Design of Software Systems

Fall 2017, BSc Software Engineering

Software Architecture: Quality and Tactics

Assist. Prof. Dr. Ronald Jabangwe

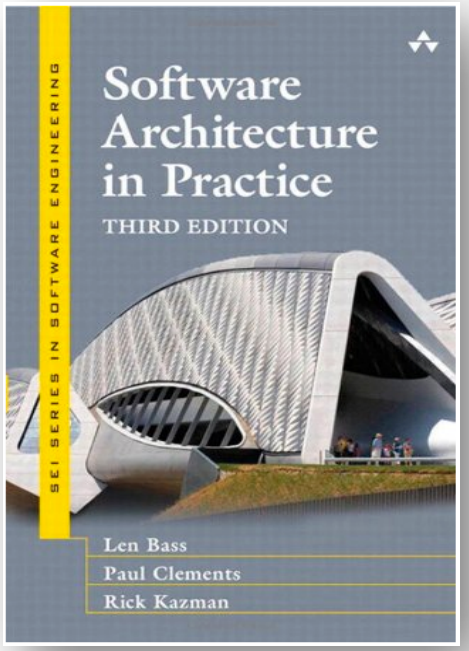
Agenda

- Recap:
  - Software architecture (mostly a recap)

- Quality and Tactics
- Modifiability Tactics
- Avaliability Tactics

What is software architecture?

"The software architecture of a system is the **set of structures** needed to reason about the system, which comprise software elements, relations among them, and properties of both." (Bass et al., 2013)



Requirements…

- **Functionality**:
- Is what the software does.
- The ability of a software to do work (i.e., provide functions) that it is intended to do
- Functional suitability is the “degree to which a product or system provides functions that meet stated and implied needs when used under specified conditions” (ISO/IEC FDIS 25010)

- **Non-functional**:
- How well the software does it.
- Sometimes referred to as quality attributes
- “a software requirement that describes not what the software will do but how the software will do it.” ISO/IEC/IEEE 24765:2010
- E.g., availability, security, modifiability, usability, etc.
- They determine the the architecture

Requirement…

- **Functionality**:
- "When a user presses X the phone number shall appear in the dialog box".

- **Nonfunctional/Quality**
- Captures: how quickly the phone number appears on the screen.
- "The phone number shall appear within (0.5 seconds) after the button X is pressed."
  - Nonfunctional/Quality attribute is Performance

How is Architecture Influenced?

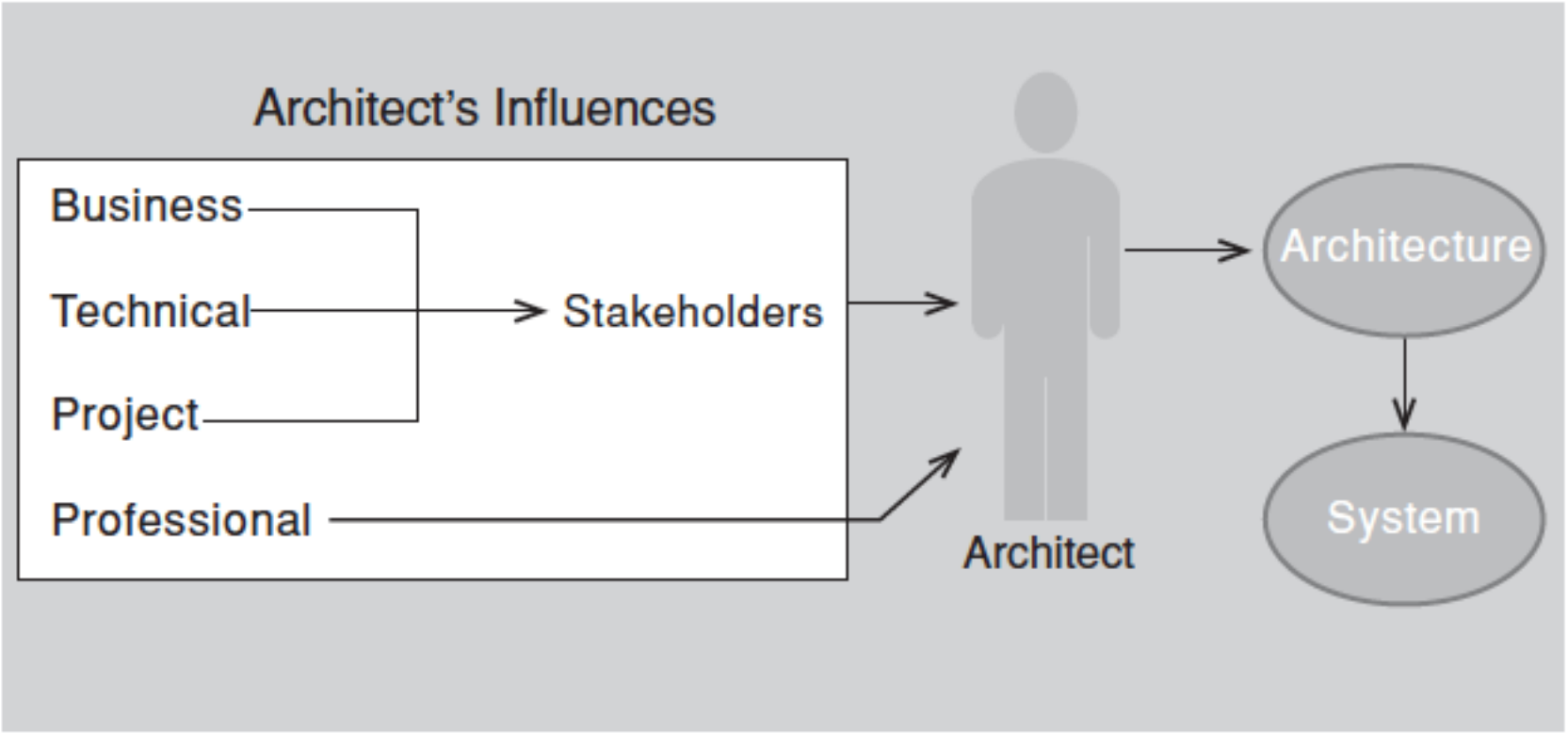


FIGURE 3.4 Influences on the architect

Definition: Software Architecture

A more pragmatic point of view:

Software architecture as a discipline is a smart combination of proven concepts, best practices, and tools of a computer scientist/engineer:

- **Abstraction**: emphasize the key problem and abstract from details
  → leads to the concept: Interface
- **Divide and Conquer**: break down a complex problem to many smaller, less complex but better manageable (sub-)problems
  → leads to: Component(s)
- **Reuse**: awareness regarding a component's embodiment and its later use dramatically increases reusability
  → don't re-invent the wheel (use and improve stuff that's already there, don't make mistakes twice…)

IMPORTANCE OF SOFTWARE ARCHITECTURE

Design of Software Systems

Why Architecture Design? Its importance is…

1. An architecture will **inhibit or enable a system's driving quality** attributes. For example:
   - **Performance**: You must manage the use of shared resources
   - **Modifiability**: minimize ripple effect of changes
   - **Security**: Manage and protect access which information;
     - you may also need to introduce specialized elements (such as an authorization mechanism).
   - **Reusability**: Restrict inter-element coupling, so that when you extract an element, it does not come out with too many attachments to its current environment.

