SOFTENG 325:
Software Architecture

Part II, Lecture 1a: Overview

Ewan Tempero
Department of Computer Science

Agenda

- Class schedule
  - No formal labs
- Software Architecture Overview
- Reading.
  - *The architecture of open source applications* Amy Brown & Greg Wilson (editors), Volume II, Chapter 1
  - *Essential Software Architecture* Ian Gorton, Chapter 1

References

- *The architecture of open source applications* Amy Brown & Greg Wilson (editors), Volumes I & II `http://aosabook.org` (also available through library)
- *Essential Software Architecture* Ian Gorton (2006) available on-line through the library
- *Software Architecture in Practice* Len Bass, Paul Clements, Rick Kazman — 1st (1998), 2nd (2003), and 3rd (2013) editions
- *Software architecture : perspectives on an emerging discipline* Mary Shaw, David Garlan (1996)

One of these is not like the others

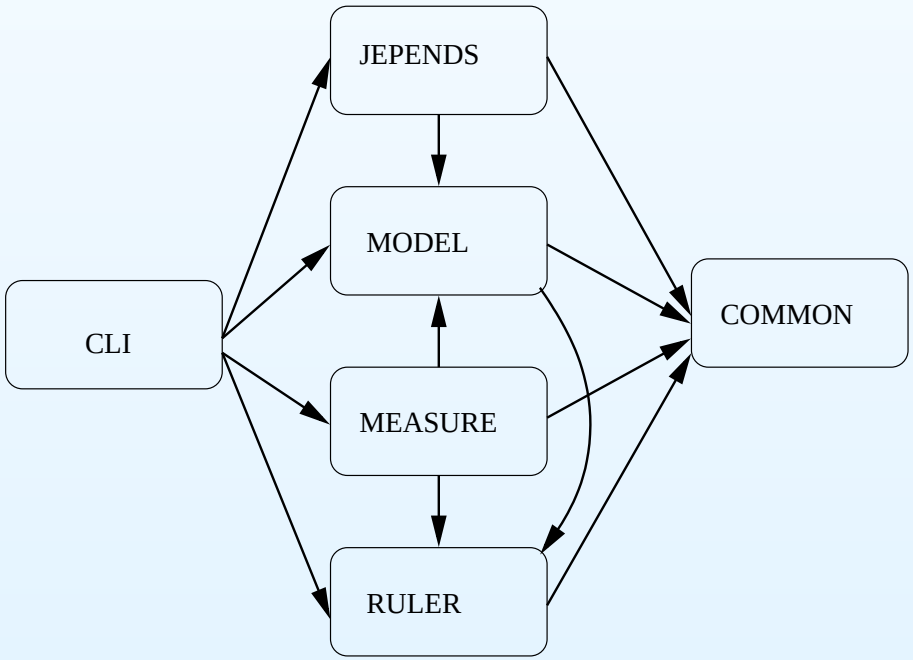
Show picture of 4 bridges.

Example: `mete`

- Your task: add a new way to analysis Java source code to Ewan's research software system `mete`
- `mete` consists of 178 *extremely well documented* `.java` files (not counting 400+ `.java` files used for testing)
- Where do you start?

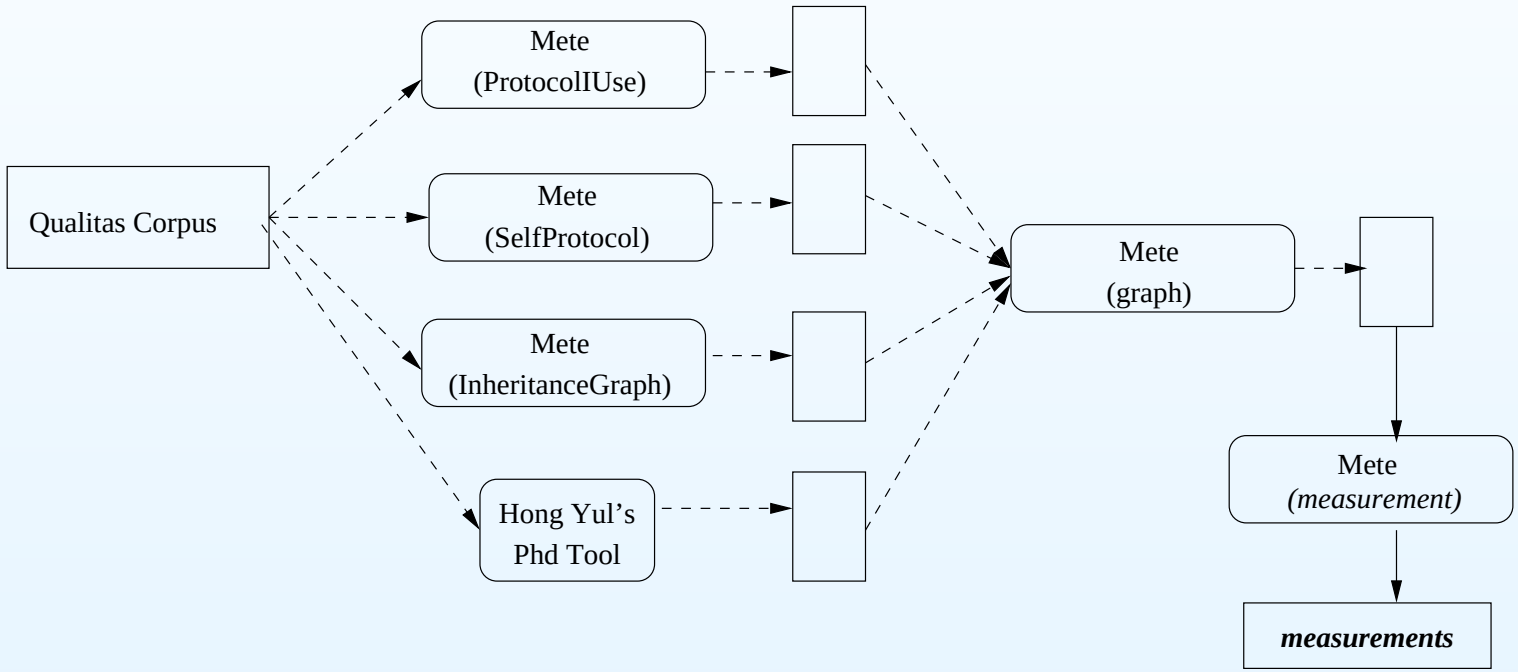
Example: `mete`

- Your task: add a new way to analysis Java source code to Ewan's research software system `mete`
- `mete` consists of 178 *extremely well documented* `.java` files (not counting 400+ `.java` files used for testing)
- Where do you start?



Example: `mete`

- Your task: replicte the “bad inheritance study” using `mete`
- Where do you start?



Definitions of Software Architecture

There are lots —
`http://www.sei.cmu.edu/architecture/definitions.html`
has over 100

- The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time. (Garlan and Perry, Guest editorial, *IEEE Transactions on Software Engineering*, April 1995)
- An architecture is the set of significant decisions about the organization of a software system, the selection of the structural elements and their interfaces by which the system is composed, together with their behavior as specified in the collaborations among those elements, the composition of these structural and behavioral elements into progressively larger subsystems, and the architectural style that guides this organization—these elements and their interfaces, their collaborations, and their composition (Booch, Rumbaugh, and Jacobson, *The UML Modeling Language User Guide*, 1999).
- The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them. (Bass, Clements, Kazman *Software Architecture in Practice*

Other phrases

- “large scale components”
- “system level abstraction”
- “components and interaction between components”
- “system level analysis”
- “load-bearing walls”

Why Software Architecture?

- systems are getting bigger and more complicated
  - $\Rightarrow$ need "standard" solutions to common problem *types* — **reuse**
  - $\Rightarrow$ need some way to talk about them — **understanding, communication**
  - $\Rightarrow$ need some way to organise them — **construction, evolution**
  - $\Rightarrow$ need some way to reason about them — **analysis, management**
  - $\Rightarrow$ **large scale decisions** are more important (from a total system cost viewpoint) than data structures and algorithms

In the Software Lifecycle

- Requirements
- Specification
- *Architecture*
- Detailed Design
- Implementation

Software Architecture Topics

Classification How do we organise architectures? What kinds (or “styles”) or architectures are there? How do we describe an architecture?

Analysis: Given an architecture, how do we reason about it? What do we need to know about it to reason effectively?

Develop: How do we decide on an architecture for a system? How do we create a new architecture? What issues to we have to keep in mind?

Documentation: How should architectures be described? What are the important things to show?

What Software Architecture is Not

- Software Architecture is not System (or Computer) Architecture
- Software Architecture is not about functionality
- Software Architecture is not a box-and-line drawing

What Software Architecture is Not

- Software Architecture is not System (or Computer) Architecture
- Software Architecture is not about functionality
- Software Architecture is not a box-and-line drawing

What is Software Architecture?

> The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them. (Bass, Clements, Kazman *Software Architecture in Practice*)

What is Software Architecture?

> The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them. (Bass, Clements, Kazman *Software Architecture in Practice*)

- multiple structures
- software elements and relationships
- externally visible properties

Describing Architectures

- An architecture rarely can be fully understood by just looking at one picture
- The architecture for a system of a reasonable size will consists of several *structures*
- Any structure may be described in several different *views*

Good & Bad Architectures

- An architecture is not inherently “good” or “bad”
- The only criteria of relevance is “meets quality requirements” or “has *quality attributes*”
- Quality is not *absolute*. It can only be measured relative to what criteria are considered important
  - “I don’t care if it blue-screens once a day so long as it is cheap/the same as everyone else uses/accessible to me”
  - “I don’t care if it’s free, I want someone else to install/maintain/fix it (cheaply)”
  - “I have particular requirements so I need to be able to change the code myself”

Quality Attributes relevant to SA

- Extensibility
- Performance
- Security
- Understandability
- Readability
- Comprehensibility
- Modifiability
- Maintainability
- Portability
- Buildability
- Scaleability
- Availability
- Reliability
- . . .

Properties of software

- Which is more important:
  - Getting the system built quickly (buildability)
  - Making changes to it cost effectively, whether to fix faults or to cope with changing conditions (modifiability or maintainability)
  - Making changes to it to add new functionality cost effectively (extensibility)
  - Be able to move systems onto different hardware and/or operating systems (portability)
  - Having developers new to the project be able to understand it quickly (understandability, readability, comprehensibility)

Client-Server

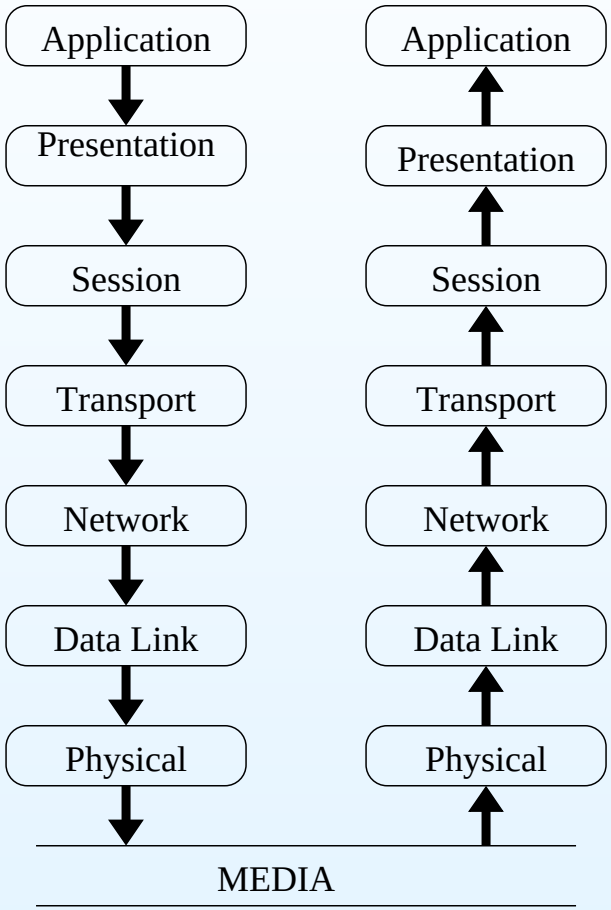
- “seminal architectural model which remains the most widely adopted”
- What makes something “Client-Server”?
- When should we use Client-Server?
- When should we *not* use Client-Server?
- Is Client-Server necessary and sufficient for :
  - SVN
  - Cecil
  - Google
  - Amazon
  - Excel
  - SPSS
  - ?

Client-Server: Relevant Quality Attributes

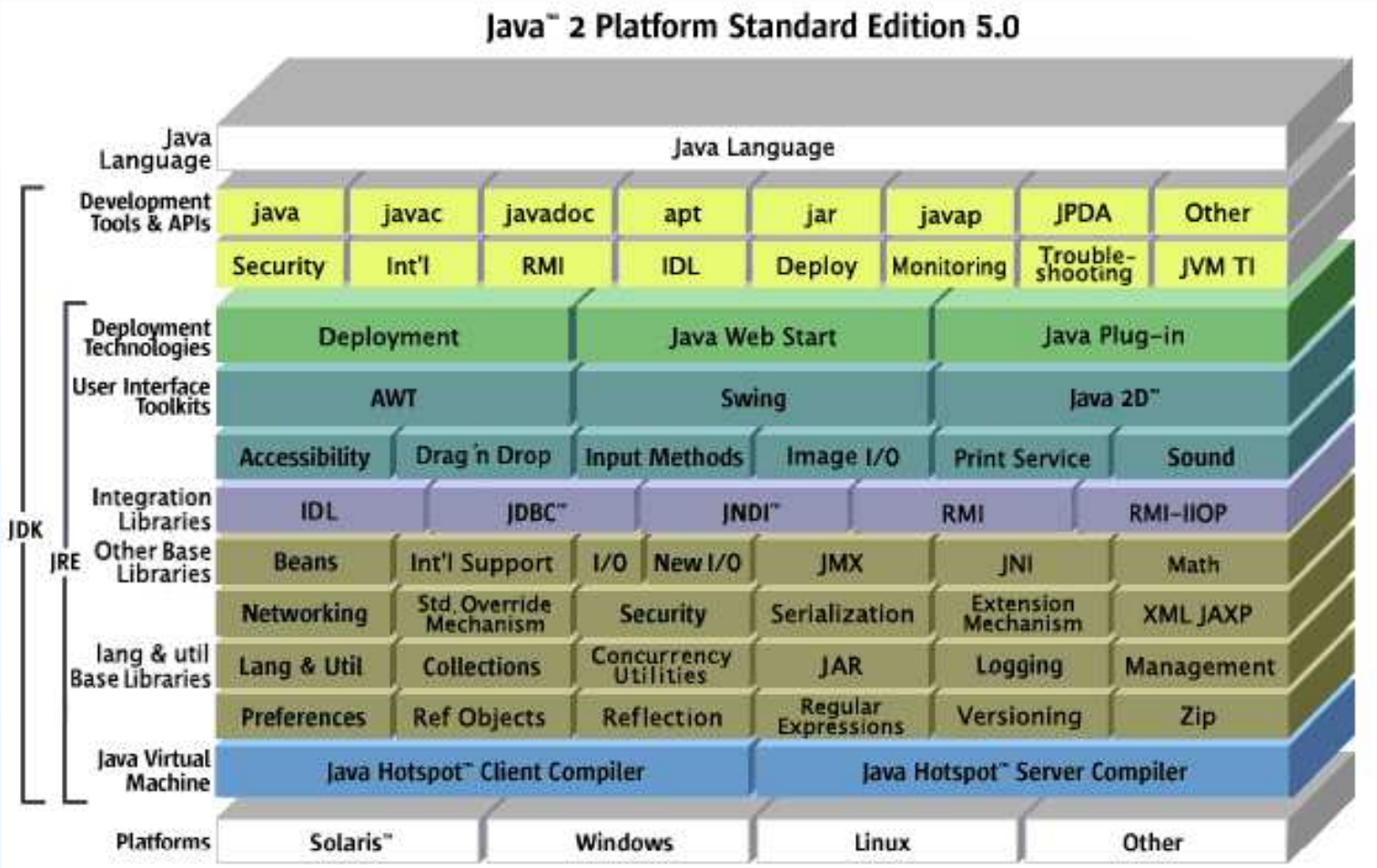
- Extensibility — Only the server needs to change
- Performance — Add more servers
- Security — communications channel is a weak link
- Understandability, Readability, Comprehensibility — Can deal with client (UI only) separately from server (no UI), added complication of protocol between client and server
- Modifiability — can change client and server independently
- Maintainability — faults in client won't affect server (?)
- Portability — change change client and server hardware independently
- Buildability — More work to do
- Scaleability — More clients can be added
- Availability — System can be unavailable even though the client (server) is still working (because the server (client) is not)
- Reliability — More things to fail
- . . .

Layers

- a layer encapsulates everything below it



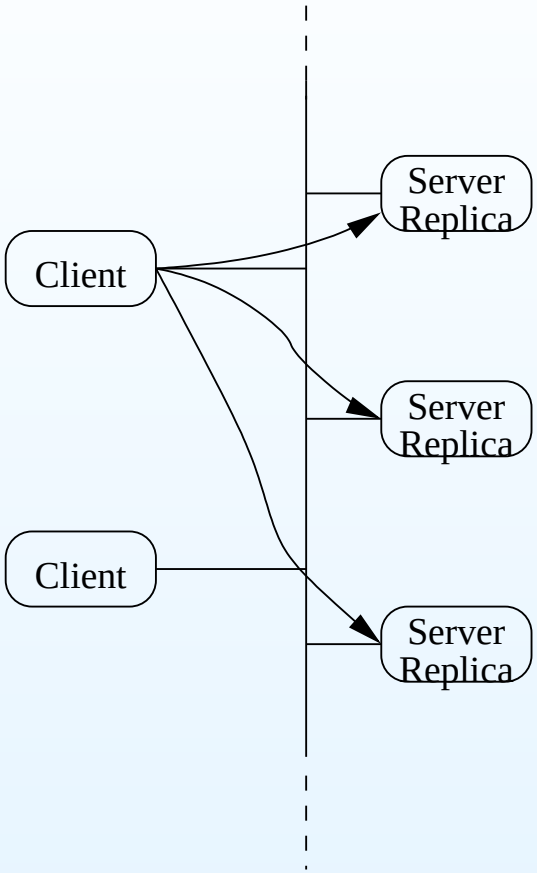
Layers

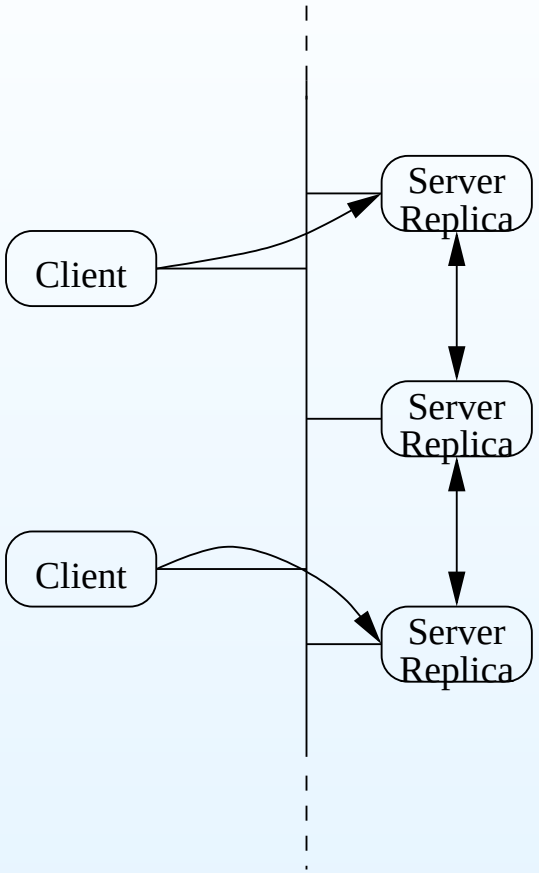


Quality Attributes relevant to Layers

- Performance — more layers $\Rightarrow$ slower
- Buildability/Modifiability/Understandability/. . . — only need to deal with one layer at a time
- Portability — only the lower layer(s) need to change

Replication





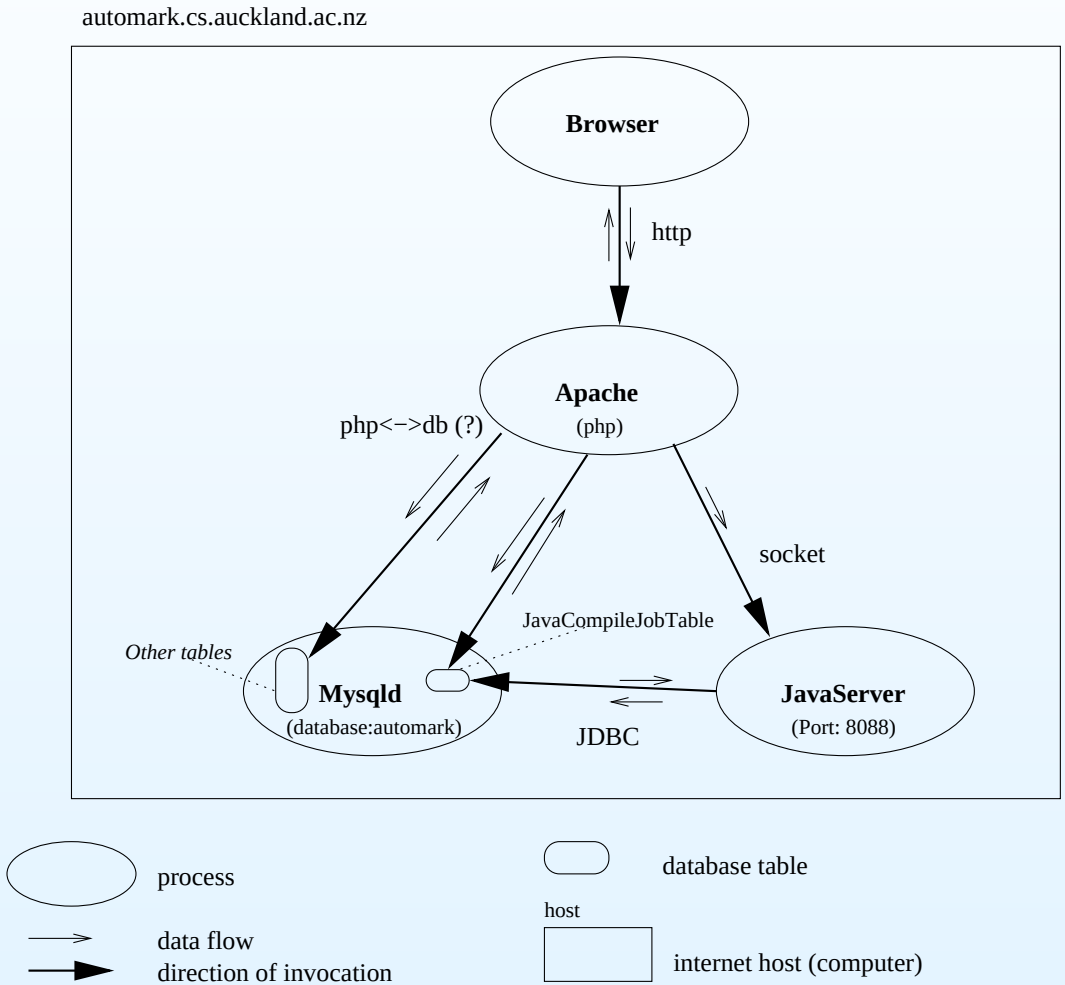
- Under which circumstances should different forms be used?

