Roadmap of the course

- What is software architecture?
- Designing Software Architecture
  - Requirements: quality attributes or qualities
  - Today
    - How to achieve requirements: tactics
- Next time:
  - How do tactics lead to architectural styles
  - Case studies on architectural styles
Tactics

- Qualities achieved via design decisions
- What design decisions needed for achieving a specific quality?

**Tactic**
- Design decision that influences the control of a quality attribute response

**Architectural strategy**
- Collection of tactics
Architectural style

• A **package** of tactics
  • Tactics can **refine** other tactics
    • Redundancy **is refined by** data redundancy, code redundancy

• **Example**
  • One **availability** tactic: introduce **redundancy**
  • Implication: we also need **synchronization** of replicas
    • To ensure the redundant copy can be used if the original fails
Availability tactics

- **Failure**
  - Deviation from intended functional behavior
  - Observable by system users

- **Failure vs fault**
  - **Fault**: event which may cause a failure

- **Availability tactics**
  - Keep faults from becoming failures
  - Make possible repairments
Availability tactics (2)

**Availability**

- Detection
  - Ping/Echo
  - Heartbeat
  - Exception

- Recovery-preparation and repair
  - Voting
  - Active redundancy
  - Passive redundancy
  - Spare

- Recovery-reintroduction
  - Shadow
  - State
  - Resynchronization
  - Rollback

- Prevention
  - Removal
  - From service
  - Transactions
  - Process
  - Monitor

Fault ➔ Availability ➔ Fault masked ➔ Repair made
Fault detection: **Ping/Echo**

- Comp. 1 issues a "ping" to comp. 2
- Comp. 1 expects an "echo" from comp. 2
- Answer within predefined time period
- Usable for a group of components
  - Mutually responsible for one task
- Usable for client/server
  - Tests the server and the communication path
- Hierarchy of fault detectors improves bandwidth usage
Fault detection: **Heartbeat**

- Comp. 1 emits a "**heartbeat**" message periodically
- Comp. 2 **listens** for it
- If **heartbeat** fails
  - Comp. 1 assumed failed
  - Fault correcting comp. 3 is notified
- **Heartbeat** can also carry data
Fault detection: **Exceptions**

- **Fault classes:** omission, crash, timing, response
- When a fault class recognized, exception is raised
  - A fault consequently is recognized
- Exception handler
  - Executes in same process that introduced the exception
  - Typically does a semantic transformation of fault into a processable form
Component recovery: Voting

- Processes running on redundant processors take equivalent input and compute output
- Output sent to voter
- Voter detects deviant behavior from a single processor => it fails it
- Method used to correct
  - Faulty operation of algorithm
  - Failure of a processor
Component recovery: active redundancy

- All redundant components respond to events in parallel => all in same state
- Response from only one comp used
- **Downtime**: switching time to another up-to-date component (ms)
- Used in client-server config (database syst)
  - Quick responses are important
- **Synchronization**
  - All messages to any redundant component sent to all redundant components
Component recovery: **passive redundancy**

- Primary component
  - responds to events
  - informs standby components of state updates they must make
- Fault occurs:
  - System checks if backup sufficiently fresh before resuming services
- Often used in control systems
- Periodical switchovers increase availability
Component recovery: spare

- Standby spare computing platform configured to replace many different failed components
  - Must be rebooted to appropriate SW config
  - Have its state initialized when failure occurs
- Checkpoint of system state and state changes to persistent device periodically
- Downtime: minutes
Component reintroduction: **shadow operation**

- Previously failed component may be run in "shadow" mode
  - For a while
  - To make sure it mimics the behavior of the working components
  - Before restoring it to service
Component reintroduction: state re-synchronization

- Passive and active redundancy
  - Restored component upgrades its state before return to service
- Update depends on
  - Downtime
  - Size of update
  - Number of messages required for the update
    - One preferable; more lead to complicated SW
Component reintroduction: checkpoint/rollback

- **Checkpoint**
  - Record of a consistent state
  - Created periodically or in response to specific events

- Useful when a system fails unusually, with detectably inconsistent state: system restored using
  - A previous checkpoint of a consistent state
  - Log of transactions occurred since
Component prevention

- Removal from service
  - Comp removed from operation to undergo some activities to prevent anticipated failures
  - Exp: rebooting comp to prevent memory leaks
  - Automatic (design architecture) or manual (design system)

- Transactions
  - sequential steps bundled together, s.t. the whole bundle can be undone at once