Why Architecture Design? Its importance is… (Other reasons)

1. The analysis of an architecture **enables early prediction of a system's qualities**.

2. The architecture is a carrier of the **earliest and hence most fundamental, hardest-to-change design decisions**.

3. The architecture **dictates the structure of an organization**, or vice versa.

4. An architecture is the key artifact that allows the architect and project manager to **reason about cost and schedule**.

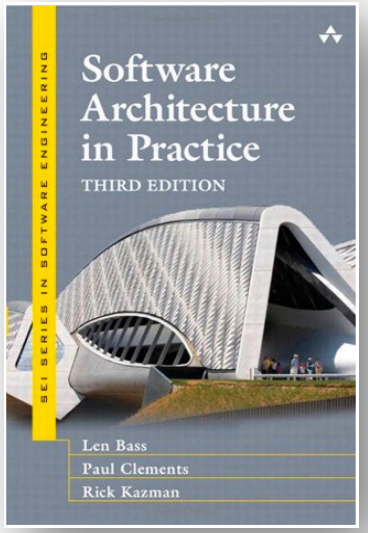
5. **Communication and Training**.

Recommended Reading…

Reflective Reading and Preparation

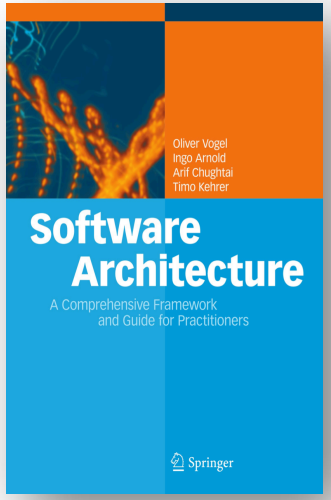
Read Bass et al. book: On importance of software architecture, and requirements in software architecture for reflective reading in Bass et al.

- Ch1-3



Other sources: on requirements in software architecture for reflective reading in Vogel et al.

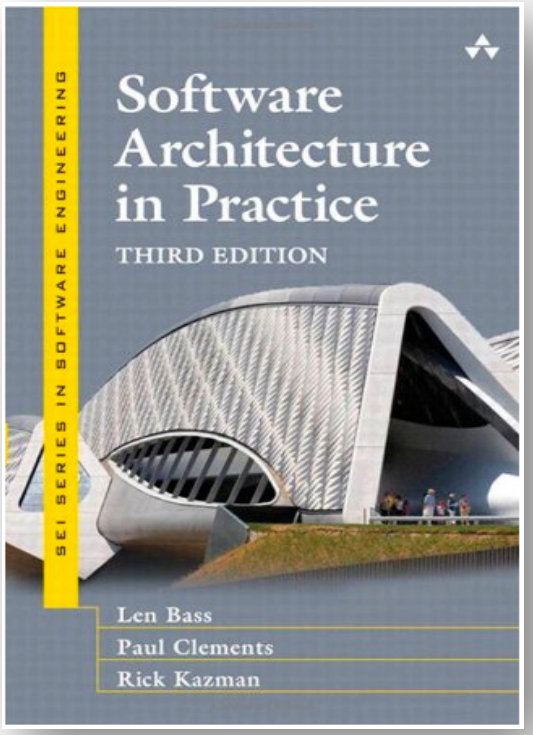
- Chapter 5



QUALITY AND TACTICS IN DESIGN

Design of Software Systems

Content is based on Bass et al.



Some important descriptions…

- **Pattern** is a commonly known solution for a recurring problem

- **Architecture design pattern**:
  - is the design at a high-level of abstraction
  - what components will you have in each layer? How many servers? Data collection and storage? Cloud-storage system?
  - E.g., Multi-tier architecture

- **Software design patterns**
  - are low level code and implementation specific.
  - This is where you define the classes, its methods, etc. and design relation between classes
  - Implement software design patterns,
    - eg., Factory class– i.e., instantiating an object at runtime without knowing the class of the object that needs instantiating

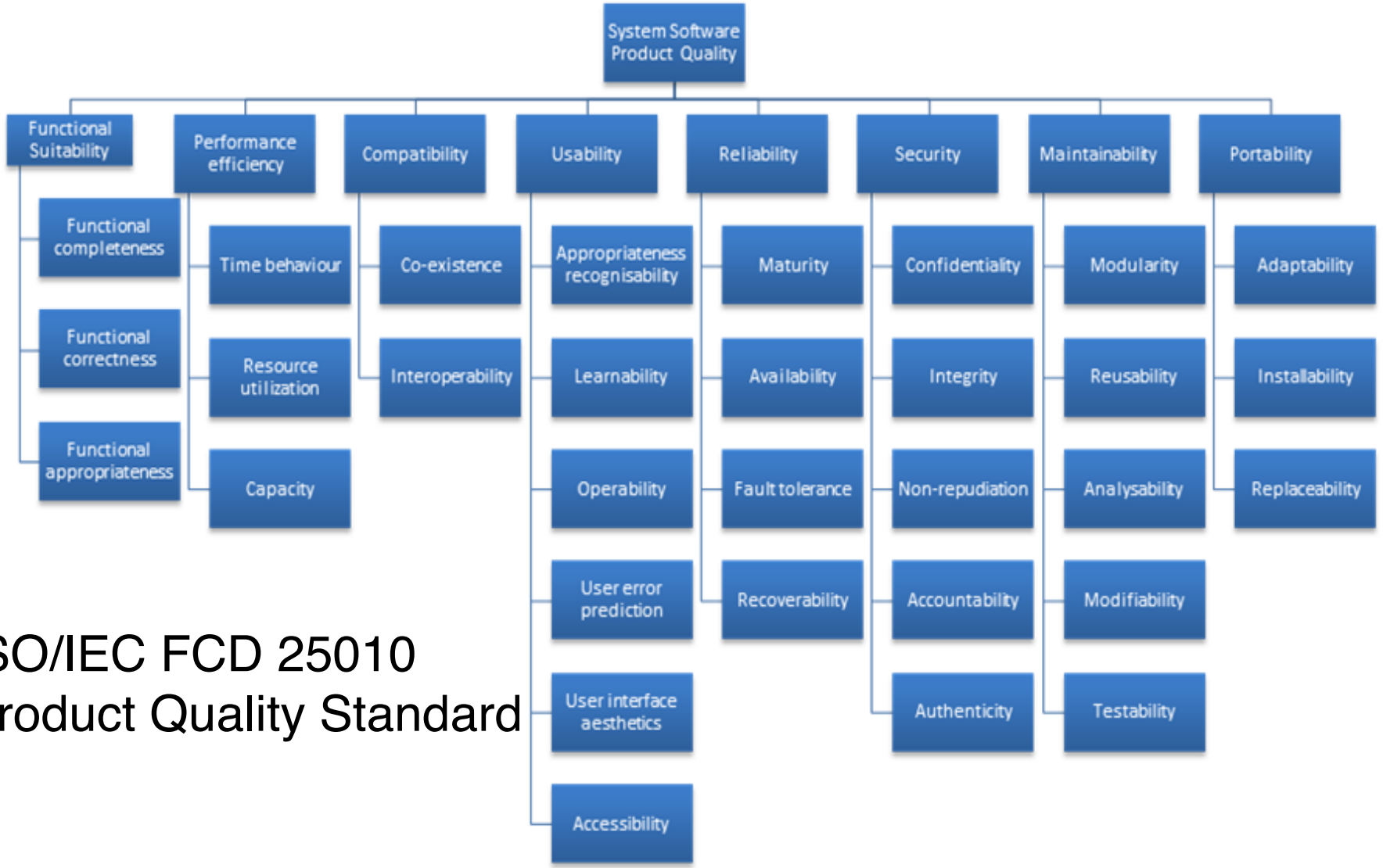
Software Quality

- "… degree to which a software product **satisfies stated and implied needs** when used under specified conditions."