Quality Attributes relevant to Replication

- Availability, Performance — broadcast style increases by reducing effect of server failures, and not having to wait for the slowest server, but broadcast in general does not scale
- Scaleability — can add as many servers as are needed (sometimes at the cost of performance)

Conveying Information

- What information is this diagram providing?



Eclipse Plug-in Architecture

- Provides Extensibility

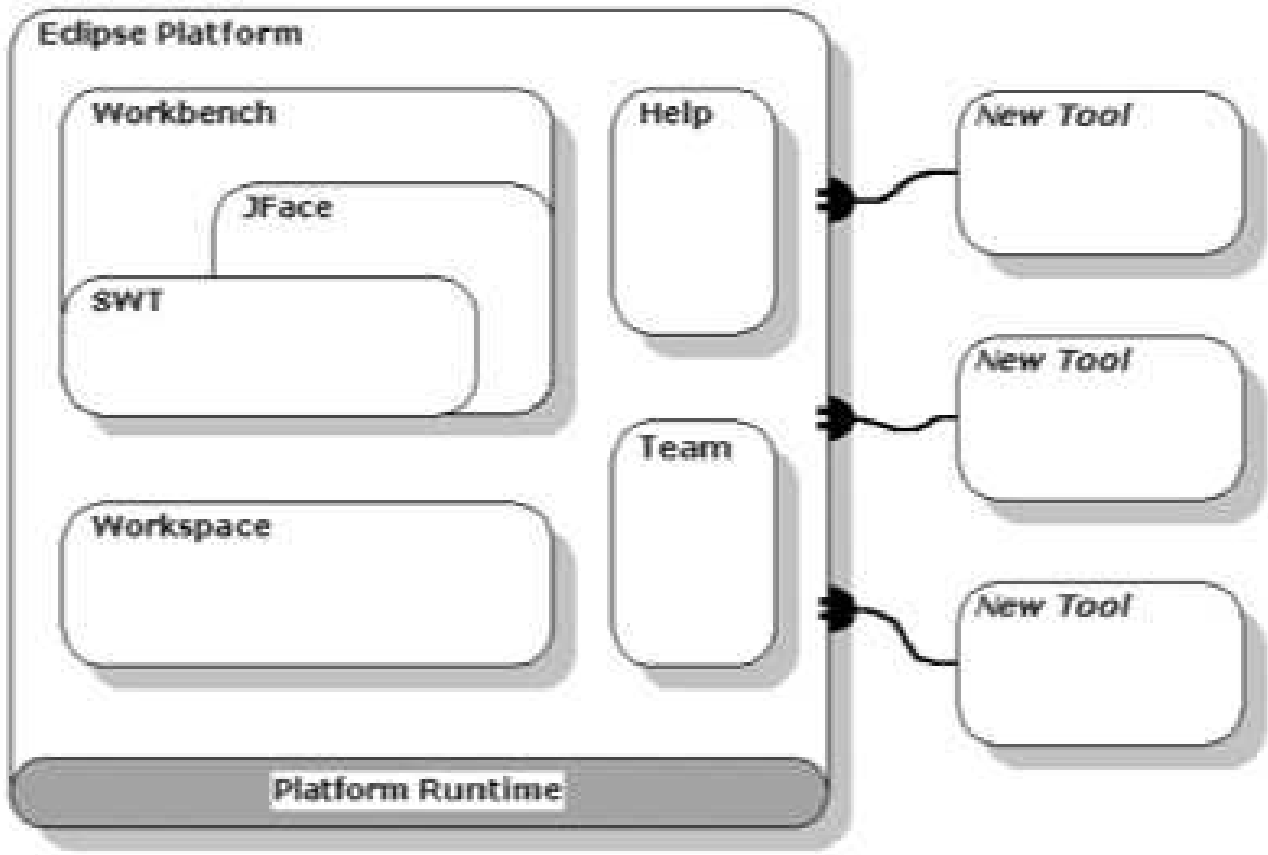
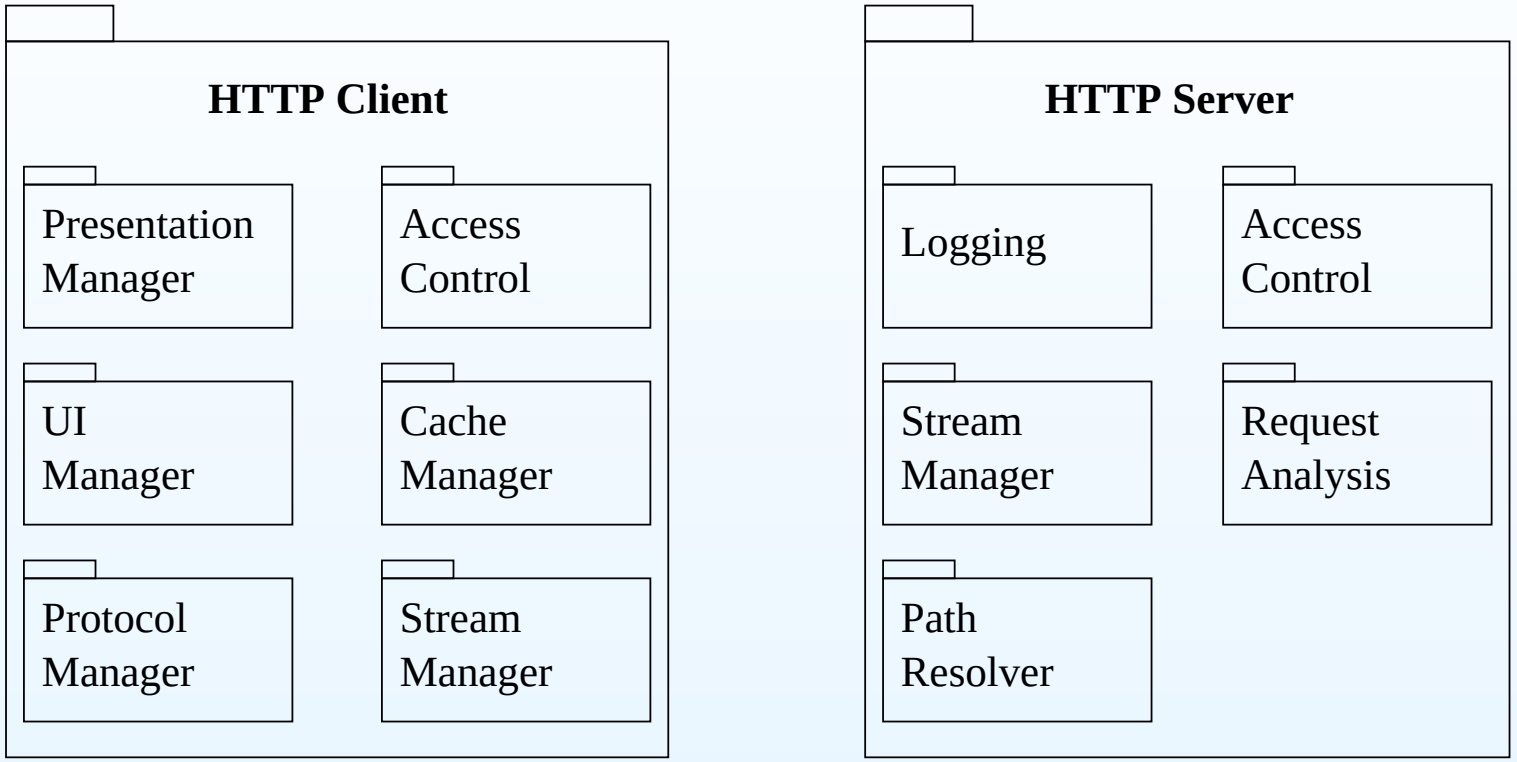


Figure 2. Eclipse Platform architecture.

