Software Achitecture

`Review`

```
Eser Manuscript
```

Summary

Chapter 1

- SA (Software Architecture)
  - The SA of (a program or computing system) 理解
    - **structure or structures** of the system (*comprises*)
      - **elements**
      - **relationships** among them
      - **properties** of both
  - Architecture = Components + Connectors + Constrains
  - **Structures** 掌握
    - **a set of elements** held together by **a relation**
    - SftSystem are composed of many structures
      - Module
        - partition system into **implementation units**
        - are assigned sepecific responsibilities
      - Component and Connector
        - **runtime entity**, runtime properties
        - the way elements interact with each other
        - help answer questions like:
          - What are the major shared data stored?
          - What parts of the system can run in parellel
      - Allocation

        - mapping from software structures to environments
        - answer questions like:
            - which processor does each element execute on?
    - Examples -- some useful **module structures**
        - Uses structure
            - units related by the **uses relation**--use another unit or being used...
        - Layer structure
            - modules here called layers
            - which is a "virtual machine" provides a **cohesive set of services** through a managed interface.
        - Class structure
            - Unit: **Class**
            - Relation: inherit from, instance of or composed of
        - Data model
            - describe static infomation strutures in terms of entities && relationships
    - Examples -- some useful **C&C strutures**
        - Service struture
        - Concurrency struture
    - Examples -- some useful **Allocation strutures**
        - Deployment structure
        - Implementation structure
        - Work assignment structure
- Views 了解
    - **Representations** of structures
        - A module view is a representation of a module structure, documented and used by stakeholders
- the ABC (Architecture Business Cycle)
    - **cycle of influences** (Environment -> Architecture -> Environment)
        - Business/organizational **Environment** --(affects)--> **Architectural design decision**
            - **Architectural design** --(part of)--> every step of the **development process** --(affect)--> **organization's bottom line**

        - **SA** --(changes)--> **Business Environment**
- **Stakeholders** 了解
    - Any one who **has a stake in the sucesss of the system**
    - Have **different concerns**
    - Identify and engage them to solicit their needs and expectations.
    - Example
        - Developing origanization's management Stakeholder
            - low cost...
        - Marketing Stakeholder
            - neat feature, short time to market...
        - End user Stakeholder
            - Behavior, performance, security
        - Maintenence Organization Stakeholder
            - Modifibility
        - Customer Stakeholder
            - Low cost, timely delivery...
- Good "Rules of Thumb" for an architecture 了解
    - 一位架构师Architect或其领导小组
    - A掌握功能需求，并有划分优先级的质量属性列表
    - 架构文档完备
    - 让stakeholders积极参与设计方案评审
    - 分析架构人身，得出可应用的量化度化指标
    - 架构设计应有助于增量式实现 lend itself to incremental implementation
    - 允许资源争用，但应给出资源争用的解决方案

Chapter 2

- Importance of SA *理解*
    - Influence **quality attributes**
    - **Reason about and manage changes**
    - **Predict qualities**
    - Enhancing **Communication among stakeholders** 涉众交流手段
    - Capture the **earliest Design Decisions**(most fundamental, hardest to change)

- Define **Constraints on Implementation** (Budget control)
- Dictates the **Structure of an Organization**
- Enabling Evolutionary **Prototyping** (be analyzed&prototyped as a skeletal sys)
- Improving **Cost&Schedule Estimates**
- **Transferable, Reusable Model** 可传递、可重用的系统抽象
- Focuse attention on the **assembly of components**, rather than on their **creation**
- **Restricting Design Vocabulary/Reducing design&sys complexity** (Restricts to a relatively small number of choices of elements&their interactions
- **Foundation(Basci) for training** a new team

Chapter 3

- 理解
  - Technical Context
    - Factors include
      - **set of attributes** that Arch helps to achieve
      - **current technical environment**(Standard industry practices, techniques, library...)
  - Project Life-cycle Context
    - Standard approaches for developing software sys -- imposing a **discipline**
    - **Four dominant software development processes**:
      - WaterFall
      - Iterative
      - Agile
      - Model-driven development
  - Business Context
    - Arch&sys serve some business purpose
    - Architect need to understand (Who customers are? What goals are?)
  - Professional Context
    - For an Architect
      - Many duties beyond technical skills
        - diplomatic, negotiation, communicate ideas clearly

        - up-to-date knowledge
    - Influence between Architecture and context

Chapter 4

- Sys requirements
    - Functional requirements - not determine Architecture
    - Quality attribute requirements
    - Constraints -- design decision already been made
- Quality Attribute(QA) 理解
    - QA scenarios consist of:
        - Stimulus
        - Stimulus source
        - Response
        - Response measure
        - Environment
        - Artifact
    - Specifying QA Requirements 掌握
        - Source of Stimulus
            - people, hardware, physical infrastrue...
        - Stimulus
            - Fault: omission, crash, incorrect timing...
        - Environment
        - Artifact -- Some artifact is stimulated.
            - Whole sys/ parts of sys
            - like Process, Storage, Processor...
        - Response
            - Detect fault, Recover from fault...
        - Response measure
- Tactics 理解
    - a collection of **primitive design techniques**
    - not invent but capture what architects do in practice
    - Why?

        - Design pattern are complex. If exist --> augment to achieve QA
        - If not exits, Tactics help to construct a design fragment
        - Make design more systematic
- Guiding Quality Design Decisions 掌握
    - Architecture design -- A systematic approach to making design decisions
    - **Seven Design Decision** an architect needs to make:
        - Allocation of responsibilites
        - Coordination model
            - identify elements that must coordinate and that are prohibited from coordinating
            - properties of coordination like timeliness, currency...
            - choosing a communication mechanisms that can realize them
        - Data model
            - Major data abstractions, operation, properties
            - Metadata
            - Organization (How data to be kept)
        - Management of resources
            - what resources
            - which sys elements manage each resources
            - how shared and what strategies employed when there is a contention
            - impact of saturation
        - Mapping among architectural elements
            - Modules <---> Runtime elements
            - assignment of (runtime elements -> processors)
            - assignment of (items in data model -> data stores)
        - Binding time decisions
            - The decisions in the other categories have an associated binding time decison.
            - keep runtime updating
        - Choice of technology

- 理解：概念

- 了解：公式和 一般场景
- 掌握：战术Tatics以及设计清单Design for

Chapter 5

- Availability 可用性 理解
  - A property of software that it is there and ready to carry out its task when you need it to be
  - = Reliability + Recovery (Repair)
  - Fundamentally, iit is about minimizing service outage time by mitigating faults 通过减轻错误最小化服务中断时间
- Sample Scenario 了解
  - The heartbeat monitor determines that the server is nonresponsive during normal operations. The system informs the operator and continues to operate with no downtime
- Tatics 掌握
  - Goals
    - Fault -potential-> Failure
    - Enable a sys to endure faults
    - Keep Faults from becoming Failures at least bound the effects of the fault and make repair possible
  - Availability Tactics!!!
    - **Detect Faults**
      - Ping/echo -- request/response message exchanged between nodes --> chech reachability & round-trip delay
      - Monitor -- a Component used to monitor the state of health of other parts of the sys
      - Heartbeat -- a periodic message exchanged between sys monitor and processed
      - Timestamp -- detect incorrect sequences of events
      - Sanity Checking -- checks the validity or reasonableness of a component's operations or output
      - Condition Monitoring -- checking conditions in a process or device or validating assumtions

