another up-to-date component (ms)

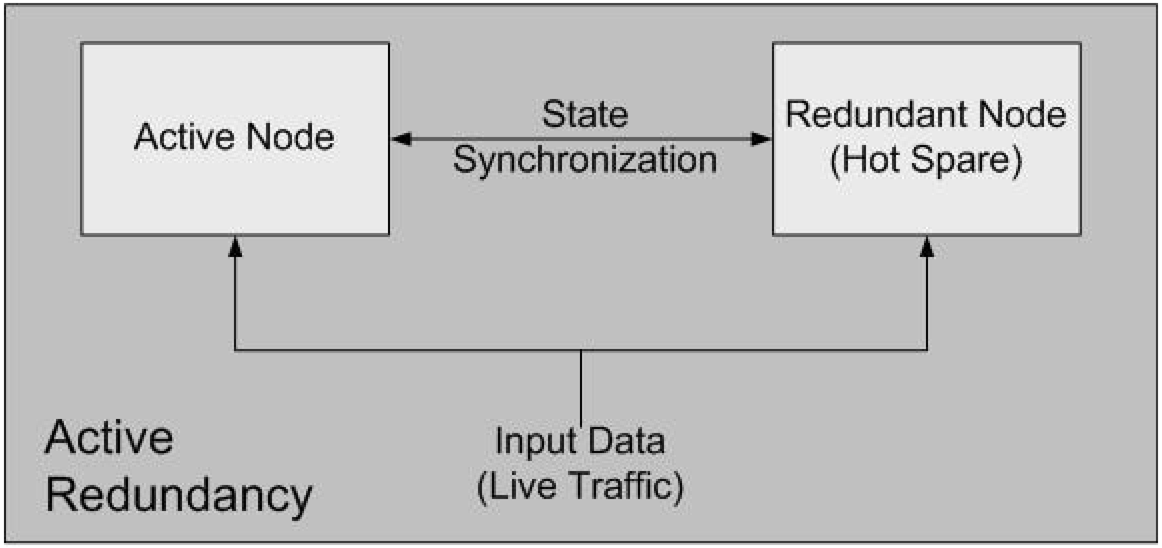
Used in client-server config (database system)

- Quick responses are important.

Synchronization

- All messages to any redundant component sent to all redundant components.

Preparation-and-repair: Active redundancy



Preparation-and-repair: Passive redundancy

Primary component

- responds to events
- informs standby components of state updates they must make

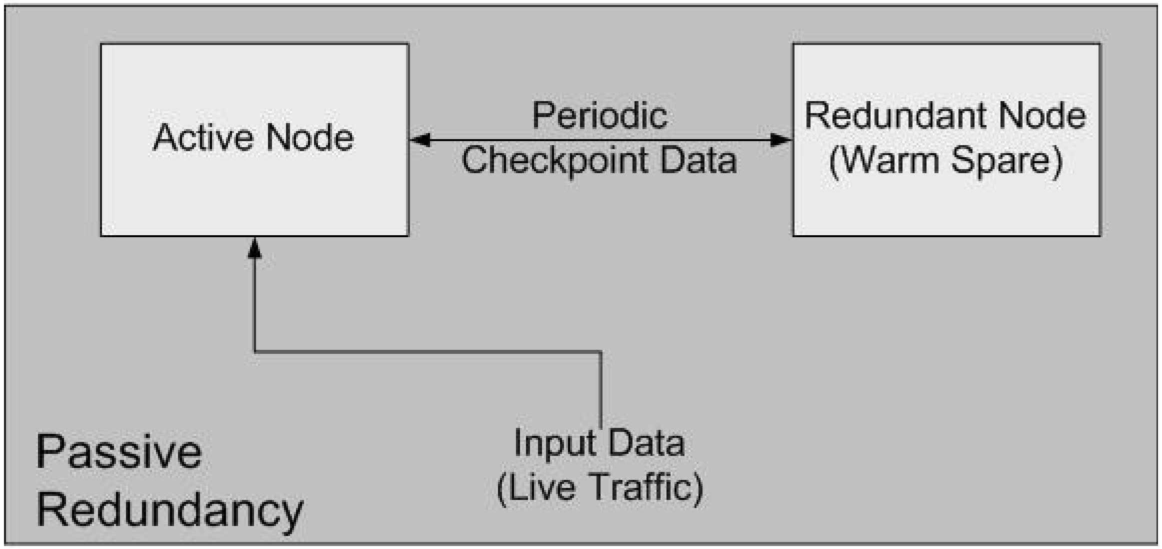
Fault occurs:

- System checks if backup sufficiently fresh before resuming services

Often used in control systems

Periodical switchovers increase availability

Preparation-and-repair: Passive redundancy



Preparation-and-repair: Spare

Standby spare computing platform configured to replace many different failed components

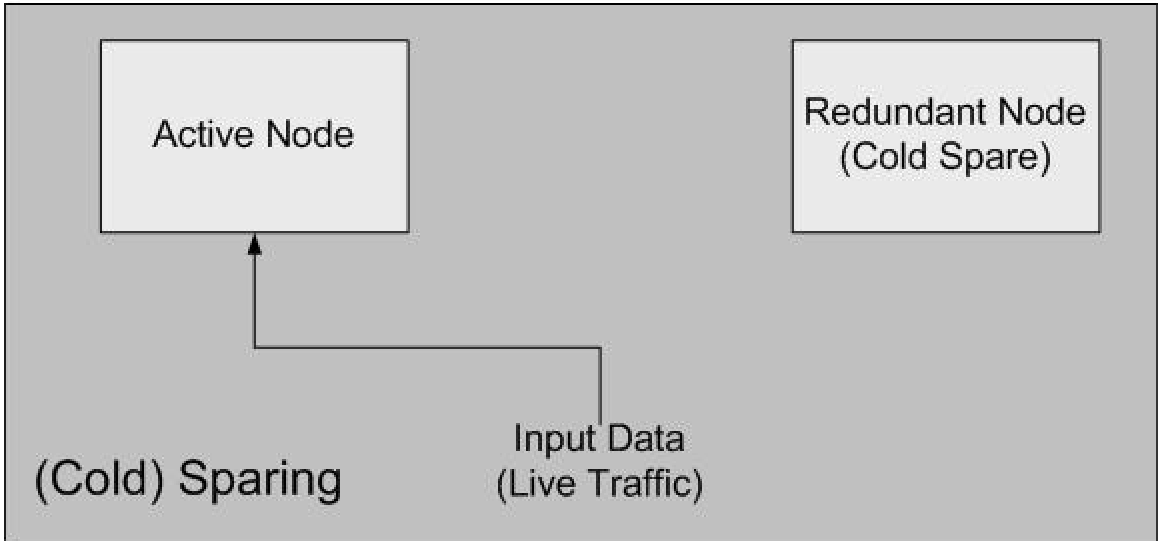
- Must be rebooted to appropriate SW config
- Have its state initialized when failure occurs

Checkpoint of system state and state changes to persistent device periodically.

Downtime: minutes

- Suited for systems having only high-reliability instead of high-availability.

Preparation-and-repair: Spare



Preparation-and-repair: Others

Exception Handling: dealing with the exception by reporting it or handling it, potentially masking the fault by correcting the cause of the exception and retrying.

Rollback: permits the system to revert to a previous known good state upon the detection of a failure.

Software Upgrade: in-service upgrades to executable code images in a non-service-affecting manner.

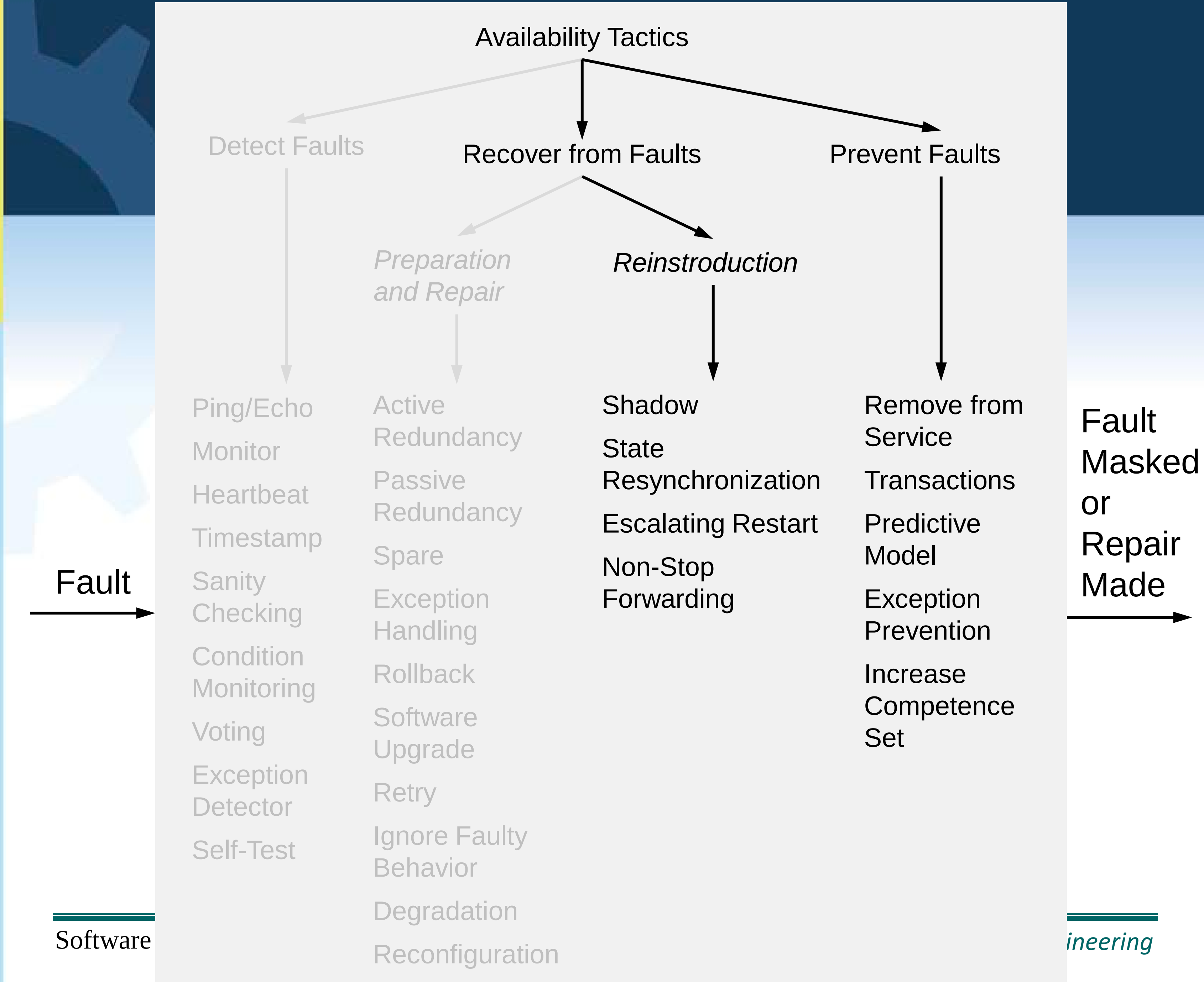
Retry: where a failure is transient retrying the operation may lead to success.