Eclipse Plug-in

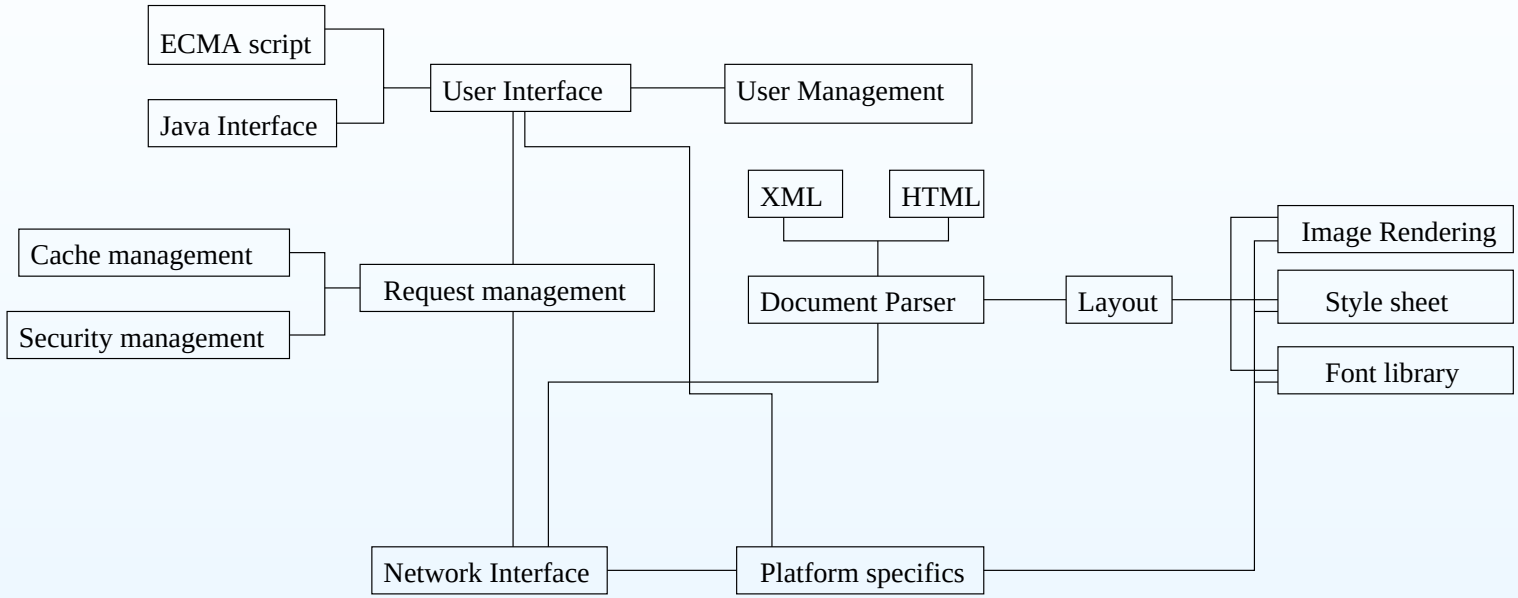

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
  name="JUnit Testing Framework"
  id="org.junit"
  version="3.7"
  provider-name="Eclipse.org">
  <runtime>
    <library name="junit.jar">
      <export name="*"/>
    </library>
  </runtime>
</plugin>
```

Describing the “code”



- Decompose along functional lines into modules
- Different modules get deployed in different places

HTTP Client Architecture



- systems have subsystems
- subsystems have architecture

Specifying Quality

- software architecture depends on *quality attribute* requirements
- how do we specify these requirements so that we can determine if a given software architecture meets them?

Summary

- The software architecture of a system is the “large-scale” view of its design
- The choice of architecture for a system depends on the quality attributes the system must meet.

SOFTENG 325:
Software Architecture

Part II, Lecture 1b: Quality Attributes

Ewan Tempero
Department of Computer Science

Potential Assessment Question

Which of the following are **not** relevant to the need for discussing software architecture?

(a) Makes it easier to come up with solutions
(b) Makes easier to learn how systems work
(c) Makes it easier to decide which data structures are appropriate
(d) Makes it easier to determine if a proposed solution will meet the requirements
(e) None of the above.

Justify your answer.

Agenda

- PAQ
- Assignment 2
- Quality attributes
- Specifying quality — Quality Attribute Scenarios
- **References**
  - *Software Architecture in Practice* Bass et al. — Chapter 4.

Assignment 2

- Create an architecture that could meet a set of Quality Attribute Scenarios for **BigBrotherDriver**.
- Write (part of) a report explaining the architecture (views of relevant structures etc), explaining how the architecture meets the quality attributes and explains how the architecture was developed (tactics)

BigBrotherDriver (BBD)

- Track all vehicles in New Zealand, report their movements, take action as necessary.
- Vehicles have occupants that run an app (BBD-V) that reports all movements and driving patterns.
- Requirements for performance, modifiability, and security

Assignment 2

- Due 23hr Monday 16 October
- Worth 7%
- Submit via `adb.auckland.ac.nz`
- Architecture assessed mainly on how convincing it is, not how “correct” it is
- More details on Canvas.

Quality Requirements Example

Create a web-based travel booking system. It must be buildable, reliable, secure, fast.

- Choice of architecture is determined by quality attributes, not functionality
- $\Rightarrow$ need to know which quality attributes are important to determine best (or at least acceptable) architecture
- Change important quality attibutes $\Rightarrow$ (potentially) change architecture that is most acceptable

Example Quality Requirements

Create a web-based travel booking system. It must be modifiable, reliable, secure, fast.

Example Quality Requirements

Create a web-based travel booking system. It must be modifiable, reliable, secure, fast.

- what are these requirements really saying?
- how do we know when the requirements have been met?

Quality Attributes

A **quality attribute** is a measurable or testable property of a system that is used to indicate how well the system satisfies the needs of its stakeholders

- quality — some notion of how "good" the product is $\Rightarrow$ "satisfies the needs of its stakeholders"
- quality attribute — some specific aspect of quality (e.g., integrity, correctness, reliability, maintainability, flexibility, reusability, modularity, etc) $\Rightarrow$ "testable property"
- there are many architectures for any given set of functional requirements, they differ on the "non functional" quality attributes

Achieving quality attributes

- choice of architecture is crucial to realising quality attributes
- choice of architecture gives the *possibility* of achieving quality attributes
- choice of architecture does **not** guarantee achieving quality attributes
- *not all quality attributes are affected by architecture*

System Quality Attributes

- Advocated by Bass, Clements, Kazman
  - **availability** — is the system providing the service it is supposed to when it is expected to
  - **modifiability** — can reasonable changes be made at reasonable cost
  - **performance** — does the system respond quickly enough or without using too many resources
  - **security** — can authorised people access the system services while unauthorised people can or will not
  - **testability** — can the system be artificially exercised to a degree that ensures confidence in correctness
  - **usability** — how easy is it for the user to do what is desired
  - **interoperability** — the degree to which two or more systems can usefully exchange meaningful information

Business Quality Attributes

- Can also affect choice of architecture, more difficult to operationalise
  - **time to market** — is there pressure to deliver quickly
  - **cost and benefit** — what does the budget allow
  - **system lifetime** — how long must the system be useful
  - **market** — what does the market consider important
  - **roll-out schedule** — in what manner will the system be made available
  - **integration with legacy systems** — must the system have to fit with existing systems

Other Architecture Qualities

- Can also affect choice of architecture, very difficult to operationalise
  - **conceptual integrity** — is there a single theme
  - **correctness and completeness** — is it possible to provide the functional requirements
  - **buildability** — can be the system be built efficiently?

What about other "qualities"?

- There are more system quality attributes than the six listed — what about the others?
- In principle, all other system qualities can be expressed in terms of those six:

  Portability The system must be changed to run on different hardware/operating system ⇒ modifiability

  Maintainability ("fixing bugs") The system must be changed to remove the invalid behaviour ⇒ modifiability

  Scaleability The system must be changed to have more capacity (can handle more requests than currently, store more information,. . .) ⇒ modifiability

  Operability The system must be "easy" to use by being "easy" to learn ⇒ usability

  Reliability The system must not fail "too often" ⇒ availability

  Integrity Data presented by the system must be not able to be changed in an unauthorised or unexpected manner ⇒ security

Example

Create a web-based travel booking system. It must be modifiable, reliable, secure, fast.

- what are these requirements really saying?
- how do we know when the requirements have been met?
- $\Rightarrow$ need description of quality requirements that is operational

Quality Attribute Scenarios

- Provides an operational description of the requirements for quality attributes
- Identifies and categorises the necessary information for making such requirements operational
- Available in concrete and general forms
- Process is to first determine relevant general QAS, refine into concrete QAS

Quality Attribute Scenarios

- **stimulus** — condition or event that needs to be considered, occurs at (some part of) the system
- **source of stimulus** — some entity generates a stimulus
- **environment** — conditions under which the stimulus occurs
- **artifact** — the thing that is stimulated
- **response** — what the artifact should do on arrival of the stimulus
- **response measure** — how to measure the response to determine that it is satisfactory

Example: “fast” (1)

- stimulus — “form submission”
- source of stimulus — “the user”
- environment — “normal working conditions”
- artifact — “the system”
- response — “load and display page indicating successful purchase”
- response measure — “the page is loaded in less than 2 seconds 95% of the time”

Example: “fast” (1)

- stimulus — “form submission”
- source of stimulus — “the user”
- environment — “server is not overloaded, no network congestion”
- artifact — “the system”
- response — “load and display page indicating successful purchase”
- response measure — “the page is loaded in less than 2 seconds 95% of the time”

Example: “fast” (1)

- stimulus — “form submission”
- source of stimulus — “the user”
- environment — “server is not overloaded, no network congestion”
- artifact — “the system”
- response — “load and display page indicating successful purchase”
- response measure — “the page is loaded in less than 2 seconds ~~95% of the time~~”

Example: “fast” (2)

- stimulus — “form submission”
- source of stimulus — “the user”
- environment — “normal working conditions”
- artifact — “the server”
- response — “verify and record purchase”
- response measure — “process 1000 transactions per second”

Example: “reliable (1)”

- stimulus — “single cpu failure”
- source of stimulus — “random event”
- environment — “normal working conditions”
- artifact — “server”
- response — “notify operator”
- response measure — “no loss of service”

Example: “secure” (1)

- stimulus — “Attempts to change details on web server”
- source of stimulus — “Disillusioned Youth”
- environment — “normal working conditions”
- artifact — “server”
- response — “notify operator with details”
- response measure — “no loss of service”

Example: “secure” (2)

- stimulus — “Attempts to change details on web server”
- source of stimulus — “Disillusioned Webmonkey”
- environment — “normal working conditions”
- artifact — “server”
- response — “record access”
- response measure — “identity of person making change is available”

Example: “secure” (2)

- stimulus — “Attempts to change details on web server”
- source of stimulus — “Disillusioned Webmonkey”
- environment — “normal working conditions”
- artifact — “server”
- response — “record access”
- response measure — “actions and identity of person making change is available”

Example: “reliable (2)”

- stimulus — “requests (some service)”
- source of stimulus — “the user”
- environment — “receiving more than 1000 requests per second”
- artifact — “the system”
- response — “reject request, report to operator”
- response measure — “operator must be notified within 5 minutes”

Example: “modifiable (1)”

- stimulus — “requests 100GB extra storage capacity”
- source of stimulus — “the system administrator”
- environment — “normal working conditions”
- artifact — “the system”
- response — “extra storage is provided”
- response measure — “no loss of service”

Example: “modifiable (2)”

- stimulus — “requests new site made active with 10-100 users”
- source of stimulus — “the owner”
- environment — “normal working conditions”
- artifact — “the system”
- response — “new site is activated
- response measure — “downtime does not exceed 48 hours”

Summary

- Most system quality requirements can be expressed in terms of the system quality attributes
- In order to know whether a given architecture is adequate, the quality requirements need to be presented in an operational manner
- Quality Attribute Scenarios provide a means to specify quality requirements operationally

SOFTENG 325:
Software Architecture

Part II, Lecture 2a: Quality Attribute Scenarios

Ewan Tempero
Department of Computer Science

Potential Assessment Question

Which of the following quality attributes best describe “Extensibility”?

(a) Usability
(b) Modifiability
(c) Security
(d) Interoperability
(e) None of the above.

Justify your answer.

Agenda

- PAQ
- Admin
  - Next week $\Rightarrow$
    - Wednesday — reading must be completed
- Determining Quality attributes
- General Quality Attribute Scenarios
- **Reading**
  - *The architecture of open source applications* Amy Brown & Greg Wilson (editors), Volume II, Chapter 1
- **References**

  Software Architecture in Practice Bass et al. — Chapter 4.

Previously in SOFTENG 325

- We care about quality requirements because they most influence choice of architecture
- There are many aspects to quality, which we describe as *quality attributes* (QA)
- A quality attribute is a **measurable** or **testable** property of a system that is used to indicate how well the system satisfies the needs of its stakeholders
- For a QA to be measurable or testable, we need to describe the requirements with respect to the QA in an operational form.
- Quality attribute scenarios (QAS) provide an operational description of quality requirements — unambiguous and testable
- A QAS has six parts: stimulus, source, artifact, environment, response, measure.
  - two QASs that are the same except in the value of one part may lead to different choices of architecture

Finding Scenarios

- There are lots of possible stimuli, possibly multiple sources for any given stimulus, many different environmental factors, and numerous other variations
- Do we list them all or if not, how do we ensure we've got the "important" ones?

  $\Rightarrow$ the client dictates which are the important ones

  $\Rightarrow$ abstract the possible variations and list all possible abstractions — general scenarios

General vs Concrete Scenarios

- general — system independent
- concrete — specific to a system under consideration, typically many for the same general scenario
- set of concrete scenarios = “non functional requirements” (or at least those that are not fixed design decisions)

Example: General Scenario

- stimulus — sporadic events arrive
- source of stimulus — independent source
- artifact — system
- environment — normal mode
- response — process the stimulus
- response measure — latency

  - quality attribute this applies to is determined by (typically) choice of response measure and stimulus
  - no indication of what system this applies to (because it applies to many systems)

Requirements Elicitation

- for each quality attribute . . .
- . . . consider all possible values for each scenario part . . .
- . . . to create a general scenario . . .
- . . . and then derive concrete by choosing system specific values.

Performance: possible values

- stimulus — periodic events arrive; sporadic events arrive; stochastic events arrive
- source of stimulus — internal to system; external to system
- artifact — system; one or more components
