中国科学技术大学
University of Science and Technology of China

Software Architecture

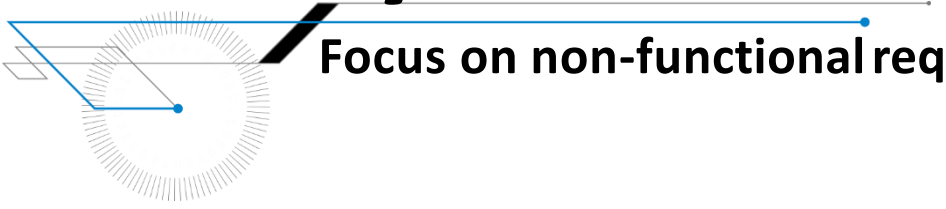
SSE USTC Qing Ding
dingqing@ustc.edu.cn
http://staff.ustc.edu.cn/~dingqing

Quality Attributes

Outline

- Quality Attributes
- Runtime QA
- Non-runtime QA
- Business QA
- Non-Functional Concepts

Quality Attributes



Focus on non-functional requirements.

Need to address QAs

- Without any need for performance, scalability, ... any implementation of functionality is acceptable
- However, we always need to take into account the broader context
- E.g. hardware, technological, organizational, business, ...
- The functionality must be there but without proper addressing of QA it is worth nothing

Influence on QAs

- Typically, a single component can not address a QA completely
- Any QA is influenced by multiple components and their interactions
- E.g. a UI component has a high degree of usability: however, usability of the system is compromised if a data management component has poor performance in accessing the data → users need to wait long → poor usability
- Components and their interactions → software architecture

  QAs are directly influenced by software architecture

Quality Attributes

Runtime Quality Attributes

QAs that affect the runtime behaviour

PURS

- PURS (performance, usability, reliability, security)
- Performance: time performance, memory, disk, or network utilization
- Usability: human factors, easy to learn, easy to use, ...
- Reliability: availability, safety, ...
- Security: authentication, data protection, ...

Performance

- Time performance is most obvious
- Measured in the number of operations per second 从系统的角度分析
- Also, latency: the time from receiving an input and producing an output 以个人的角度分析
- Other measures: memory, disk, network utilization or throughput

Performance

- Different measures are typically traded off against each other
- E.g. increasing throughput may increase latency
- Time performance might be increased with more memory
- True performance of the system is not only defined by performance of single components
- But also by their interactions and the overall processes in the system

Performance factors

1. Choice of algorithms
2. Database design
3. Communication
4. Resource management

Choice of algorithms

以复杂度来分析

- Performance of **algorithms** is typically measured by their complexity (big O notation)
- E.g. linear complexity: $O(n)$
- Running time increases in direct proportion to the size of the data
- E.g. polynomial complexity: $O(n^2)$
- It does not scale: double size of the data - running time increased by factor of 4
- → "friendly" runtime complexities: Realistic goal: $O(nlog(n))$
- (typically, only theoretically achievable: $O(n)$, $O(1)$)
- However, O does only describe one aspect of the performance (worst case behaviour vs. typical behaviour)

Database design

- Performance of database queries can dominate the overall performance
- The design of the tables has enormous impact on the overall performance
- Techniques to improve it: lazy evaluation, replication, caching
- Some additional cost to manage replication and/or caching
- In-memory databases (real-time systems) Developing a new
- database (search engines)

Communication

- Network overhead
- Package data according to a protocol, sending data over network
- Each layer means additional overhead
- Think how to use network: packaging binary data as XML!? Use
- more compact formats, e.g. JSON vs XML

Resources management

- Overloaded components need to be avoided A chain is only as
- strong as its weakest link!
- E.g. a single-threaded shared resource is in use: all other threads are blocked
- Very difficult to track down

Performance Tactics

- Generate a response to an event within some time-based constraint Two basic contributors to response time:
- ‣ Processing time (e.g. computation)
- ‣ Blocking time (e.g. waiting for other components, resources)
- Tactics:
  1. Control resource demand - reduce demand on resources
  2. Manage resources - work more efficiently

Performance Tactics - Control Resource Demand

- **Increase performance by manage the demand for resources:**
- Manage the sample rate - reduce the number of events being generated
- Limit event response - use queueing systems (in the worst case, throw away events if queue is full)
- Prioritize events - ignore low-priority events
- Increase resource efficiency - invest time to improve algorithms Reduce overhead - e.g. merge components to reduce latency →
- negative effect on modifiability
- Bound execution times - sacrifice accuracy to achieve higher performance, e.g. using faster algorithms that just approximate the correct behaviour

Performance Tactics - Manage Resources

- **Increase performance by manage resources:**
- Increase resources - faster processors, additional memory, ... (scale vertically)
- Introduce concurrency - parallel computing
- Maintain multiple copies of computations - more than one machine for computations (scale horizontally)
- Maintain multiple copies of data - caches for faster data access

Bound queue size - control the number of incoming events

- Schedule resources - introduce scheduling for resources (e.g. priorities)

Usability

- Usability is a very rich field
- If usability is important you will need a usability expert
- Combination of many factors: responsiveness, graphical design, user expectations, confidence
- Measuring with time taken to complete task, error rate, time to response, ...