ISO/IEC FDIS 25010

** it replaced the previous quality standard: ISO/IEC 9126

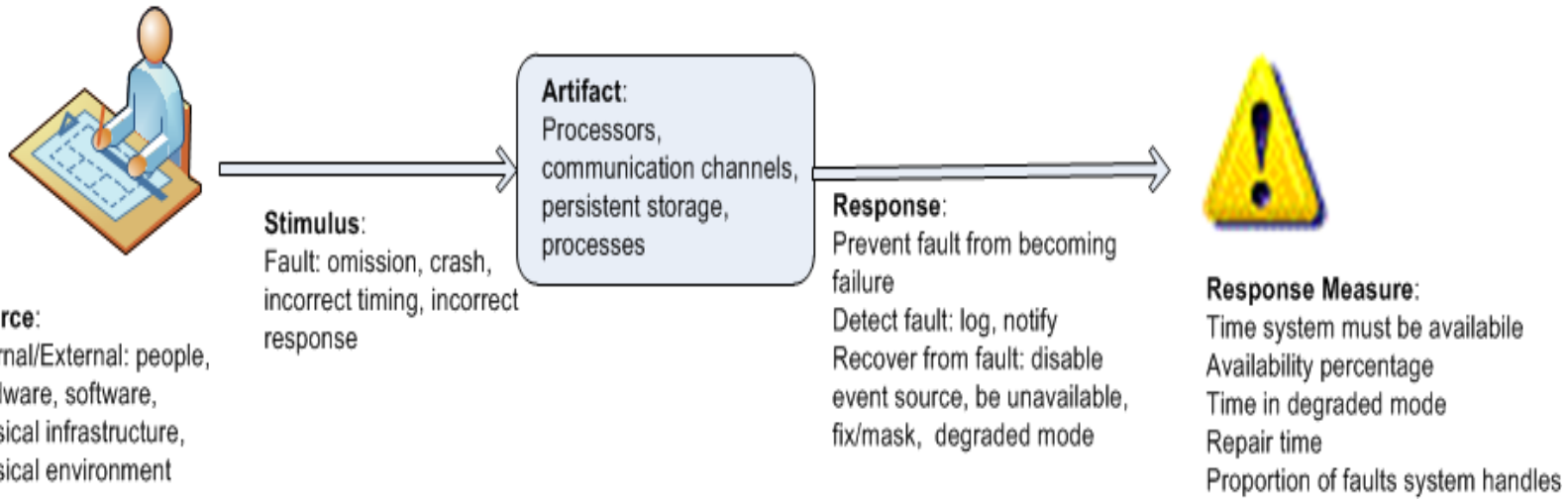
Standard Lists of Quality Attributes



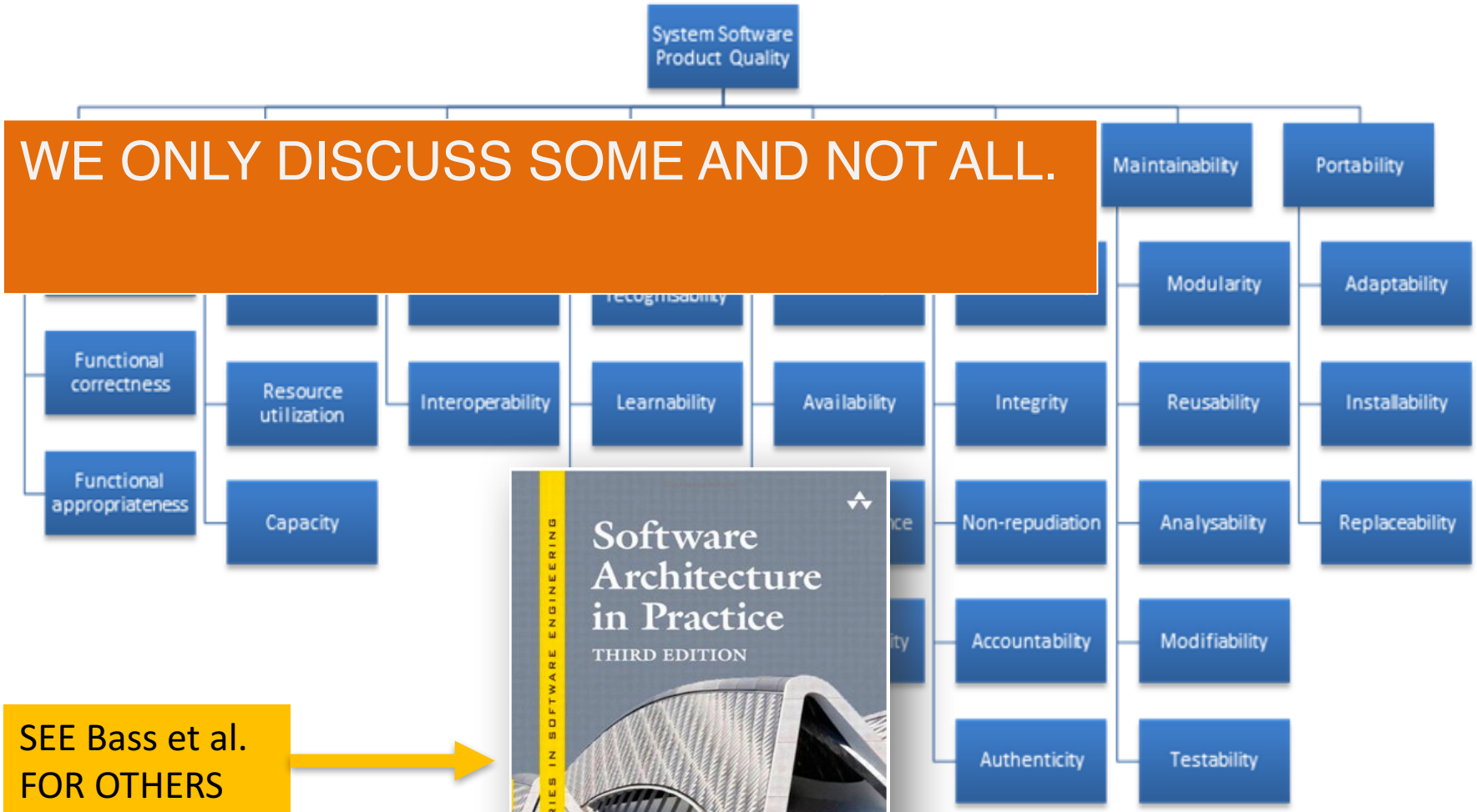
ISO/IEC FCD 25010
Product Quality Standard

How to Achieve Quality Attributes?