        - Voting -- check replicated component are producing the same results
        - Exception Detection
        - Self-test
    - **Recover from Faults**
        - Active Redundancy(hot spare) -- readiness to overcome an otherwise significant start-up delay
            - All nodes in a protection group receive&process identical inputs in parallel
            - few seconds to recover
        - Passive Redundancy(warm spare)
            - Only the active members of the group process input traffic
            - Primary members update other nodes
            - few minutes to recover
        - Spare(cold spare)
            - Redundant spares out of service
            - several hours
        - Exception Handling
        - Rollback, Upgrade, Retry, Ignore Faulty Behavior,
        - Degradation -- maintains the most critical sys functions, dropping others
        - Reconfiguration, shadow...
    - **Prevent Faults**
        - Removal From Service -- Temporarily disable component that may cause fault
        - Transactions
        - Predictive Model
            - Monitor the state of health of process
            - Take corrective action when conditions are detected abnormal.
        - Exception Prevention
        - Increase Competence Set -- handle more cases--faults--as part of normal operation
- **Design for Availability!!!**
    - Adopt suitable architectural pattern & tactics
    - Establish an availability model/formula

        - Design a dedicated availability view if necessary
        - Detemine the sys responsibilities that need to be highly available.
            - Ensure that additional responbilities have been allocated to detect a fault
        - Ensure that there are responsibilities to:
            - **log** the fault
            - **notify** appropriate entities(People or Sys)
            - **disable source** of events causing the fault
            - be temporarily unavailable
            - **fix or mask** the fault/failure
            - operate in a degraded mode

Chapter 6

- Interoperability
- Definition
    - about the degreee to which two or more sys can usefully exchange meaningful information
- Sample Scenario
    - Our vehicle information sys sends our current location to the traffic monitoring sys. The traffic monitoring sys combines our location with other information, overlays this information on a Google Map, and broadcasts it.
- Tactics
    - must
        - know each other
        - exchange infos in a semantically meaningful fashion
    - Discover Service
    - Manage Interfaces
- Design for Interoperability
    - Adopt suitable architectural patterns and tactics
    - **Design** a dedicated interoperability **view** if necessary
    - Determine **which** responsibilities will **need** to interoperate with other sys
    - Responsibilities have been allocated to
        - accept the request

        - exchange information
        - reject the request
        - notify appropriate entities (people | sys)
        - log the request (for audit trail)
    - **coordination mechanisms** --meet--> **critical quality attribute requirements**
        - Volume of traffic on the network, timeliness.....
    - Determine (syntax&semantics)<--major data abstractions that may be exchanged
    - Major data abstraction --consistent with--> data from interoperating sys.

Chapter 7

- Modifiability
    - is about change, and our interest in it centers on the cost and risk of making changes
    - What can change? likelihood? When & Who?
- Samle 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
- Tactics
    - Goal --> controling the complexity, as well as the time and cost
    - Reduce Size of a Module
        - split module
    - Increase Cohesion (SRP)
    - Reduce Coupling (OCP)
    - Defer Binding -- the latter, the better
- Design Modifiability
    - Adopt suitable architectural patterns and tactics
    - Design a dedicated modifiability view if necessary
    - Build a modifiability model if possible by introducing some measures
    - Determine sth likely to change

        - which (categories of) changes are likely to occur
        - which devices, protocols, and communication paths
        - elements for which modifiability is a concern -> reduce coupling

---

Chapter 8

- Performance
    - is about time and the software sys's ability to meet timing requirements
- Sample Scenario
    - Users initiate transactions under normal operations. The system processes the transactions with an average latency of two seconds.
- Tactics Goal
    - to generate a response to an event arriving at the system within some time-based constraint
- Tactics
    - Control Resource Demand //computational Resource like processors
        - Manage Samplilng Rate
            - reduce sampling frequency at which a stream of data is captured -- loss of fidelity
        - Limit Event Response
            - Process events only up to a set maximum rate
        - Prioritize Events
        - Reduce Overhead -- removing intermediaries to decrease the resources consumed in processing an event stream and improves latency
        - Bound Execution Times -- limited time to respond to an event
        - Increse Resource Efficiency -- improve algorithms used in critical area
    - Manage Resources
        - Increase Resource -- faster processors, additional memory
        - Increase Concurrency
        - Maintain Multiple Copies of Computations -- reduce the contention where computations took place on a single server

        - Maintain Multiple Copies of Data
        - Bound Queue Size
        - Schedule Resources
- Design for Performance
    - Adopt suitable architectural patterns and tactics
    - Establish an performance model/formula
    - Design a dedicated performance view if necessary
    - Determine the sys's responsibilities that will involve heavy loading, have time-critical response requirements, are heavily used, or impact other portions of sys that do. For those responsibilities, identify(即对high-loaded的部分做到以下）
        - Process requirements of each & determine whether they may cause bottlenecks
        - Additional responsibilities to recognize & process requests appropriately including
        - Responsibilities crossing process or processor boundaries
        - Responsibilities to manage the threads
        - Responsibilities for scheduling shared resources or managing performance-related queues, buffers, and caches
    - Determine the elements of the sys that must coordinate with each, choose communication and coordination mechanisms that 对于合作的部分，选这样的交流协作机制
        - supports any introduced concurrency, event priorization, or scheduling strategy
        - ensures that the required performance response can be delivered
        - have the appropiate properties of the communication mechanisms like stateful/stateless, synchronous/asynchronous, throughput, latency...
    - Detemine those portions of the data model that ...(heavily loaded...). For those data abstraction, determine
        - maintaining multiple copies of key data benefit performance?
        - partitioning data beneficial?
        - whether (reducing the processing requirements)/(adding resources) to (reduce bottlenecks) for the (creation, initialization,persistence, manipulation, translation, or destruction) of (the enumerated data

abstractions) if possible
    - Determine which resources in your sys are critical for performance --> to be monitored & managed under normal/overloaded sys operation
        - sys elements that need to be aware of, and managed; time & other performance-critical resources
        - process/thread models
        - prioritizaiton of res and access to resources
        - scheduling and locking strategies
        - deploying additional resources on demand to meet increased loads

---

Chapter 9

- Security
    - is a measure of the sys's ability to protect data and information from unauthorized access while still access to people and sys that are authorized
    - Characteristics (CIA):
        - Confidentiality（机密性）/ˌkɔnfi,denʃi'æliti/
        - Integrity /ɪn'tɛgrəti/ //integrality /ˌinti'grælәti/完整性
        - Availability
- Sample Scenario
    - A disgruntled employee from a remote location attempts to modify the pay rate table during normal operations. The sys maintains an audit trail and the correct data is restore within a day
- Tactics
    - Detect Attacks
        - Detect Instrusion ///ɪn'truʒn/侵入侵扰
        - Detect Service Denial
        - Verify Message Integrity
        - Detect Message Delay
    - Resist Attacks
        - Limit Exposure
        - Encrypt Data

        - Separate Entities
            - differnt servers -> different networks
            - use of virtual matchines, "air gap"
        - Change Default Settings
    - React to Attacks
        - Revoke Access -- limit access to sensitive res even for normally legitimate users&uses when an attack is suspected
        - Lock Computer -- limit access when repeated failed attempts to access it
        - Inform Actors -- notify operators...
    - Recover from Attacks
        - In addition to the Acailabilty tactics for recovery of failed resources there is **Audit**
            - That is, keep a record of user&sys actions&their effects, to help trace the actions of and to identify an attacker
- Design for Security
    - Adopt suitable architectural patterns and tactics
    - Establish an security model/formula
    - Determine which system responsibilitieds need to be secure. Ensure that additional responsibilities have been allocated to:
        - Identify, authenticate&authorize the actor
        - grant|deny access to data or services
        - record attempts to access or modify data or services
        - encrypt data
        - recognize DOS attack
        - recover from an attck
        - verify checksums and hash values