Preparation-and-repair: Others

Ignore Faulty Behavior: ignoring messages sent from a source when it is determined that those messages are spurious.

Degradation: maintains the most critical system functions in the presence of component failures, dropping less critical functions.

Reconfiguration: reassigning responsibilities to the resources left functioning, while maintaining as much functionality as possible.



Availability Tactics

Detect Faults

Recover from Faults

Prevent Faults

Preparation and Repair

Reinstroduction

Ping/Echo

Monitor

Heartbeat

Timestamp

Sanity Checking

Condition Monitoring

Voting

Exception Detector

Self-Test

Active Redundancy

Passive Redundancy

Spare

Exception Handling

Rollback

Software Upgrade

Retry

Ignore Faulty Behavior

Degradation

Reconfiguration

Shadow

State Resynchronization

Escalating Restart

Non-Stop Forwarding

Remove from Service

Transactions

Predictive Model

Exception Prevention

Increase Competence Set

Fault

Fault Masked or Repair Made

Reintroduction: Shadow

Previously failed component may be run in “shadow” mode.

- For a while
- To make sure it mimics the behavior of the working components
- Before restoring it to service

Reintroduction: State resynchronization

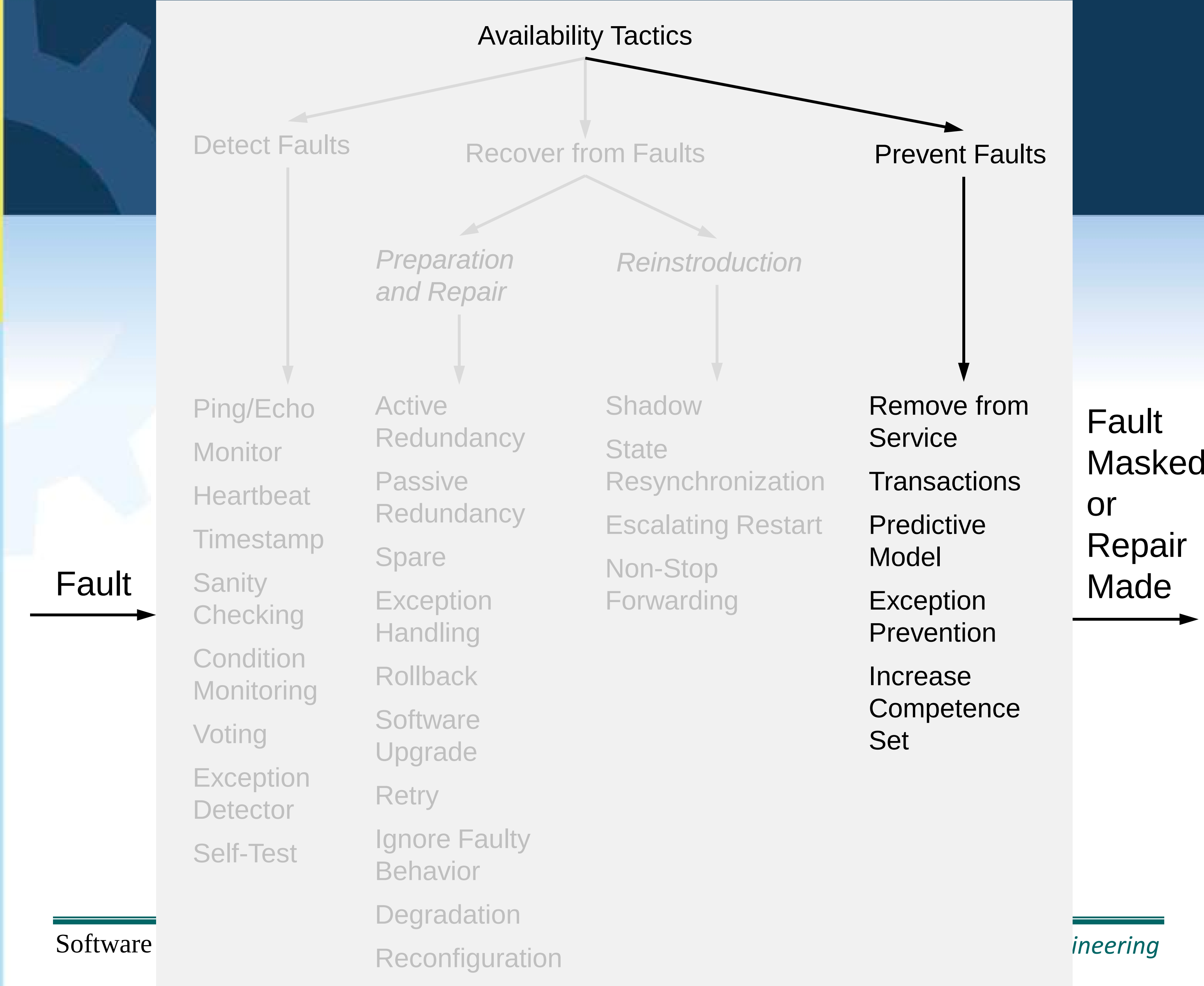
Partner to passive and active redundancy

- Restored component upgrades its state before return to service.
- Active redundancy: checksum, MD5…
- Passive redundancy: checkpoint

Reintroduction: Others

Escalating restart: allows the system to recover from faults by varying the granularity of the component(s) restarted and minimizing the level of service affected.

Non-stop forwarding: used to enable graceful degradation of high-availability systems.



Availability Tactics

Fault

Detect Faults

- Ping/Echo
- Monitor
- Heartbeat
- Timestamp
- Sanity Checking
- Condition Monitoring
- Voting
- Exception Detector
- Self-Test

Recover from Faults

Preparation and Repair

- Active Redundancy
- Passive Redundancy
- Spare
- Exception Handling
- Rollback
- Software Upgrade
- Retry
- Ignore Faulty Behavior
- Degradation
- Reconfiguration

Reinstroduction

- Shadow
- State Resynchronization
- Escalating Restart
- Non-Stop Forwarding

Prevent Faults

- Remove from Service
- Transactions
- Predictive Model
- Exception Prevention
- Increase Competence Set

Fault Masked or Repair Made

Fault prevention

Removal from service

- Comp removed from operation to undergo some activities to prevent anticipated failures
- Exp: rebooting comp to prevent memory leaks