- Through Architectural Tactics - a collection of primitive design techniques that an architect can use to achieve a quality attribute response.
- We call these architectural design primitives *tactics*.
- Patterns package tactics to achieve certain quality attributes



Standard Lists of Quality Attributes



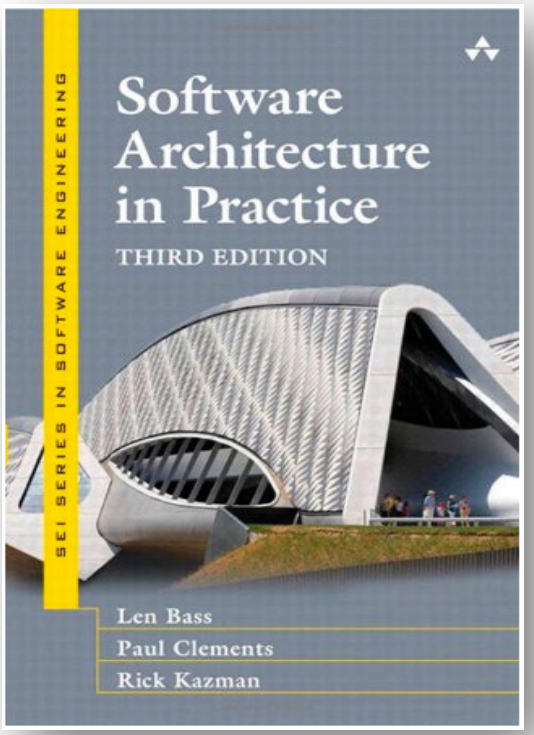
WE ONLY DISCUSS SOME AND NOT ALL.

SEE Bass et al.
FOR OTHERS

MODIFIABILITY TACTICS

Design of Software Systems

Content is based on Bass et al.



What is Modifiability?

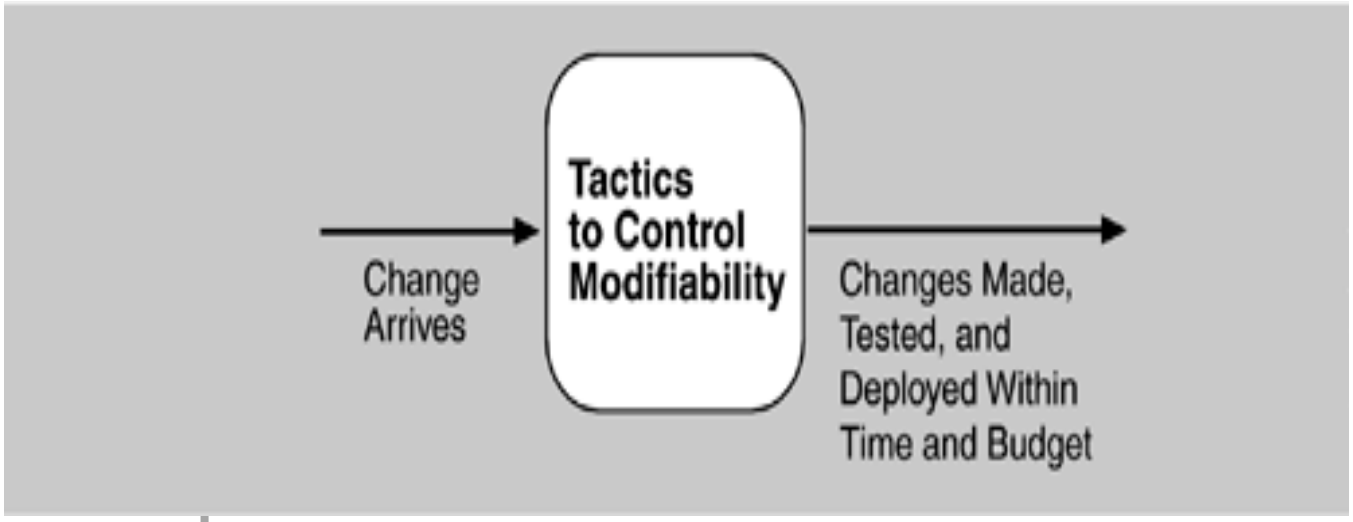
- Modifiability is about change and our interest in it is in the cost and risk of making changes.
- To plan for modifiability, an architect has to consider three questions:
  - What can change?
  - What is the likelihood of the change?
  - When is the change made and who makes it?

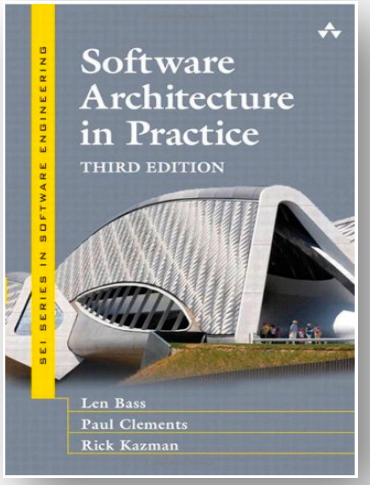
- Cost in terms of:
  - number, size, complexity of affected artifacts
  - Effort and calendar time
  - Extent to which this modification affects other functions or quality attributes
  - Few defects introduced
  - Ripple-effect of changes

Goal of Modifiability Tactics

- Goals of the Tactics to control modifiability:
  - Controlling the complexity of making changes,
  - Control time and cost needed to make changes.

- Example 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.





Modifiability Tactics (only some)

- **Reduce size of a module**
  - Split module
    - Example avoid God classes

- **Increase cohesion**
  - Increase Semantic Coherence: If the responsibilities A and B in a module do not serve the same purpose, they should be placed in different modules.

- **Reduce coupling**
  - Restrict Dependencies: restricts the modules which a given module interacts with or depends on.