Responsiveness and data availability

- An example of relations between QAs
- Usability requires that the system responds to user actions within a certain period of time
- If it is a complex system this need translates into performance along the path of the user action

Discussion on relations between QAs

- If we support security even if it is not needed
- Very often QAs exercise opposing forces on the system
- Security requires a lot of checking: performance will suffer → usability will suffer
- A minimalistic approach: develop only what is required!

Usability Tactics

- General advise: separate the user interface
- ... also support modifiability Tactics:
-

1. Support user initiative - how easy it is for the user to accomplish a task
2. Support system initiative - support of the system for the user

Usability Tactics

- **Support user initiative**
- Cancel - allow the user to cancel long running operations Undo - prove means to rollback the state
- Pause/resume - control the execution of long running operations
- Aggregate - prevent micro-management by allowing batch operations on multiple objects (e.g. select multiple files)

Usability Tactics

- **Support system initiative**
- Maintain task model - assess the user’s context to provide assistance (e.g. word completion in a text editor)
- Maintain user model - system behaviour should match users expectations (even allow the user to control the model), e.g. scrolling speed
- Maintain system model - monitor the system’s behaviour, e.g. display progressbar for long running operations

Reliability

- In traditional engineering disciplines reliability measures the failure rate of the system
- Failure rate specified by mean time to failure *MTTF*
- A related measure: mean time between failures *MTBF*

  MTTR is mean time to repair
- *A* is availability

Reliability

- $MTBF = MTTF + MTTR$
- $A = \frac{MTTF}{MTBF}$
- $A = \frac{MTTF}{MTTF+MTTR}$
- E.g. expected (theoretically optimal) availability of production systems:

Reliability

- $MTBF = MTTF + MTTR$
- $A = \frac{MTTF}{MTBF}$
- $A = \frac{MTTF}{MTTF + MTTR}$
- E.g. expected (theoretically optimal) availability of production systems: 1 (always up-and-running)
- $\Longrightarrow \quad MTTF \to \infty$

Reliability

- Increasing reliability involves testing
- However, impossible to prove that a system is correct, i.e. without bugs
- Acceptability of errors depends on the nature of a system
- Personal desktop use: bugs are typically tolerated Enterprise level:
- medium reliability level
- High-reliable systems: bugs can be fatal
- Failure analysis: combine the probability of a failure with its effect

Reliability Tactics

- Use higher level programming languages
- Use existing proven components
- Tactics:
  1. Detect faults - identify non-working components
  2. Recover from faults - what to do in case of faults
  3. Prevent faults - how to increase availability

Reliability Tactics

- **Detect faults**
- Monitor - explicit component to monitor the system
- Ping/echo - test the reachability of components by asynchronous test messages
- Heartbeat - fault detection mechanism by periodic messages Time stamp - detect incorrect sequence of messages
- Sanity checking - check the validity of messages Condition monitoring - e.g. checksums for messages
- Voting - e.g. three systems do the same thing to detect inconsistencies
- Exception detection - e.g. CPU register for division by zero, read access to uninitialised memory
- Self-test - deliberate health checks by the components themselves

Reliability Tactics

Recover from faults

- Active redundancy - identical system running a backup, instantaneous up-to-date (hot spare)
- Passive redundancy - identical system running a backup, periodic updates (warm spare)
- Spare - backup system needs to be started in case of fault (cold spare)
- Exception handling - how to deal with exceptional behaviour
- Rollback - go back to an older (good) system state
- Software upgrade - patch the system (or sub-systems)
- Retry - repeat failed action
- Ignore faulty behaviour - only, if the error messages are spurious
- Degradation - explicit state of reduced functionality Reconfiguration - reassign jobs to unaffected components

Reliability Tactics

- **Prevent faults**
- Removal from service - take components off-line to prevent (or restart) Transactions - prevent race conditions by clear transaction semantics Predictive model - state of the health of a system
- Exception prevention - e.g. smart pointers (specialised data structure)
- Increase competence set - improve components to deal with exceptional states

Security

- Increasingly important aspect of systems is security

  Because systems are exposed to threats
- Especially networked systems
- As with other QAs security is a set of related responses to user needs

Authentication

- Requirement for identification of users with a system
- Users present credentials so that the system can identify them
- Typically username and password
- Other forms: certificates, smart cards, biometric features

Authorization

- After authentication authorization which functions and what data is available for users
- This information is captured in an authorization model
- Access control lists (ACL) define who can access and how a resource might be accessed
- E.g. read access, write access, delete access, ...