- Process monitor
  - If process fault detected, then monitoring process deletes non-performing process and creates new instance of it
    - initialized to some appropriate state as in the spare tactic
Modifiability tactics

- **Goal**: controlling time and cost to implement, test, modify and deploy changes

- **Sets of tactics**
  - *Localize modifications*
    - Reduce nr of modules affected by a change
  - *Prevent ripple effects*
    - Limiting modifications to localized modules
  - *Defer binding time*
    - Controlling deployment time and cost
Modifiability tactics (2)

Changes arrive

Modifiability

- Localize changes
  - Semantic coherence
  - Anticipate expected changes
  - Generalize module
  - Abstract common services

- Prevention of ripple effects
  - Hide information
  - Maintain existing interface
  - Restrict communication paths
  - Use an Intermediary

- Defer binding time
  - Runtime registration
  - Configuration files
  - Polymorphism
  - Component replacement
  - Adherence to defined protocols

Changes Made, Tested, Deployed Within Time and Budget
Localize modifications: maintain semantic coherence

- **Semantic coherence**: relationships among responsibilities in a module
- **Goal**: ensure all these responsibilities work together w/o excessive reliance on other modules
- **Goal achievement**: design modules with responsibilities in semantic coherence
Localize modifications: **abstract common services**

- Subtactic of semantic coherence
- Provides common services through specialized modules
- Reuse and modifiability
  - Modifications to common services done only once rather than in every module using them
  - Modifications to modules using common services do not impact other users
Localize modifications: *anticipate* expected changes

- Considering the set of envisioned changes provides way to evaluate particular assignment of responsibilities
- Questions
  - For a change: does proposed decomp. limit the set of modules needing modif?
  - Fundamentally diff. changes: do they affect the same modules?
- Goal: minimizing effects of changes
Localyze moudifications: **generalize the module**

- Generalize a module by making it compute a broader range of functions due to its *input type*
- Input → defining language for the module
  - Making constants input parameters
  - Implementing the module as an interpreter and making the input parameters be programs in that interpreter’s language
- The more general the module
  - The most likely is that requested changes can be made by adjusting input language
Prevent ripple effects

- **Localize modifications** vs **limit modifications to localized modules**
  - **There are modules directly affected**
    - Whose responsibilities are adjusted to accomplish change
  - **There are modules indirectly affected by a change**
    - Whose responsibilities remain unchanged BUT implementation needs to be changed to accommodate the directly affected modules
Ripple effects

- Ripple effect from a modification
  - The necessity of making changes to modules not directly affected by it
  - This happens because said modules are SOMEHOW dependent on the modules directly dealing with the modification
Dependency types

• We assume
  • Module A changed to accomplish particular modification
  • Module B changed only because A changed

• There are several dependency types
  • Syntax, semantics, sequence, identity of interface, location of A, quality of service, existence of A, resource behavior of A
Syntax dependency

- **Of data**
  - B consumes data produced by A
  - Type and format of data in both A and B need to be consistent

- **Of service**
  - B invokes services of A
  - Signature of services produced by A need to be consistent with B’s assumptions
Semantics dependency

- **Of data**
  - B consumes data produced by A
  - Semantics of data produced by A and consumed by B need to be consistent with B’s assumptions

- **Of service**
  - B invokes services of A
  - Semantics of services produced by A need to be consistent with B’s assumptions
Sequence dependency

- **Of data**
  - B consumes data produced by A
  - B must receive the data produced by A in a fixed sequence

- **Of control**
  - A must have executed previously within certain time constraints
Identity of interface of A

- A may have multiple interfaces
- B uses one of them
- For B to compile and execute correctly, the identity (name or handle) of the interface must be consistent with B’s assumptions
Other dependencies

- **Runtime location of A**
  - Must be consistent with B’s assumptions
- **Quality of service/data provided by A**
  - Properties involving the above quality must be consistent with B’s assumptions
- **Existence of A**
  - For B to execute, A must exist
- **Resource behavior of A**
  - Must be consistent with B’s assumptions
Tactics for ripple effect prevention

- Information hiding
- Maintain existing interfaces
- Restrict communication paths
- Use intermediary
Information hiding

- Decomposition of responsibilities into smaller pieces and choosing which information to make private and which public
- Public information available through specified interfaces
- **Goal:** isolate changes within one module and prevent changes from propagating to others
  - Oldest technique from preventing changes from propagating
  - Strongly related to "anticipate expected changes" (it uses those changes as basis for decomposition)
Maintain existing interfaces