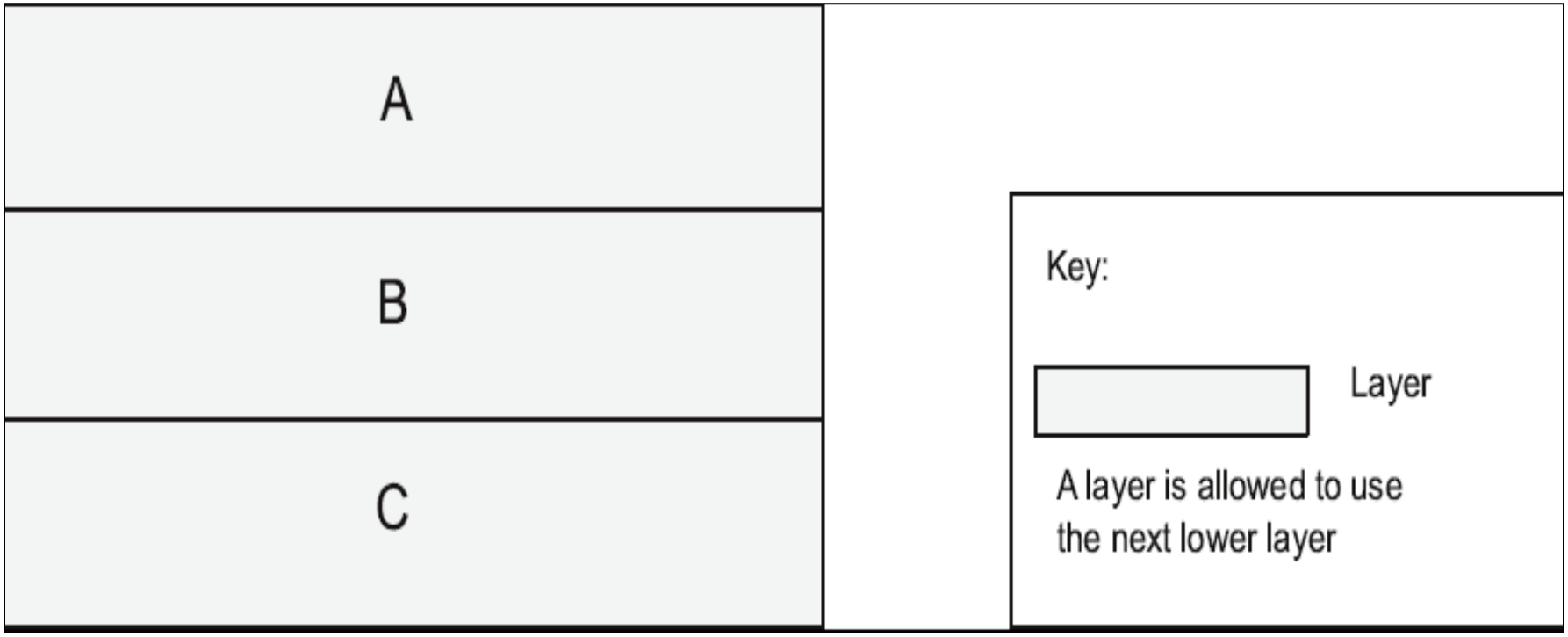
Summary

- Modifiability deals with change and the cost in time or money of making a change, including the extent to which this modification affects other functions or quality attributes.

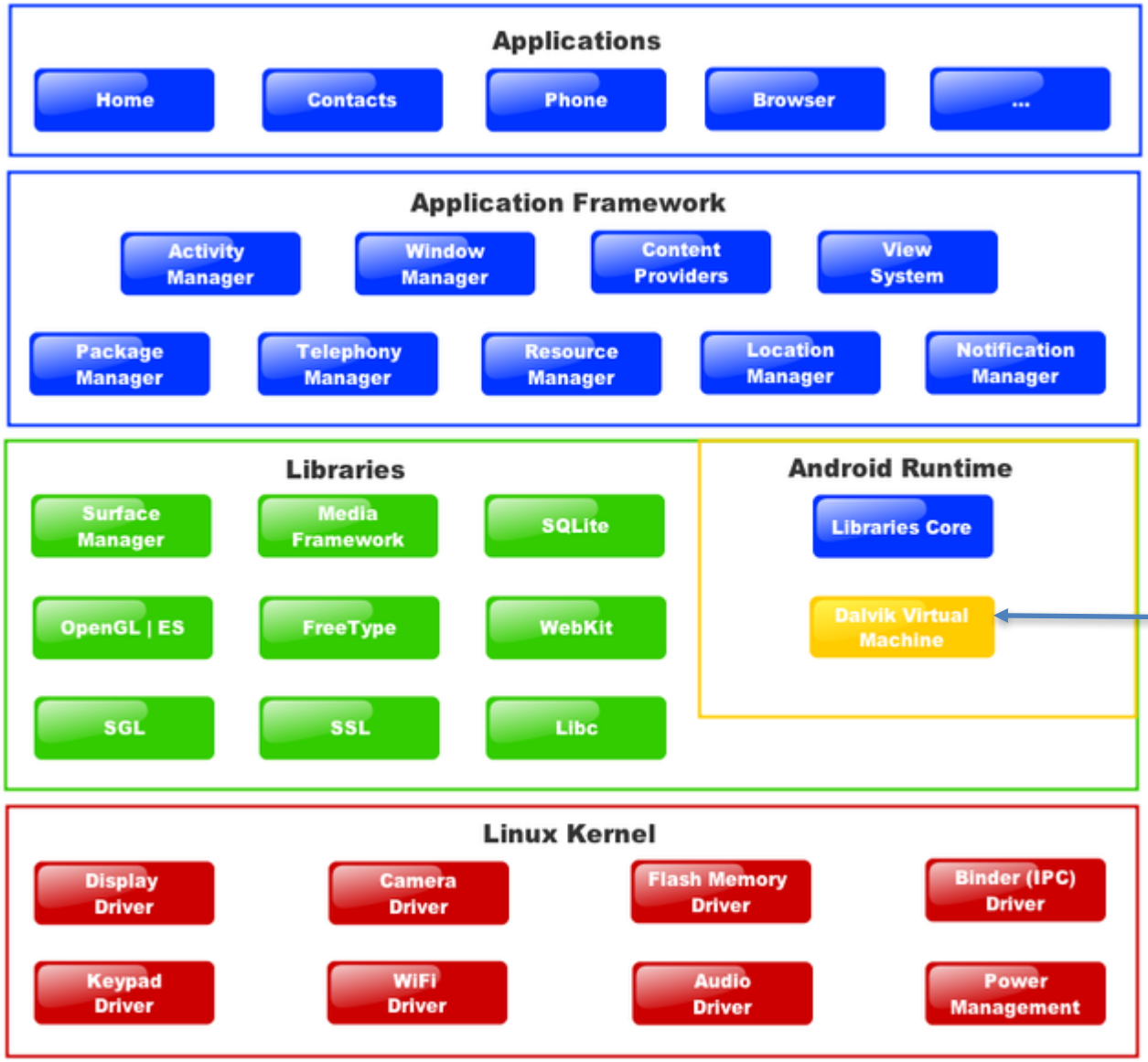
- Tactics to reduce the cost of making a change include
  - making modules smaller,
  - increasing cohesion, and
  - reducing coupling.

Summary

- Patterns package various tactics
- Lets look at an example… Layered pattern



FOR ANDROID…. OFTEN SEE THE FOLLOWING



High level Abstraction mult-Layer software stack

Slightly outdated, e.g., Replaced by Android Runtime (ART) from Android 5.0

Layered Pattern Example

- Group similar functionality and separation of other functions
  - Expectation is to increase modifiability and in turn maintainability
  - Then have a platform-specific layer to abstract the details of the underlying layer
    - The rest of the system then accesses the underlying layer through these abstractions

- Tactics:
  - *Localization Changes* tactics to increase cohesion
  - *Prevent Ripple Effects* tactics to reduce coupling
  - …. Etc.

Layered Pattern Example

- Group similar functionality and separate of other functions
  - Expectation is to increase modifiability and in turn maintanability
  - Then have a platform-specific layer to abstract the details of the underlying layer
    - The rest of the system then accesses the underlying layer through these abstractions

- Tactics:
  - *Localization Changes* tactics to increase cohesion
  - *Prevent Ripple Effects* tactics to reduce coupling
  - …. Etc.
  - What if we add an *intermediary tactic* (when we break dependency between A and B by using an intemediary, e.g., shared-data repository)?