- environment — operational mode: normal; emergency; peak load; overload
- response — processes stimuli; changes level of service
- response measure — latency; deadline; throughput; jitter; miss rate

Performance: possible values

- stimulus — periodic events arrive; sporadic events arrive; stochastic events arrive
- source of stimulus — internal to system; external to system
- artifact — system; one or more components
- environment — operational mode: normal; emergency; peak load; overload
- response — processes stimuli; changes level of service
- response measure — latency; deadline; throughput; jitter; miss rate

Performance Stimuli

- periodic — stimuli occur at regular (and known) intervals (“predictable”)
- sporadic — time between stimuli is completely unpredictable (“random”)
- stochastic — time between stimuli follows particular probability distribution (“likelihood increases over time”)

Performance response measures

- latency — time between stimulus and response
  - page must be shown within 2 seconds of link clicked
- deadline — response must be by specific time independent of the stimulus
  - information must be updated an refreshed at each minute boundary
- throughput — rate at which stimuli are responded to
  - must be able to record 60 frames per second
- jitter — variation in successive responses
  - Display must be updated every 10-15 seconds
- miss rate— events not processed
  - Must provide responses to 99.99% page requests.

Example: General Scenario

- stimulus — sporadic events arrive
- source of stimulus — independent source
- artifact — system
- environment — normal mode
- response — process the stimulus
- response measure — latency

Example: General Scenario

- stimulus — sporadic events arrive
  ⇒ *form submission*
- source of stimulus — independent source
  ⇒ *the user*
- artifact — system
  ⇒ *system*
- environment — normal mode
  ⇒ *server is not overloaded, no network congestion*
- response — process the stimulus
  ⇒ *load and display page indicating successful purchase*
- response measure — latency
  ⇒ *the page is loaded in less than 2 seconds*

General Scenario: Availability

- stimulus — fault: omission, crash, incorrect timing, incorrect response
- source of stimulus — internal to system, external to system: hardware, software, physical infrastructure, physical environment
- artifact — system's processors, communication channels, persistent storage, processes
- environment — normal operation, startup, shutdown, repair mode, degraded operation, overloaded operation
- response — Prevent fault from becoming failure; Detect the fault:
  - log the fault
  - notify appropriate people or systems

  Recover from the fault:
  - disable sources of events that cause fault
  - fix or mask the fault/failure, or contain damage
  - be unavailable repair for a pre-specified interval
  - continue to operate in degraded mode during repair
- response measure — time interval when the system must be available, availability percentage, time to detect fault, time to repair fault, time interval in which system can be in degraded mode, proportion or rate of class of faults that are prevented or handled without failing

General Scenario: Availability

- stimulus — fault: omission, crash, incorrect timing, incorrect response
- source of stimulus — internal to system, external to system: hardware, software, physical infrastructure, physical environment
- artifact — system's processors, communication channels, persistent storage, processes
- environment — normal operation, startup, shutdown, repair mode, degraded operation, overloaded operation
- response — Prevent fault from becoming failure; Detect the fault:
  - log the fault
  - notify appropriate people or systems

  Recover from the fault:
  - disable sources of events that cause fault
  - fix or mask the fault/failure, or contain damage
  - be unavailable repair for a pre-specified interval
  - continue to operate in degraded mode during repair
- response measure — time interval when the system must be available, availability percentage, time to detect fault, time to repair fault, time interval in which system can be in degraded mode, proportion or rate of class of faults that are prevented or handled without failing

General Scenario: Availability

- stimulus — crash
- source of stimulus — internal to system
- artifact — system's processors
- environment — normal operation
- response — System should detect stimulus and notify appropriate parties
- response measure — availability percentage

General Scenario: Availability

- stimulus — crash
  ⇒ *single processor failure*
- source of stimulus — internal to system
  ⇒ *random event*
- artifact — system's processors
  ⇒ *server*
- environment — normal operation
  ⇒ *normal working conditions*
- response — System should detect stimulus and notify appropriate parties
  ⇒ *notify operator*
- response measure — availability percentage
  ⇒ *no loss of service* (100% availability)

General Scenario: Modifiability

- source of stimulus — end user; developer; system administrator
- stimulus — wishes to add/delete/modify/vary functionality; quality attribute; capacity; technology
- artifact — code; data; interface; platform, environment; resources; configuration; system that interoperates with target system
- environment — at runtime, compile time, build time, design time; initiation time
- response — locates places in architecture to be modified; makes modification without affecting other functionality; tests modification; deploys modification
- response measure — cost in terms of number of elements affected, effort, money; time; extent to which this affects other functions or quality attributes; defects introduced

Exercise

What is the general scenario that would generate this concrete scenario?

- source of stimulus — “the owner”
- stimulus — “requests new site added with 10-100 users”
- environment — “normal working conditions”
- artifact — “the system”
- response — “new site is provided”
- response measure — “downtime does not exceed 48 hours”

Effectiveness of QAS

- quality attribute scenarios provide a *tool* to help identify architectural requirements
- not a lot of reported use, so effectiveness not yet established
- better than nothing!
- *a "correct" scenario is one that unambiguously conveys useful information about the quality requirements of the system*

$\Rightarrow$ Don't feel constrained by the values given by the general scenarios

Summary

- Quality Attribute Scenarios provide an operational description of quality requirements
- General scenarios provide framework for identifying concrete scenarios

SOFTENG 325:
Software Architecture

Part II, Lecture 2b: Architecture Structures

Ewan Tempero
Department of Computer Science

Potential Assessment Question

Which of the following statements is most correct with respect to the reasons general quality attribute scenarios exist?

(a) General scenarios are necessary for determining the relevant concrete scenarios for a system.

(b) General scenarios are necessary only when the architecture type being used is not a standard one.

(c) General scenarios are necessary for determining the relevant architecture for a system.

(d) General scenarios are completely optional when determining the relevant concrete scenarios for a system.

(e) None of the above.

Justify your answer.

Agenda

- PAQ
- Defining architectures — structures
- Kinds of structures
- **References** *Software Architecture in Practice* Bass et al. — Chapter 1.

What is a Software Architecture

The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.

(Bass, Clements, Kazman *Software Architecture in Practice*)

Architectural Structures and Views

- structure — a set of coherent set of elements and relations between them
- view — a representation of a structure
- an architecture will typically consists of *multiple structures*
- there can be *multiple views* of a structure
- structures are usually described by a set of views

Structure types

Module — **elements** are modules (units of implementation) representing functional responsibility, **relationships** include *uses*, *inherits*, is *submodule*

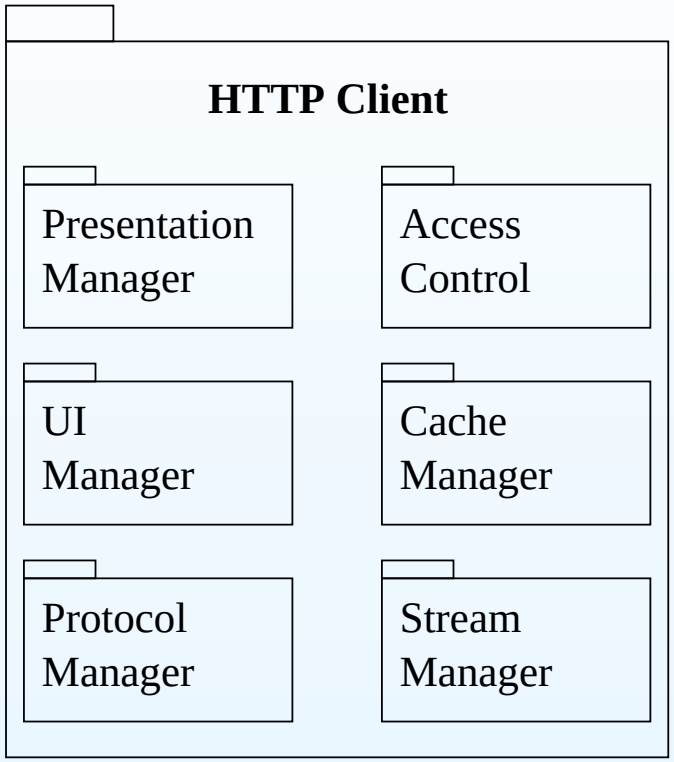
Component-and-connector — **elements** are run-time components (units of computation), **relationships** include *connectors* (communications channels)

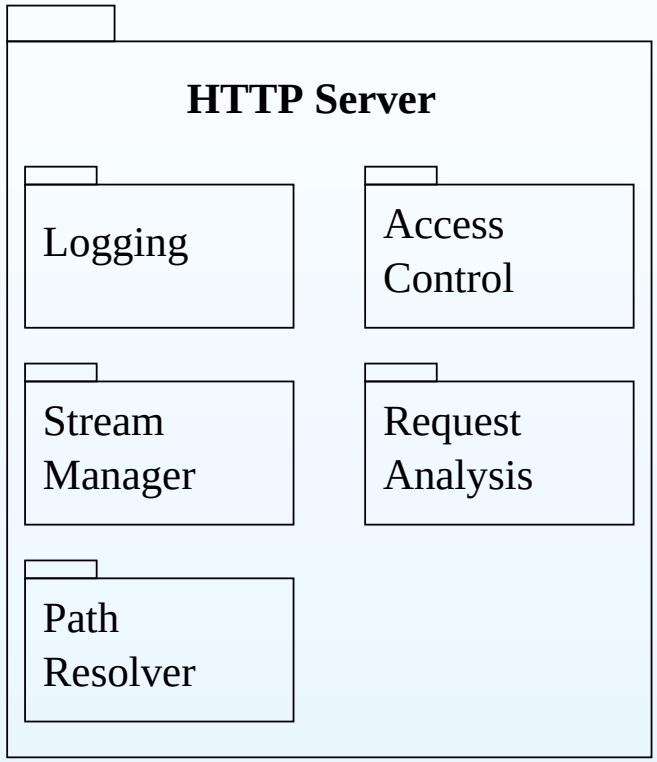
Allocation — **elements** are software elements and environment elements (e.g., hardware systems, teams), **relationships** include *executes on*, *is tested by*

Module Structure Examples

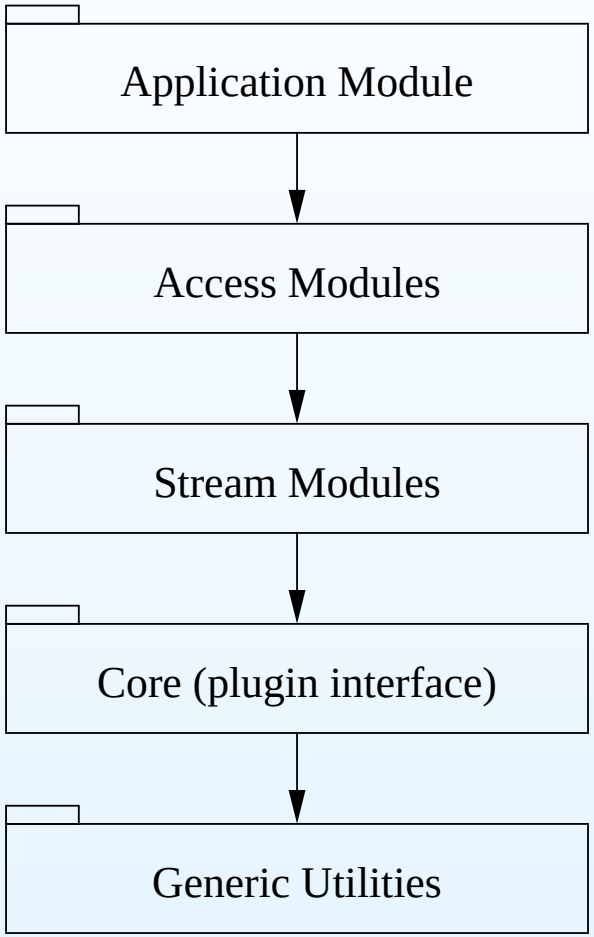
- uses — relationship is “uses” (i.e., needs the existence of), useful for identifying functional subsets
- decomposition — relationship is “is a submodule of”, useful for organisation
- generalisation — relationship is “generalises” (i.e., inherits from)
- layered — restricted form of “uses”, layer n uses only the services of layer $n - 1$
- data model — data entities and their relationships

Decomposition Example





Layered Example



caching, logging, proxy management

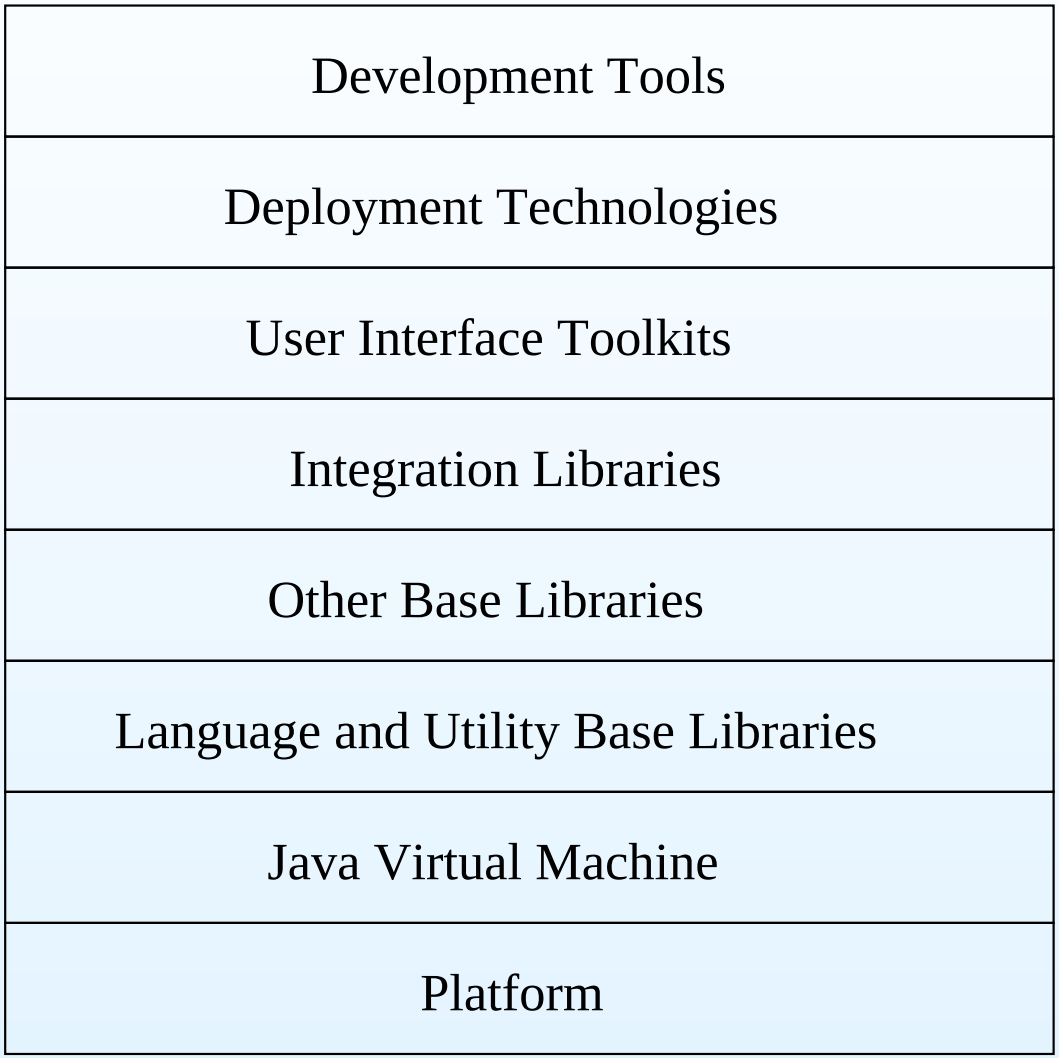
protocol management

transport of data to network

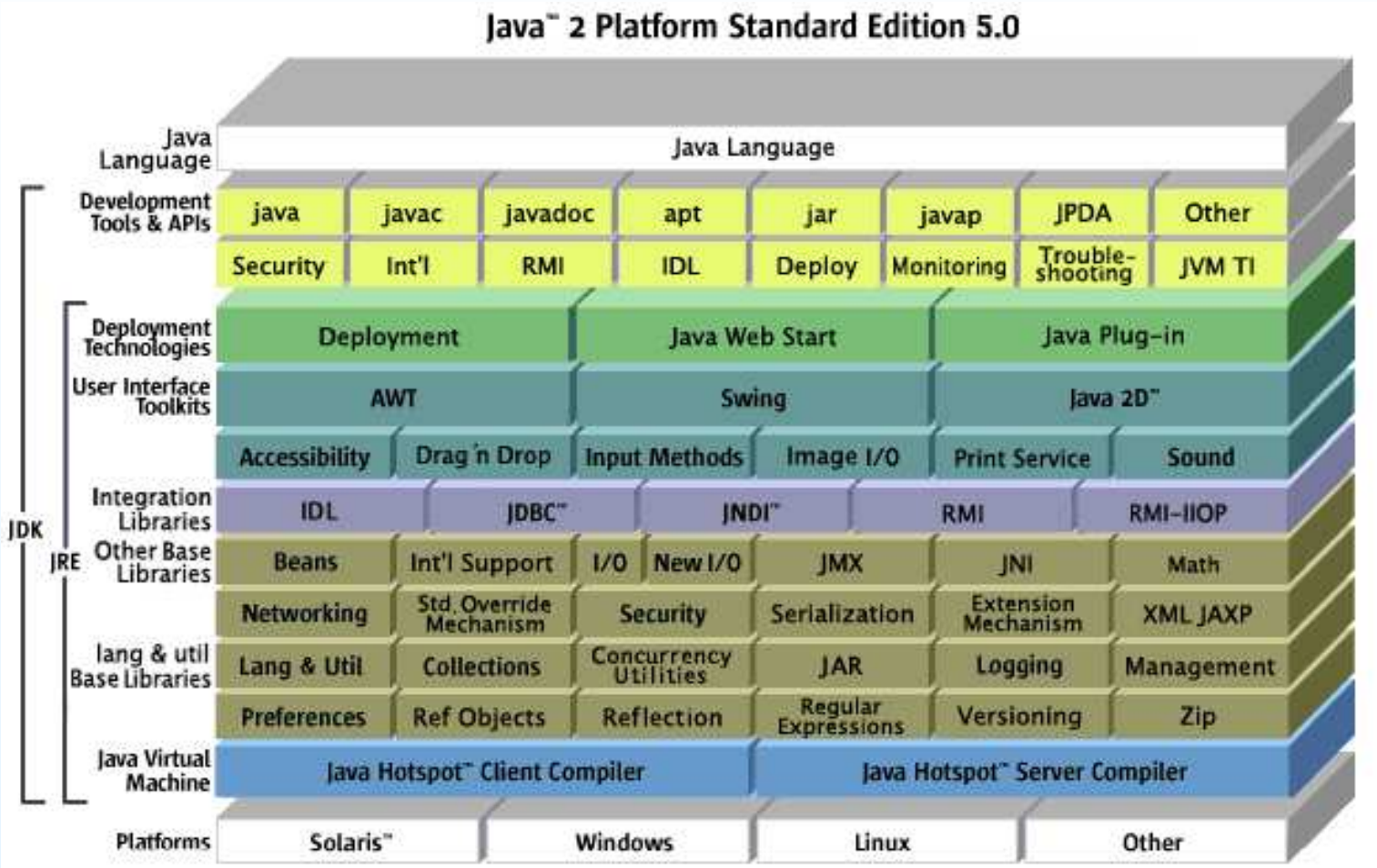
network access, parsing, logging

network management, string manipulation

Example 2



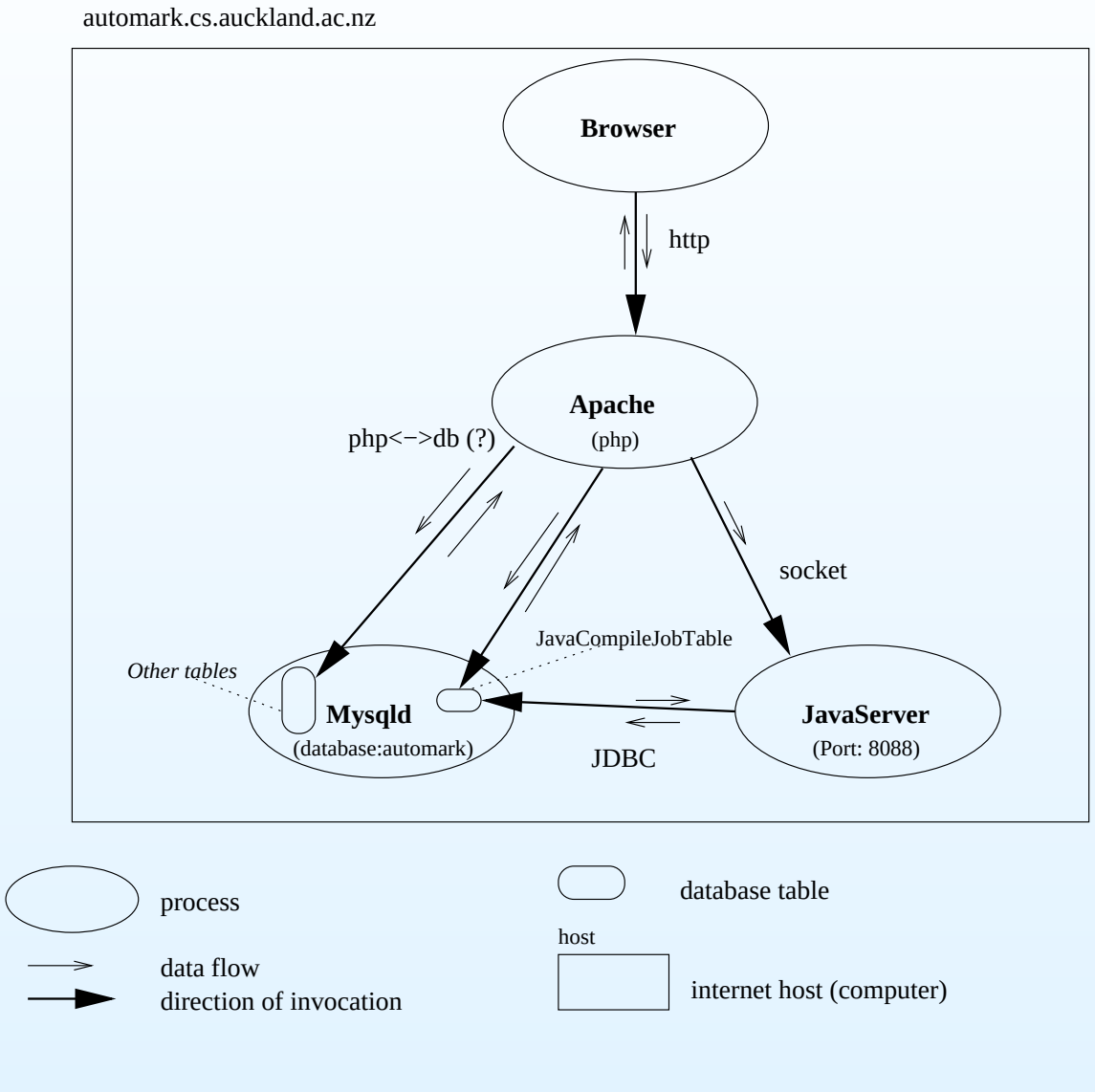
Example 2: Different View



Component-and-Connector Structure Examples

- concurrency — processes/threads that (can) run in parallel, connected by communication, synchronisation, and/or exclusion
- service — elements are services that interoperate with each other by service coordination mechanisms

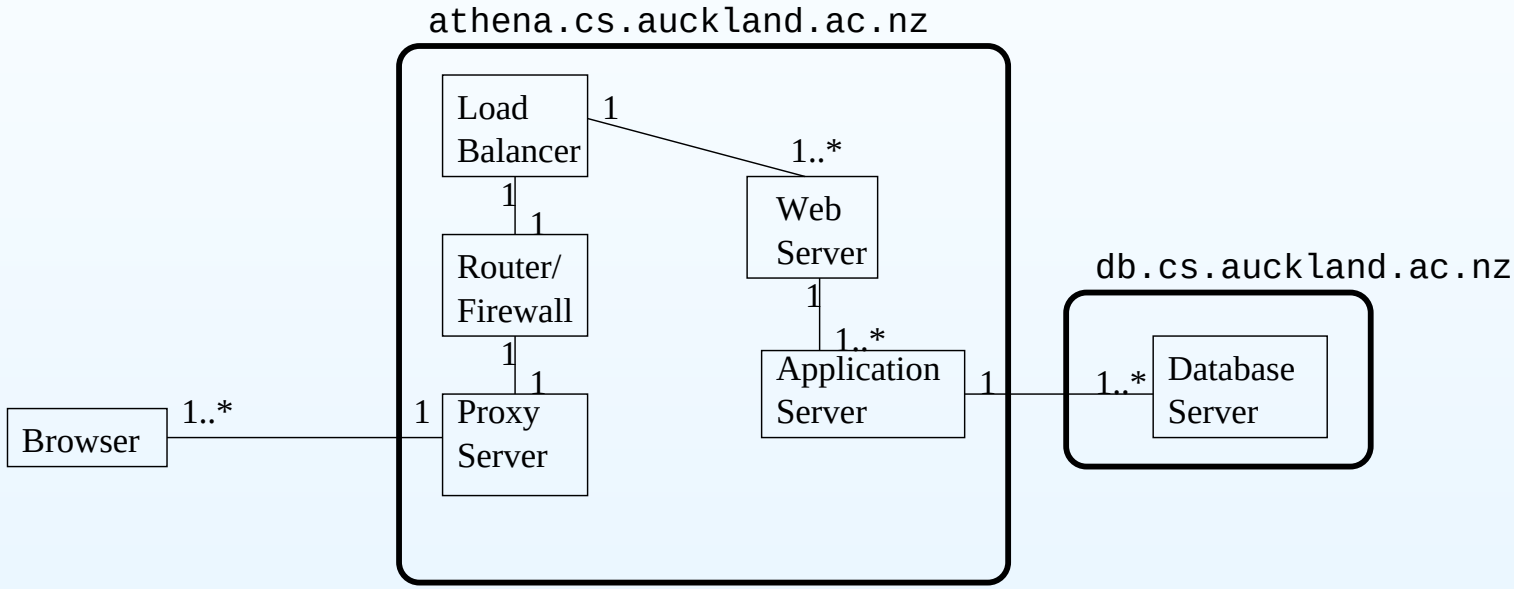
Processes Example



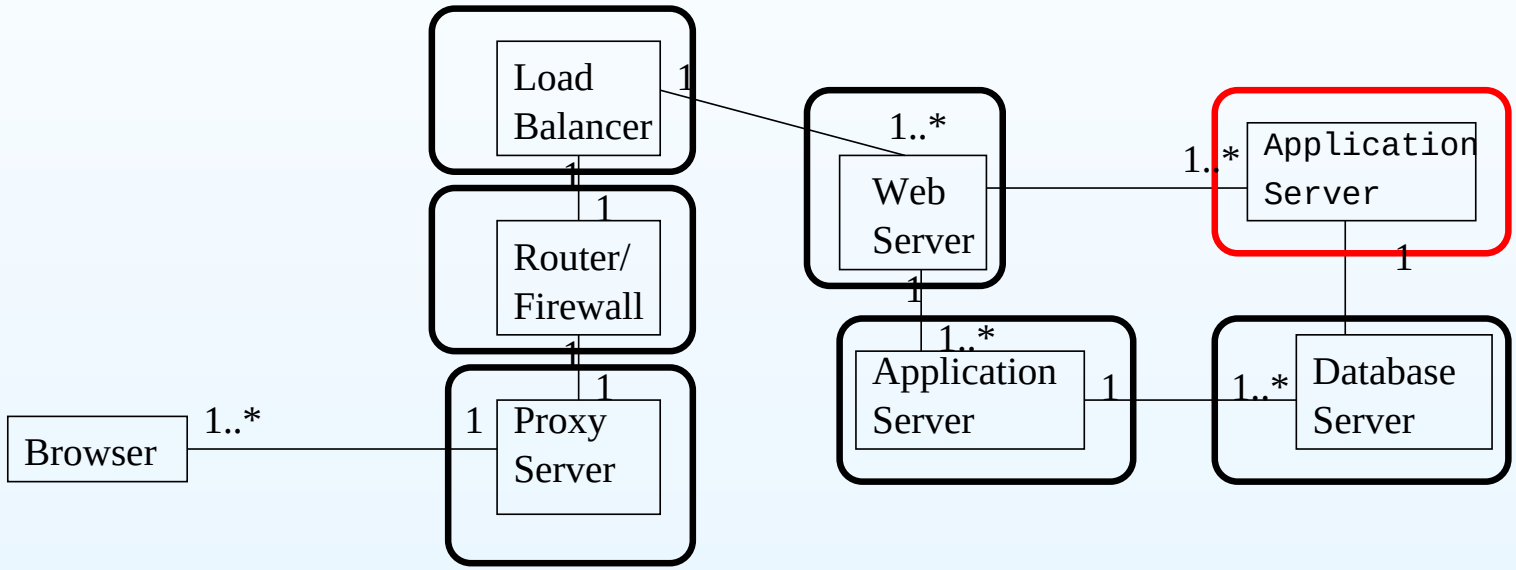
Allocation Structure Examples

- deployment — show software is assigned to computation and communication elements (including migration)
- implementation — how software and other control information is assigned to physical files, *or what modules make up which processes*
- work assignment — who's implementing (testing, documentation) what

Deployment Example

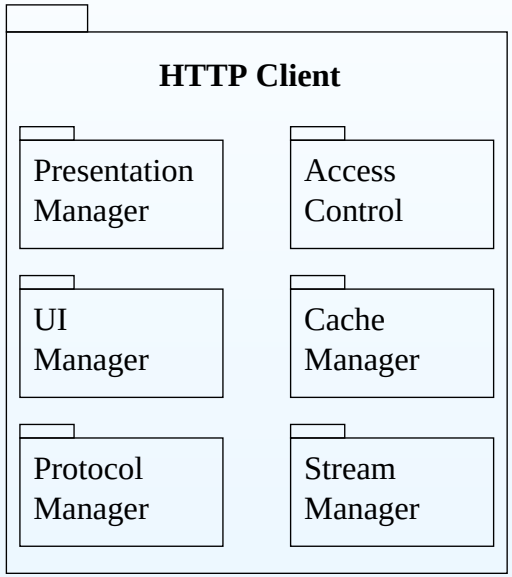


Deployment Example

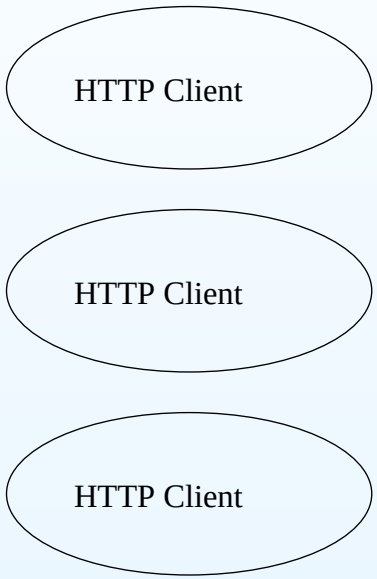


Deployment Example

Modules

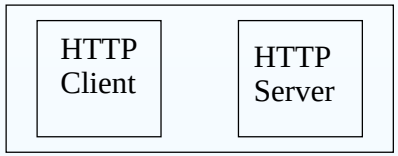


Processes

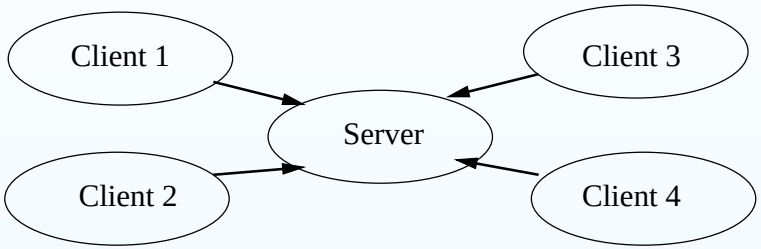


Relationship between Structures

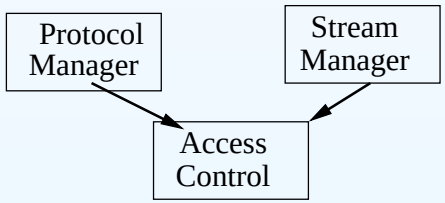
Module Structure (Decomposition)



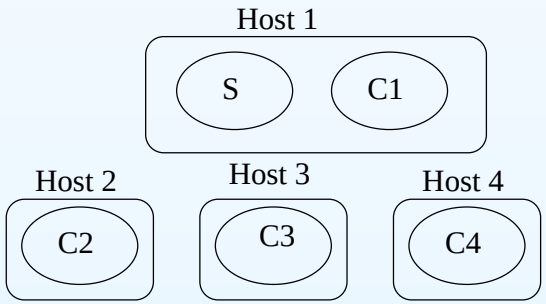
Component and Connector Structure



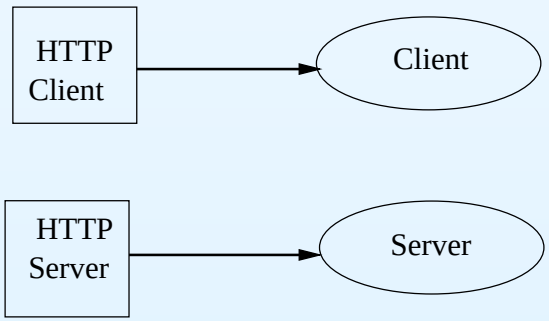
Module Structure (Uses)



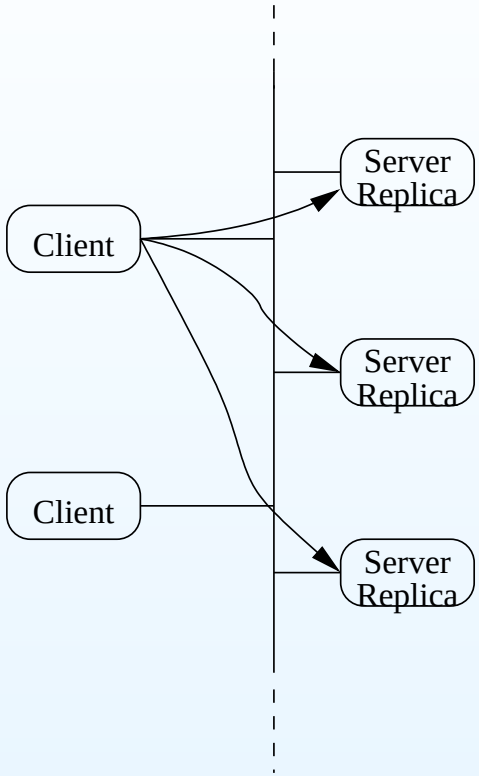
Allocation Struture (Deployment)

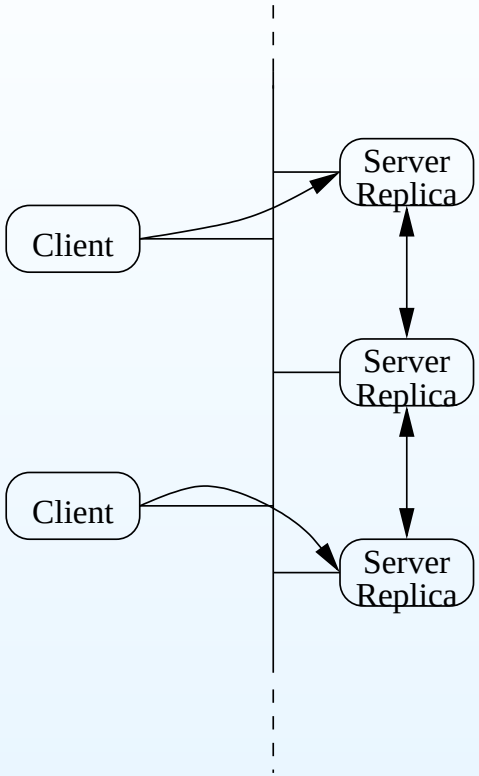


Allocation Structure (Implementation)



Class Exercise





- What kinds of structures?

Use of Structures/Views

- In order to determine whether or not an architecture meets modifiability scenarios relating to change of functionality, what kinds of software elements should be considered?
- In order to determine whether or not an architecture meets modifiability scenarios relating to change run-time performance behaviour, what kinds of software elements should be considered?
- In order to determine which things to run on the same machine, what kinds of relationships should be considered?
- In order to determine whether or not an architecture meets its security goals, what kinds of relationships should be considered?

Summary

- An architecture has (typically) multiple structures
- Structures have elements and relationships between them relevant to reasoning about the architecture
- Structures (and architecture) are abstract, so we typically deal with (concrete) views of a structure
- Different structures (different sets of elements and relationships) are useful for reasoning about different quality attributes

SOFTENG 325:
Software Architecture

Part II, Lecture 3a: Tactics — Achieving Qualities

Ewan Tempero
Department of Computer Science

Potential Assessment Question

In the course, we discussed 3 types of structures: *module*, *component and connector*, and *allocation*. Explain why we need each of the three types to describe most interesting architectures. Give examples.

Agenda

- PAQ
- Admin
  - Assignment 2?
  - Wednesday — reading must be completed
- Tactics — how to achieve quality attribute requirements
- Tactics for Availability
- **Reading**
  - *The architecture of open source applications* Amy Brown & Greg Wilson (editors), Volume II, Chapter 1
- **References**

  Software Architecture in Practice Bass et al.

  — Chapters 4, 5.

Creating Architectures

- We know how to specify the requirements that determine which architectures are sufficient — quality attribute scenarios
- We know how to describe architectures — views of structures
- How do we **create** architectures?

  $\Rightarrow$ tactics

Tactics

- design decisions that affect quality attributes
- affecting quality attributes means controlling the responses to stimuli
- tactics are sometimes related
- tactics often affect more than one quality attribute (not always positively)

Example: Performance tactic

- use multiple processors to process events
  - $\Rightarrow$ reduce time to process events (latency)
  - $\Rightarrow$ increase cost
  - $\Rightarrow$ sychronisation becomes a concern
  - $\Rightarrow$ communication bandwidth may become a concern
- (so, provide multiple copies of data sets to reduce network contention)
- same tactic can be used to increase availability

Tactics for Improving Availability

- system failure — system does not deliver expected services *as observed by user*
- system fault — an incorrect aspect of the system that may lead to system failure
- faults may be corrected or hidden from the user
- a system that has failed is not available until it is repaired
- availability (metric) — probability the system delivers services when expected

$$\text{availability} = \frac{\text{mean time to failure}}{\text{mean time to failure} + \text{mean time to repair}}$$

What makes up “Time to Repair”

- problem recognition time
- administrative delay time
- maintenance tools collection time
- problem analysis time
- change specification time