Chapter 10

- Testability
    - refers to the ease with which software can be made to demonstrate its faults through testing

    - more specifically, it refers to the probability, assuming that the software has at least one fault, that it will fail on its next test execution
    - if a fault is present in a sys, then we want it to fail during testing as quickly as possible
    - Requirement
        - able to control each component's inputs and observe output (possibly its internal state)
- Sample Scenario
    - The unit tester completes a code unit during development and performs a test sequence whose results are ccaptured and that gives 85% path(即code coverage 被测试代码覆盖率) coverage within 3 hours of testing
- Tactics Goals
    - Allow for easier testing when an increment of software development has completed
- Tactics
    - Control&Observe sys state
        - Built-in Monitors -- monitor states&record
        - Specialized Interfaces -- to control or capture variable values
        - Record/Playback
        - Localize State Storage -- start anything in an arbitrary state for a test (store state in a single place)
        - Abstract Data Sources -- Abstracting the interfaces makes substitute test data more easily
        - Sandbox -- be unconstrained by the worry about having to undo the consequences of the experiment
        - Executable Assertions
    - Limit Complexity
        - Limit Strutural Complexity
        - Limit Non-determinism //非决定论
- Design for Testability
    - Adopt suitable architectural patterns and tactics
    - Establish an testability model/formula

    - Determine which sys responsibilities are most critical and hence need to be most thoroughly tested
    - Additional sys responsibilities
        - execute test suite and capture results capture(log) the activity that resulted in a fault
        - control and observe relevent sys state for testing
    - Test-driven development
    - Isolateability

Chapter 11

- Usability 易用性
    - is concerned with how easy it is for the user to accomplish a desired task and the kind of user support the sys provides
    - What is?
        - Learning sys features
        - Using a sys efficiently
        - Minimizeing the impact of errors
        - Adapting the sys to user needs
        - Increasing confidence(信任) and satisfaction
- Sample Scenario
    - The user downloads a new app and is using it productively after two minutes of experimentation
- Tactics
    - Separate User Interface
        - Isolate interface from implementation
        - Late binding
    - Support User Initiative
        - Cancel, Pause/Resume, Undo, Aggregate, Default settings
    - Support sys Initiative
        - Maintain Task Model
        - Maintain User Model

        - Maintain sys Model
- Design for Usability
    - Adopt suitable architecture patterns and tactics
    - Adopte various usability prototypes
    - Ensure that additional sys responsibilitiees have been allocated, as needed, to assist the user in
        - learning how to use
        - efficiently achieving the task at hand
        - adapting and configuring the sys
        - recovering from user and sys errors

Chapter 12

- Goal
    - 了解
        - 其它软件质量属性如可变性、可移植性、开发可分布性、伸缩性、可部署性、移动性、可监控性、生命财产安全性。其它类别的质量属性如架构质量属性、商业属性、系统质量属性。ISO/IEC FCD 25010 产品质量标准
    - 理解
        - 如何处理未知的质量属性
- Other Important Quality Attributes
    - Variability
        - Special form of modifiability
        - refers to the ability of a sys and its supporting artifacts to **support the production of a set of variants that differ from each other in a preplaneed fashion**.
    - Portability
        - Special form of modifiability
        - refers to the ease with which software that built to run on one platform can be changed to run on a different platform
    - Development Distributability
        - designing the software to support distributed software development
    - Scalability

        - Horizontal scalability(Scaling out) -- add more res to logical units (another server..)
        - Vertical scalability(Scaling up) -- add more res to a physical unit (more memory to a computer)
    - Deployability
    - Mobility移动性
        - deals with the problems of movement and affordances of a platform(e.g. size, type of IO, battery life...)
    - Monitorability
    - Safety生命财产安全
        - ability to avoid entering states that cause or lead to damage, injury or loss of life, and to recover and limit damage.
        - concerns identical with those for availabilty(prevent, detect, recover...)
- Other Categories of QA
    - Conceptual Integrity
        - Same thing is done in the same way through the architecture
    - Marketability
        - the use of the system with respect to market competition.
- Dealing with "X-ability"
    - Model the quality attribute
    - Assemble a set of tactics for the quality attribute
    - Construct design checklists

Chapter 13

- Pattern
    - A context
    - A problem
    - A solution
        - A set of element types
        - A set of interaction mechanisms or connectors
        - A topological layout of the components

        - A set of semantic constraints covering topology, element behavior, and interaction mechanisms

Pattern Catalogue 掌握

- Module Pattern
    - Layer Pattern
        - Context: Separation of concerns
        - Problem: The software needs to be segmented in such a way that the model can be **developed and evolved separately** with listtle interaction amoong the parts, supporting portability, modifiability, and reuse
        - Solution: Divides the software into units called layers. Each layer is **a grouping of modules** that offers **a cohesive set of services**. The usage must be unidirectional
        - Overview: Defines **layers** and **a use relation**
        - Element: Layer
        - Relations: Allowed to use //allowed-to-use relations should not be circular
        - Constraints:
            - Every piece of software is allocated to exactly one layer
            - At least two layers
            - The allowed-to-use relations should not be circular(a lower layer cannot use a layer above)
    - OOP Pattern
- Component&Connector Pattern
    - Broker Pattern
        - Context: Distributed services interoperate with each other
        - Problem: How do we **structure distributed software** so that service users do not need to know **the nature and location** of service providers, making it easy to **dynamically change the bindings** between users and providers?
        - Solution: **Separates** users of services(**clients**) from providers of services(**servers**) by **inseting an intermediary**, called a broker
        - Overview: The **broker** mediates the communication between a number of **clients** and **servers**
        - Element:

        - Client(a requester of services)
        - Server(a provider of services)
        - Broker(an intermediary)
        - Client-side proxy
        - Server-side proxy
    - Constraints: C/S can only --attach to--> a broker
- Weakness: communication bottleneck; up-front complexity; security attacks; difficult to test
- MVC Pattern
    - Context: separate view from model
    - Problem: How can user interface functionality be kept separate from application functionality and yet still be responsive to user input, or to changes in the underlying application's data? And how can multiple views of the user intercace be created, maintained, and coordinated when the underlying application data changes?
    - Solution: The MVC pattern separates application functionality into three kinds of components
    - Element: M, V, C
    - Relations: The notifies relation connects instances of model, view and controller, notifying elements of relevant state changes
    - The model component should not interact directly with the view
- Pipe and Filter Pattern
    - Context: process data streams
    - Problem: Such systems need to be divided into **reusable**, **loosely coupled components** with **simple, generic interaction mechanisms**
    - Solution: The pattern of interaction in the pipe&filter pattern is characterized by successive transformations of streams of data
    - Overview: Data is transformed by filters connected by pipes
    - Element: Filter and Pipe
    - Relations: The attachment relation associates (**the output of filters**) with (**the input of pipes**) and vice versa
    - Constraint:
        - **filter output ports** --Pipe(connect)-- **filter input ports**