Layered Pattern Example

- Group similar functionality and separate of other functions
  - Expectation is to increase modifiability and in turn maintanability
  - Then have a platform-specific layer to abstract the details of the underlying layer
    - The rest of the system then accesses the underlying layer through these abstractions

- Tactics:
  - Localization changes tactics to increase cohesion through
  - Prevent ripple effects tactics to increase coupling through
  - …. Etc.
  - What if we add an intermediary tactic (when Breaking dependency between A and B by using an intemediary, e.g., shared-data repository)?

Layered Pattern Example

- Group similar functionality and separate of other functions
  - Expectation is to increase modifiability and in turn maintanability
  - Then have a platform-specific layer to abstract the details of the underlying layer
    - The rest of the system then accesses the underlying layer through these abstractions

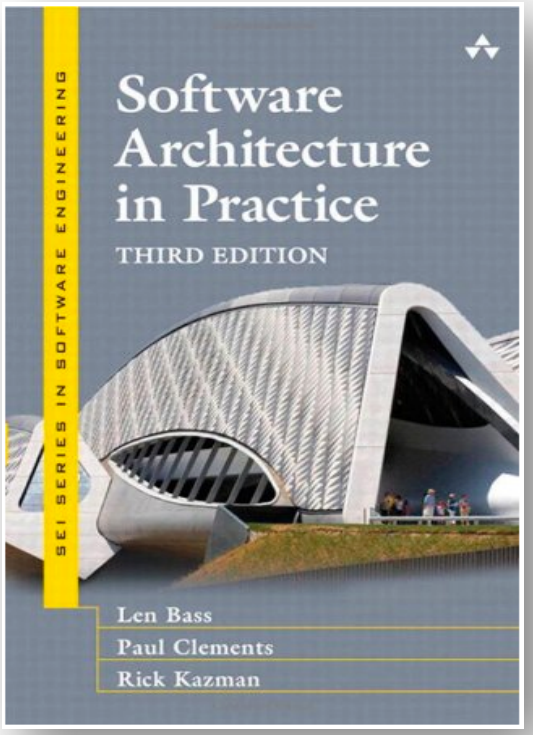
- Tactics:
  - Localization changes tactics to increase cohesion through
  - Prevent ripple effects tactics to increase coupling through
  - …. Etc.
  - What if we add an intermediary tactic (when Breaking dependency between A and B by using an intemediary, e.g., shared-data repository)?
    - Improves modifiability by reducing coupling between A and B
    - But Adds a third component, which adds to effort, cost and maintenance work needed

Example from “Modifiability Tactics”, Technical Report by Felix Bachmann, Len Bass and Robert Nord, from Software Engineering Institute

AVALIABILITY TACTICS

Design of Software Systems

Content is based on Bass et al.

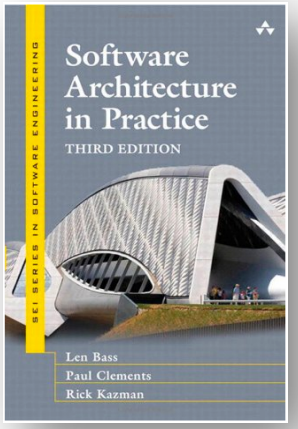


What is Availability?

- **“ degree to which a system, product or component is operational and accessible when required for use”** - ISO/IEC FDIS 25010:2010

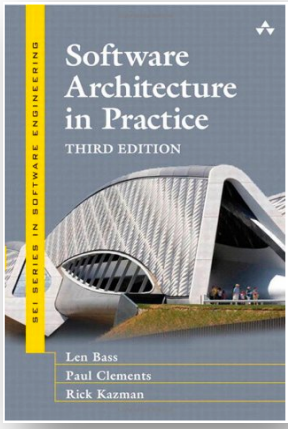
- This is a broad perspective and encompasses what is normally called reliability.
  - by adding the notion of recovery (repair).

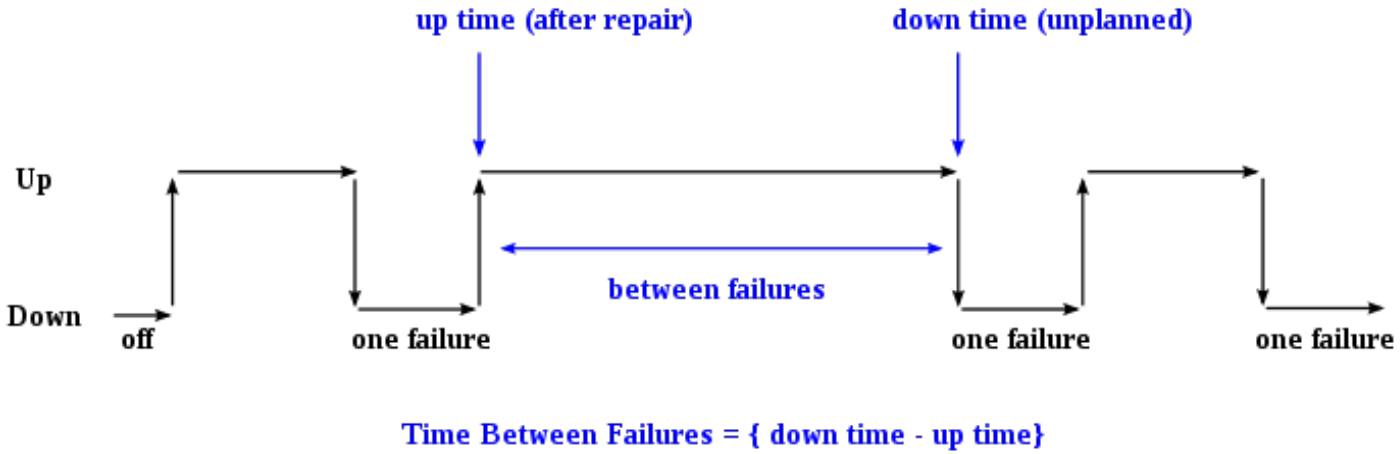
- Fundamentally, availability is about minimizing service outage time by mitigating faults.



Common Availability Calculation

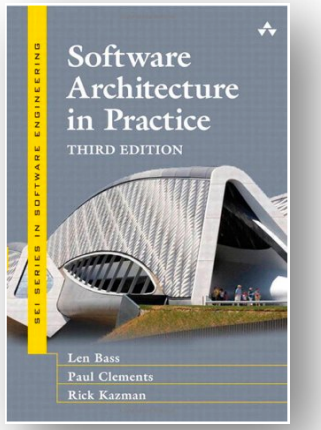
- **MTBF – mean time between failures**
  - Associated with repairable components
- **MTTR – mean time to repair/recovery**
  - Associated with repairable components
- They are also used for reliability
- They are often used for hardware
- In the context of software: What will make your software fail; how likely is it to occur; and there will be time to repair it