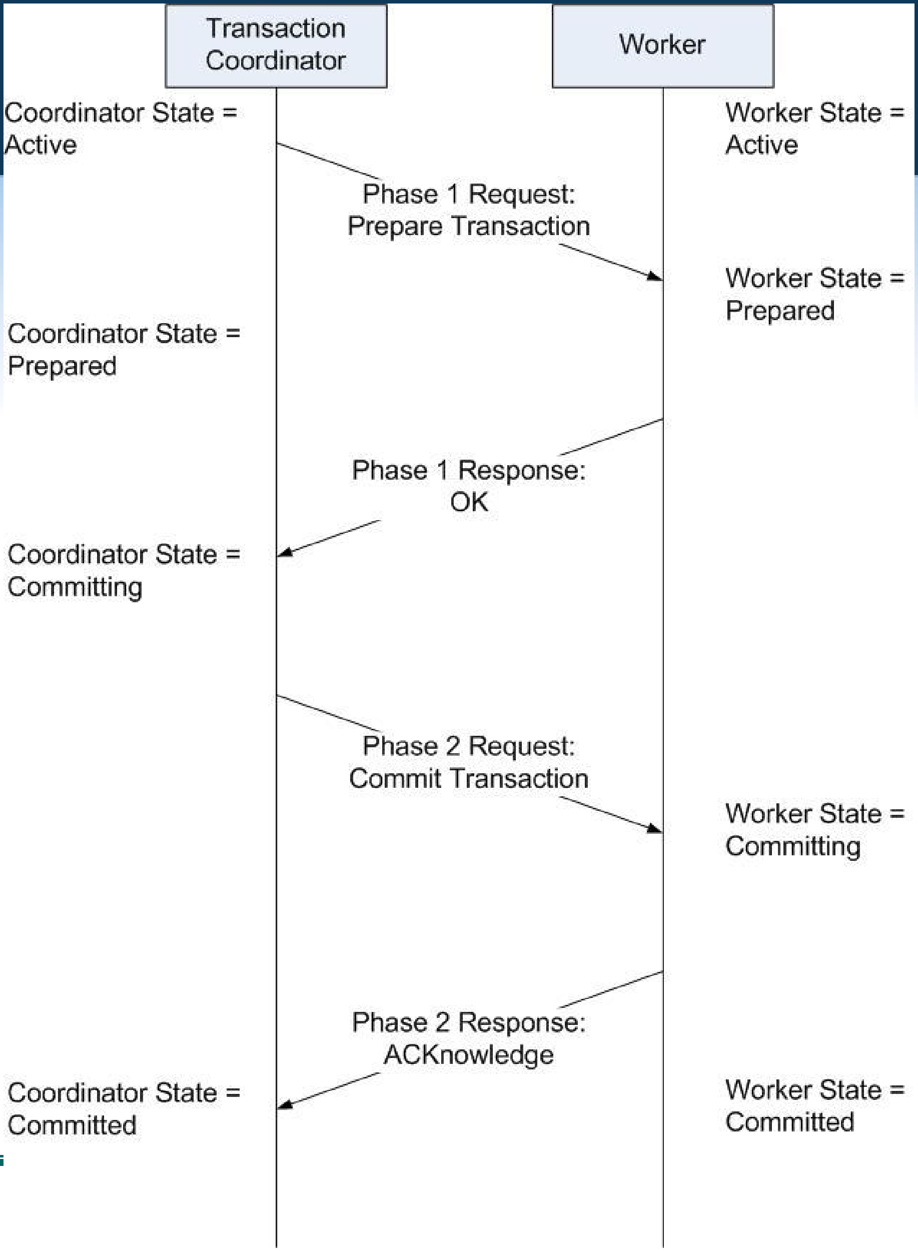
Transactions

- sequential steps bundled together, s.t. the whole bundle can be undone at once
- Atomic, consistent, isolated, and durable (ACID property)

Fault prevention

Transactions tactic: Two-Phase commit

- Prevent race condition


Transaction Coordinator
Worker
Coordinator State = Active
Worker State = Active
Phase 1 Request: Prepare Transaction
Worker State = Prepared
Coordinator State = Prepared
Phase 1 Response: OK
Coordinator State = Committing
Phase 2 Request: Commit Transaction
Worker State = Committing
Phase 2 Response: ACKnowledge
Coordinator State = Committed
Worker State = Committed

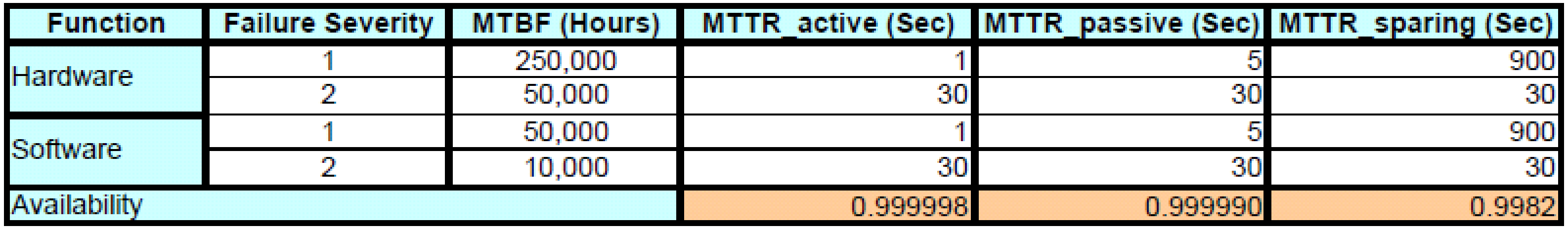
Fault prevention: Others

Predictive Model: monitor the state of health of a process to ensure that the system is operating within nominal parameters; take corrective action when conditions are detected that are predictive of likely future faults. □

Exception Prevention: preventing system exceptions from occurring by masking a fault, or preventing it via smart pointers, abstract data types, wrappers.□

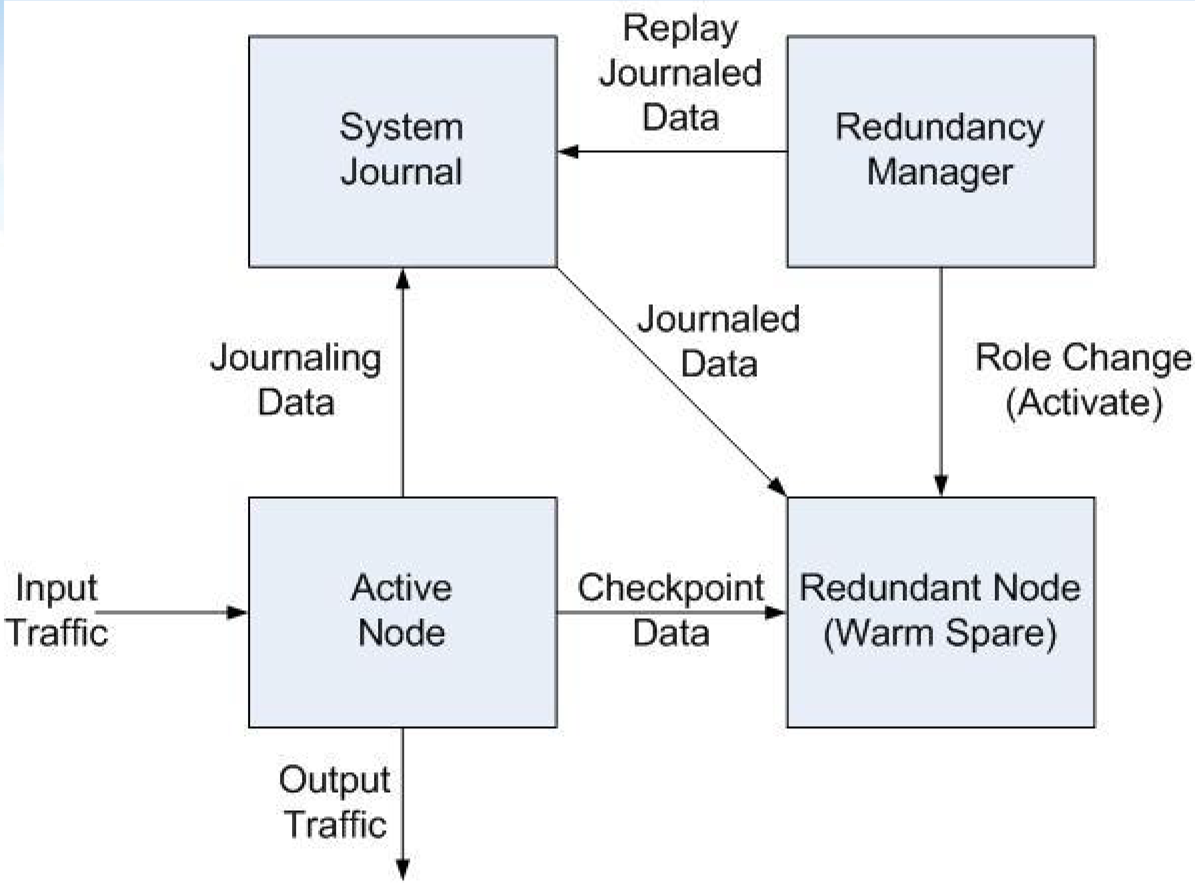
Increase Competence Set: designing a component to handle more cases—faults—as part of its normal operation.

Example for availability tactics



Function	Failure Severity	MTBF (Hours)	MTTR_active (Sec)	MTTR_passive (Sec)	MTTR_sparing (Sec)
Hardware	1	250,000	1	5	900
	2	50,000	30	30	30
Software	1	50,000	1	5	900
	2	10,000	30	30	30
Availability			0.999998	0.999990	0.9982

Example for availability tactics



Discuss questions

1. Write a concrete availability scenario for a program like Microsoft Word.
2. Redundancy is often cited as a key strategy for achieving high availability. Look at the tactics presented in this chapter and decide how many of them exploit some form of redundancy and how many do not.
3. Consider the fault detection tactics (ping/echo, heartbeat, system monitor, voting, and exception detection). What are the performance implications of using these tactics?

Achieving QAs through tactics

Tactics to achieve two types of quality attributes:

- Availability
- Modifiability

What is modifiability?

Modifiability is about change and our interest in it is the cost and risk of making changes.