- change time (including testing and review)

General Scenario: Availability

- stimulus — fault: omission, crash, incorrect timing, incorrect response
- source of stimulus — internal to system, external to system: hardware, software, physical infrastructure, physical environment
- artifact — system's processors, communication channels, persistent storage, processes
- environment — normal operation, startup, shutdown, repair mode, degraded operation, overloaded operation
- response — Prevent fault from becoming failure; Detect the fault:
  - log the fault
  - notify appropriate people or systems

  Recover from the fault:
  - disable sources of events that cause fault
  - fix or mask the fault/failure, or contain damage
  - be unavailable repair for a pre-specified interval
  - continue to operate in degraded mode during repair
- response measure — time interval when the system must be available, availability percentage, time to detect fault, time to repair fault, time interval in which system can be in degraded mode, proportion or rate of class of faults that are prevented or handled without failing

Improving Availability

- types of faults (stimuli)
  omission component fails to respond to input
  crash component repeatedly suffers omission faults
  timing component responds but the response is early or late
  response component responds with incorrect value
- to increase availability, increase mean time to failure or reduce mean time to repair
- increase mean time to failure by protecting from possible faults, or hiding faults so that they don't lead to failure
- reduce mean time to repair by reducing problem recognition time, reducing administrative delay, (etc)

Goals of availability tactics

- monitor health to detect faults or failures
- (automatically) recover from faults or failures
- prevent faults or failures

Omission/Crash Tactics

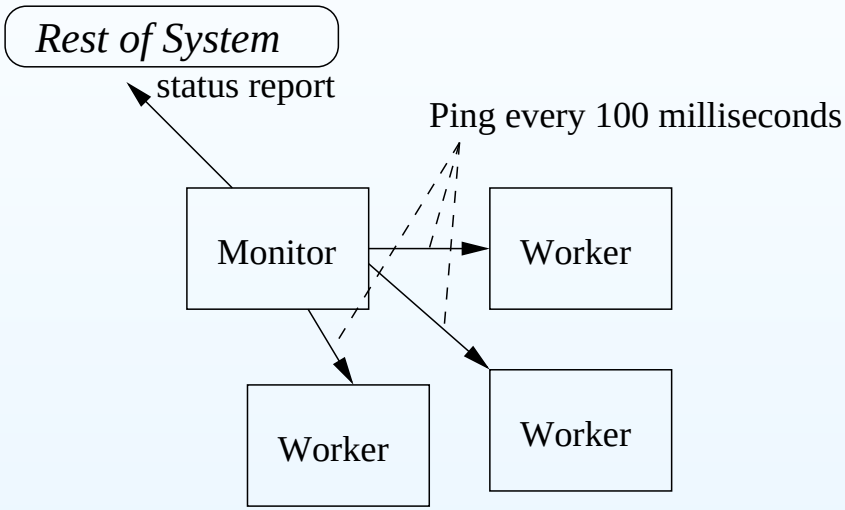
detection

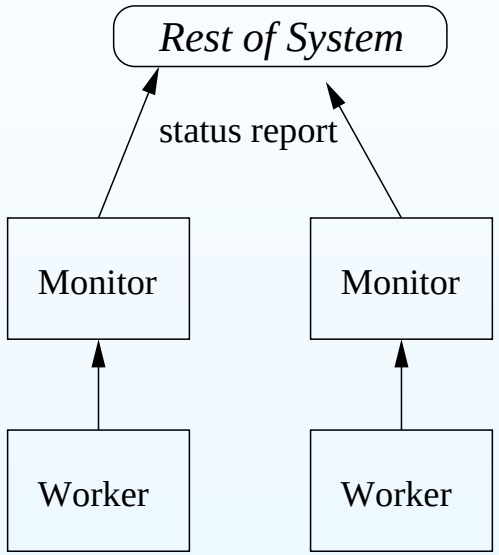
- ping — “are you alive”?
- heartbeat — “I am alive, really”
- exception — “I died!”

repair

- voting — redundant subsystems with arbitrator to decide which subsystem is to be ignored (failed)
- active redundancy (hot restart) — keep multiple systems in identical state, any one can be use
- passive redundancy — non-primary systems are notified of state changes, explicitly make one of these primary when failure detected
- spare — separate system that must be started and put in correct state after failure detected

Example application of tactics





heatbeat every 0.5 seconds

Omission

reintroduction of failed component

- shadow — run in parallel to check correction works
- state resynchronisation — ensure state matches current state
- rollback — keep record of valid states

prevention

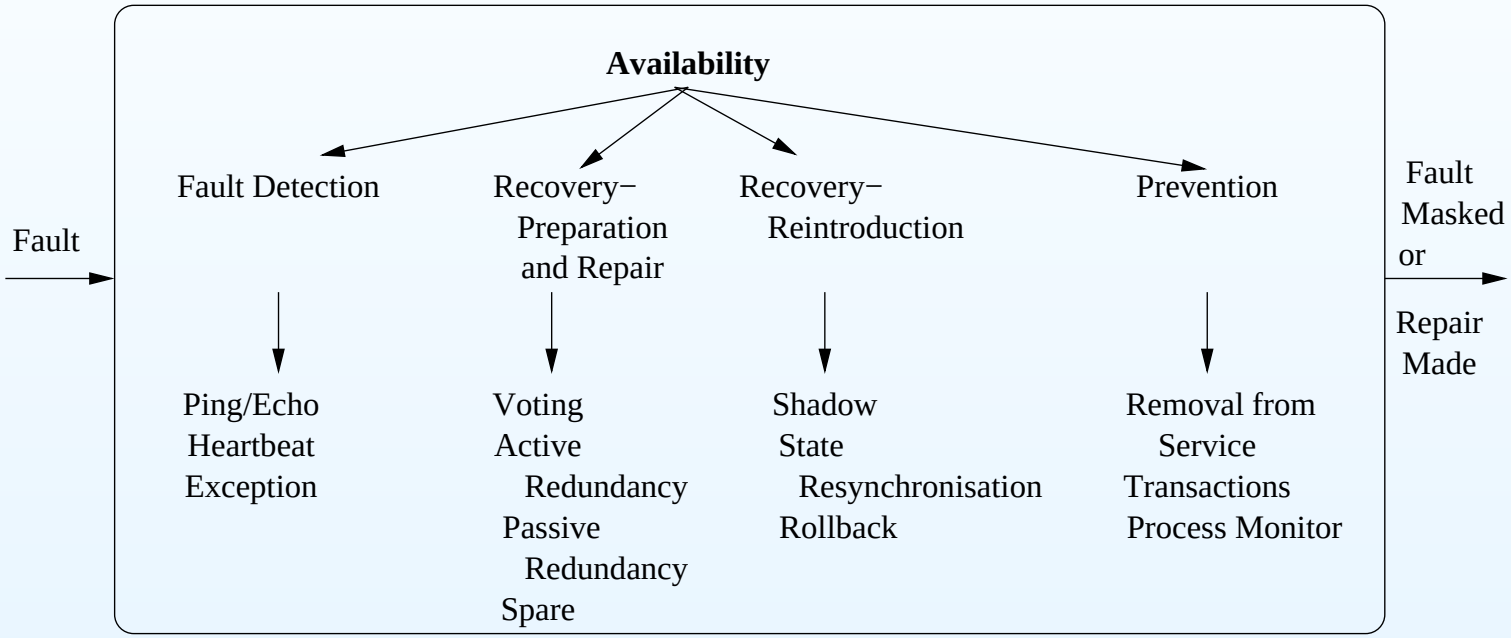
- removal from service — scheduled downtime to deal with potential problems
- transactions — only allow “valid” states, rollback otherwise
- process monitor — determine that process is in faulty state before failure occurs

Class exercise: On-line Travel Booking

Part	Details
Stimulus	A crash occurs
Source	. . . due to a random event
Artifact	. . . to one of the servers
Environment	. . . with no other system or networking problems.
Response	The failure is recorded
Measure	. . . within 10 minutes and there is no loss of service.

- What might be useful tactics to consider when developing an architecture that has to meet this requirement?

Availability Tactics Summary



SOFTENG 325:
Software Architecture

Part II, Lecture 3b: Tactics for Performance and Modifiability

Ewan Tempero
Department of Computer Science

Potential Assessment Question

Consider a system for providing information on when the next bus will arrive at a bus stop (such as Auckland Transport's Real time information board). A central system uses location information from buses to determine estimated arrival times, which it then sends to displays at bus stops. Which of the following tactics would be most suitable for dealing with the failure of the communications link from the central system to the displays?

(a) Voting
(b) Shadow
(c) Process Monitor
(d) Heartbeat.
(e) None of the above.

Bonus points: Why is the correct answer still not a very good choice?

Agenda

- Performance tactics
- Modifiability tactics
- **References**

  Software Architecture in Practice Bass et al.

  — Chapters 4, 7, 8.

Performance: possible values

- stimulus — periodic events arrive; sporadic events arrive; stochastic events arrive
- source of stimulus — internal to system; external to system
- artifact — system; one or more components
- environment — operational mode: normal; emergency; peak load; overload
- response — processes stimuli; changes level of service
- response measure — latency; deadline; throughput; jitter; miss rate

Performance factors

Processing time

- time during which system is doing work towards response
- processing consumes resources, which takes time depending on the nature of the resources
- as resource utilisation approaches 100%, time taken can increase
- some of the processing is not directly task related (e.g. marshalling parameters for network communication)

blocked time

- time during which no progress is made due contention for resources, unavailable resources, waiting for other work to be completed

Performance

- affect time of response
  - $\Rightarrow$ resources consumed
  - $\Rightarrow$ resources needed
- goals
  - resource demand
  - resource management
  - resource arbitration

Resource Demand

- reduce resources required for processing stimuli
  - increase computational efficiency
  - reduce computational overhead
- reduce number of events to process
  - manage event rate
  - control frequency of sampling
- control resources consumed
  - bound execution times
  - bound queue sizes

On-line Travel Booking

- increase computational efficiency — use faster search algorithms
- reduce computational overhead — co-locate processes so as not to pay the cost of remote communication
- manage event rate — not applicable(?)
- control frequency of sampling — not applicable(?)
- bound execution times — if a search takes too long just give up
- bound queue sizes — if queueing search requests on a processor and its queue is full just drop new jobs

Resource Management

- introduce concurrency
- maintain multiple copies of either data or computations
- increase available resources

On-line Travel Booking

- introduce concurrency — have multiple servers processing requests
- maintain multiple copies of either data or computations — have multiple copies of the data about flights, one per server
- increase available resources — use multiple communications links

Resource Arbitration

- contention $\Rightarrow$ resource must be scheduled
- scheduling $=$ priority assignment $+$ dispatching
- scheduling criteria included...
  - optimal resource usage
  - minimising number of resources used
  - ensure fairness
  - prevent starvation
  - optimal processing of “important” requests
- dispatching may require preemption
  - can occur any time
  - can occur only at specific preemption points
  - can occur only when nothing else is executing

Scheduling Policies

- first-in/first-out
- fixed-priority
  - semantic importance
  - deadline importance
  - rate monotonic
- dynamic priority
  - round robin
  - earliest deadline first
- cyclic executive (static)

Class Exercise

- One option for improving performance is to use a caches. What tactics are associated with caches?

General Scenario: Modifiability

- source of stimulus — end user; developer; system administrator
- stimulus — wishes to add/delete/modify/vary functionality; quality attribute; capacity; technology
- artifact — code; data; interface; platform, environment; resources; configuration; system that interoperates with target system
- environment — at runtime, compile time, build time, design time; initiation time
- response — locates places in architecture to be modified; makes modification without affecting other functionality; tests modification; deploys modification
- response measure — cost in terms of number of elements affected, effort, money; time; extent to which this affects other functions or quality attributes; defects introduced

Modifiability

Change happens

- what can change? — functionality, hardware, operation system, middleware, protocols, qualities (performance, reliability, modifiability), capacity
- what is the likelihood of change? — designing for change that won't happen wastes effort
- when is the change made and who makes it? — the developer, system administrator, end user, design time, implementation time, compile time, build time, configuration (installation) time, start time, execution time
- what is the cost of change? — introducing a mechanism to support change; making the change

Cost of making changes

- the bigger the unit being changed, the more it costs — more to understand, more to check, more likely the change will impact another unit
- if multiple units have to be changed, it costs more than changing a single unit the size of their sum — have to understand interactions
- the more likely a change in one unit forces a change in another, the more it costs
- the more the change of a unit is under the control of a computer, the less it costs — faster, fewer errors, cheaper people to do the change (e.g. end users)

Modifiability tactics: goals

- coupling — reduce the probability that a change in one module will impact another module
- cohesion — maximise the probability that if some part of a module changes, it all changes
- defer binding time — increase the ability of the computer to manage the change (*binding*: determining what a name means)

Managing Dependencies

- dependency — if changing A involves B in some way then A depends on B
- different kinds of dependency
  - syntax
  - semantics
  - ordering
  - location
  - existence
  - resource requirements
  - ...
- dependencies are not necessarily bad — no dependencies means no system

Tactics to increase cohesion

- split module — separate unrelated responsibilities
- increase semantic coherence — combine related responsibilities
- anticipate expected changes — located things likely to change in minimum number of modules

On-line Travel Booking

What can change?

- system has to communicate with other systems (airlines, travel agents, hotels, rental) — don't have different modules for communicating with each different system, use a single module (increase semantic coherence)
- flights for an airline will change; airlines, travel agents, hotels, rental companies that want to connect will change — have module to manage change in connection (increase semantic coherence, anticipate expected changes
- BUT of type of connection is quite different (e.g. socket versus web service) keep in separate modules (split modules)
- search algorithms — have a module that is responsible for only searching (maintain semantic coherence, anticipate expected changes

Tactics to reduce coupling

- encapsulate — provide explicit interface to module
- restrict dependencies — hide details (and modules), restrict which modules can be used, require authorisation
- use intermediary — breaks dependencies between modules
- reduce syntactic duplication (“refactor”) — reduce implicit dependencies
- abstract common services — reduce semantic duplication, reduce implicit dependencies

On-line Travel Booking

What can change?

- airline flight information systems — provide a module that presents the same interface (encapsulate, use an intermediary, abstract common services)

Tactics to defer binding

Compile time

- component replacement (e.g. as specified in a build script)
- compiler parameterisation
- aspects

Deployment time

- configuration files

Initialisation

- resource files

Runtime

- runtime registration
- dynamic lookup (names, services)
- plug-ins
- publish-subscribe
- polymorphism
- standard protocols

On-line Travel Booking

What can change?

- which airlines bookings can be made with
  - providing list of airlines at system initiation (configuration files)
  - airlines can join or leave any time (runtime registration)
- What systems must the airlines use to integrate with the on-line travel booking system?
  - doesn't matter so long as they can send and receive the necessary data via a common protocol (standard protocols)

SOFTENG 325:
Software Architecture

Part II, Lecture 4a: Tactics for Security, Usability, and Testability

Ewan Tempero
Department of Computer Science

Potential Assessment Question

Which one of the following statements is **false** regarding the tactic “control frequency of sampling”?

(a) It is a tactic that can improve performance.

(b) It is a tactic that reduces the number of events that have to be processed.

(c) It is a tactic that can reduce availability.

(d) It is a tactic that is intended to influence resource demand.

(e) None of the above.

Agenda

- Security tactics
- Usability tactics
- Testability tactics
- **References**

  Software Architecture in Practice Bass et al.

  — Chapters 4, 9, 10, 11.

Security Properties

- Confidentiality — the property that data or services are protected from unauthorized access.
- Integrity — the property that data or services are being delivered as intended.
- Availability — the property that the system will be available for legitimate use.
- Authentication — the property that the parties to a transaction are who they purport to be.
- Nonrepudiation — the property that a transaction (access to or modification of data or services) cannot be denied by any of the parties to it.
- Authorisation — the property that a actor has the necessary privileges to perform a task

Security General Scenarios

Security is a measure of the system's ability to resist unauthorized usage while still providing its services to legitimate actors.

Part	Details
Source	Individual or system that is (correctly identified, identified incorrectly, of unknown identity) who is (internal/external, authorized/not authorized) with access to (limited resources, vast resources)
Stimulus	Tries to (display data, change/delete data, access system services, reduce availability to system services) ("threat" or "attack")