        - Connected filters must agree on the type of data being passed along the connecting pipe
    - Client-Server Pattern
        - Context: There are shared resources and services that large number of distributed clients wish to access
        - Problem: We want to improve scalability and availability by centralizing the control of these resources and services, while distributing the resources themselves across multiple physical servers
        - Solution: Clients interact by requesting services of servers, which provide a set of services
        - Elements: C/S
        - Overview: Clients initiate interactions with servers, invoking services as needed from those servers and waiting for the results of those requests
        - Relations: The attachment relation associates clients with servers
        - Constraints: Clients are connected to servers through request/reply connectors
    - Peer-to-Peer Pattern
        - Context: Distributed computational entities are considered equal
        - Problem: How can a set of "equal" distributed computational entities be connected to each other via a common protocal so that they can organize and share their services with high availability and scalability?
        - Solution: In the p2p pattern, components directly interact as peers. All peers are "equal" and no peer or group of peers can be critical for the health of the system
        - Overview: Computation is achieved by cooperating peers
        - Elements: Peer; Request/Reply connector
        - Relations: The relation associates peers with their connectors. Attachments may change at runtime
        - Constraints: Restrictions may be placed on the following:
            - The number of allowable attachments to any given peer
            - The number of hops used for searching for a peer
            - which peer know about which other peers
            - ...

- SOA Pattern (Service Oriented Architecture)
  - Context: A number of services interoperates without any detailed knowledge of their implementation
  - Problem: How can we support interoperability of distributed components running on different platforms and written in different implementation languages, provided by different organizations, and distributed across the Internet?
  - Overview: Computation is achieved by a set of cooperating services over a network
  - Element:
    - Components: Service providers/consumers (p/c)
    - ESB: an intermediary element -- route&transform messages
    - Orchestration server: coordinates the interactions between service p/c
    - Registry of services
    - Connectors
      - SOAP connector -- uses the SOAP protocal for synchronous communication between web services, typically over HTTP
      - REST connector -- relies on the basic request/reply operations of the HTTP protocol
      - Asynchronous messaging connector (p2p)
  - Relations:
    - Attachment of the different kinds of components available to the respective connectors
  - Weakness:
    - complex to build
    - performance overhead(middleware), performance bottlenecks, not performance guarantees
- Publish-Subcribe Pattern
  - Context: The precise **number** and **nature** of the data producers and consumers are not predetermined or fixed, nor is the data that they share
  - Problem: How can we create integration mechanisms that support the ability to transmit messages among the producers and consumers so they are unaware of each other's identity, or potentially even their existence?

        - Solution: In the publish-subscribe pattern, components interact via announced messages, or event
        - Overview: Components publish and subscribe to events. When an event is announced by a component, the connector infrastructure dispatches the event to all registered subscribers.
        - Element:
            - Publisher, Subscriber
            - The publish-subscribe connector
        - Relations:
            - The attachment relation associates components with the publish0subscribe connector by prescribing which components announce events and which components are registered to receive events
        - Constraints: Publish ports are attached to annouce roles and subscribe ports are attached to listen roles
    - Shared-Data Pattern
        - Context: Various computational components need to share and manipulate **large amounts of data**(shared data)
        - Problem: How can systems **store** and **manipulate persistent data** that is accessed by multiple independent components?
        - Solution: In the shared-data pattern, interaction is dominated by the **exchange of persistent data** between **multiple data accessors** and at least one **shared-data store**
        - Overview:
            - Communication between data accessors is mediated by a shared data store
        - Element:
            - Shared-data store
            - Data accessor component
            - Data reading and writing connector
        - Relations:
            - Attachment relation determines which data accessors are connected to which data stores
        - Constraints:

        - Data accessors interact only with the data store(s)
- Allocation Pattern
    - Map-Reduce Pattern
        - Context: need to quickly analyze enormous volumes of data
        - Problem: To efficiently perform a **distributed** and **parallel sort** of a large data set and provide a simple means for the programmer to specify the analysis to be done
        - Solution: The map-reduce pattern requires three parts:
            - A specialized infrastructure takes care of allocating the data as needed
            - A map which filters the data to retrieve items
            - A reduce which combines the results of the map
        - Overview: The map-reduce pattern provides a framework for analyzing a large distributed set of data. The map performs the extract and transform portions of the analysis and the reduce performs the loading of the results
        - Elements:
            - Map
            - Reduce
            - The infrastructure is the framework responsible for deploying map and reduce instances, shepherding the data between them, and detecting and recovering from failure
        - Relations:
            - Deploy on
            - Instantiate, monitor, and control is the relation between the infrastructure and the instances of map and reduce
        - Constraints:
            - The data to be analyzed must exist as a set of files
            - Map functions are stateless and do not communicate with each other
    - Multi-Tier Pattern
        - Context: To distribute a system's infrastructure into distinct subsets
        - Problem: How can we **split the system into** a number of computationally independent **execution structures**--groups of software and hardware--connected by some communications media?
        - Solution: The execution structures of many systems are organized as a set of

logical groupings of components. Each grouping is termed a tier
    - Overview:
        - The execution structures of sys are organized as a set of logical grouping of components
    - Element:
        - Tier
    - Relations:
        - is part of
        - communicates with
        - allocated to
    - A software component belongs to exactly one tier
- Pattern are maily determined by connectors
- Relationships Between Tactics and Patterns 理解
    - Pattern are built from tactics; if a pattern is a molecule/'mɑlɪkjul/, a tactic is an atom
    - MVC, for example utilizes the tactics:
        - Increase semantic coherence
        - Encapsulation
        - Use an intermediary
        - Use run time binding
    - Pattern solve a specific problem but are neutral or have weaknesses with respect to other qualities
    - Tactics and Interactions
        - Each tatics has pluses and minuses--side effects
        - use of tactics can help alleviate the minuses
        - but nothing is free
    - Tactics example -- all have pluses and minuses
        - Ping/Echo -- detecting fault
            - side effects: security, performance, modifiability...
        - "Increse Available Resources" -- address the performance side-effect
        - "Scheduling Policy" -- address the efficient use of resources
        - "User and Intermediary" -- address the addition of the scheduler to the sys

Dict

```
SOAP (originally Simple Object Access Protocol) is a protocol
specification for exchanging structured information in the implementation o
f web services in computer networks. Its purpose is to induce extensibility
, neutrality and independence. It uses XML Information Set for its message
format, and relies on application layer protocols, most often Hypertext Tra
nsfer Protocol (HTTP) or Simple Mail Transfer Protocol (SMTP), for message
negotiation and transmission.
```

Chapter 14

- 了解

Chapter 15

Chapter 16

理解
- ASR -- Architecturally Significant Requirement
- Architectures exist to build sys that satisfy req
- Not all req are created equal
- A requirement that will have a profound effect on the architecture

- How to find?
    - From Requirements Documnets(RD) -- usually not the case
        - not RD
        - RD pay more attention to functionality than quality attributes
        - Most of the RD dose not affect architecture
        - No architect just sits&wait until RD done
        - QA are often captured poorly -- like "The sys shall be modular..."
        - Much of what is useful to an architect is not in even the best RD
    - From Stakeholders(SH)