To plan for modifiability, an architect has to consider four questions:

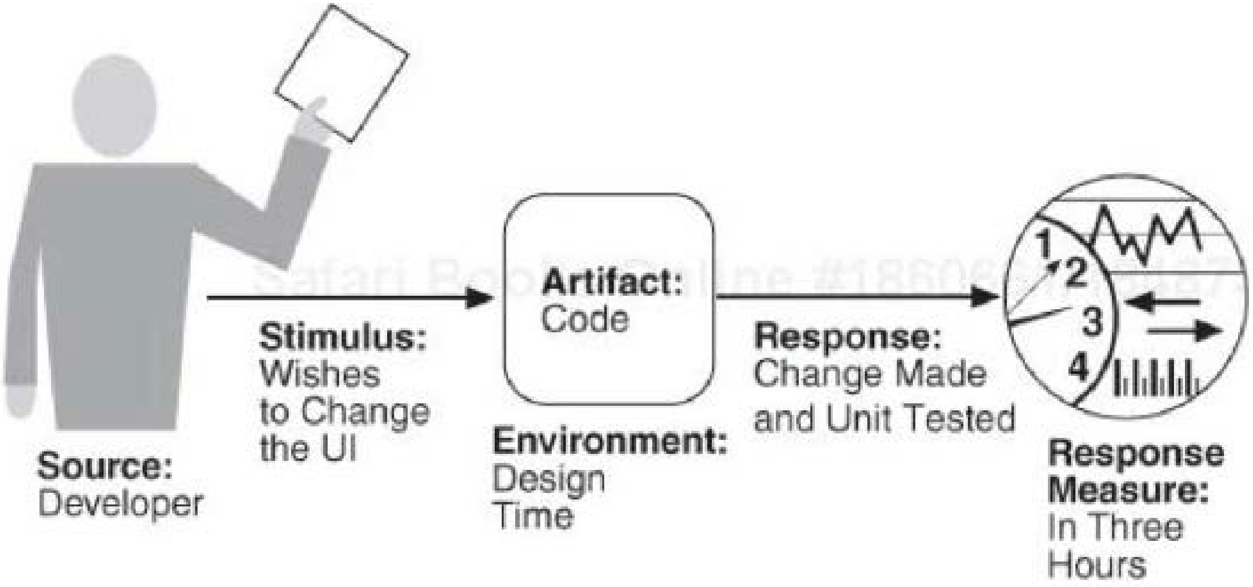
- What can change?
  - functions, platform, environment, system qualities, capacity…
- What is the likelihood of the change?
- When is the change made and who makes it?
  - implementation, compile, build, configuration setup, execution
- What is the cost of the change?
  - The cost of introducing the mechanism(s) and the cost of using it.

Modifiability general scenario

Source of stimulus	The developer, a system administrator, or an end user.
Stimulus	The addition of a function, the modification of an existing function, or the deletion of a function. Making the system more responsive, increasing its availability. Accommodating an increasing number of simultaneous users. changes may happen to accommodate new technology of some sort, the most common of which is porting the system to a different type of computer or communication network.
Artifact	Specific components or modules, the system's platform, its user interface, its environment, or another system with which it interoperates.
Environment	design time, compile time, build time, initiation time, or runtime.
Response	Make the change, test it, and deploy it.
Response measure	All of the possible responses take time and cost money; time and money are the most common response measures.

Sample concrete modifiability scenario

The developer wishes to change the user interface by modifying the code at design time. The modifications are made with no side effects within three hours.



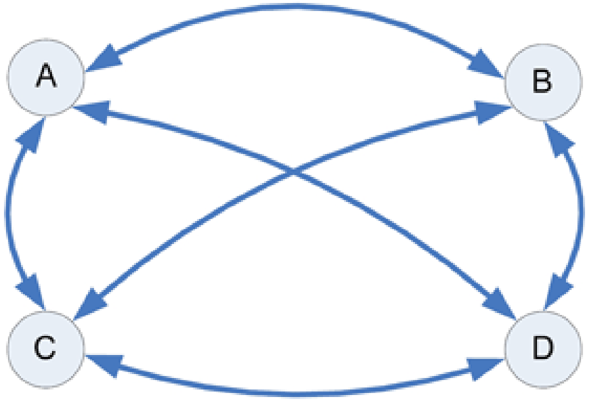
Coupling

Coupling is the overlap of two modules' responsibilities.

- It is a measure of interconnection among modules.
- "Strength" of coupling

High coupling is an enemy of modification.

- Components depend on each other.
- Strong coupling: Changes in A → changes in B, C and D. Changes in B → changes in A, C, and D.



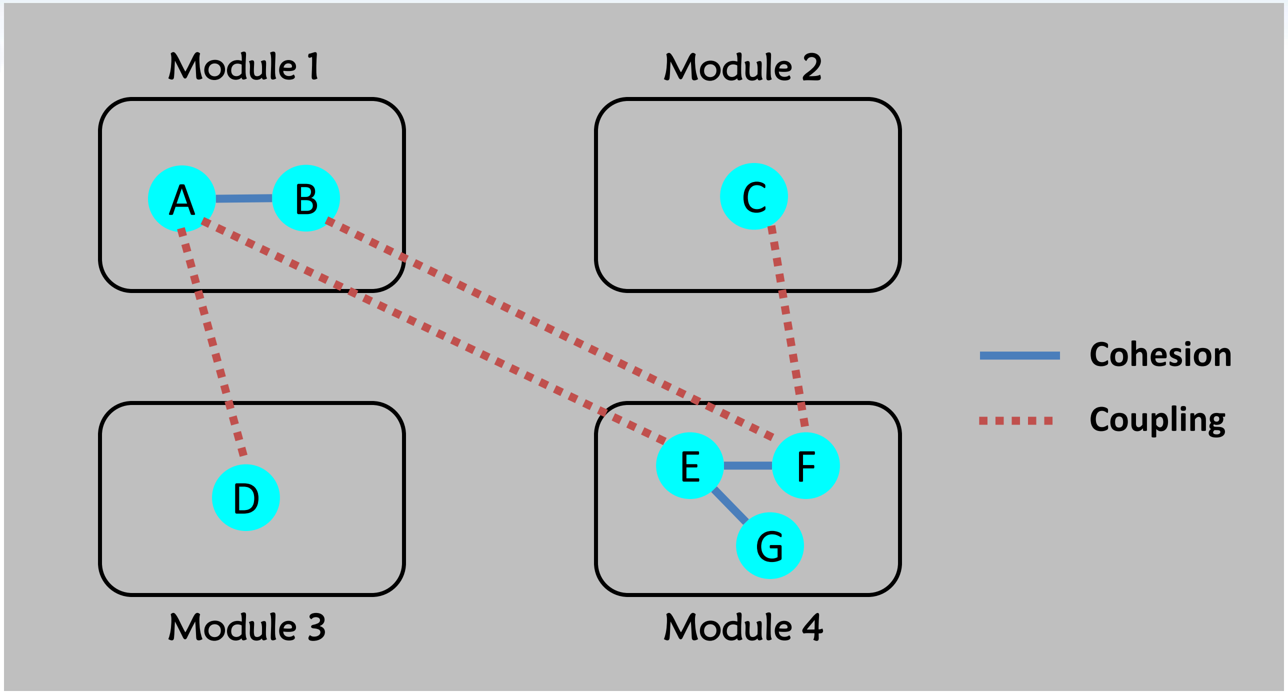
Cohesion

Cohesion is a measure of the relationship among the responsibilities of a specific module. It measures

- how strongly the responsibilities of a module are related.
- the module’s “unity of purpose”.
- Example: routine *ComputeAndDisplayFibonacci* vs. routines *ComputeFibonacci* and *DisplayFibonacci*

The higher the cohesion, the lower the probability that a given change will affect multiple responsibilities.

Coupling and cohesion



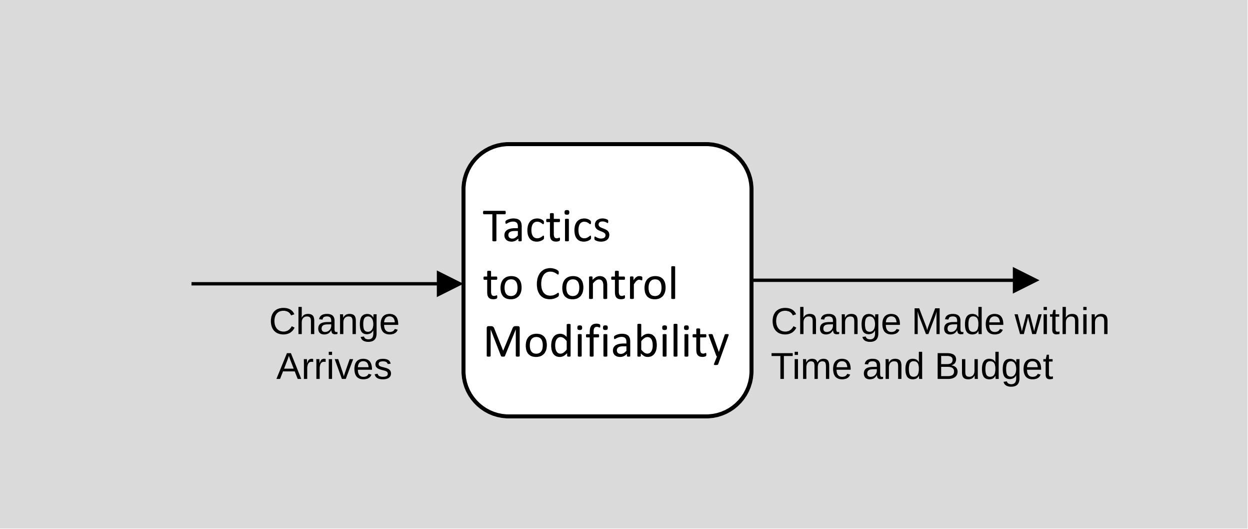
Tactics for modifiability

Parameters motivate modifiability tactics:

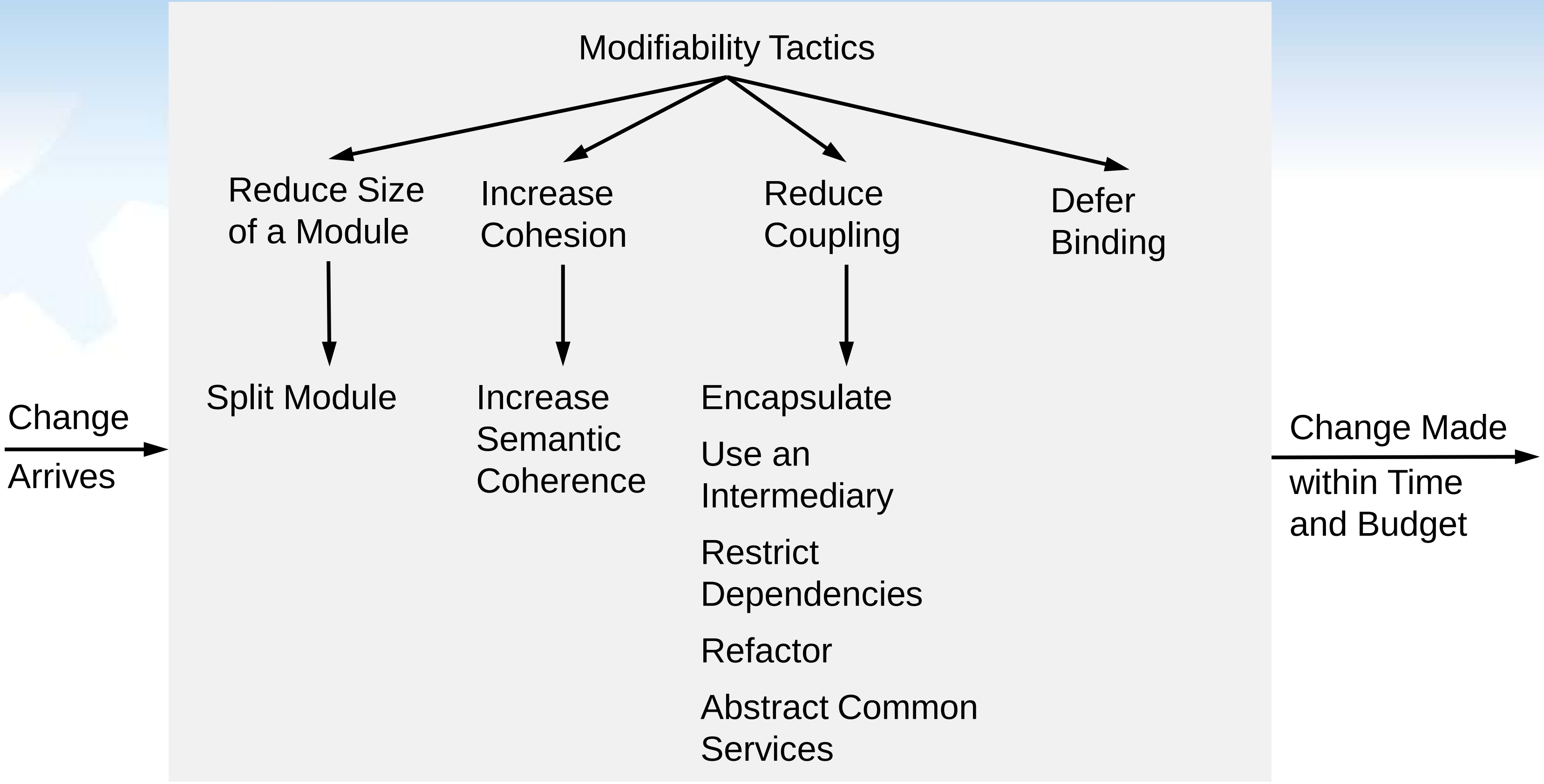
- Size of a module
- Coupling
- Cohesion
- Binding time of modification
  - Modification made late in the life cycle will cost less.

Tactics for modifiability

Goal: controlling the complexity of making changes, as well as the time and cost to make changes.


Change Arrives
Tactics to Control Modifiability
Change Made within Time and Budget

Tactics for modifiability



Modifiability Tactics

Reduce Size of a Module

Split Module

Increase Cohesion

Increase Semantic Coherence

Reduce Coupling

Encapsulate

Use an Intermediary

Restrict Dependencies

Refactor

Abstract Common Services

Defer Binding

Change Arrives

Change Made within Time and Budget

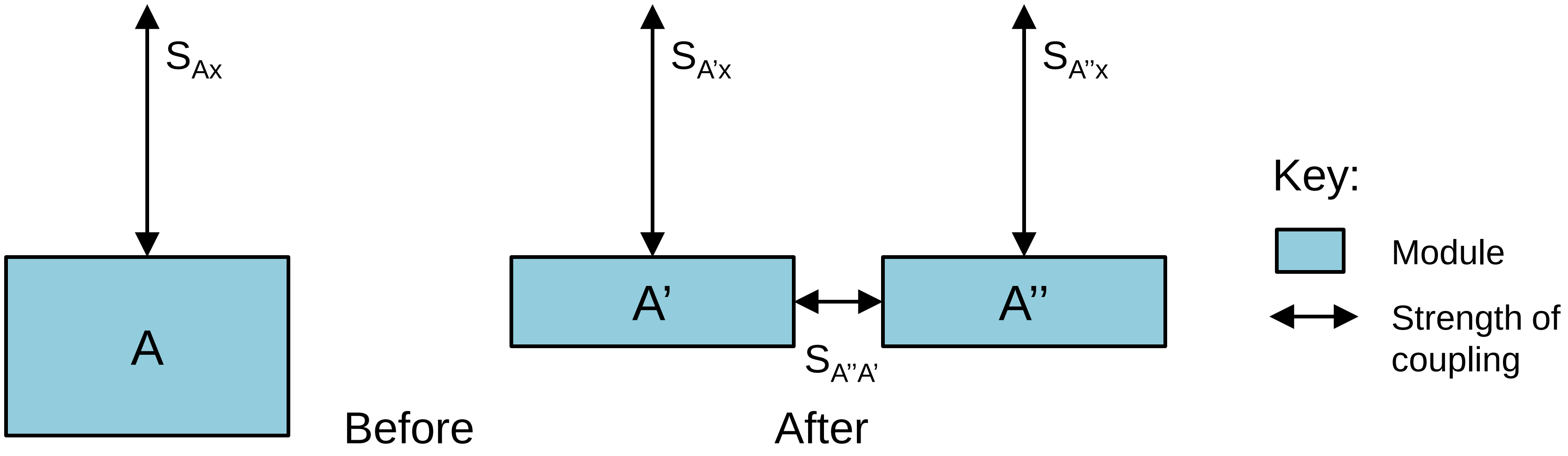
Reduce size of a module: Split module

Refining the module into several smaller modules should reduce the average cost of future changes.

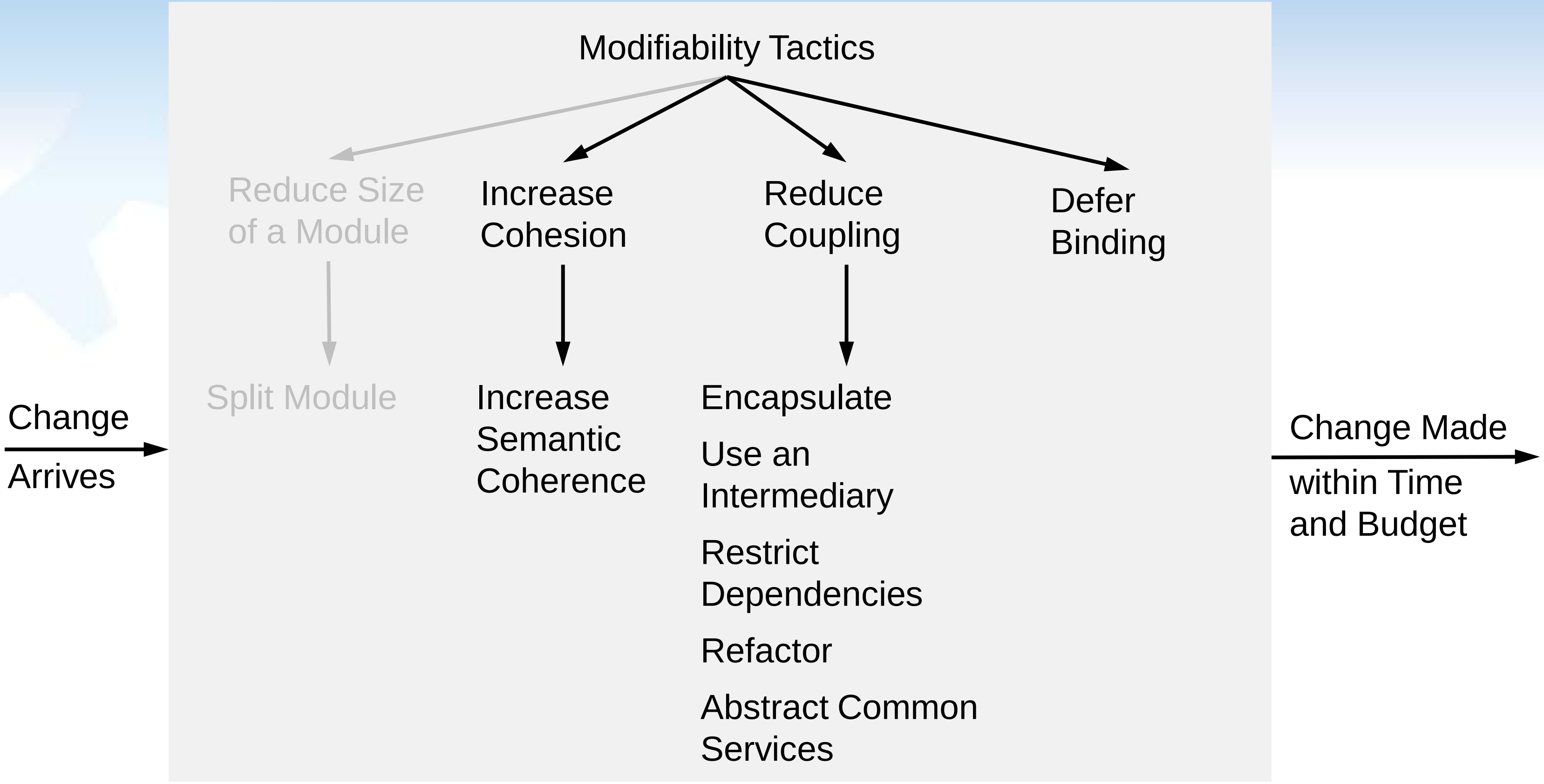
- High capability → high cost of modification