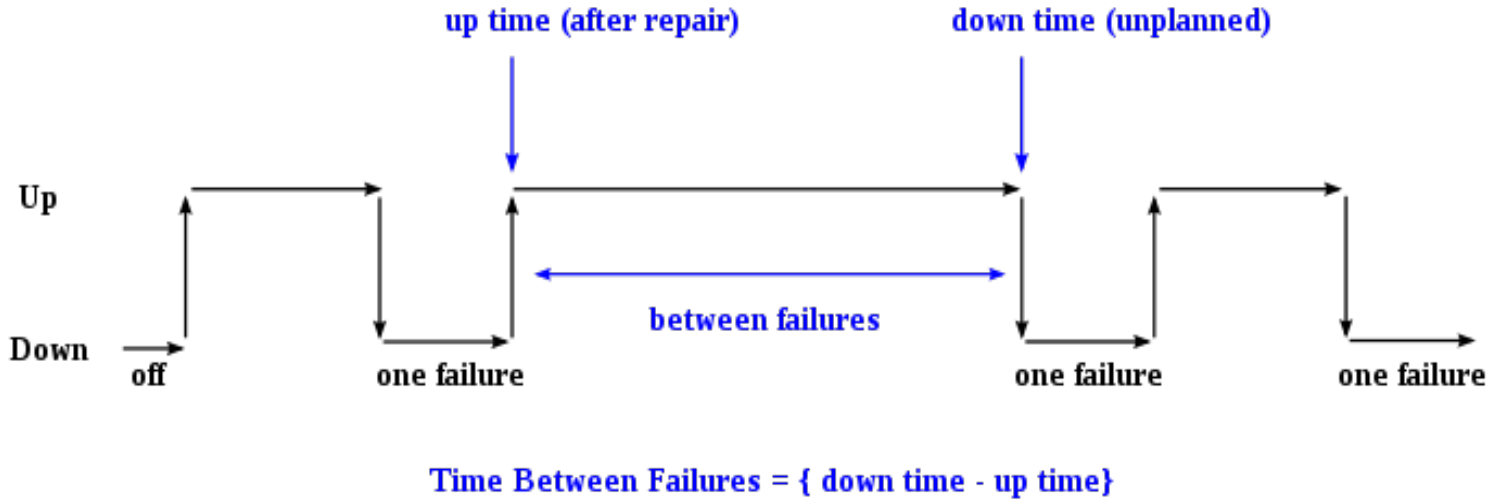




Common Availability Calculation

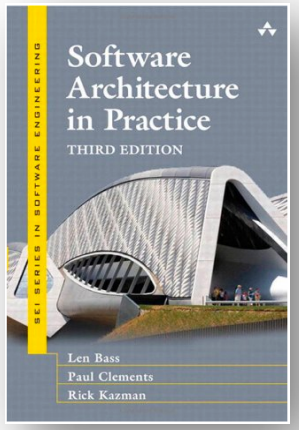
- Mean time between start of down time to start of up time or
- MTBF/(MTBF+MTTR)
  - Very simplistic. What contextual information is not considered?

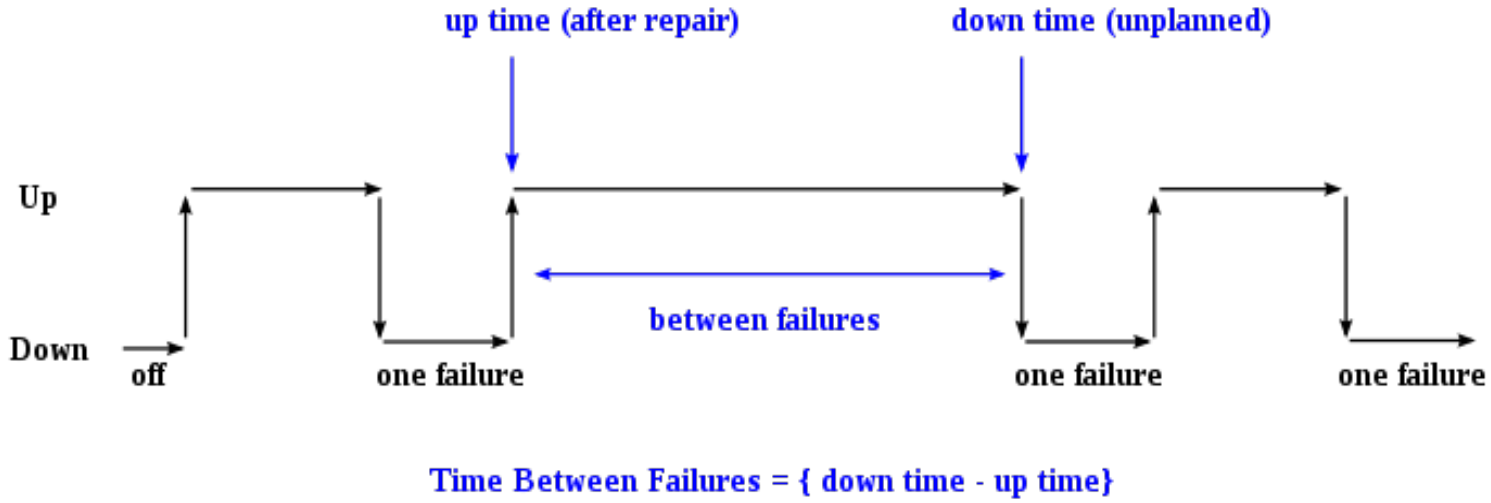




Common Availability Calculation

- Mean time between start of down time to start of up time or
- MTBF/(MTBF+MTTR)
  - Very simplistic. What contextual information is not considered?
  - Different complexity across components, (e.g., different fixing times and failure occurrences), etc.





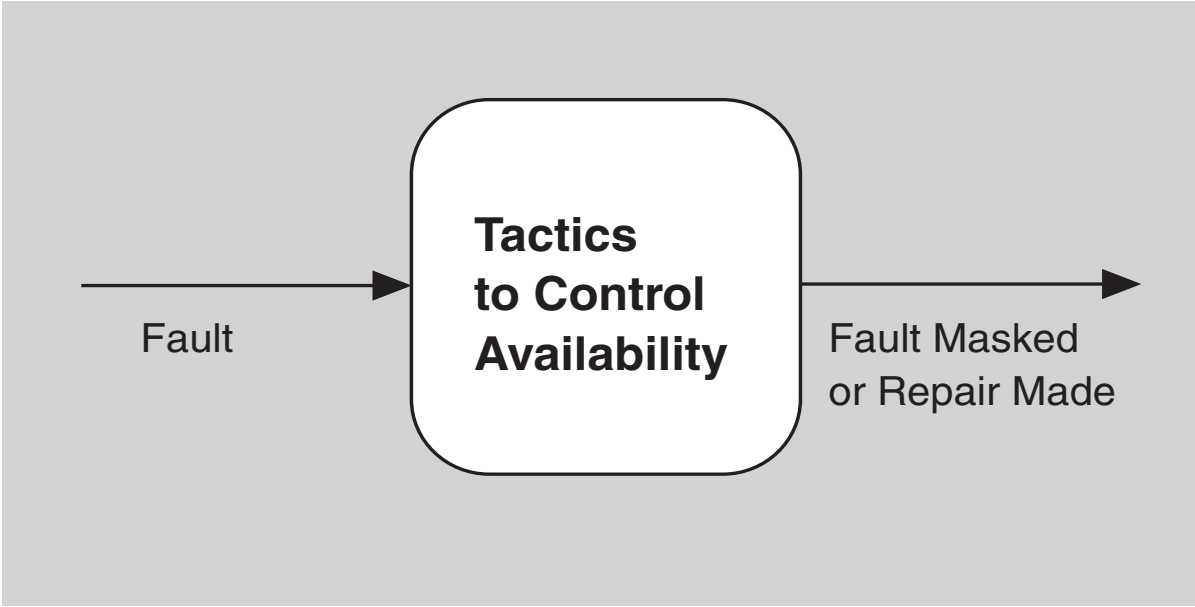
Goal of Availability Tactics

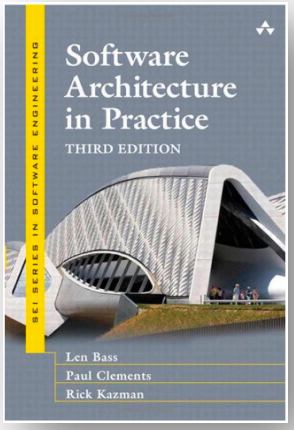
- **A failure** occurs when the system no longer delivers a service consistent with its specification
  - this failure is observable by the system's actors.
- **A fault** (or combination of faults) has the potential to cause a failure.