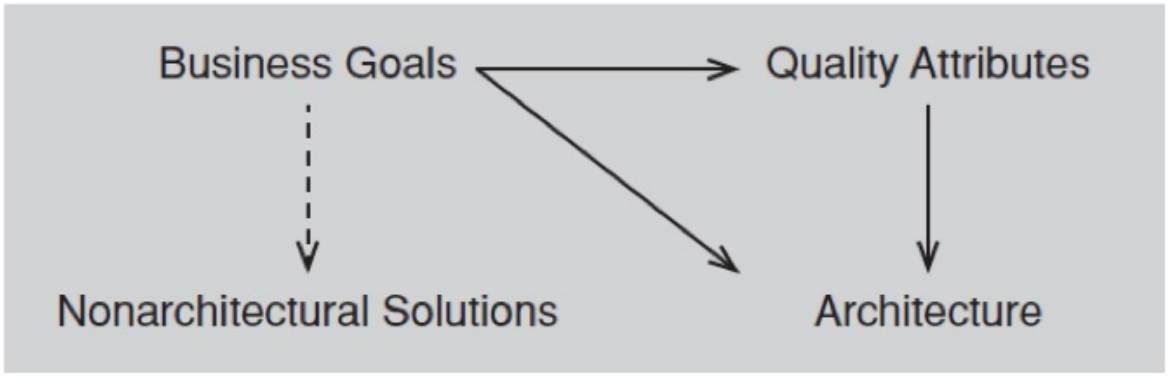
        - SH often have no idea what QAs they want
            - inaccurate measures
            - unreasonable requirements
        - Architects knows, so to interview the relevant SH
        - Result
            - a list of architectural dirvers
            - a set of QA scenarios that the SH prioritized
    - Quality Attribute Workshop (QAW)
        - a facilitated, SH-focused method to generate, prioritized and refine QA scenarios before SA is completed
        - sys-level; role that software will play in the sys
        - QAW Steps
            - QAW Presentation&Introductions
            - Business/Mission Presentation
            - Architectural Plan Presentation
            - Identification of Architectural Drivers
            - Scenario Brainstorming
            - Scenario Consolidation
            - Scenario Prioritization
            - Scenario Refinement
    - From Business Goals
        - 

FIGURE 3.2 Some business goals may lead to quality attribute requirements (which lead to architectures), or lead directly to architectural decisions, or lead to nonarchitectural solutions.

        - Business Goal Scenario 了解
            - Goal-source

        - Goal-subject -- who wish it to be true
        - Goal-object -- entities to which the goal applies
        - Environment -- context (social, legal,competitive, customer, technological...)
        - Goal
        - Goal-measure
        - Pedigree and value -- The degree of confidence the person who stated the goal has in it.
- PALM -- A Method for Eliciting Business Goals 理解
    - seven-step method
    - over 1.5 day in a workshop
    - by Architects and SH
    - Steps
        - PALM overview presentation
        - Business drivers presentation
        - Architecture drivers presentation
        - Business goals elicitation /i,lisi'teiʃən/引出
        - Identification of potential QAs from business goals
        - Assignment of pedigree to existing QA drivers
        - Exercise conclusion
- An ASR must
    - A profound impact on the architecture
    - A high business or mission value
- Utility Tree 掌握
    - A way to record ASRs all in one place
    - Establishes priority of each ASR in terms of
        - Impact on architecture
        - Business or mission value
    - ASRs are captured as scenarios
    - Root of tree is placeholder node called "Utility"
    - Second level of tree contains broad QA categories
    - Third level of tree refines those categories

- Example:

	Quality Attribute	Attribute Refinement	ASR
Utility	Performance	Transaction response time	A user updates a patient's account in response to a change-of-address notification while the system is under peak load, and the transaction completes in less than 0.75 second. (H,M)
			A user updates a patient's account in response to a change-of-address notification while the system is under double the peak load, and the transaction completes in less than 4 seconds. (L,M)
		Throughput	At peak load, the system is able to complete 150 normalized transactions per second. (M,M)
	Usability	Proficiency training	A new hire with two or more years' experience in the business becomes proficient in Nightingale's core functions in less than 1 week. (M,L)
			A user in a particular context asks for help, and the system provides help for that context, within 3 seconds. (H,M)
		Normal operations	A hospital payment officer initiates a payment plan for a patient while interacting with that patient and completes the process without the system introducing delays. (M,M)
	Configurability	User-defined changes	A hospital increases the fee for a particular service. The configuration team makes the change in 1 working day; no source code needs to change. (H,L)
	Maintainability	Routine changes	A maintainer encounters search- and response-time deficiencies, fixes the bug, and distributes the bug fix with no more than 3 person-days of effort. (H,M)
			A reporting requirement requires a change to the report-generating metadata. Change is made in 4 person-hours of effort. (M,L)
		Upgrades to commercial components	The database vendor releases a new version that must be…

Key:
H=high
M=medium
L=low

- Next Steps about UT
  - Pay attention to searching for unrecorded ASRs
  - Pay most attention to ASRs that rate a (H,H) rating
  - SH can review the utility tree to make sure their concerns are addressed

---

Chapter 17

- Design Strategy 理解
  - Decomposition
  - Designing to Architecturally Significant Requirements (ASR)
  - Generate and Test
    - View the current design as a hypothesis --> whether satisfies the requirement(test) --> if not, generate a new one
    - Initial Hypothesis
      - Desirable sources (existing sys, frameworks)

        - Less desirable sources (patterns&tactics...)
    - How test?
        - use the analysis techniques already covered
        - design strtegies from QA descussion
        - ASR
        - Output of test: List of requirements not met
    - Generate Next
        - Add missing responsiblities
        - use tactics to adjust QA behavior
    - until all ASRs are satisfied or run out of budget

> - Green Field Project
>   In software development, a greenfield project could be one of developing **a system for a totally new environment**, **without concern for integrating with other systems**, especially not legacy systems. Such projects are deemed as **higher risk**, as they are often for new infrastructure, new customers, and even new owners. For this reason, ***agile software development*** is often deemed the best approach, as it proposes how to handle those risks by developing small slices of complete functionality and getting them in the hands of customers (internal or external) quickly for immediate feedback

- ADD 掌握
    - The Atttribute-Driven Design Method
    - I/O
        - Input: Requirements
            - Functional
            - Quality
            - Constraints
        - Output: Containers
            - Responsibilities
            - Interactions
            - Information flow
    - Steps of ADD

    - Choose an element of the sys to design
        - chosen element is
            - for green field designs -- the whole sys
            - for legacy designs -- portion to be added
        - next element
            - if using new technology or a team needs work --> depth first
            - otherwise --> breadth first
    - Identify the ASRs for the chosen element
        - switch (Chosen Element):
            - case (the whole sys): use a utility tree; break;
            - case (further down the decomposition tree): generate a utility tree from the requirements for that element
    - Generate a design solution for the chosen element
        - Apply (**Generate&test**) method
    - Inventory remaining requirements and select the input for the next iteration
        - for each functional requirement, if not satisfied, then add responsibilities to satisfy the requirement
            - Add them to container with similar requirements(coherence)
    - Repeat steps 1-4 until all the ASRs have been satisfied

---

Chapter 18

- 理解：架构编档的重要性。架构文档的用途、读者。
- 了解：如何选择视图进行编档。
- 掌握：如何对视图进行编档。视图文档模版。架构文档摘要信息。如何对系统行为进行编档。如何对质量属性进行编档。

Architecture Documentation (AD)

- Audientce (stakeholders such as)
    - new employees
    - developers
    - analysts
- Three Uses for AD