Criterion for splitting:

- Children module can be modified independently.



Tactics for modifiability



Increase cohesion: Increase semantic cohesion

If responsibility A and B in a module do not serve the same purpose,

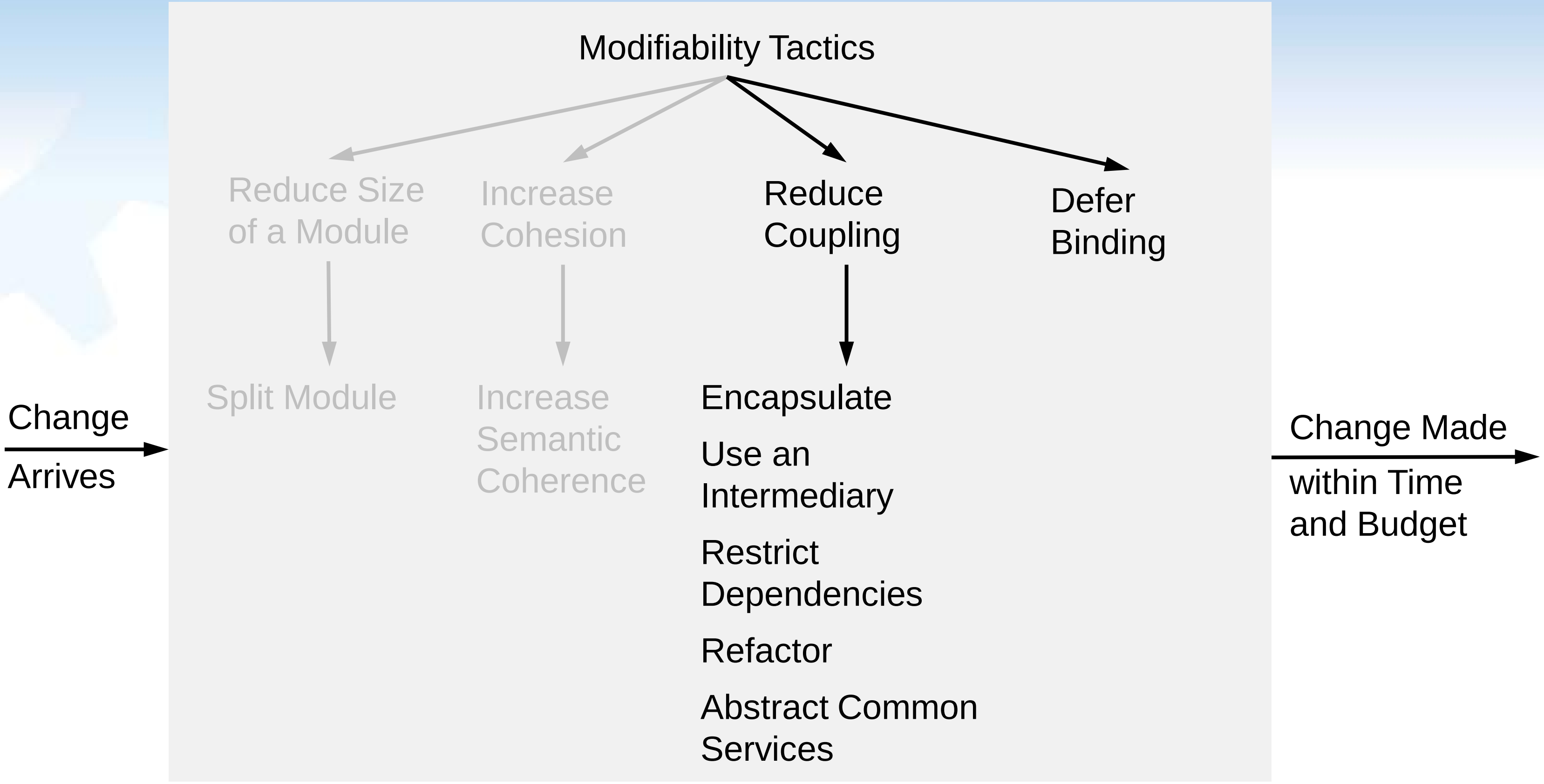
- place them in different modules.

The purpose of moving responsibilities from one module to another is to reduce the likelihood of side effects affecting other responsibilities in one module.

If some responsibilities are not affected by changes,

- they should be removed from the original module.

Tactics for modifiability



Reduce coupling: Encapsulate

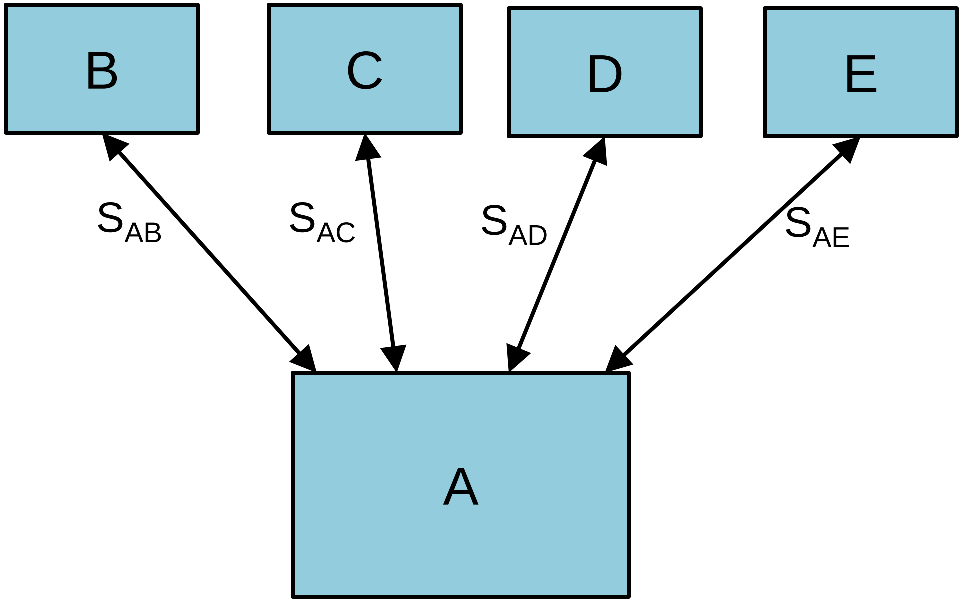
The purpose is to reduce the probability that a change to one module propagates to other modules

- by introducing an explicit interface.

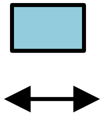
The interface limits the ways in which external responsibilities can interact with the module.

Interface should hide details of the module.