Artifact	System services; data within/produced/consumed by system
Environment	Either: online or offline, connected or disconnected, firewalled or open
Response	Authenticates actor; hides identity of the actor; blocks access to data and/or services; allows access to data and/or services; grants or withdraws permission to access data and/or services; records access/modifications or attempts to access/modify data/services by identity; stores data in an unreadable format; recognizes an unexplainable high demand for services, and informs a actor or another system, and restricts availability of services
Response Measure	Time/effort/resources required to circumvent security measures with probability of success; probability of detecting attack; probability of identifying individual responsible for attack or access/modification of data and/or services; percentage of services still available under denial-of-services attack; restore data/services; extent to which data/services damaged and/or legitimate access denied

Security Goals

- Detect attacks
- Resist attacks
- React to attacks
- Recover from attacks

Detecting Attacks

- **Detect instrusion** — examine communication or service requests within the system for unusual patterns
- **Detect service denial** — examine communication or service requests from external to the system for unusual patterns
- **Verify message integrity** — confirm that data has not been changed (e.g through use of checksums or similar techniques)
- **Detect message delay** — specific to man-in-the-middle attacks
- **Intrusion detection system** — provide infrastructure to compare current patterns of use with historical use, and identify when the patterns are different.

Resisting attacks

- **Identify actors** — identify the source (users or systems) of external input
- **Authenticate actors** — ensuring that a actor or remote computer is actually who it purports to be.
- **Authorize actors** — ensuring that an authenticated actor has the rights to access and modify either data or services.
- **Limit access** — reduce the opportunity for attacks.
- **Limit exposure** — allocate services to sub-units so that if one unit is compromised, exposure is reduced
- **Encrypt data** — protect data from unauthorized access.

React to Attacks

- **Revoke access** — limit users and uses
- **Lock computer** — prevent users and uses
- **Inform actors** — notify relevant parties

Recover from Attacks

- *restore state* — similar to recovery for **Availability** scenarios
- **Audit Trail** — maintain copy of each transaction applied to the data in the system together with identifying information.

Usability Goals

- **Learning system features** If the user is unfamiliar with a particular system or a particular aspect of it, what can the system do to make the task of learning easier?
- **Using a system efficiently** What can the system do to make the user more efficient in its operation?
- **Minimizing the impact of errors** What can the system do so that a user error has minimal impact?
- **Adapting the system to user needs** How can the user (or the system itself) adapt to make the user's task easier?
- **Increasing confidence and satisfaction** What does the system do to give the user confidence that the correct action is being taken?

Usability General Scenarios

Part	Details
Source	End user, possibly in specialised role
Stimulus	Wants to learn system features; use system efficiently; minimize impact of errors; adapt system; feel comfortable
Artifact	System; Part of system
Environment	runtime; configuration time
Response	Provides features needed; anticipates needs
Measure	Task time, number of errors, number of tasks accomplished, user satisfaction, gain of user knowledge, ratio of successful operations to total operations; amount of time/data lost when error occurs

Support User Initiative

Generally: give the user feedback as to what the system is doing and provide user with the ability to issue usability-based commands

- **Cancel** — ensure that the system can respond in a timely manner
- **Undo** — ensure that the necessary state is recorded and available
- **Pause/resume** — ensure long-running operations can be suspended and resumed
- **Aggregate** — ensure repetitive operations or operations on a large number objects can be aggregated

Support System Initiative

Generally: provide support for the system to predict possible future behaviour and reduce user burden through creating and maintaining models

- **User Model** — determines the user's knowledge of the system, the user's behavior in terms of expected response time, and other aspects specific to a user or a class of users.
- **System Model** — determines the expected system behavior so that appropriate feedback can be given to the user.
- **Task Model** — determines the context so the system can have some idea of what the user is attempting and provide various kinds of assistance.

Testability Goals

- Increase Observability
- Increase Controllability
- Limit Complexity

Testability General Scenarios

Part	Details
Source	Unit developer, Increment integrator, System verifier, Client acceptance tester, System user
Stimulus	Analysis, architecture, design, class, subsystem integration completed; system delivered
Artifact	Piece of design, piece of code, complete application
Environment	At design time, at development time, at compile time, at deployment time
Response	Provides access to state values; provides computed values; prepares test environment
Response Measure	Percent executable statements executed, Probability of failure if fault exists, Time to perform tests, Length of longest dependency chain in a test, Length of time to prepare test environment

Control and Observe State

- **Specialize access routes/interfaces** — allows the capturing or specification of variable values for a component through a test harness as well as independently from its normal execution.
- **Localise state storage** — allow “start” of system in a pre-determined state
- **Abstract data sources** — allow substitution of different data sources for testing purposes
- **Record/playback** — capturing information crossing an interface and using it as input into the test harness (output from one component is input to another)
- **Sandboxing** — isolate instance of system from real world to allow experimenting without adverse consequences
- **Executable assertions** — program statements to check internal state is as expected

Limit Complexity

- **Limit structural complexity** — isolate components (e.g. limiting or resolving cyclic dependencies, or avoid other problematic dependencies), limit dynamic variation (e.g polymorphism, reflection) (See *Modifiability, reduce coupling*)
- **Limit nondeterminism** — limit behavioural complexity

SOFTENG 325:
Software Architecture

Part II, Lecture 5a: Architecture Patterns

Ewan Tempero
Department of Computer Science

Potential Assessment Question

Which of the following best evaluates the statement: “The best way to improve modifiability is to remove all dependencies”

(a) Correct, but you need to have the (Module) Uses structure to do so.

(b) Correct, but you need to have the (Component and Connector) Communicating Processes structure to do so.

(c) Correct, but you need to use one of the tactics for avoiding Ripple Effects.

(d) Not correct.

Agenda

- PAQ
- Admin
  - Assignment 2?
- Architecture Patterns (aka Architecture Styles)
- **References**
  - *Software Architecture in Practice* Bass et al. — Chapter 13
  - *Software architecture : perspectives on an emerging discipline* Mary Shaw, David Garlan. Prentice Hall, 1996.

Bridges Again

More pictures of bridges here

Software Architecture Patterns

- a way to organise architectures
- a vocabulary for discussing architectures
- a meta-level description of architectures
- architecture-level design patterns

Definition

- a package of design decisions (tactics) found repeatedly in practice
- has known properties that permit reuse
- a class or family of architecture meeting same set of constraints
- architecture-level design patterns $\Rightarrow$
  - context — recurring common situation giving rise to a . . .
  - problem — generalised describing of forces that need to be resolved
  - solution — a successful architectural abstract resolution to problem
    - elements types
    - interaction mechanisms
    - topological rules (element relationships)
    - semantic constraints

Common Patterns

Distributed
- client/server, n-tier
- peer-to-peer

Dataflow
- Batch sequential
- Pipes and filters
- Broker

Repositories
- databases
- hypertext systems
- blackboards

Call-and-return
- Main program and subroutine
- object-oriented
- functional
- hierarchical layers

Independent Components
- communicating processes
- event-based

Virtual machines
- interpreters
- rule-based systems

(Not complete! Mostly from Garlan and Shaw)

Broker

Context

- System constructed from distributed services
- Services change locations and protocols

Problem

- How to use services without knowing their identity, location, or nature

Solution

- **Broker** mediates communication between requester of service (client) and service provider
  1. Client sends request to Broker
  2. Broker forward request to service provider
  3. Provider returns response back to Broker
  4. Broker returns response to Client

Broker and Tactics

- increase semantic coherence (cohesion)
- abstract common services (cohesion)
- encapsulate (cohesion)
- restrict communication paths (coupling)
- use an intermediary (coupling)
- raise abstract level (coupling)
- runtime registration (defer binding)

(From *Bachmann, Bass, Nord "Modifiability Tactics" CMU Technical Report SEI-2007-TR-002, 2007*, slightly different names)

Dealing with Pattern Weaknesses

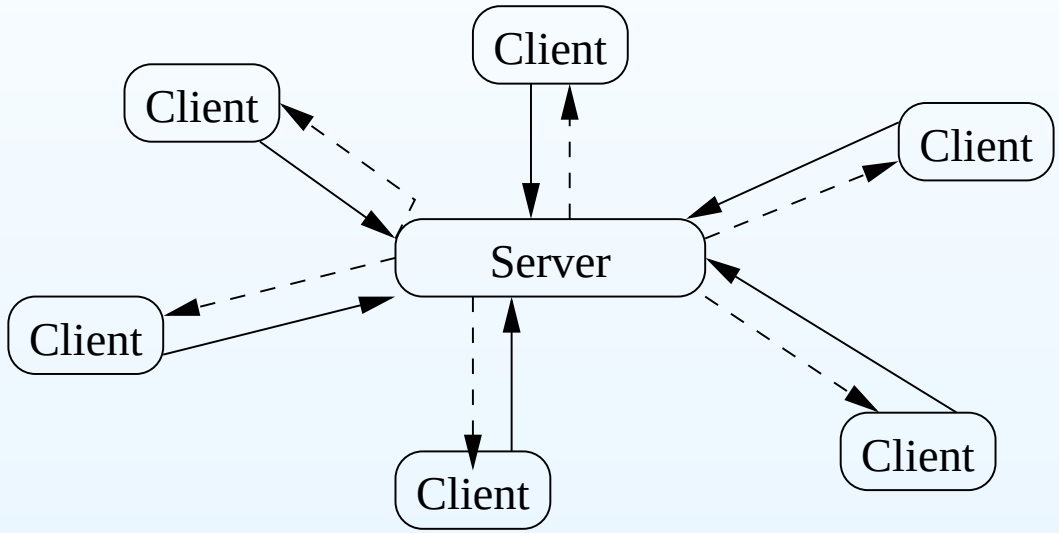
- Architecture Patterns don't necessarily solve all problems
  - trade-offs may not always line up with requirements
  - some quality attributes just not addressed by pattern
- $\Rightarrow$ apply other tactics to deal with weaknesses that matter
- E.g. Broker weaknesses
  - Availability — broker is single point of failure
  - Performance — indirection means overhead $=$ higher latency
  - Testability — context is complex, but no support for testing in pattern
  - Security — more communication $\Rightarrow$ more vulnerabilities
- Possible solutions
  - Availability: Ping/Echo to detect failure, Redundancy (probably active, also supports Performance)
  - Performance: increase available resources, maintain multiple copies of data and computations
  - Testability: Record/playback (also supports Security), specialise routes/interfaces
  - Security: Audit Trail

Client/Server

Elements: *Client* and *Server* processes

Interaction: Standard protocol

Topological rules:



Constraints:

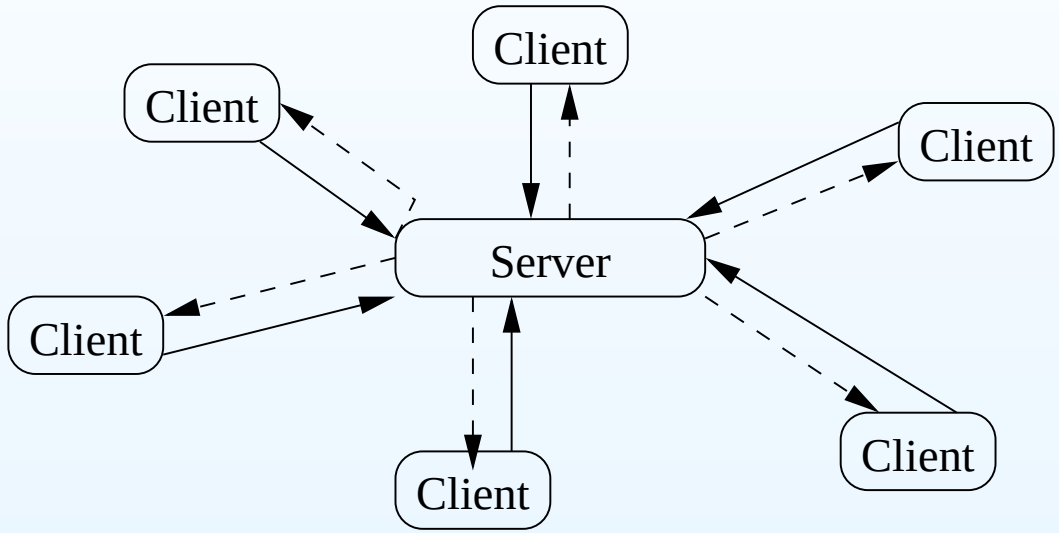
- only communication by server is responses to requests, clients only communicate with server
- server has no knowledge of clients outside of processing requests

Client/Server

Elements: *Client* and *Server* processes

Interaction: Standard protocol

Topological rules:



Constraints:

- only communication by server is responses to requests, clients only communicate with server
- server has no knowledge of clients outside of processing requests

Tactics

Modifiability/Scalability *anticipate expected changes* (more clients), *hide information* (who are the clients, what is the server), *runtime registration*,

Modifiability/Portability *adherence to defined protocols*

Modifiability/Maintainability *abstract common services*, *semantic coherence*

Usability *separate user interface*

Performance *increase available resources* (more processors), (anti) *increase computational overhead* (protocol processing), (anti) *increase contention* (server is bottleneck)

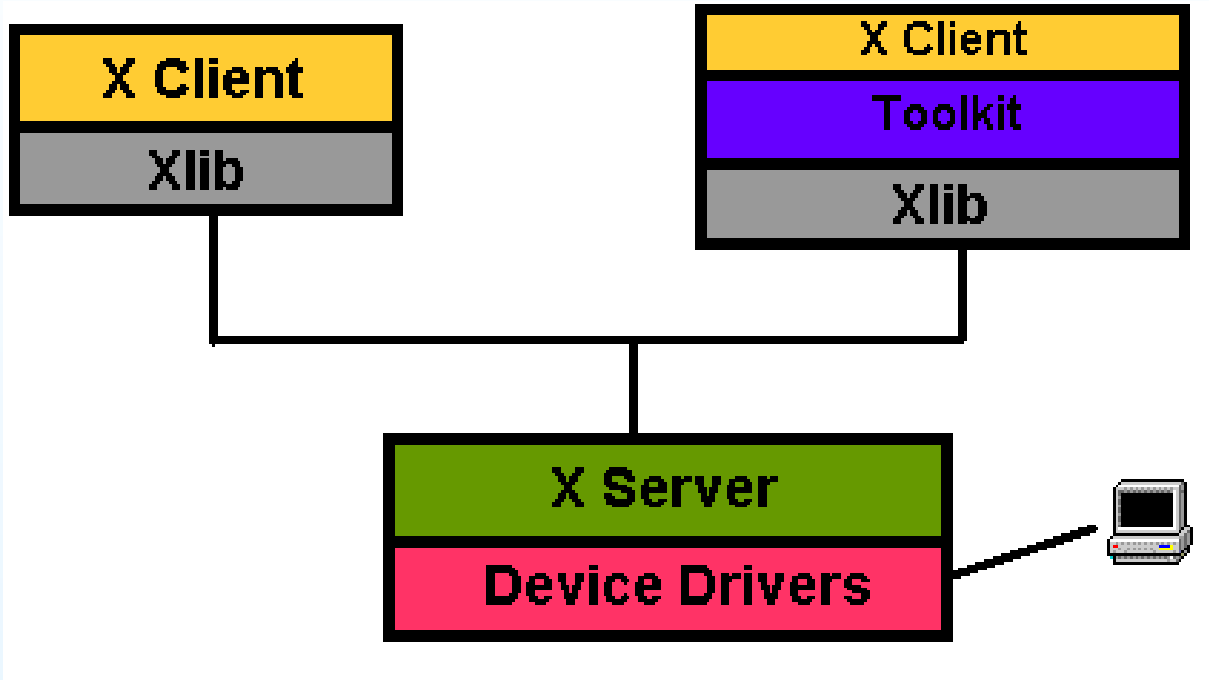
Availability *removal from service*, various forms of *redundancy*

Security (anti) *increases exposure*, (anti) harder to *authenticate users*

Other Quality Attributes

- time to market — provide more functionality only at server or client
- integration with legacy systems — introduce intermediary as server
- deployment — clients easier to install than whole system

Example: X Window System



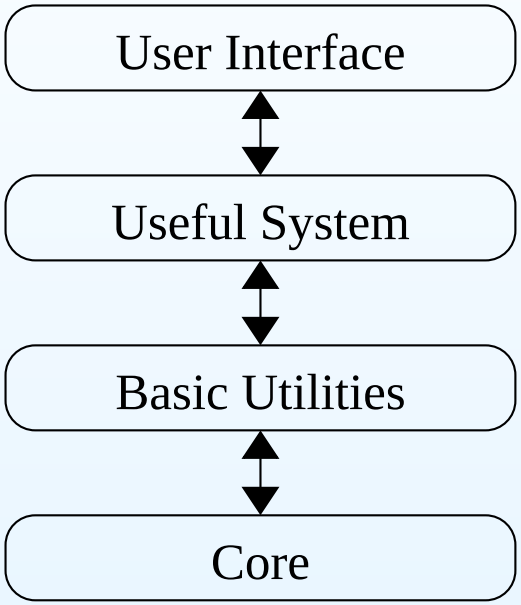
(www.x.org)

Layered/Virtual Machine

Elements: *Layer*

Interaction: procedure call, message passing, standard protocol

Topological rules:



Constraints:

- Typically synchronous calls (module structure)

Tactics

Modifiability/Portability *adherence to defined protocols*, *hide information* (what the lower layers are/do),

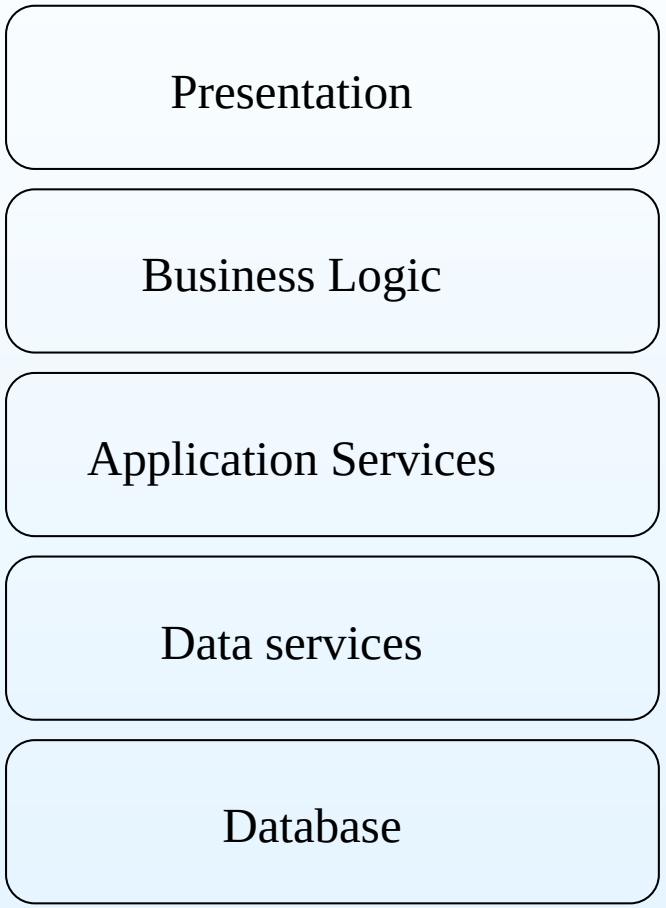
Modifiability/Maintainability *abstract common services*, *semantic coherence*, *anticipate expected changes*, *maintain existing interface*, *restrict communication paths*

Performance (anti) *increase computational overhead*,

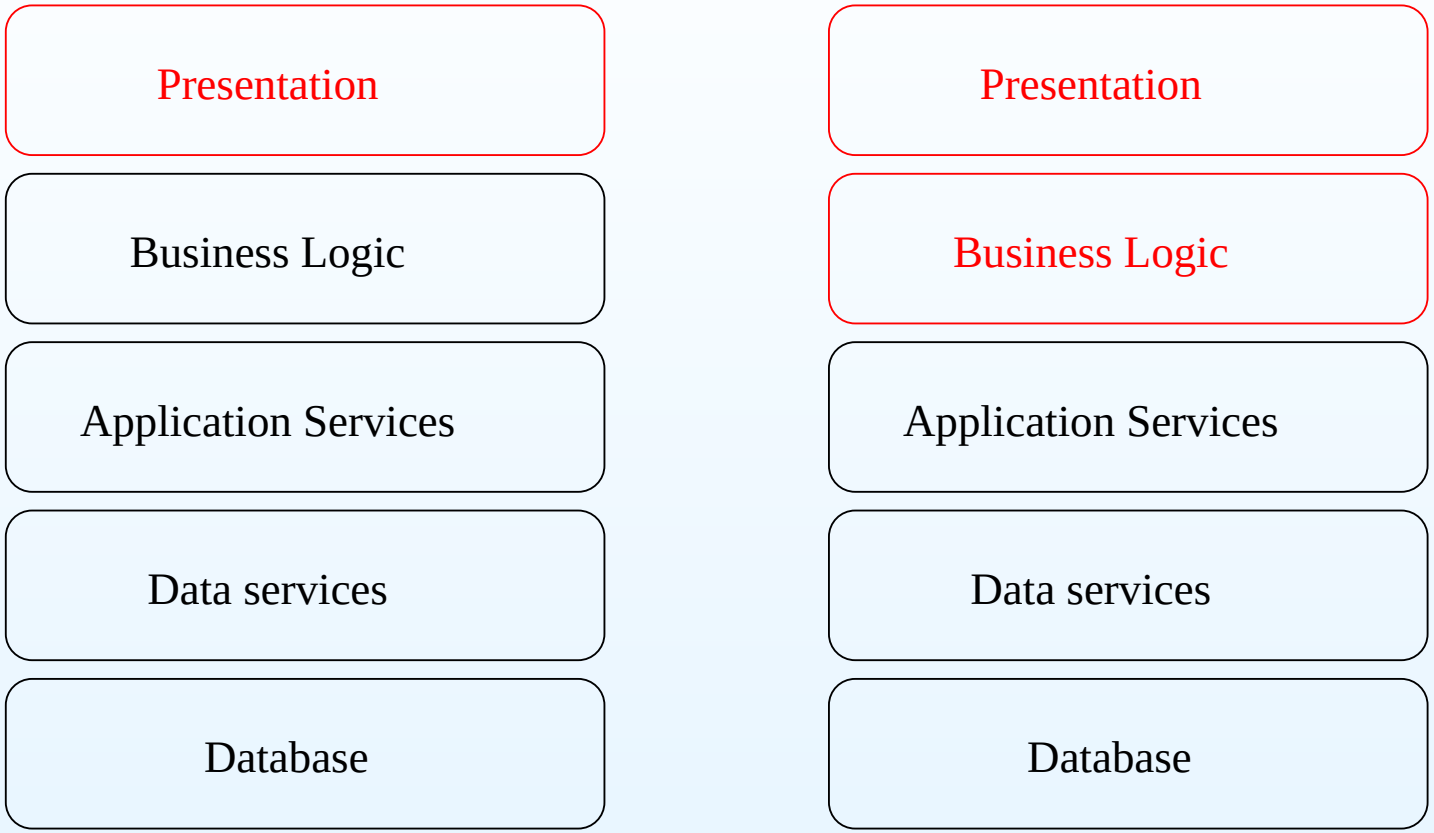
Testability *Separate interface from implementation*

Combining Patterns: n-tier

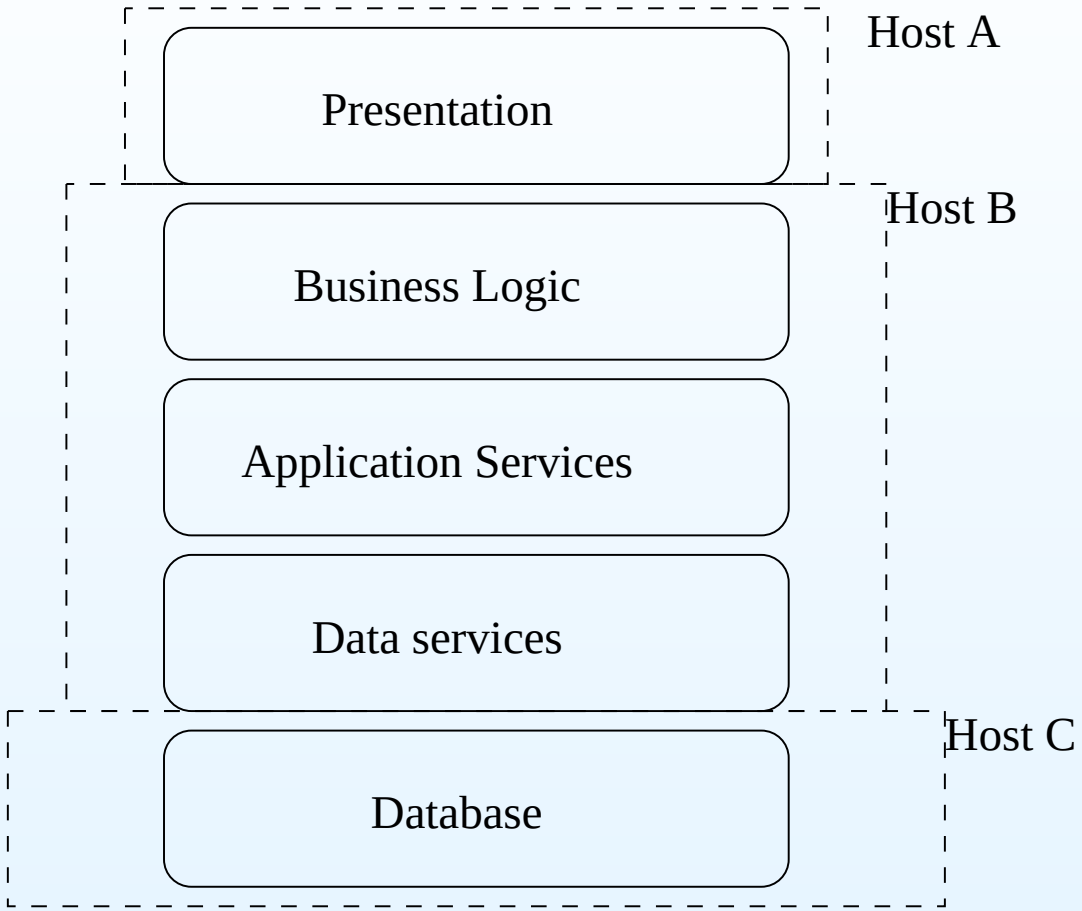
- Combine client/server with layered



Thick vs. Thin



Deployment

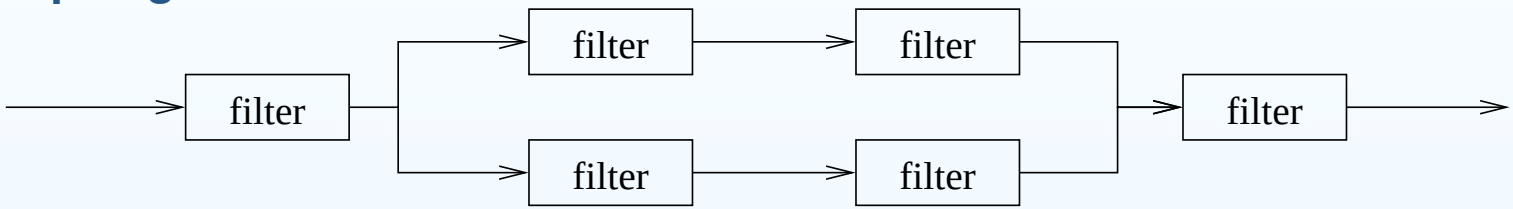


Pipe and Filter

Elements: *Filter*, *Pipe*

Interaction: Pipes carry output from one filter to the input of the next

Topological rules:



Constraints:

- no shared state between filters
- filter doesn't know what filters are upstream or downstream

Notes

- Next filter can begin as soon as it has enough input, *or not* (batch sequential)

Tactics

Modifiability/Portability *adherence to defined protocols hide information* (where the input comes from, where the output goes),

Modifiability/Maintainability *abstract common services*, *semantic coherence*, *anticipate expected changes*, *maintain existing interface*, *restrict communication paths*

Performance (anti) *increase computational overhead* (protocols), *introduce concurrency*

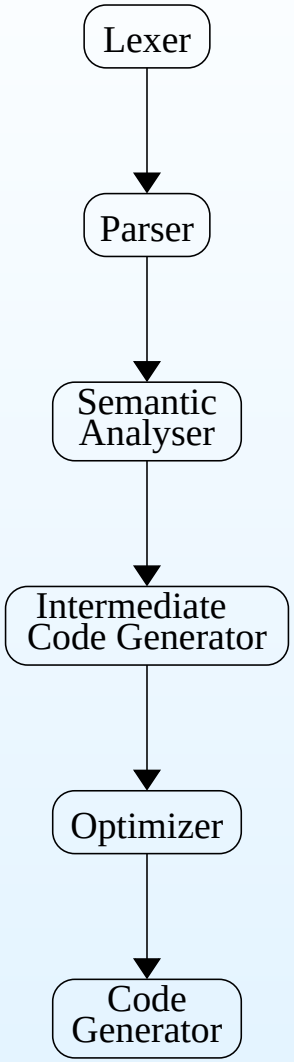
Usability (anti) *little support for either user or system initiative*

Buildability simple interactions, easy composition properties

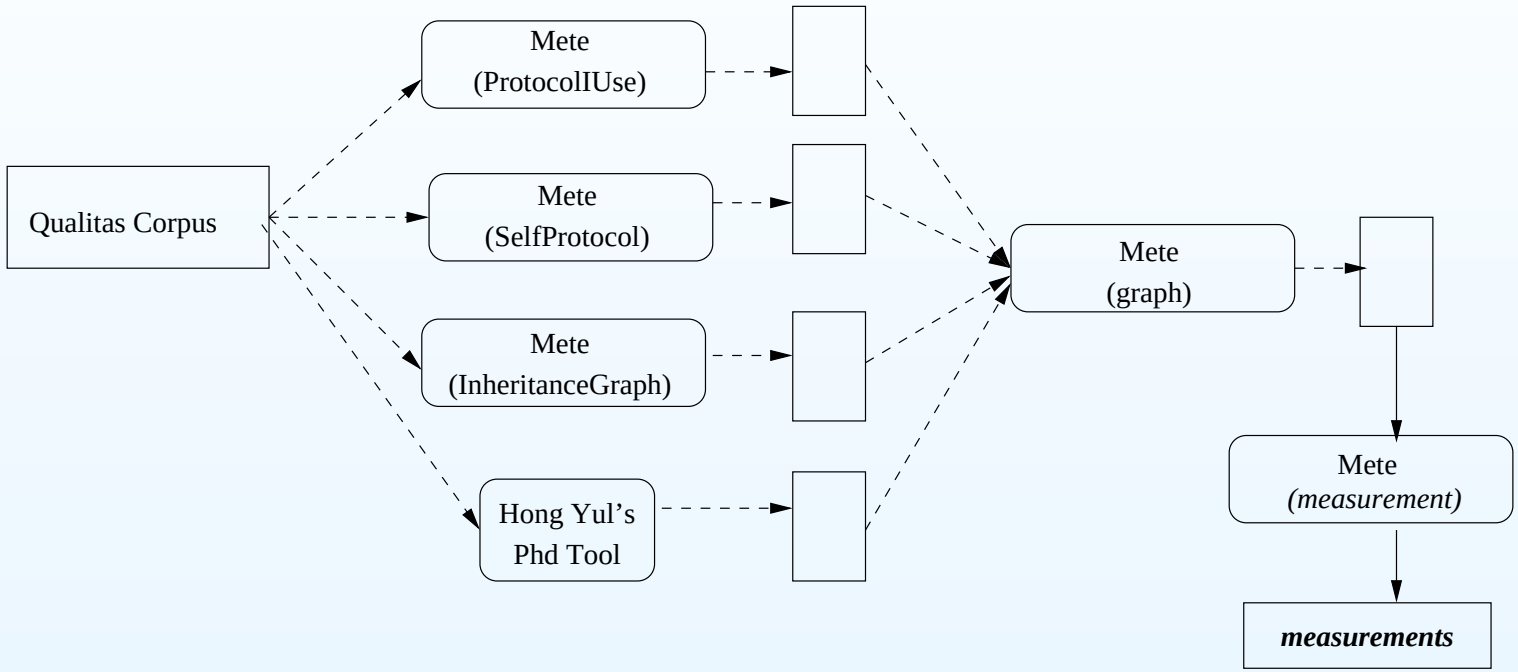
Example: Unix pipes and filters