- In essence:
  - Availability tactics enable a system to endure faults so that services remain compliant with their specifications.
  - The tactics keep faults from becoming failures or at least bound the effects of the fault and make repair possible.

Categorization of Availability Tactics

- Detect Faults
- Recover from Faults
- Prevent Faults





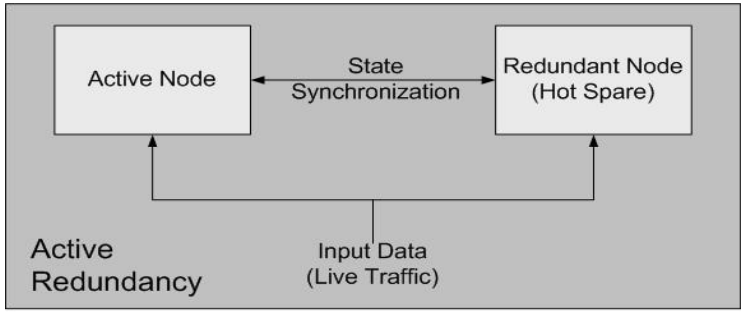
Detect Faults

- **Monitor:**
  - a component used to monitor the state of health of other parts of the system, e.g., detect hung processes, etc.
  - A system monitor can detect failure or congestion in the network or other shared resources, such as from a denial-of-service attack.

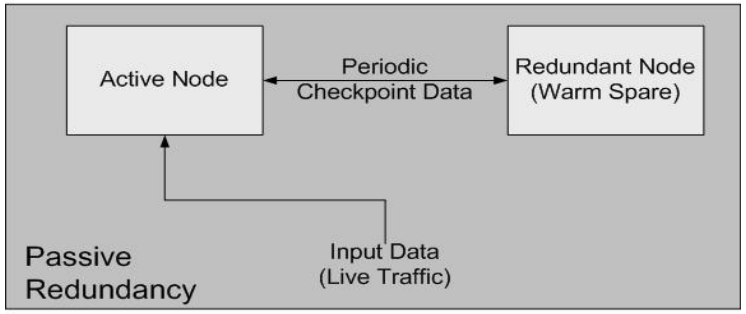
- **Voting:**
  - to check that replicated components are producing the same results.
  - *Triple Modular Redundancy* is a common realization of Voting
    - 3 processing units receive the same inputs – then forward outputs to a vote logic component to detect any inconsistencies

- **Exception Detection:**
  - detection of a system condition that alters the normal flow of execution, e.g. system exception, parameter fence, timeout.

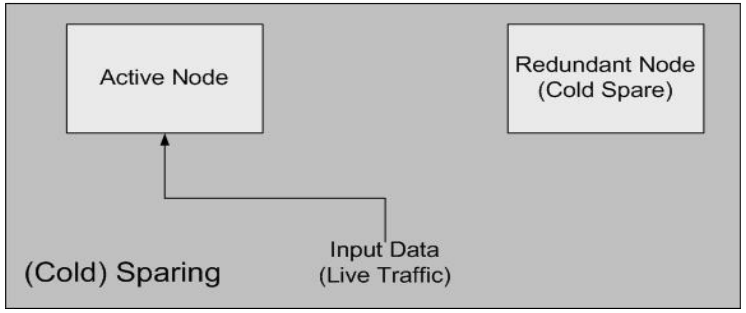
Recover from Faults (Preparation & Repair)



- **Active Redundancy** (hot spare):
  - allowing redundant spare(s) to maintain synchronous state with the active node(s).



- **Passive Redundancy** (warm spare):
  - active node(s) provide the redundant spare(s) with periodic state updates.



- **Spare (cold spare):**
  - redundant spares 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.
  - Better suited when having high-reliability requirements as opposed to having high-availability requirements

Figure from "Realizing and Refining Architectural Tactics: Availability", Technical Report by James Scott, Boeing Corporation Rick Kazman, from Software Engineering Institute

Prevent Faults

- **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 abstract data types, etc.

Overall:

- Every design decision has both:
  - Benefits
  - Side effects

- When selecting tactics review the context:
  - Development context (technology, architectural style/pattern, development teams, etc.)
  - Intended use of the software
  - Prioritization of quality attributes (e.g., modifiability vs performance)
  - Make tradeoffs

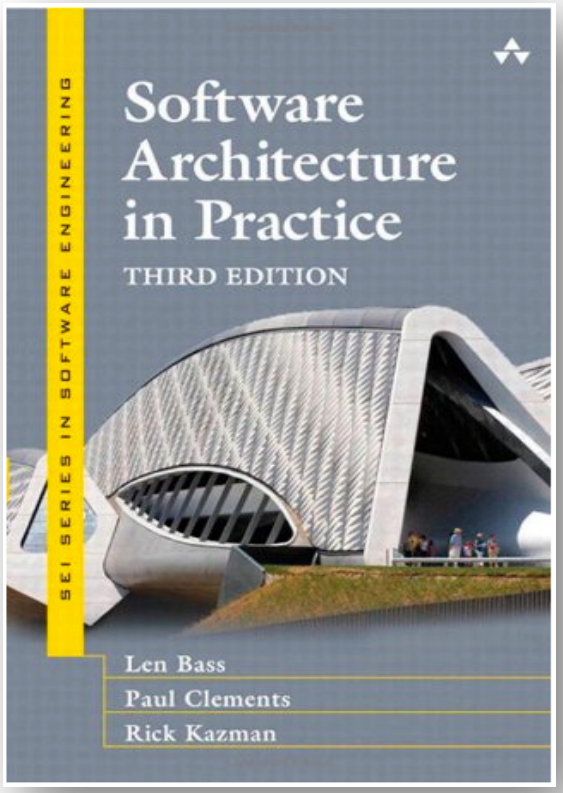
Recommended Reading…

Reflective Reading and Preparation

Read Bass et al. book:

- Ch4, Ch5 (availability), and Ch7 (modifiability),

- FYI - More tactics including checklists for quality attributes covered today or others, e.g., performance. See Part 2 of the book.



Course

- Today's Lecture:
  - Recap: Software architecture and Requirements
  - Quality and Tactics in design

- Today's Lab:
  - Lab: Intro to C#

- Next class and Lab:
  - Lab: Assignment 2 due in class (UML and C#)
  - Lecture: No Lecture. Work on the assignment which will be due at 1600.