Reduce coupling: Encapsulate



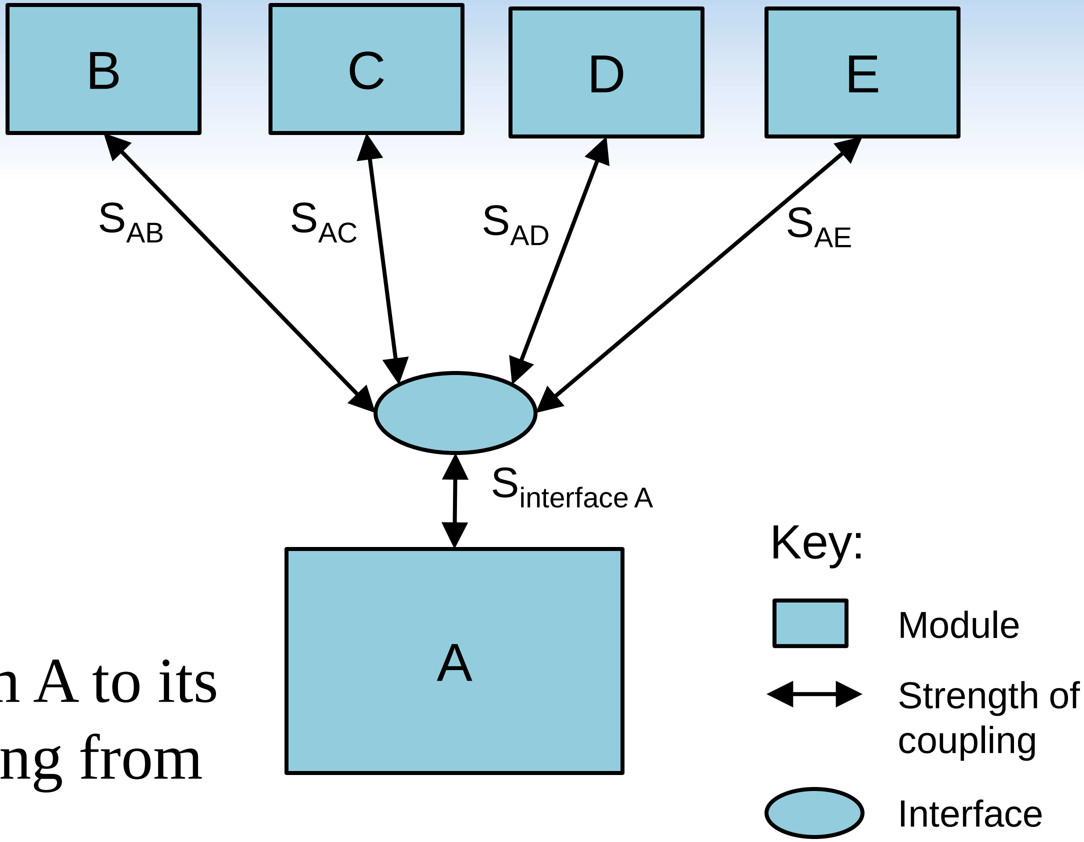
Key:



Reduce coupling: Encapsulate

Changes in the architecture:

- An explicit interface is added.
- Coupling between A and x → coupling between A and its interface.
- Strong coupling from A to its interface; low coupling from its interface to A.



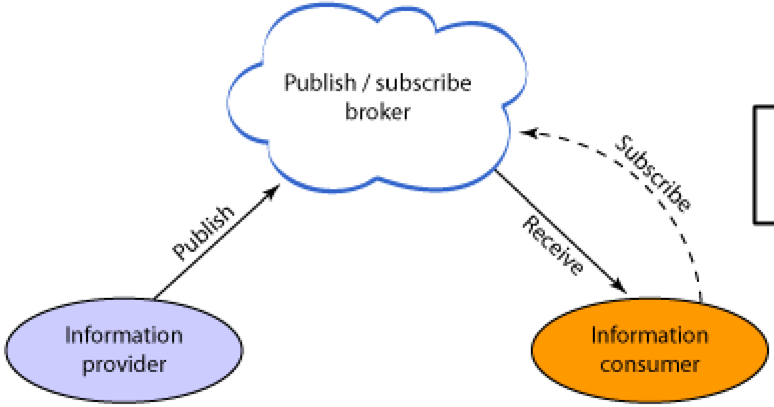
Reduce coupling: Use an intermediary

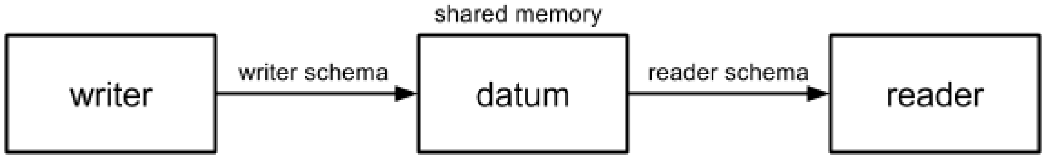
An *intermediary* breaks a dependency.

- Carrying out B requires carrying out A first

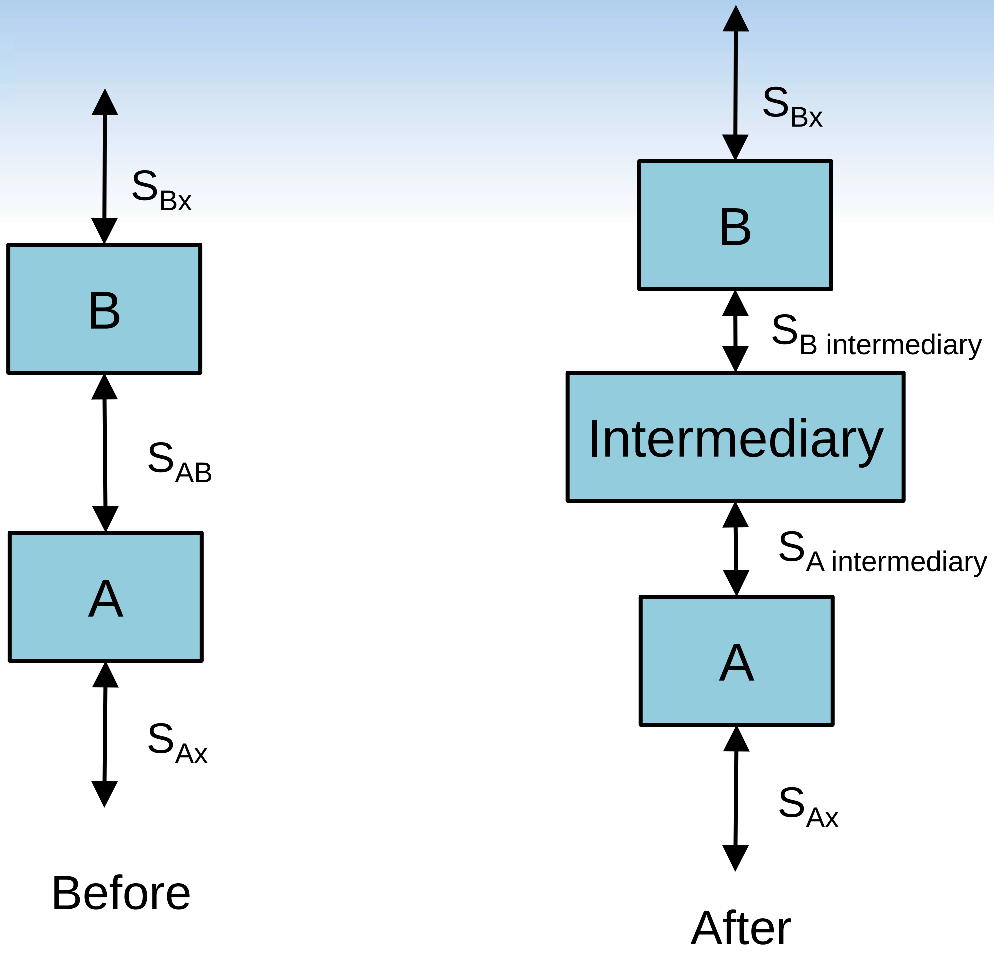
The type of intermediary depends on the type of dependency.

- A is a data producer and B is a data consumer: use a Publisher-Subscriber intermediary.
- In a shared data repository, separates readers from writers.





Reduce coupling: Use an intermediary

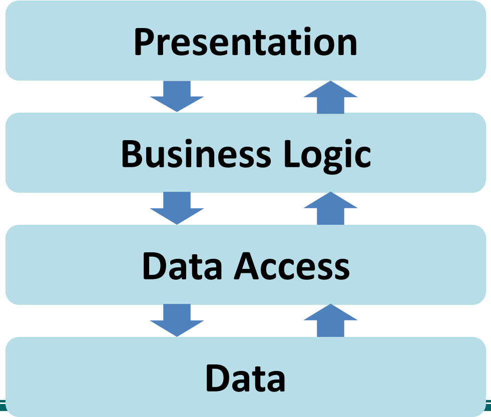


Reduce coupling: Restrict dependence

Restrict dependencies is a tactic that restricts the modules that a given module interacts with or depends on. It is achieved by

- restricting a module’s visibility;
- authorization.

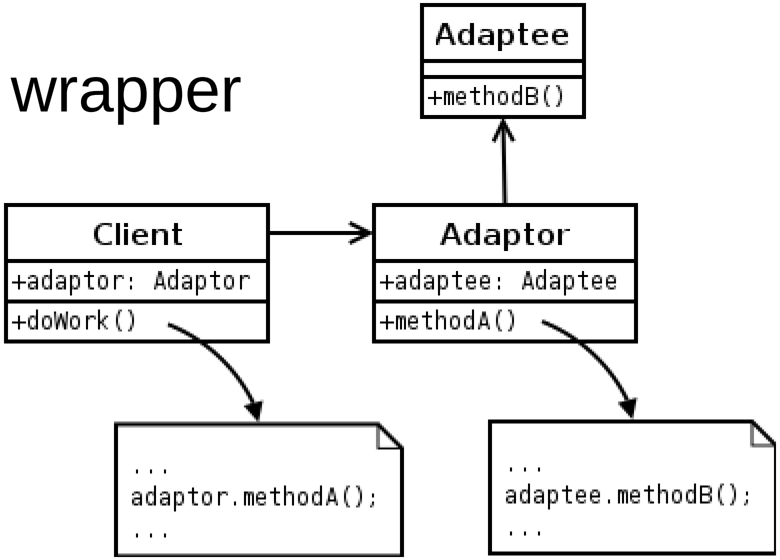
Examples: layered architecture, wrapper


Presentation

Business Logic

Data Access

Data



```
...
adaptor.methodA();
...
```

```
...
adaptee.methodB();
...
```

Reduce coupling: Refactor

Refactor is a tactic undertaken when two modules are affected by the same change because they are duplicates of each other.