```
grep Curve *.[ch] | cut -f1 -d: - | uniq | sort > CurveDefs.txt
```

Example: Compiler



Example: mete



Summary

- Architecture Patterns (styles) support vocabulary for describing possible solutions with respect to quality concerns
- *family* of architectures constrained by choice elements, element relationships and interaction mechanisms, and semantic constraints

SOFTENG 325:
Software Architecture

Part II, Lecture 6a: Documenting Architectures

Ewan Tempero
Department of Computer Science

Potential Assessment Question

Which of the following statements best describes the difference between the Client/Server and Peer-to-Peer styles?

(a) The two styles have different kinds of elements.

(b) The two styles both have processes as the elements but have different kinds of relationships between the processes.

(c) The two styles both have process communication as the relationships between the processes but the direction of the communication is different.

(d) The two styles both have processes as the elements that communicate with each other but which processes can initiate communication is different.

Agenda

- Admin
  - Schedule:
    - Test Wednesday 25 October 15-16hr (usual lecture time), Eng1401/401-401 (NOT usual lecture place)
  - Assignment 2?
- Documenting architectures
- **References** *Software Architecture in Practice* Bass et al. — Chapters 18

Why Document Architecture?

- Educate people about the system — new team members, external analysts, new architect
  - reading code isn't likely to help
  - understanding includes what the architecture is and why it is what it is
- Communication vehicle for stakeholders
  - perhaps the most important one being the future you
- Basis for evaluation
  - preferably before the system is built

Rationale

- Common questions that are asked (usually when things have gone wrong):
  - “Why didn’t you do X (e.g. .NET) in your architecture?”
  - “Why did you choose an architecture that ignores Y (e.g. security)?”
  - “Why did you do Z (e.g. Blackboard) in your architecture?”
- Record the reasons for the decisions made (or not made) so that they can be revisited (or not) in an informed way

What to Document With

- informal notations — boxes and lines + text
- semi-formal notations — existing standard non-architecture notations re-purposed for architecture, eg UML.
- formal notations — architecture description languages (ADLs)

Architecture Description Language (ADL)

- is a (formal) description of an architecture
- allows mutual communication
- provides embodiment of early design decisions suitable for analysis
- provides transferable (intra and inter system) abstraction of the system
- *not* a requirements description language (although requirements organisation can often imply aspects of architecture)
- *not* a programming language (although code structure usually implies architecture)
- *not* a modelling language (or rather, is a special purpose modelling language)

What should ADLs do?

- describe elements (especially their interfaces) and their relationships
- describe structures
- allow analysis of quality attribute scenarios

Existing ADLs

- mainly from research — emphasis on formal syntax, semantics to provide various forms of analysis e.g., C2, Wright, Rapide, UniCon, Aesop, LILEANNA, SADL MetaH,
- little commercial — graphical, only some formalism and hence only some analysis e.g., ROOM, C2
- different ADLs tend to emphasise different aspects of architecture, e.g., restrict kinds of style, restrict kinds of elements, restrict kinds of analysis

Why not UML?

- how to distinguish elements and connectors?
- how to express behaviour and constraints?
- how to describe styles
- how to analyse architectures
- *but*, is a standard

In a world without ADLs. . .

- Should we document architectures anyway, and if so how?
- Documentation provides communication among stakeholders
  - prescribes what should be true by placing constraints on decision to be made
  - describes what is true by recording decisions already made
  - can provide a training vehicle, whether about the system or the corporate culture
  - provides basis on which to perform analysis (even if not formal)
- Document it by whatever means works
  - any diagrams can work, provided their meaning is clear to all stakeholders
  - even UML. . .

Who are the Stakeholders?

Who are the Stakeholders?

- Client (person paying the bills)
- Client (person responsible for operations)
- End User
- Project Manager (e.g., work assignment)
- Development team members (including testers, integrators, user manual writers)
- Future Client (.e.g, new operations manager)
- Future Architect
- Customer?

How much documentation is needed?

- As much as is necessary and not one bit more
- What is necessary depends on the stakeholders that will use it, or *what questions are likely to be asked*
- *Intent* or *rationale* is often hardest to divine if not explicitly stated
- Avoid documentation for documentation's sake

How to document

- Choose the relevant **views**
- Document each view
- Document information that applies across views

Relevant Views

- Client (operations) probably doesn't care about module structures but does care about deployment
- Project manager probably doesn't care about implementation but does care about work allocation
- Architect probably doesn't care about work allocation but does care about most others
- Possible minimum set:
  - Uses structures — what chunks there are and how they interact
  - Process structure — how the chunks actually run and communicate
  - Deployment structure — where the chunks actually run

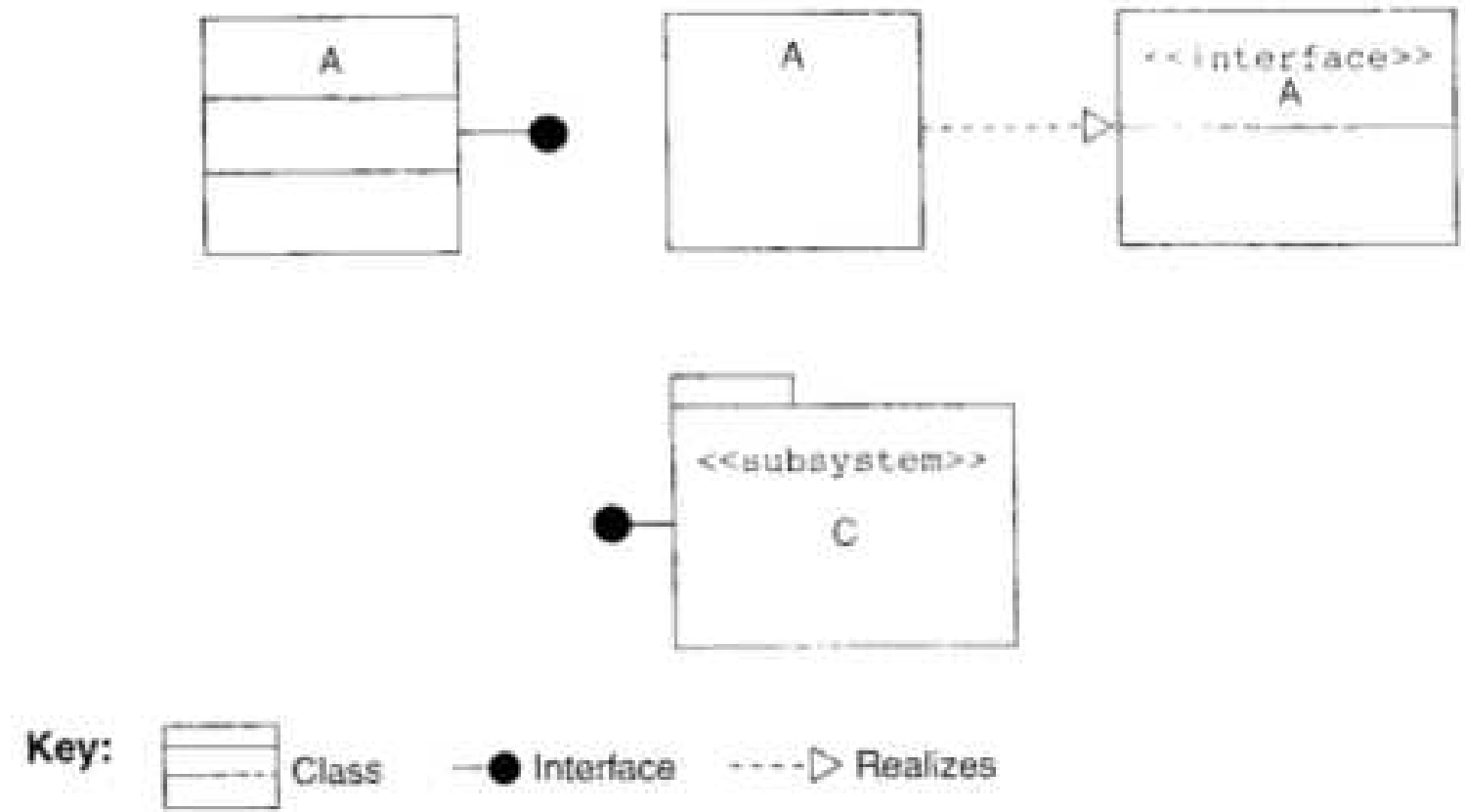
Across Views

- What to expect in the documentation
  - *E.g. standards used*
- Give an overview of what the system is meant to do
  - *"Read the 150 page System Requirements Specification" is not likely to be palatable*
- Defining features of the architecture