    - Education -- introducing people to the sys
    - Primary vehicle for communication among stakeholders
        - Especially architect to developers
        - Especially architect to future architect!
    - Basis for system analysis and construction
- Notations
    - Informal notations
        - Use **general-purpose** diagramming&editing tools
        - Description are characterized in **natural language**
        - Cannot be formally analyzed
    - Semiformal notations(UML)
    - Formal notations
        - Precise semantics
        - Formal analysis of syntax&semantics is possible
        - Architecture description languages(ADLs)
- View
    - let us divide a software architecture into a number of manageable representations of the sys
- Principle of AD:
    - Document the relevant views
    - Add documentation that beyonds views
- Module Views
    - Element: Modules
    - Relations:
        - Is part of
        - Depends on
        - Is a
    - Constraints:
        - topological constraints
        - behavioral constraints
    - Usage:
        - Blueprint for construction of the code

        - Change-impact analysis
        - Planning incremental development
        - Requirements traceability analysis
        - Work assignments, schedules, and budget information
    - at least one module view for a AD
- C&C Views
    - Element: Components&Connectors
    - Relations:
        - Attachments
        - Interface delegation
    - Constraints:
        - C can only be attached to C
        - ...
    - Usage:
        - Show how the sys interacts
        - Help reason about runtime system qualities
- Allocation Views
    - Element:
        - Software element
        - Environmental element
    - Relations:
        - Allocated to
    - Constraints:
        - Varies by view
    - Usage
        - Reasoning about performance, availability, security, and safety
        - Reasoning about distributed development and allocation of work to teams
        - Reasoning about concurrent access to software versions
        - Reasoning about the form and mechanisms of sys installation
- Quality Views
    - it can be tailored for specific SH or to address specific concernes
    - it is formed by extracting the relevant pieces of structural views and packaging

them together

Examples

- Security View
  - Components with security role or responsibility
  - The Behavior part of it
    - Show operation of security protocols and where and how human interact with the security elements
    - Capture how the sys would respond to specific threats and vulnerabilities
- Communications View
  - Show all of the component-to-component channels, the various network channels, quility-of-service parameter values, and areas of concurrency
  - The behavior part of this view could show how network bandwidth is dynamically allocated
- Exception or error-handling view
  - Show how components detect, report, and resolve faults or errors
  - Help identify the src of errors and appropriate corrective actions for each
- Reliability view
- Performance view
- .....

-.-

- Method for Choosing the Views
  - Build a stakeholder/view table
    - Rows: SHs
    - Columns: Views that apply to system
  - Combine views to reduce their number
  - Prioritize and stage
    - The decomposition view release early
    - Don't have to satisfy all the info needs
    - Don't have to complete one view before starting another
- Documentation package consists of

    - Views
    - Documentation beyond views
- Documenting a View
    - The Primary Presentation
        - shows Elements, relations..
        - contain the Infomation you wish to convey
        - graphical, occasionally textual
    - The Element Catalog
        - Depiction of at least the element&relations in the primary presentation
        - Properties of E&R
        - Element interfaces
        - Element behavior
    - Context Diagram
        - How the sys relates to its environment
        - Entities in the environment-- humans, other computer sys or physical objects
    - Variability Guide
        - shows variation points
    - Rationale
        - Architectural Problem with the chosen pattern, and the rational for choosing it over another
- Documenting Information Beyond Views
    - Control information
        - issuing organization, the current version number...
        - Usually captured in the front matter
    - Section1: Documentation Roadmap
        - Def: What information is in and where to find it?
        - Scope and summary
        - How the doc is organized
        - View overview
        - How SHs can use the doc
    - Section2: How a View Is Documented
        - Explain the standard organization you're using to document views

    - Section3: System Overview
        - Short description of the sys's function, users & any other import bg or constraints
        - A consistent mental model of the sys and its purpose
    - Section 4: Mapping Between Views
        - Help readers gain a powerful insight into how the architecture works as a unified conceptual whole
        - View-to-View assorciations like "is implementaed by", "included in"
    - Section 5: Rationale
        - Architectural decisions that apply to more than one view
    - Section 6: Directory
        - Set of reference material that helps readers find more information quickly

---

Chapter 19

- Architecture acts as a blueprint for implementation
- Embedding the Design in the Code
- A framework is a reusable set of classes organized around a particular theme
- Frameworks
    - 基类
    - 组件交互
    - 控制器
    - 配置文件
- Code Templates
    - A collection of code within which the programmer provides application specific portions
    - Process:
        - Use the code template for every critical component that must have a hot standby
        - Place application specific code in fixed places within the template
- Advantages of Code Templates
    - Components with similar properties behave in a similar fashion

    - Template only needs to be debugged once
    - Complicated portions can be completed by skilled personnel and handed off to less skilled personnel
- Keeping Code and Architecture Consistent
    - The implementation will drift away from the documented architecture
    - Implementers may make decisions that are not consistent either with each other or with the architecture
    - The architecture may not have foreseen all eventualities that come up
- Preventing Architecure Erosion 掌握
    - Use tools to enforce architectural constraints
        - can have architecture rules added that are enforced during a build or check in
    - Mark documentation as out of date when erosion occurs. Will give more credence to remaining portion
    - Schedule documentation/code synchronizaiton times
- Architecture and Testing
    - Unit test
    - Integration test
    - Network effects
    - Test activities
- Unit Test
    - Architecture defines the **units that are to be tested**. They are **components or modules**
    - Architecture defines the **responsibilities** and **interactions** of the units
    - Test harness(测试用具） will drive the element to be tested. The test harness can test:
        - **Responsibilities** for functional correctness
        - **Performance** through synthetic loads
        - **Availability** through fault injection
    - **Modifiability requirements** can also be tested by **assigning changes** to test teams
- Integration Test

    - As with unit test, integration test can test functionality, performance, availability, and security.
    - Security can be tested by having the test harness execute various attack scenarios.
    - Systems may degrade after being run for a long time if resources are not freed or a configuration is incorrectly specified.
- Network Effects
    - Important since an error causing a 2% performance will be applied to thousands of servers and might cause severe degradation, leading to network effects
    - NE are best found through self-aware systems, i.e. system monitors itself and makes values available externally
- Test Activities (involves)
    - Test planning, since the architect knows the sensitive areas of the sys.
    - Test development
      A technique where the next increment of the sys is developed to satisfy a predetermined test
    - Test interpretation
      The architect knows what various monitored values should be and is best equipped to interpret test results
    - Test harness creation
      The test harness has to intimately interact with the sys and this requires architecure knowledge

---

Chapter 20

Architecture Reconstruction and Conformance

了解

- Why Reconstruction
    - System already exist, but you do not know its architecture
        - never recorded

        - recorded but doc has been lost
        - recorded but the doc is no longer synchronized with the sys after a series of changes
    - How to maintain such a sys?
    - How to manage its evolution to maintain the QAs that it architecture has provided
- Purposes of Reconstruction
    - To doc an architecture where doc never existed or where it has become hopelessly out of date
    - To ensure conformance between the as-built architecture and the as-designed architecture
    - In architecture reconstruction, the as-built architecture is reverse-engineered from existing sys artifacts
- Reconstructing Mappings
    - When a system is initially developed
        - architectural elements --mapped to--> specific implementation element
        - like: functions, classes, files...
    - When we reconstrut those architectural elements --> apply the inverses of the original mappings
- Method
    - Use automated and semi-automated extraction tools
    - Probe the original design intent of the architect
    - Typically we use a combination of both techniques

理解

Reconstruction Phases

- Raw view extraction
    - **Raw info** obtained from various src like src code, execution traces and build scripts
    - Each of these set of raw info is called a **view**
- Database construction
    - raw info --converted to--> standard form