Authorization

- Drawbacks of ACLs
- It is resource based, e.g. a page in a CMS
- Often, authorization needs to address functions or tasks
- Also, managing of ACLs is difficult, e.g. subresources of resources
- Also, performance problems with checking

Authorization

- Another model: role-based access control (RBAC)
- Roles are used to manage many-to-many relations between users and permissions
- Roles are used to represent the job functions, e.g. author, teacher, student in an E-learning system
- Permissions are modelled as parts of roles, e.g. create page, create tests, ...
- Users are than assigned to a role and acquire automatically permissions of that role

Security Tactics

- Tactics:
  1. Resist attacks - prevent attacks before they happen
  2. Detect attacks - identify ongoing attacks
  3. React to attacks - response in case of attack
  4. Recover from attacks - actions taken after an attack

Security Tactics

- **Resist attacks**
- Identify actors - identify the source of an external input Authenticate actors
- Authorize actors
- Limit access - restrict access to resources, e.g. firewalls
- Limit exposure - restrict the number of access points of the system

Encrypt data - protect the communication

- Separate entities - have components on different machines (or virtual machines)
- Change default settings - force users to change the default settings of a system

Security Tactics

- **Detect attacks**
- Detect intrusion - e.g. monitor network traffic and identify malicious behaviour
- Detect service denial - e.g. compare service requests pattern with known denial-of-service attacks
- Verify message integrity - e.g. checksums, hash values
- Detect message delays - e.g. monitor time to deliver messages

Security Tactics

- **React to attacks**
- Revoke access - restrict access to potentially compromised resources
- Lock account - e.g. repeated failed logins as a trigger to lock an account
- Inform actors - means to notify relevant actors, e.g. mail to sysadmin

Security Tactics

- **Recover from attacks**
- Restore - rollback the system (and its data) to a known state (prior to the intrusion)
- Audit trail - keep a record of user and system actions to allow computer forensics

Non-runtime Quality Attributes

QA not related to runtime behaviour

MeTRiCS

- MeTRiCS (maintainability, evolvability, testability, reusability, integrability, configurability, scalability)
- Maintainability: how easy can you fix bugs and add new features
- Evolvability: how easy your system copes with changes
- Testability: how easy can you test the system for correctness

- Reusability: how easy is to use software elements in other contexts,
- e.g. a software library
- Integrability: how easy you can make the separately developed components of the system work correctly together
- Configurability: how easy can a system be configured for different installations and target groups
- Scalability: how easy the system copes with a higher performance demand

Maintainability

- This QA considers the whole lifecycle of a system

  What happens during system operation?
- Property that allows a system to be modified after deployment with ease
- E.g. extensible, modified behaviour, fixing errors

Maintainability

- At the design and implementation level
- Code comments
- Object-oriented principles and design rules
- Consistent programming styles
- Documentation

Maintainability

- Maintainability is very important because any software system will change over time
- Experience shows that such changes tend to degrade the system over time
- Software systems are subject to entropy
- The cumulative effect of changes degrades the quality of the system

Maintainability

- The systems tend to become messy systems Regardless of how a
- nice plan you had at beginning Design for change - recollect OO
- design rules
- Abstract messy parts of the system so that they can be exchanged

Maintainability

- Don’t be afraid to refactor and rewrite and redesign
- Each software vendor does this with major versions
- Create throw-away prototypes
- Think out-of-box and innovate
- Don’t always follow a hype - very often nothing new in hypes
- E.g. Web services

Maintainability Tactics

- Tactics:
  1. Reduce size of components
  2. Increase cohesion
  3. Reduce coupling Defer
  4. binding

Maintainability Tactics

Reduce size of components

- Split component - divide big components into smaller sub-components

Maintainability Tactics

- **Increase cohesion**
- Increase semantic coherence - if a single component has many responsibilities then divide this component into multiple, smaller components

Maintainability Tactics

- **Reduce coupling**
- Encapsulate - introduce interfaces and APIs, and allow access only through the interface
- Use an intermediary - break dependencies, e.g. by publish-subscribe architectures
- Restrict dependencies - reduce the visibility of components, e.g. n-tier architectures
- Refactor - avoid duplication by refactoring out common functionality into new components
- Abstract common services - for similar, but not identical functionality, build a more general, abstract components

Maintainability Tactics

Defer binding

- Parameters - plan components with built in flexibility to anticipate future changes and feature requests