- B syntax-depends on A’s interface
  - Maintaining the interface lets B stay unchanged
- Interface stability
  - Separating interface from implementation
- How to implement the tactic
  - Adding interfaces
  - Adding adapter
  - Providing a stub for A
Restrict communication paths

- Restrict number of modules with which the given module shares data
  - Reduce nr of modules that consume data produced by given module
  - Reduce nr of modules that produce data consumed by given module

⇒ Reduced ripple effect
- Data production/consumption introduces dependencies
Use an intermediary

- B dependent on A in other ways than semantically
  - Possible to introduce an intermediary to manage the dependency
  - Data (syntax)
  - Service (syntax)
  - Location of A
  - Existence of A
Defer binding time

- Decision can be bound into executing system at various times
- Binding at runtime
  - System has been prepared for that binding
  - All testing and distribution steps already completed
  - Supports end user/administrator making settings or providing input that affects behavior
Tactics with impact at load/runtime

- **Runtime registration**
  - Plug-and-play operation, extra overhead to manage registration
- **Configuration files** - set parameters at start-up
- **Polymorphism** - late binding of method calls
- **Component replacement** – loadtime binding
- **Adherence to defined protocols**
  - Runtime binding of independent processes
Performance tactics

- **Goal**: generate response to an event arriving at system within time constraint
- **Event**: single or stream
  - Message arrival, time expiration, significant state change, etc
- **Latency**: time between the arrival of an event and the generation of a response to it
- **Event arrives**
  - System processes it or processing is blocked
Performance tactics (2)

Events arrive

Performance

- Resource demand
  - Increase computation efficiency
  - Reduce computational overhead
  - Manage event rate
  - Control frequency of sampling

- Resource management
  - Introduce concurrency
  - Maintain multiple copies
  - Increase available resources

- Resource arbitration
  - Scheduling policy

Response generated within time limit
Resource demand tactic

- Source of **resource demand**: event stream
- **Demand characteristics**
  - Time between events in resource stream (how often a request is made in a stream)
  - **How much of a resource** is consumed by each request
- Reducing latency tactic
  1. Reduce required resources
  2. Reduce nr of processed events
Reduce required resources

- Increase computational efficiency
  - Processing involves algorithms => improve algorithms
  - Resources may be traded for one another
- Reduce computational overhead
  - If no request for a resource => its processing needs are reduced
  - Intermediaries removed
Reduce nr of processed events

- Manage event rate
  - Reduce sampling frequency at which environmental variables are monitored

- Control frequency of samplings
  - If no control over arrival of externally generated events => queued requests can be sampled at a lower frequency (request loss)

- Bound execution times
  - Limit over how much execution time used for an event

- Bound queue sizes
  - Controls max nr of queued arrivals
Resource management

- **Introduce concurrency**
  - Process requests in parallel
    - Different event streams processed on different threads (create additional threads)
    - Load balancing

- **Maintain multiple copies of data and computations**
  - Caching and synchronization

- **Increase available resources**
  - Faster processors and networks, additional processors and memory
Resource arbitration

- Resource contention => resource must be scheduled
- Architect’s goal
  - Understand characteristics of each resource’s use and choose compatible scheduling
  - Understand possibly conflicting criteria for scheduling
    - And effect of chosen tactic
- Scheduling policy
  - Priority assignment
  - Dispatching
Scheduling

• What: network, buffers, processors

• Competing criteria for scheduling
  • Optimal resource usage
  • Request importance
  • Minimizing nr of used resources
  • Minimizing latency
  • Maximizing throughput
  • Preventing starvation to ensure fairness

• Dispatching can happen only when assigned resource available
  • Pre-emptying might occur
Scheduling policies

- First in/first out (FIFO)
  - OK if all requests are of equal importance and take same time
- Fixed priority scheduling, priorities based on
  - Semantic importance (domain specific)
  - Deadline monotonic (real-time deadlines; shortest deadline 1\textsuperscript{st})
  - Rate monotonic (periodic streams, shorter period 1\textsuperscript{st})
- Dynamic priority scheduling
  - Round robin
  - Earliest deadline first
- Static scheduling
Security tactics

- **Goal:** resisting *attacks*, detecting *attacks*, recovering from *attacks*
- House defense analogy
  - Door lock
  - Motion sensor
  - Insurance
Security tactics (2)

Security

- Resisting attacks
  - Authenticate users
  - Authorize users
  - Maintain data confidentiality
  - Maintain integrity
  - Limit exposure
  - Limit access

- Detecting attacks
  - Intrusion detection

- Recovering from attacks
  - Restoration
  - Identification
  - See availability
  - Audit trail