- Common responsibilities (and the code that implements them) are “factor out” of the modules.
- By co-locating common responsibilities, the architect can reduce coupling.

Reduce coupling: Abstract common services

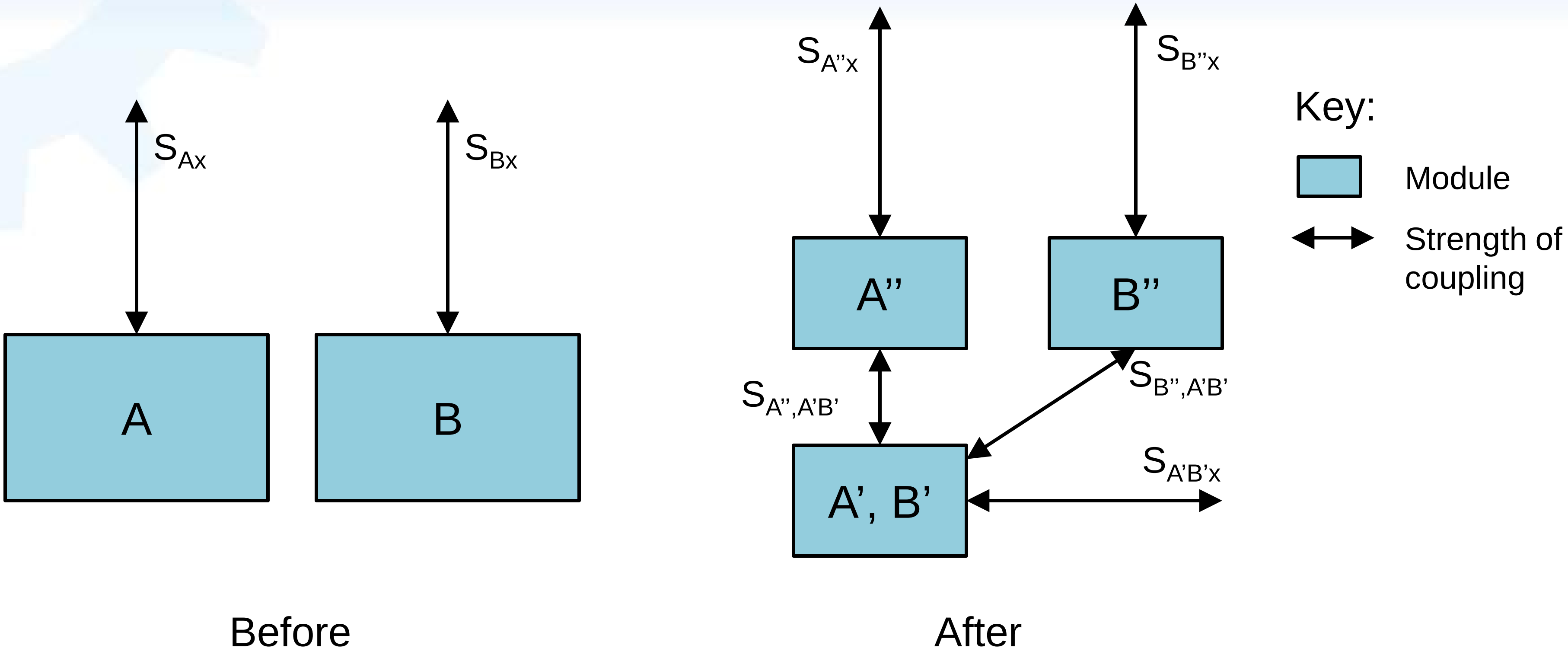
In the case where two modules provide similar services, it may be cost-effective to implement the services just once in a more general (abstract) form.

Any modification to the common service would then need to occur just in one place,

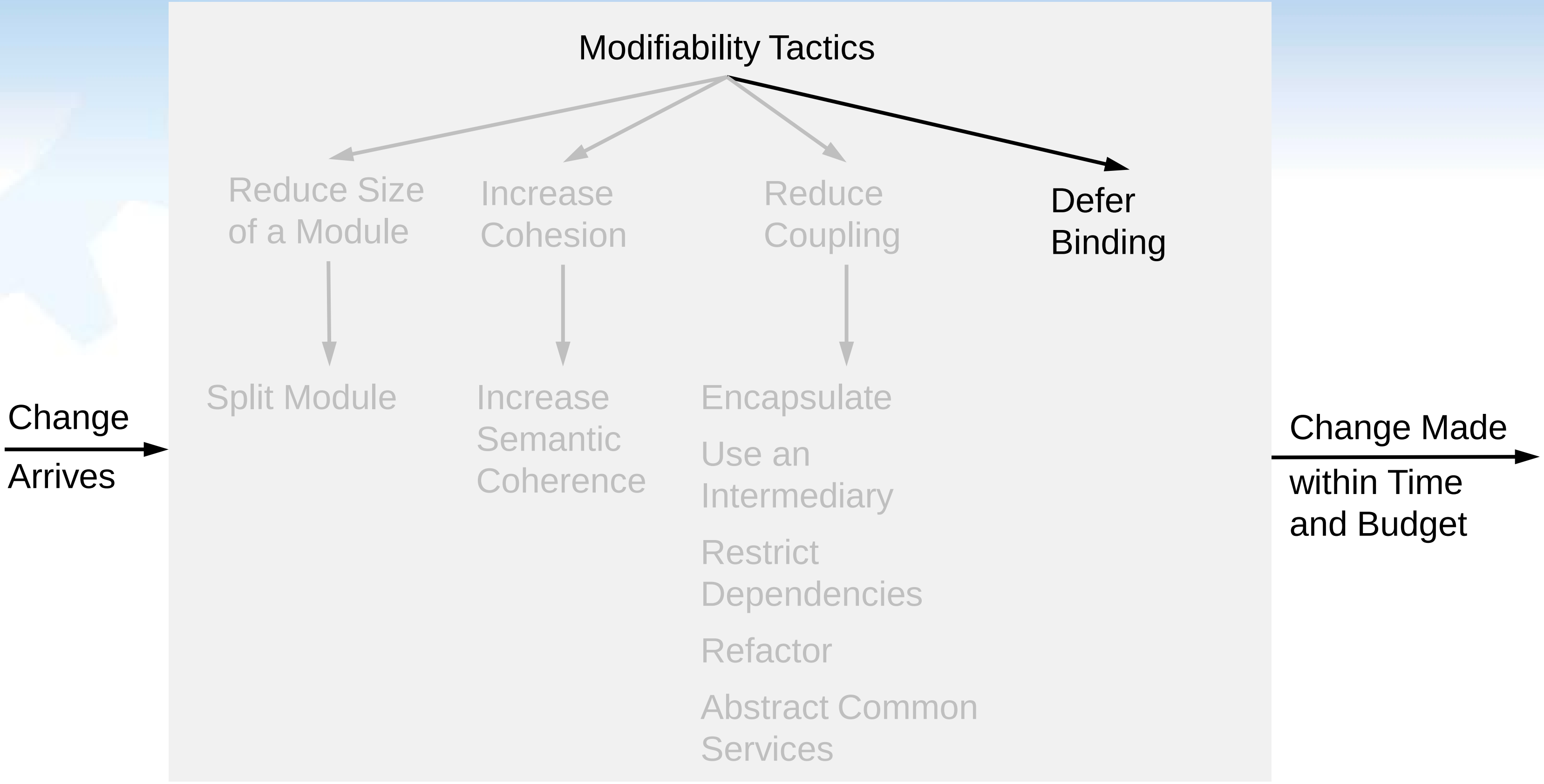
- reducing modification cost.

Reduce coupling: Abstract common services

A'' and B'' are unaffected portion of module A and B when a modification occur.



Tactics for modifiability



Defer binding

Letting computers handle a change as much as possible will reduce the cost of making that change.

The defer bindings are organized based on the system's life cycle:

- Compile time:
  - Component replacement (e.g. in a build script or makefile)
  - Compile-time parameterization
  - Aspect-Oriented programming
- Deployment time:
  - configuration-time binding

Defer binding

Letting computers handle a change as much as possible will reduce the cost of making that change.

The defer bindings are organized based on the system’s life cycle:

- Runtime:
  - Runtime registration
  - Dynamic lookup (e.g. for services)
  - Plug-ins
  - Publish-subscribe
  - Shared repositories
  - Polymorphism

Summary

Modifiability deals with change and the cost in time or money of making a change.

- To which extent this modification affects other functions or quality attributes.

Tactics to reduce the cost of making a change include

- making modules smaller,
- increasing cohesion, and
- reducing coupling.
- Deferring binding will also reduce the cost of making a change.

Discussion questions

1. In a certain metropolitan subway system, the ticket machines accept cash but do not give change. There is a separate machine that dispenses change but does not sell tickets. In an average station there are six or eight ticket machines for every change machine. What modifiability tactics do you see at work in this arrangement? What can you say about availability?
2. The abstract common services tactic is intended to reduce coupling, but it also might reduce cohesion. Discuss.

Discussion questions

3. A wrapper is a common aid to modifiability. A wrapper for a component is the only element allowed to use that component; every other piece of software uses the component's services by going through the wrapper. The wrapper transforms the data or control information for the component it wraps. For example, a component may expect input using English measures but find itself in a system in which all of the other components produce metric measures. A wrapper could be employed to translate. What modifiability tactics does a wrapper embody?

The End