    - populate a reconstruction database
    - use that to generate authoritative AD
- View fusion and manipulation
    - can improve the overall accuracy
    - can't infer runtime behavior of static views
- Architecture analysis
    - Test a set of hypotheses about the architecture(the output of view fusion) to see if they are correct
    - if not, repeat earlier steps

Detail

- Raw View Extraction
    - Analyzing a sys's existing design and implementation artifacts to construct models of it
    - These views will be refined later to support reconstruction goals
    - Blend of the ideal and the practical
    - Identify and capture the elements of interest from the source artifacts (code, header files, build files, and so on) and other artifacts(e.g., execution traces)
    - Typical List of Extracted Elements and their Relationships

Source Element	Relation	Target Element	Description
File	`includes`	File	C preprocessor `#include` of one file by another
File	`contains`	Function	Definition of a function in a file
File	`defines_var`	Variable	Definition of a variable in a file
Directory	`contains`	Directory	Directory contains a subdirectory
Directory	`contains`	File	Directory contains a file
Function	`calls`	Function	Static function call
Function	`access_read`	Variable	Read access on a variable
Function	`access_write`	Variable	Write access on a variable

    - Static info vs. Dynamic info
        - Static info is obtained by observing only the sys artifacts
        - Dynamic info is obtained by observing how the sys runs
            - Some relevant information, the precise topology of the sys.....
        - The goal is to fuse both to create more accurate sys views

- Tool

Tool	Description
Parsers	Parsers analyze the code and generate internal representations from it
AST analyzers	Build AST tree for more further analysis
Lexical analyzers	Produce tokens
Profilers	Output information about the code as it is being executed
Code instrumentation tools	Intrusive tools to monitor systems

- Database COnstruction
  - Use a database to store the extracted info
    - Amount of info is large
    - manual manipulations of the data are tedious and error-prone
  - Some reverse-engineering tools encapsulate a database
- View Fusion
  - Extracted views are manipulated to create fused views
    - Fused views combine information from one or more extracted views, each of which may contain specialized information
  - Creating a fused view is creating a hypothesis about the architecture to aid in analysis
    - These hypotheses result in new aggregations that show various abstractions or clusterings of the elements
    - By interpreting these fused views, it is possible to produce hypothesized architectural views of the sys

        - These views can be interpreted, further refined, or rejected
        - No universal completion criteria for this process
- Architectural Analysis: Finding Violations
    - Test hypotheses

Chapter 21

Architecture Evaluation

了解

- Three Forms of Evaluation
    - Evaluation by the designer within the design process
    - Evaluation by peers within the design process
    - Analysis by outsiders once the architecture has been designed
- Evaluation by the Designer
    - After every key design decision or design milestone is completed, that should be evaluated
    - It is the "test" part of the "generate-and-test"
    - "How much" depends on
        - The **importance** of the decision
        - The number of **potential alternatives**
        - Good enough as opposed to perfect
- Peer Review
    - The reviewers determine a number of quality attribute scenarios to drive the review
    - The architect presents the portion of the architecture to be evaluated
    - For each scenario, the designer walks through the architecture and explains how the scenario is satisfied
    - Potential problems are captured
- Analysis by Outsiders
    - They can cast an objective eye on an architecture

    - They possess pecialized knowledge or experience, or long experience successfully evaluating architectures
    - Managers tend to be more inclined to listen

掌握

ATAM -- The Architecture Tradeoff Analysis Method

- bg
    - used for over a decade to evaluate software architectures
    - is designed so that
        - evaluators need not be familiar with the architecture or its business goals
        - the sys need not yet be constructed, and there may be a large number of SHs
- Participants in the ATAM
    - The Evaluation Team (ET)
        - 3-5 people external to the project
    - Project decision makers
        - PM, customer, architect
    - Architecture SHs
        - include deveelopers, testers, integrators, maintainers, performance engineers, users, builders, and, possibly ohters
- Role
    - Team leader
    - Evaluation leader
    - Scenario scribe
    - Proceedings scribe
    - Questioner

Dict

```
**Articulation** is the act of expressing something in a coherent verbal fo
rm, or an aspect of pronunciation involving the articulatory organs.
```

- Outputs

    - A concise presentation of the architecture
    - Articulation of the business goals
    - Prioritized QA requirements expressed as QA scenarios
    - A set of risks and nonrisks
        - A risk if defined as an architectural decision thta may lead to undesirable consquences in light of QA requirements -- form the basis for an architectural risk mitigation plan
        - A nonrisk is an architectural decision that, upon analysis, is deemed safe
    - A set of risk themes
    - Mapping of architectural decisions to quality requirements
    - A set of identified sensitivity and tradeoff points

Dict

```
Intangible -- You can't touch this word — it is intangible.
```

- Intangible Output
    - A sense of community on the part of the stakeholders
    - Open communication channels between the architect and the SHs
    - A better overall understanding on the part of all participants of the architecture and its strengths and weaknesses
- Phases of the ATAM
    - 1 Present the ATAM
        - The Evaluation Leader(EL) presents the ATAM to all
        - EL describe the ATAM steps in brief and the output of the evaluation
    - 2 Present Business Drivers
        - the **context** for the sys and the **primary business drivers** motivating its development
        - The (Project decision maker) present **a system overview** from a business perspective
        - which includes:
            - The sys's **most importanct functions**
            - Any relevant technical, managerial, economic, or political **constraints**

        - The **business goals and context** as they relate to the project
        - The majoir SHs
        - The architectural drivers (that is, **ASR**)
    - 3 Present the Architecture -- The lead architect
        - A presentation describing the architecture
        - Covers technical constraints, and other sys with which the sys must interact
        - Describes the architectural approaches used to meet the requirements
        - Present the views that help to reason about the most important QA concerns of the sys
    - 4 Identify Architectural Approaches
        - Focuse on analyzing an architecture by understanding its architectural approaches, especially **patterns** and **tactics**
        - The ET **catalogs** the **patterns** and **tactics** that have been identified
        - The list is publicly captured and will serve as the basis for later analysis
    - 5 Gernerate Utility Tree
        - Important QA goals were named in Step2
        - (The ET) works with (the project decision makers) to **identify, prioritize**, and **refine** the sys's most important **QA goals**
        - Expressed as **Scenarios** -- populate the leaves of the utility tree
        - The scenarios are assigned a rank of importance (G,M,L)
    - 6 Analyze Architectural Approaches
        - The ET examines the highest-ranked scenarios one at a time
        - The architect is asked to explain how the architecture supports each one
        - Evalution team members probe for the architectural approaches that the architect used to carry out the scenario
        - Along the way, the ET documents the relevant architectural decisions and identifies and catalogs their risks, nonrisks, sensitivity points, and tradeoffs
    - 7 Brainstorm and prioritize Scenarios
        - The SHs brainstorm scenarios that are operationally meaningful with respect to the SHs' individual roles
        - The purpose is to take the pulse of the larger SH community
        - Then they are prioritized by voting
        - The list of prioritized scenarios is compared with those from the utility tree

exercise
    - 8 Analyze Architectural Approaches
        - The ET performs the same activities as in step 6, using the highest-ranked, newly generated scenarios
        - The ET guides the architect in the process of carrying out the highest ranked new scenarios
        - The architect explains how relevant architectural decisions contribute to realizing each one
    - 9 Present Results
        - The ET confers privately to group risks into risk themes, based on some common underlying concern or systemic deficiency
        - For each risk theme, ET identifies which of the business drivers listed in step 2 are affected
        - The collected information from the ET is summarized and presented to SHs
        - Outputs:
            - The **Architectural Approaches** documented
            - The **set of scenarios** and their **prioritization** from the brainstorming
            - The **utility tree**
            - The **risks** discovered
            - The **nonrisks** documented
            - The **sensitivity points** and **tradeoff points** found
            - **Risk themes** and the **business drivers** threatened by each one

---

Macro

- Design for (A)
    - Adopt suitable architectural patterns and tactics
    - Establish an (A) model/formula
    - Design a dedicated (A) view if necessary
    - Determine which portion of the sys pertains to (A) and Ensure that.....