Testability

- Means to improve testability
- Test cases: if something fails there is a bug
- Separation of the testing framework and the system, i.e. testing with scripts from outside
- Logging

Testability Tactics

- Tactics:
  1. Control and observe system state - make testing easier
  2. Limit complexity - complex systems are harder to test, thus make them simpler

Testability Tactics

- **Control and observe system state**
- Specialised interfaces - additional testing interfaces (e.g. enable verbose output, set state)
- Record/playback - means to track in internal state Localise state storage - store a specific system state
- Abstract data sources - e.g. mock object data access instead a “real” database
- Sandbox - e.g. by virtualisation
- Executable assertions - pre/post conditions with documentation

Testability Tactics

- **Limit complexity**
- Limit structural complexity - remove dependencies, reduce coupling
- Limit non-determinism - limit behaviour complexity by identifying sources of non-determinism (e.g. unconstrained parallelism)

Configurability

- Ability of a system to vary its operational parameters without re-compiling or re-installing
- E.g. selecting appropriate database drivers, configuring network parameters, ...
- Typically, realized by a set of configuration files
- E.g. Apache Web server configuration file sets host name, virtual hosts, ...

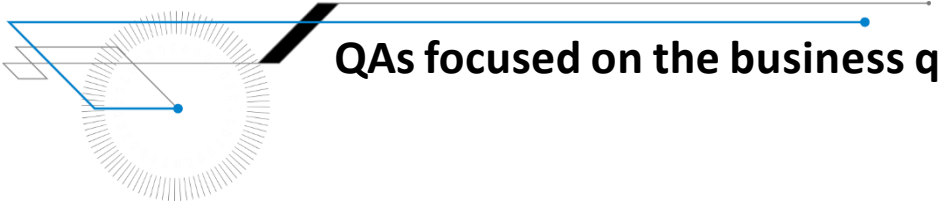
Configurability

- Configurability interacts with other QAs such as testability, maintainability, reliability
- High degree of configurability tends to have a negative impact on those QAs
- Testing of different system configuration becomes more difficult →
- reliability compromised
- Configurable components will be strongly parametrized → decreased maintainability

Scalability

- Ability of a system to increase its capacity without re-compiling or re-installing
- E.g. serving additional Web pages means only copying these Web pages into a Web server file system
- Sometimes increasing capacity means increasing hardware, e.g. Web server clusters
- Managing user session on the client side, means only providing additional code-on-demand from the server

Business Quality Attributes



QAs focused on the business quality goals

Overview

Two types of business quality attributes

- Related to costs and schedule
- Related to marketing

Overview

Time to market

- Time it takes to ship the (finished) product
- Highly depends on already existing components
- ... relates to reusability
- ... and the selection of the infrastructure components

Overview

Costs

- The budget to develop the system
- Architecture plays an important role in the actual costs

Overview

Lifetime of project

- The type of the project plays an important (business) role
- Project, that does not need to be maintained afterwards
- Products, which are planned to be maintained
  - Multiple releases in the future
  - ... will require an architecture more focused on evolvability

Overview

- **Market and target users**
  - For the mass market vs. niche markets
  - Expectations will be different for each market (e.g. level of usability)
  - … need to take this into account

Overview

- **Integration with other/legacy systems**
  - What is the expected infrastructure?
  - Which other components/systems will be present
  - … need to integrate interfaces to these systems

Non-Functional Concepts

Implications of cross-cutting concerns

Functionality Shared between Software Components

- Requirements not covered by individual components
- Instead requirements needed to be addressed **by each component**
- Or constraints imposed on each component
- ⇒ **Cross-cutting concerns**

Examples for Cross-Cutting Concerns

- Logging
- Auditing
- Monitoring
- Exception/error handling
- Security (e.g. authentication, authorisation)
- Caching
- Localisation
- Configuration management

Cross-Cutting Functionality Implications

- **Why are cross-cutting aspects a problem?**
- They do need to be consistently used/implemented by all components Some components might not allow this
- If the cross-cutting functionality need to be adapted, all components need to be modified
- E.g. changes due to new security constraints require all components to be inspected

Cross-Cutting Concerns on the Implementation Level

- Cross-cutting concerns also exist on the implementation level Addressed by **aspect oriented programming**
- Aspects are factored out and moved away from the functionality
- E.g. logging is done in a separate module and at runtime injected into the method calls

通过拦截的方式，调用log模块代码，这样就可以把与功能无关的代码发在其他地方

Cross-Cutting Concerns on the SA Level

- Keep cross-cutting functionality in separate components (is possible)
- E.g. dedicated component for authentication/authorisation Use/develop components that allow modifications
- E.g. Interceptor architectural pattern