System Detects, Resists, or Recovers from attacks

8.11.2011
Resisting attacks

- Authenticate users
  - Passwords, one-time passwords, digital certificates, biometric id
- Authorize users
  - Access control patterns
- Maintain data confidentiality
  - Encryption to data and communication links
- Maintain integrity
  - Checksums, hash results help
- Limit exposure: limited services on each host
- Limit access: firewalls, DMZ
Detecting attacks

- Intrusion detection system
  - **Compared** network traffic patterns to database
  - **Misuse** => pattern compared to historical patterns of known attacks
  - **Anomaly** => pattern compared to historical baseline of itself
- Filtering
  - Protocol, TCP flags, payload size, addresses, port numbers
- **Must have**
  - Sensor to detect attacks
  - Manager for sensor fusion
  - Databases for storing events for analysis
  - Tools for offline reporting and analysis
  - Control console to modify intrusion detection actions
Intrusion detectors

- Sensor to detect attacks
- Managers for sensor fusion
- Databases for storing events for later analysis
- Tools for offline reporting and analysis
- Control console
  - Analyst can modify intrusion detection actions
Recovering from attacks

- Tactics for restoring state
  - Recovering a consistent state from an inconsistent one: availability tactics
  - Redundant copies of system adm data
    - Passwords, access control lists, domain name services, user profile data: special attention
- Tactics for attacker identification
  - For preventive or punitive purposes
  - Maintain audit trail
Audit trail

- Copy of each transaction applied to data in the system + identifying information
- Can be used to
  - Trace the actions of an attacker
  - Support non-repudiation
    - Provides evidence that a particular request was made
  - Support system recovery
- Often attack targets
  - Should be maintained in trusted fashion
Testability tactics

- **Goal**: allow for easier testing when some increment of software development is completed
- Enhancing testability not so mature but very valuable
  - 40% of system development
- Testing a running system (not designs)
- Test harness
  - SW that provides input to the SW being tested and captures the output
  - Goal: find faults
Testability tactics (2)

- Manage Input/Output
  - Record/Playback
  - Separate Interface from Implementation
  - Specialized Access Routines/Interfaces

- Internal Monitoring
  - Built-In Monitors

Completion of an Increment → Faults detected
I/O Tactics: record/playback

- Refers to
  - Capturing info crossing an interface
  - Using it as input to the test harness

- **Info crossing an interface** at normal operation
  - Output from one component, input to another
  - Saved in repository
    - Allows test input for one component
    - Gives test output for later comparisons
I/O Tactics: interface vs implementation

- Separating interface from implementation
  - Allows substitution of implementations for various testing purposes
    - Stubbing implementations let system be tested without the component being stubbed
    - Substituting a specialized component lets the component being replaced to act as test harness for the remainder of the system
  - Tactic also achieves modifiability
I/O Tactics: specialize access routes/interfaces

- **Having** specialized testing interfaces
  - Captures/specifies variable values for components
    - Via test harness
    - Independently from normal execution
- **Specialized access routes/interfaces**
  - Should be kept separate from required functionality
- **Hierarchy** of test interfaces
  - Test cases can be applied at any architectural level
  - Testing functionality in place to observe responses
Internal monitoring tactic

- **Built-in monitors**
  - Component can maintain *state*, *performance load*, *capacity*, *security*, etc accessible through interfaces
    - Permanent interface or temporarily introduced for testing
  - Record events when monitoring states activated
    - Additional testing cost/effort
    - Increased visibility
Usability tactics

- **Usability concerns**
  - How easy can a user accomplish desired task
  - What is the system support for the user

- **Tactics**
  - *Runtime*: support user during system execution
  - *Design time*: support interface developer
    - Iterative nature of interface design
    - Related to modifiability tactics
    - Refinement of semantic coherence tactic to localize expected changes
Runtime tactics

- User feedback as to what is the system doing
- Providing user with ability to issue usability commands
  - Cancel
  - Undo
  - Aggregate
  - Show multiple views
Initiatives

- HCI terminology
  - User initiative
  - System initiative
  - Mixed initiative
- User initiative
  - Architect designs response as for any other functionality
- System initiative
  - System needs models: user, task, system state
Usability tactics

Usability

Runtime tactics

System initiative

User initiative

Design time tactics

Separate user interface

User given appropriate feedback and assistance
Todays take away

- There are many ways to achieve a QA – tactics
  - Can work well or less well
  - Can interact with other tactics for achieving other QAs
- A Software Architecture – determined by the collection of tactics