Appendix

Appendix 来自林连南的笑声

> 此为录音人肉翻译，有误免责声明，爱看不看

Chap3 架构 influencecycle;arhictecture influence what,

what influnce architecture 简答或不怎么考

Part2

八个基本质量属性一个扩展质量属性，具体描述，场景的概念也要清楚

由具体到抽象，区分战术，给你一个场景有哪种质量属性。

战术大家要熟悉

以下设计战术可以提高什么（心跳…。）

以下什么战术可以提高安全性(C限制访问…）

具体什么场景用什么战术

车门下降到几秒，与什么质量相关（选择）

注意考试考英文

10-18
为什么一直架构考试都是上课讲过的东西，为什么总是选择题那么低分
就是有某个小障碍干扰

简答题呢尤其是课程作业一定要做我布置了你们

质量属性是有限的吗？那肯定是无限的啊！

注意布置的作业。。万一考到

简答题————（
怎么捕获ASR

ATM的步骤和输出
什么是甜点，场景等基本概念）

要么考作业，要么就是你一定会知道的（笑声）

13
Pattern 的分类，，三大类
Pattern统一程序方法。 ，，，通过场景，问题和方案（方案分具体元素关系和限制），，，，简答题（描述某个pattern，通过前面这几点）
面向对象在这书就是一种架构模式
开始念名字，遍历三大类各自pattern
其中SOA和Shared data就是和大作业有关的
最基本就是分层模式
课程大作业要高度重视，占了40%。。。。。。。

18-28
架构模式—从CPS, ERC这几点来理解，知道优缺点，在约束条件里边讨论，（这里涉及到一个问题）——怎么使用战术来加强 augment
课本例子 (Broker)，优缺点，，，（ ！！！强调了这个—》如何用某个战术来加强broker ————》变化无常）
以前的试卷可以看，但不能背答案，，，因为什么教学评估怎么怎么样，有查重率，不一样的考法（（以前考优缺点，现在可能变成考如何加强）

怎么综合运用多种战术加强，，课本的那个例子

15章
有个什么甜点图，得看懂，280页的图，曾经考过问答题，选择题

16
效用树
ADD缩写，步骤要知道
一般选The whole system

18
课程大作业有关，但是考试关系不大

Architecture view，概念级了解，Architecture Revolution

考过 —— 将AV的概念搬上来，然后让选是 AV ，还是Architecture style之类

——
要了解什么是产品线，大概有什么用
——
云计算基本机制，考过service model
CAP view?要了解，122，为什么三者不能兼得

Appendix 张平键版本

Chapter 1. What is Software Architecture?
理解：软件体系结构（软件架构）的定义、架构模式的概念。
掌握：软件系统有哪几类结构？在每类结构里，元素及其之间的关系是什么？每类结构各有哪些常见的结构？其特点是什么？
了解：结构与视图是什么关系？好的结构的一些经验法则。

Chapter 2. Why is Software Architecture Important?
理解：13个理由。

Chapter 3. The Many Contexts of Software Architecture
理解：技术环境、项目生命周期、商业环境、架构师职业环境中的软件体系结构。架构与环境的相互影响。
了解：涉众。

Chapter 4. Understanding Quality Attributes
了解：系统的功能需求。功能需求与系统架构的关系。功能需求与质量需求的关系。系统约束。
理解：系统的质量需求。战术的概念。
掌握：质量属性场景的概念和举例。质量设计的7种决策。

Chapter 5. Availability
理解：可用性概念。
了解：可用性公式。可用性一般场景。

掌握：可用性战术。可用性设计清单。

Chapter 6. Interoperability
理解：互操作性概念。
了解：互操作性一般场景。
掌握：互操作性战术。互操作性设计清单。

Chapter 7. Modifiability
理解：可修改性概念。
了解：可修改性一般场景。
掌握：可修改性战术。可修改性设计清单。

Chapter 8. Performance
理解：性能概念。
了解：性能公式。性能一般场景。
掌握：性能战术。性能设计清单。

Chapter 9. Security
理解：安全性概念。
了解：安全一般场景。
掌握：安全性战术。安全性设计清单。

Chapter 10. Testability
理解：可测试性概念。
了解：可测试性一般场景。
掌握：可测试性战术。可测试性设计清单。

Chapter 11. Usability
理解：易用性概念。
了解：易用性一般场景。
掌握：易用性战术。易用性设计清单。

Chapter 12. Other Quality Attributes
了解：其它软件质量属性如可变性、可移植性、开发可分布性、伸缩性、可部署性、移动性、可监控性、生命财产安全性。其它类别的质量属性如架构质量属性、商业属性、系统质量属

性。ISO/IEC FCD 25010 产品质量标准。
理解：如何处理未知的质量属性。

Chapter 13. Patterns and Tactics
了解：架构模式（架构风格）的概念。
掌握：层次模式、代理模式、MVC模式、管道-过滤器模式、CS模式、P2P模式、SOA模式、发布订阅模式、共享数据模式、Map-Reduce模式、多级模式。
理解：模式与战术的关系。

Chapter 14. Quality Attribute Modeling and Analysis
了解：模型。常见质量属性模型的成熟度。
了解：思想实验。粗略分析。原型。模拟仿真。实验。

Chapter 15. Architectures in Agile Projects
了解：敏捷开发思想与准则。
理解：敏捷开发的甜蜜点。
了解：敏捷开发与架构编档。敏捷开发与架构演化。

Chapter 16. Architecture and Requirements
理解：ASR。ASR的几种获取方法。QAW。
了解：商业目标场景。
理解：PALM方法。
掌握：效用树。

Chapter 17. Designing an Architecture
理解：Generate and Test（架构设计的假设检验法）。初始化、迭代、终结。
掌握：ADD方法。

Chapter 18. Documenting Software Architectures
理解：架构编档的重要性。架构文档的用途、读者。
了解：如何选择视图进行编档。
掌握：如何对视图进行编档。视图文档模版。架构文档摘要信息。如何对系统行为进行编档。如何对质量属性进行编档。

Chapter 19. Architecture, Implementation, and Testing

理解：实现与架构的一致性。
掌握：将架构嵌入代码。框架方法。代码模版方法。防止架构侵蚀。
了解：架构师在测试中的角色。

Chapter 20. Architecture Reconstruction and Conformance
了解：架构重构的背景和目的。
理解：架构重构的阶段。每个阶段的方法。

Chapter 21. Architecture Evaluation
了解：架构评审的3种形式及其特点。轻量级架构评审。
掌握：ATAM方法：目的、参与人员、步骤、采用的方法、结论。

Chapter 26. Architectures for the Cloud
了解：云计算的特点、主要机制、技术。
掌握：云计算下的质量属性。

考题类型
选择题（30%）
概念的理解
简答题（30%）
架构方法的掌握
分析设计题（40%）
针对具体问题，给出质量属性场景、所用战术、架构模式