  - *E.g. Main architectural requirements*
  - *E.g. Aspect of architecture that took the most effort and why*
- Appropriate indexes
  - *E.g. Catalogue of elements*

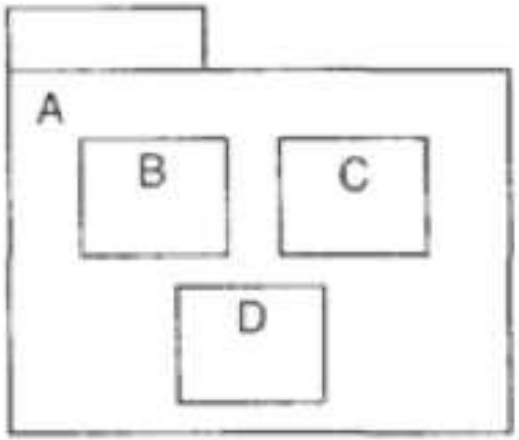
Documenting Architectures with UML

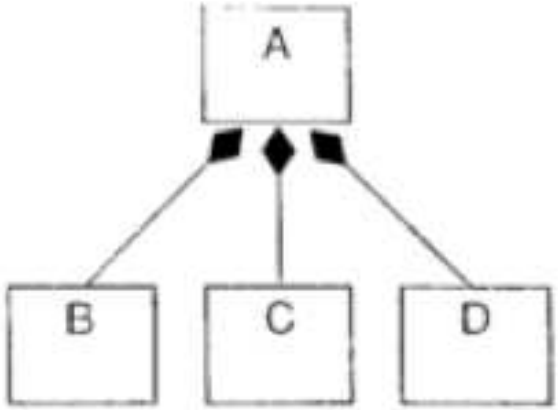
- Module structures
  - Elements — Class or Package, possibly with Interface (or other classes)
  - Relationships
    - uses — Dependency
    - decomposition — Composition or Nesting
    - generalisation — Generalisation
    - layered — Dependency (+ topology restriction)

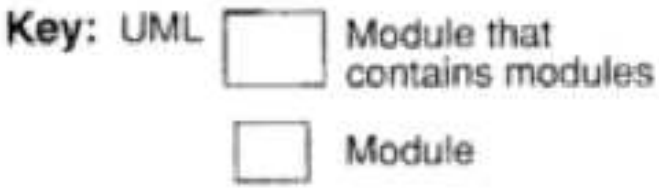
Example: element interfaces (Bass et al.)



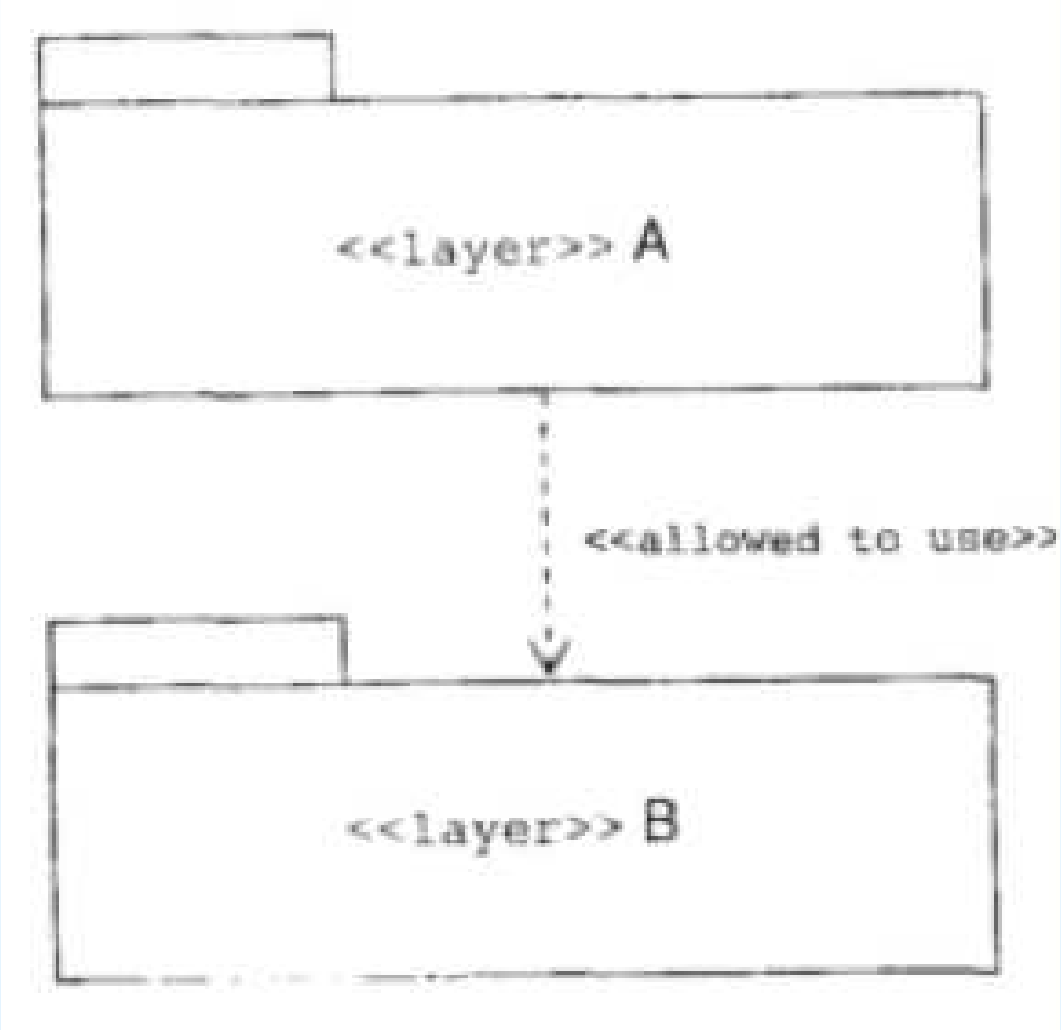
Example: decomposition (Bass et al.)







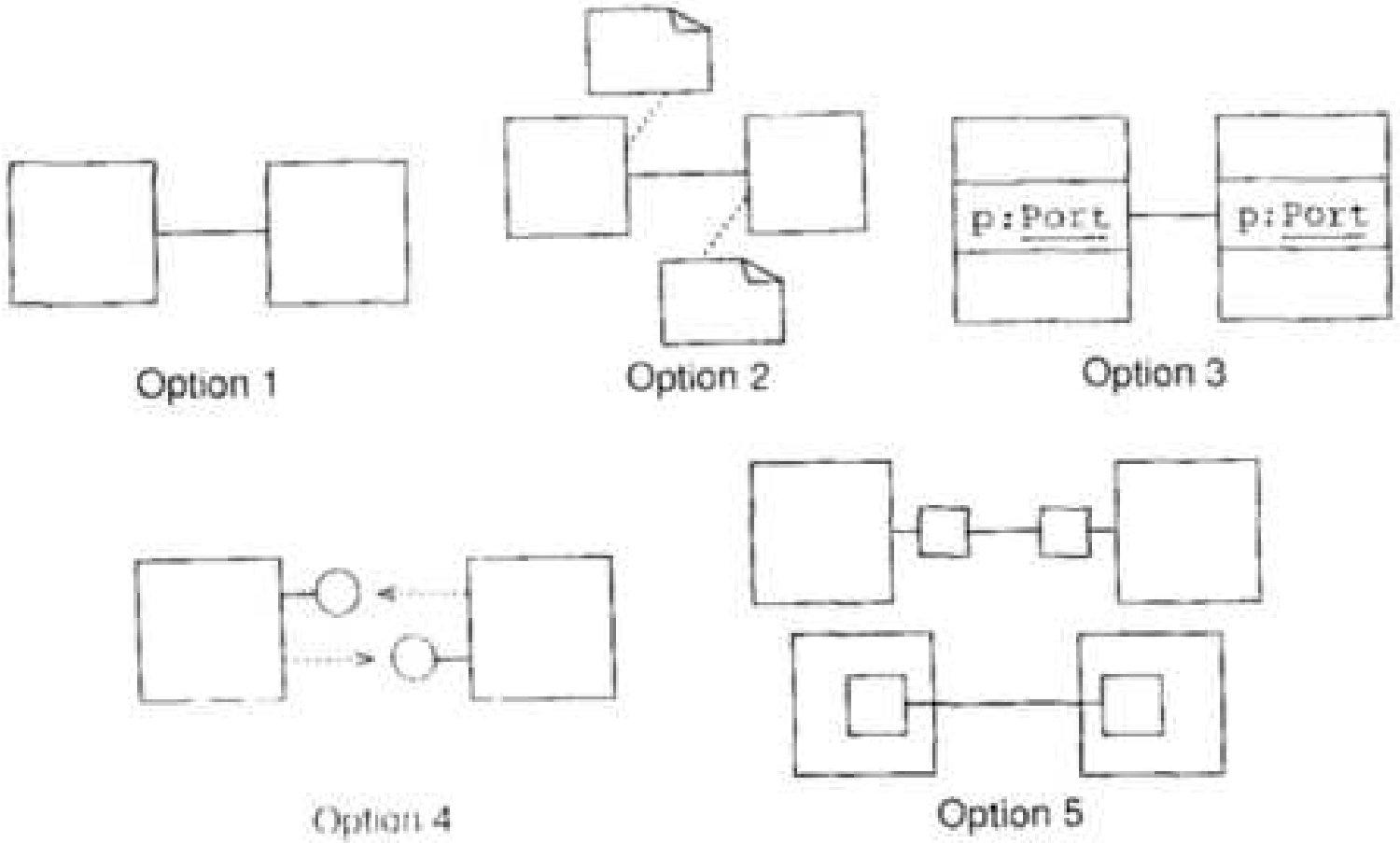
Example: layers (Bass et al.)



Documenting Architectures with UML

- Component and Connector
  - Elements — Class or Object, possibly with Interface
  - Relationships
    - (communicating) processes — associations, classes
    - concurrency — association, classes
    - shared data (or repository) — classes
    - client-server — association, classes

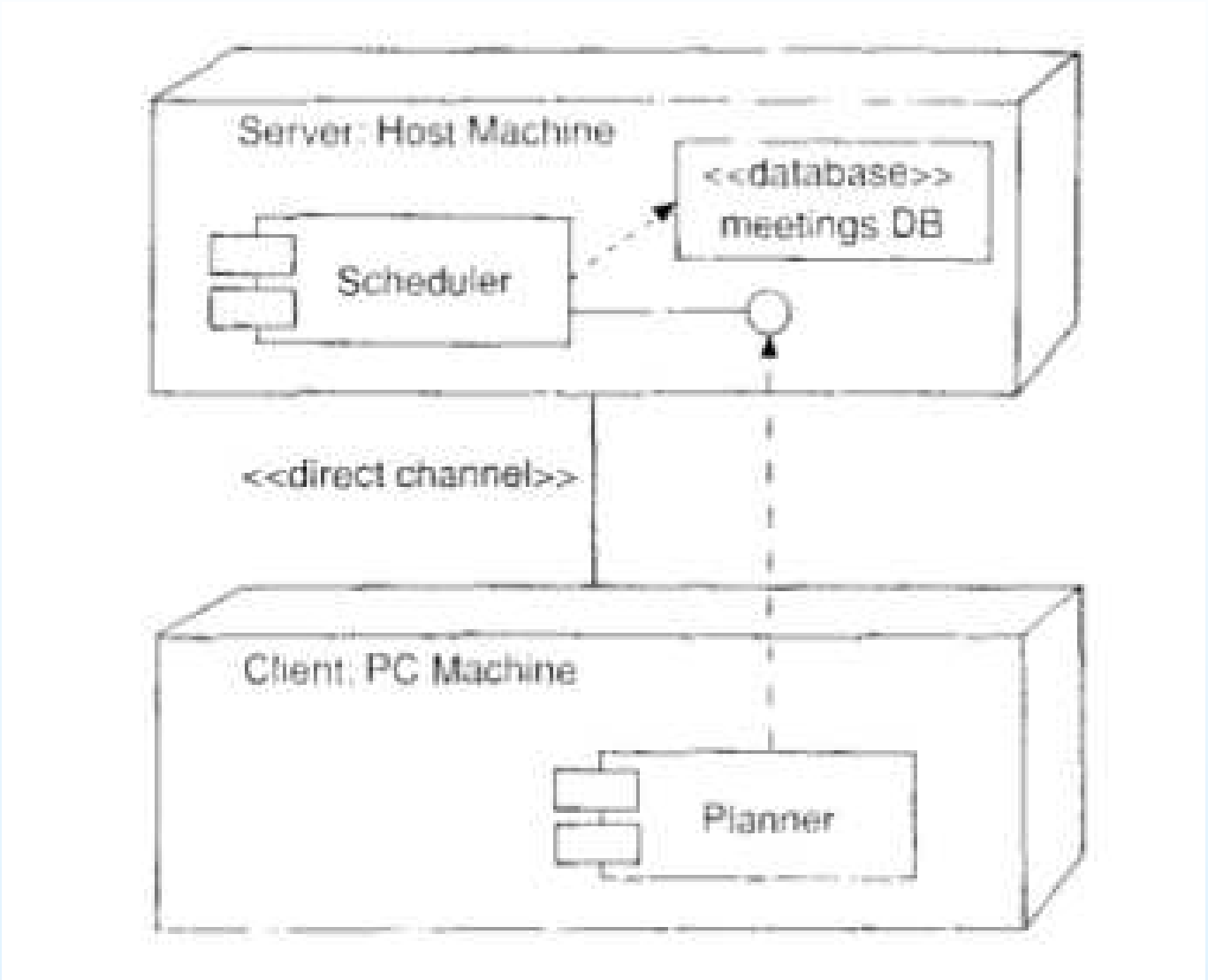
Example: representing interfaces to components (Bass et al.)



Documenting Architectures with UML

- Allocation
  - deployment — Nodes and Components, nesting
  - implementation — Components and Classes, nesting
  - work assignment — Nodes and Classes, nesting

Example (Bass et al.)



SOFTENG 325:
Software Architecture

Part II, Lecture 6b: Creating Architectures

Ewan Tempero
Department of Computer Science

Agenda

- How to create architectures — Attribute-Driven Design
- **References**

  Software Architecture in Practice Bass et al. — Chapters 17

Where to Architectures come from?

- Experience
- Quality Attribute Requirements
- Technological Constraints
- Organisation Structure

Attribute-Driven Design (ADD)

- (one) approach to defining a software architecture
- bases decomposition on required quality attributes
- uses architecturally significant requirements (ASRs) as starting point
- focuses on module decomposition

Attribute-Driven Design

1. choose module to decompose
2. refine module according to:
   (a) choose ASRs
   (b) choose architecture pattern
   (c) instantiate pattern and allocate functionality and choose views
   (d) define interfaces of child modules
   (e) verify and refine requirements to child modules
3. repeat for each child module that needs decomposition

Example: Travel Booking System

- Web-based travel booking system, accessible by anyone, but travel agents pay for access and get more functionality
- Provide ability to search for, book, and pay for flights
- Provide ability to search for, book, and pay for hotels
- Security is crucial
- Performance is important
- Must be able to add new airlines and hotels easily

Architectural ASRs

- resist attacks (communications channels, data stores)
- allow specialised access to users
- travel agents must get response within 5 seconds
- must be possible to add/remove hotels/airlines

Security Quality Attribute Scenario

Source Someone of unknown identity who is external and not authorised with access to limited resources

Stimulus tries to access

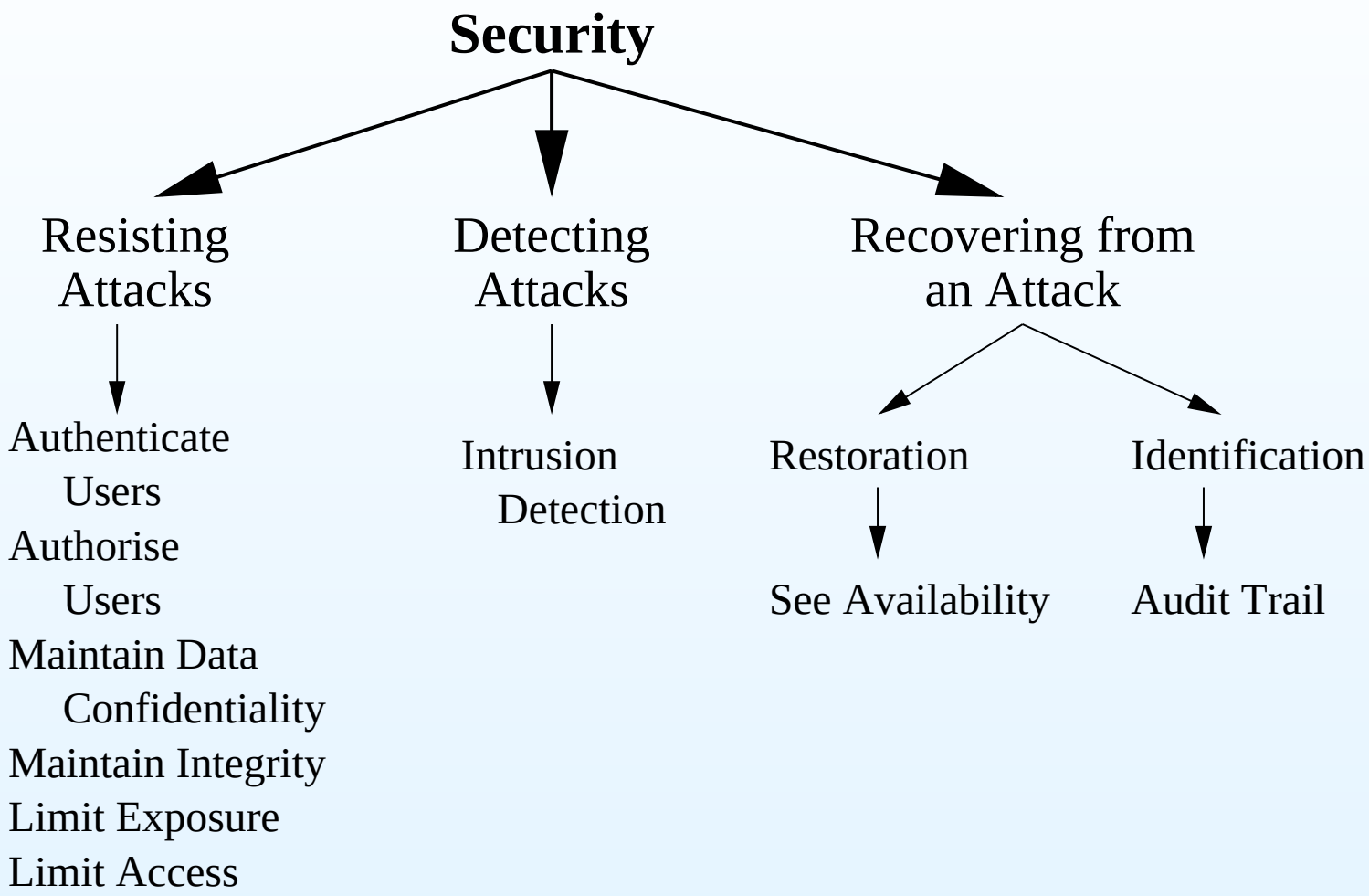
Artifact customer data (identity/financial details)

Environment to the system while it is on-line and running normally

Response The response is to block access to the data and record the access attempts

Response Measure with 100% probability of detecting the attack, 100% probability of denying access, and 50% probability of identifying the location of the individual.

Security Tactics



Security Tactics

- limit access — firewall
- maintain data confidentiality — encryption for communication (performance issue)
- authenticate users — must know who travel agents are
- authorise users — travel agents have different access than others

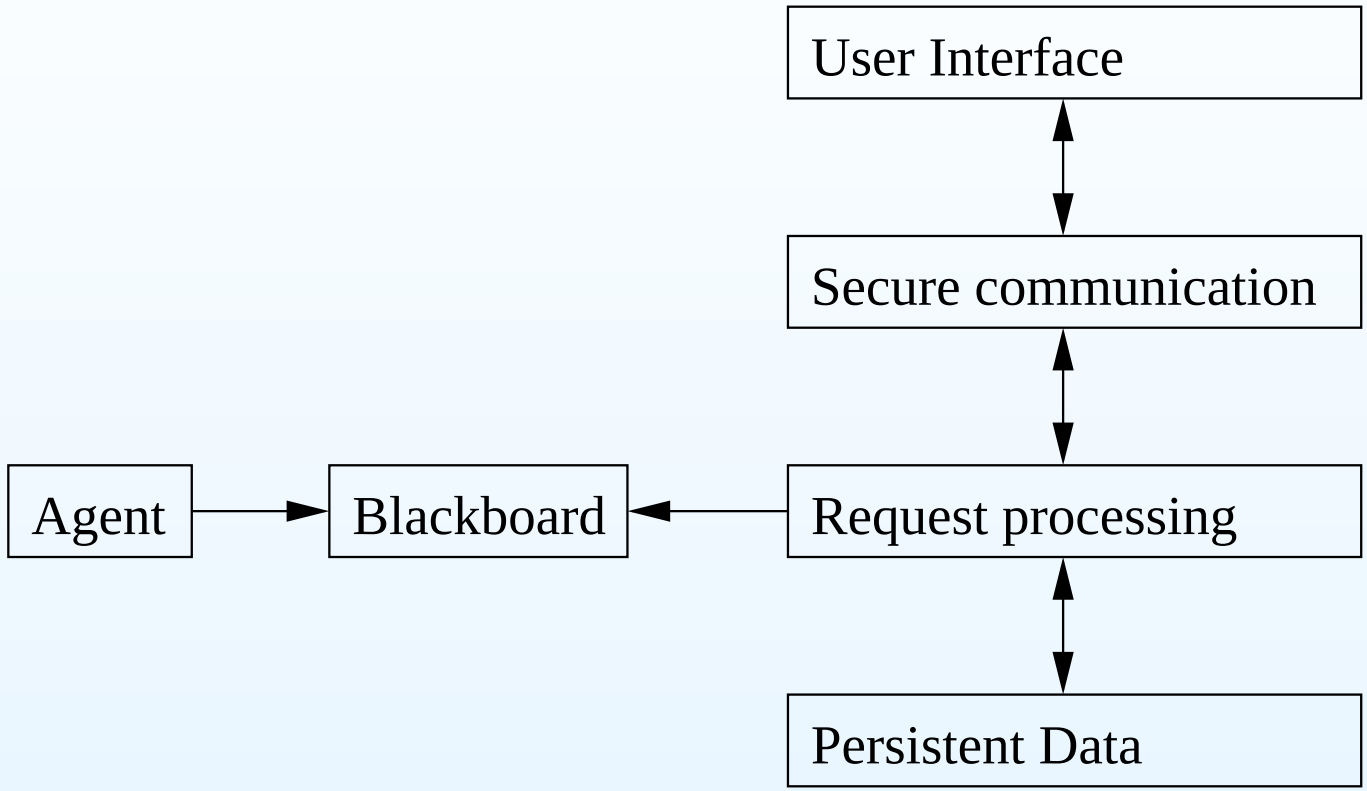
Performance

- *communicating processes* pattern
- introduce concurrency — process requests in parallel
- increase available resources — more processors
- maintain multiple copies — information that doesn't change much
- scheduling policy — travel agents have priority

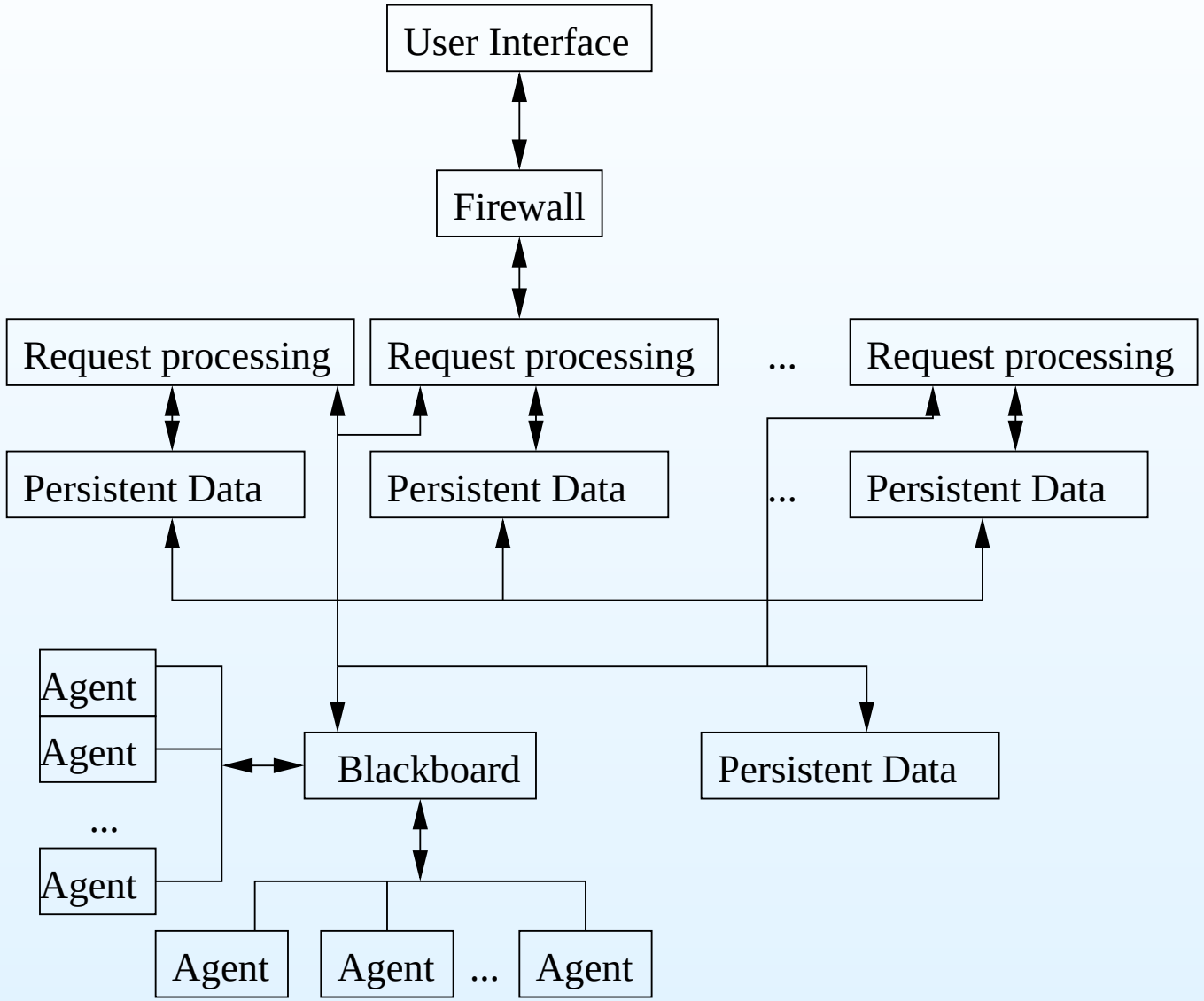
Modifiability

- n-*tier* pattern
- *blackboard* pattern — place to record what airlines and hotels are participating at any time
- semantic coherence — airlines and hotels (at least)
- information hiding — exactly which airlines and which hotels (and which travel agents)
- runtime registration, adherence to defined protocols — central registry to look up airlines, hotels

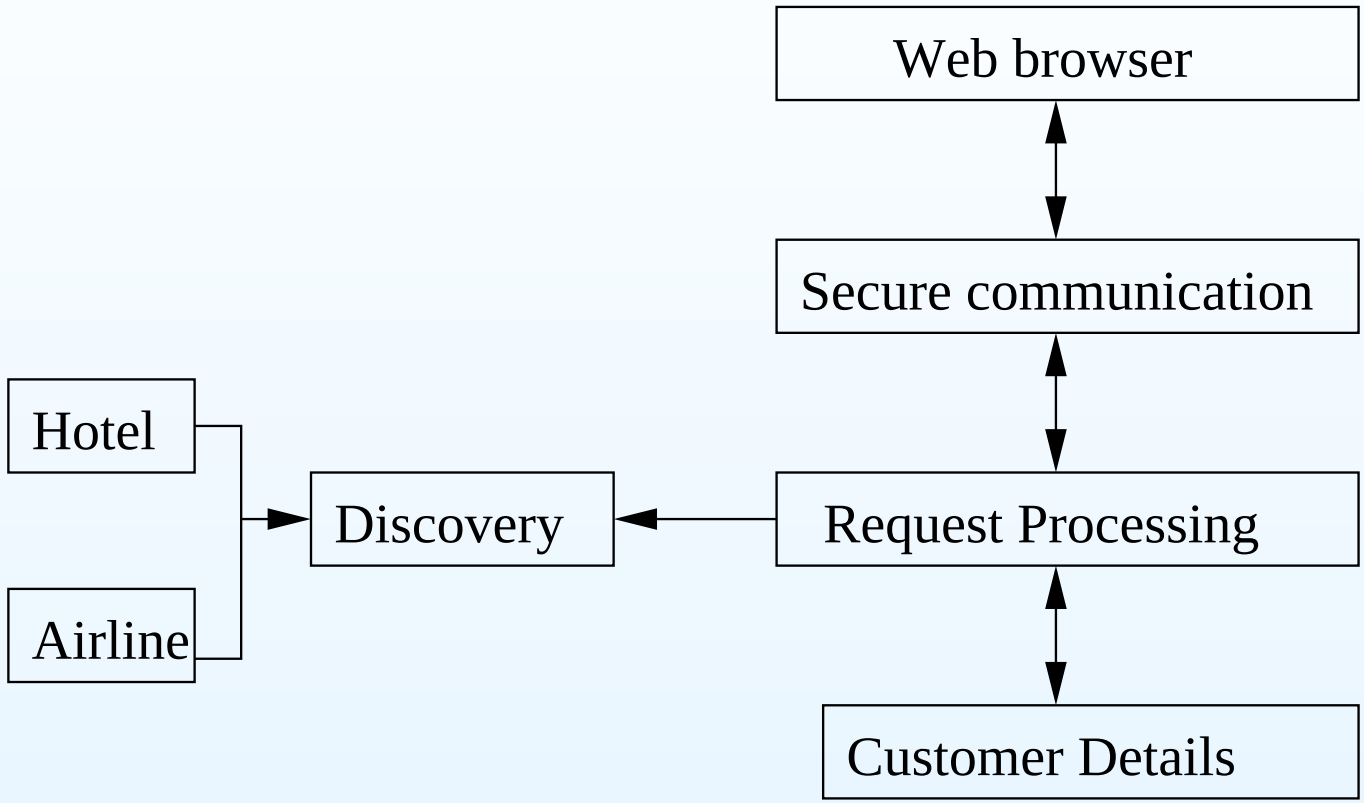
Uses Structure Pattern (Module)



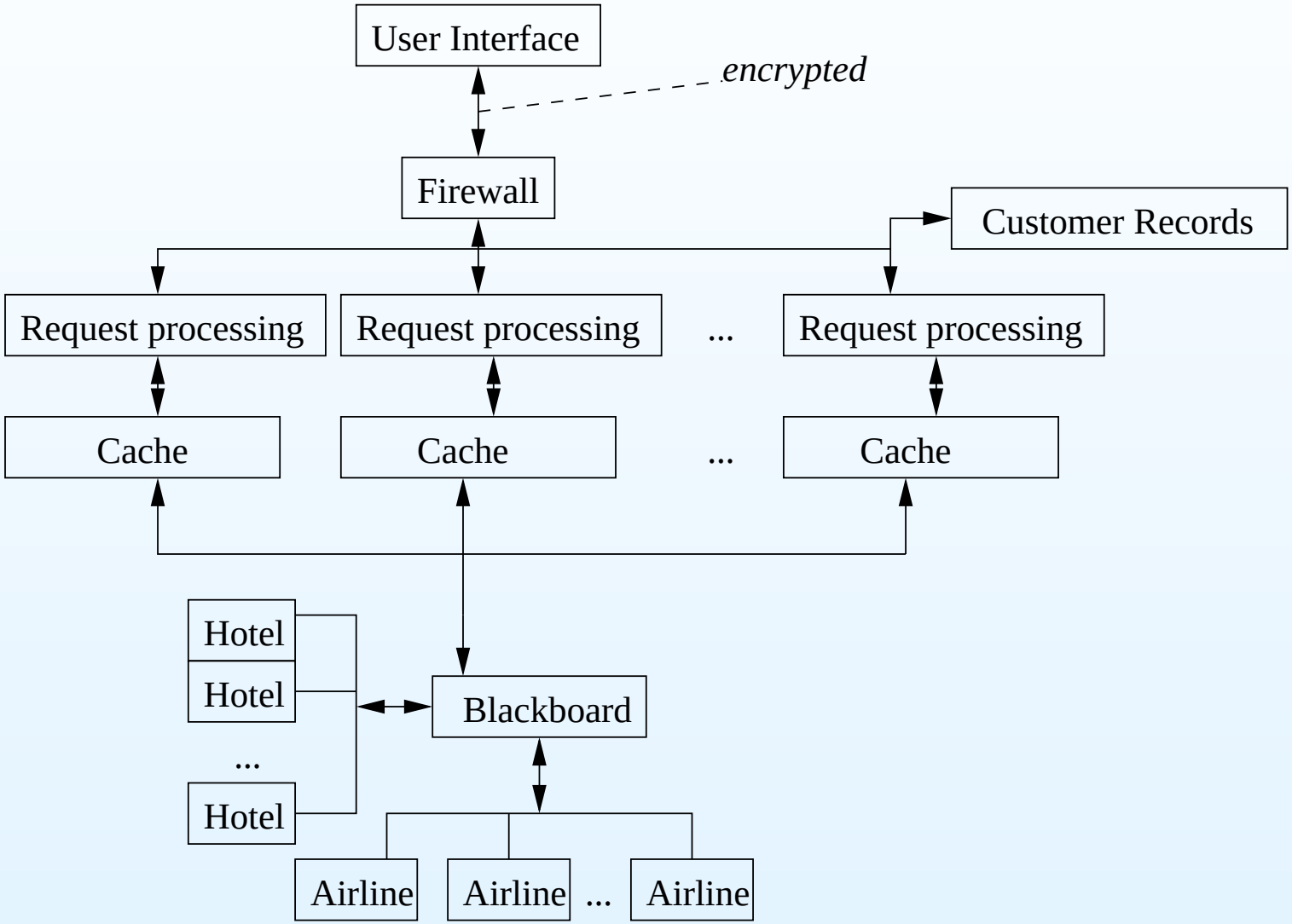
Process Structure Pattern (C&C)



Uses Structure



Process Structure



Questions

- Where is “runtime registration” described?
- Where are the various protocols shown?
- What is the scheduling policy?
- Do we have enough information to argue that the security requirement is met?
  - where is the information for authentication and authorisation kept?
- Do we have enough information to argue that the performance requirements are met?

Define interfaces of child modules

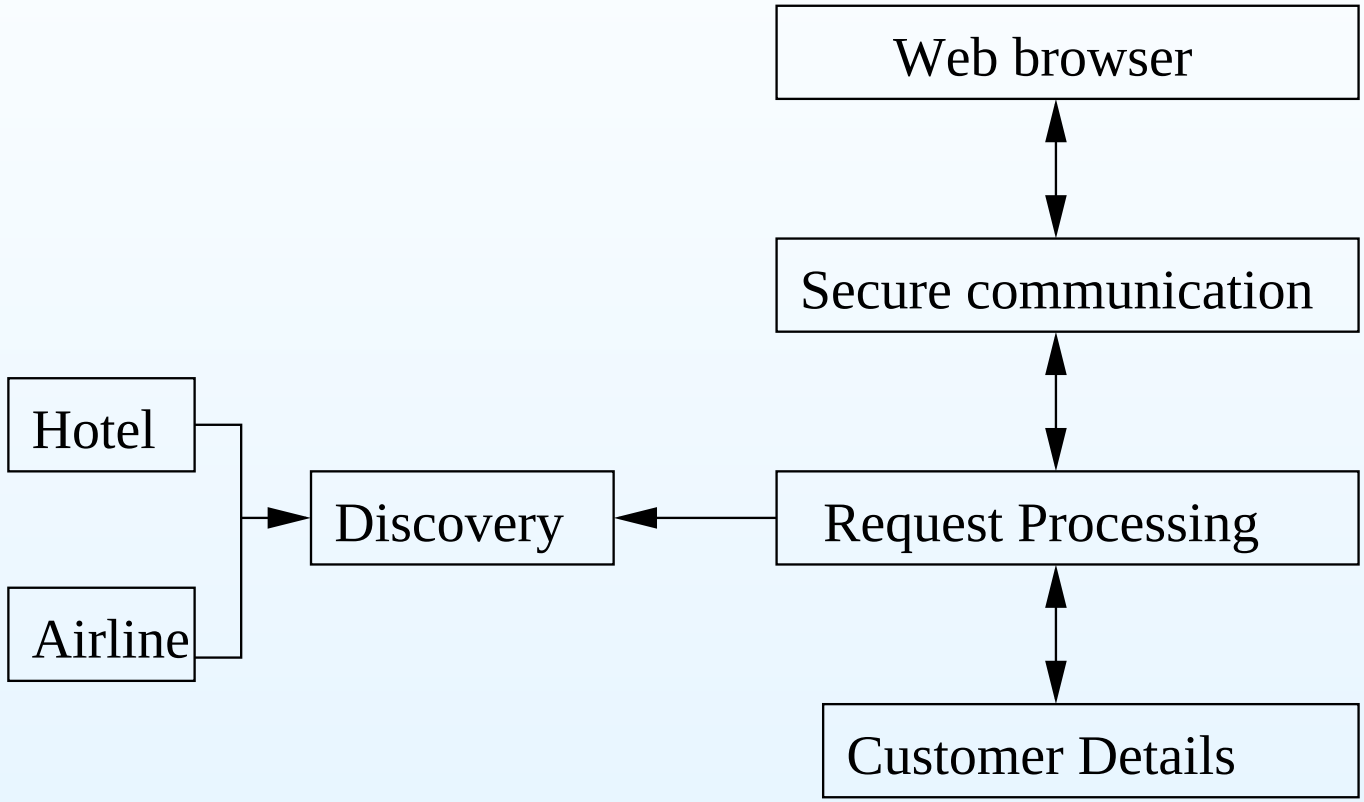
- What are the responsibilities of each of the modules?
- What about each module is visible to the other modules?
- . . . see later

ADD: Verify and Refine

- Where is all the functionality? We should be able to decompose the functional requirements in line with the modules.
- How are the quality attribute scenarios being met? For each scenario, one of the following must hold:
  - we can demonstrate it has been met
  - in order for it to be met, only **one** child element must met it
  - in order for it to be met, several child elements must meet bits of it

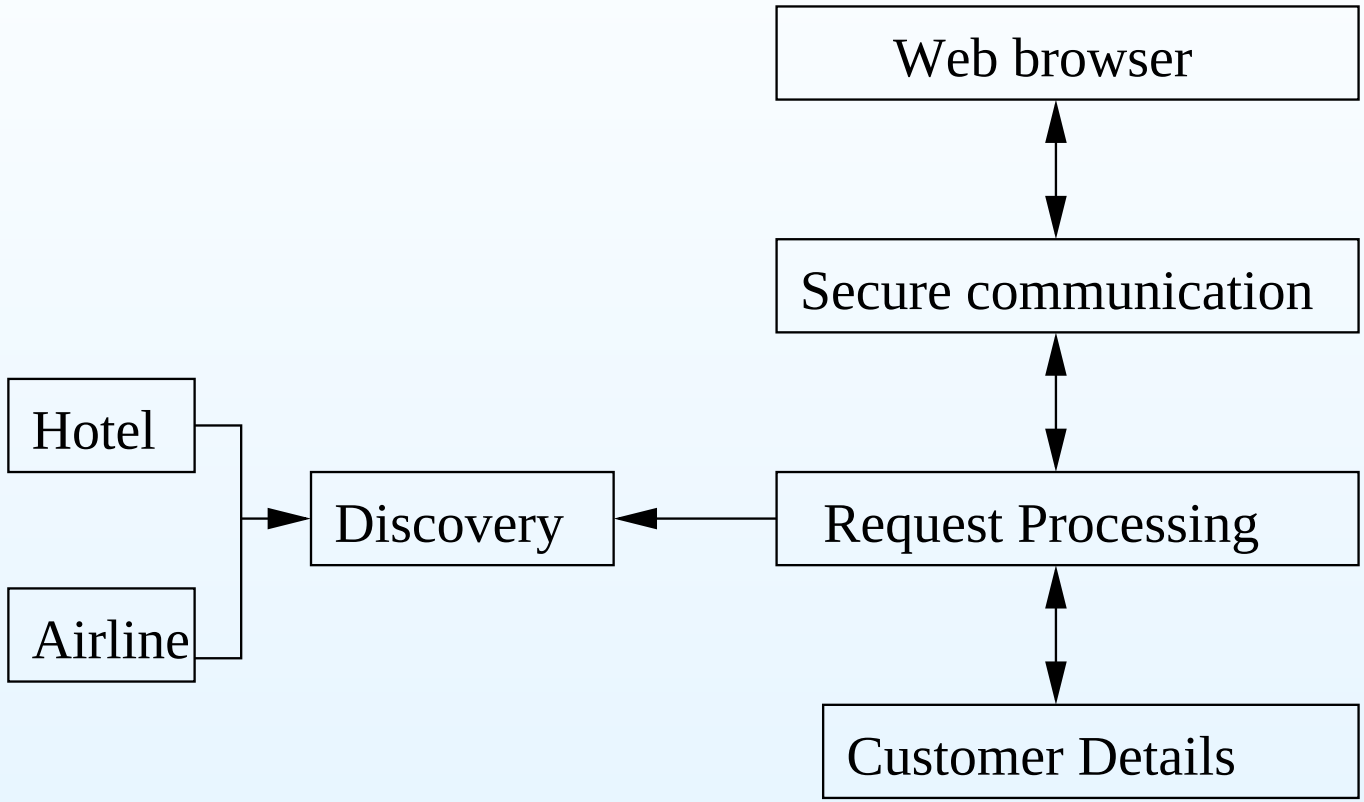
Example: Verify

- Provide ability to search for, book, and pay for flights/hotel rooms



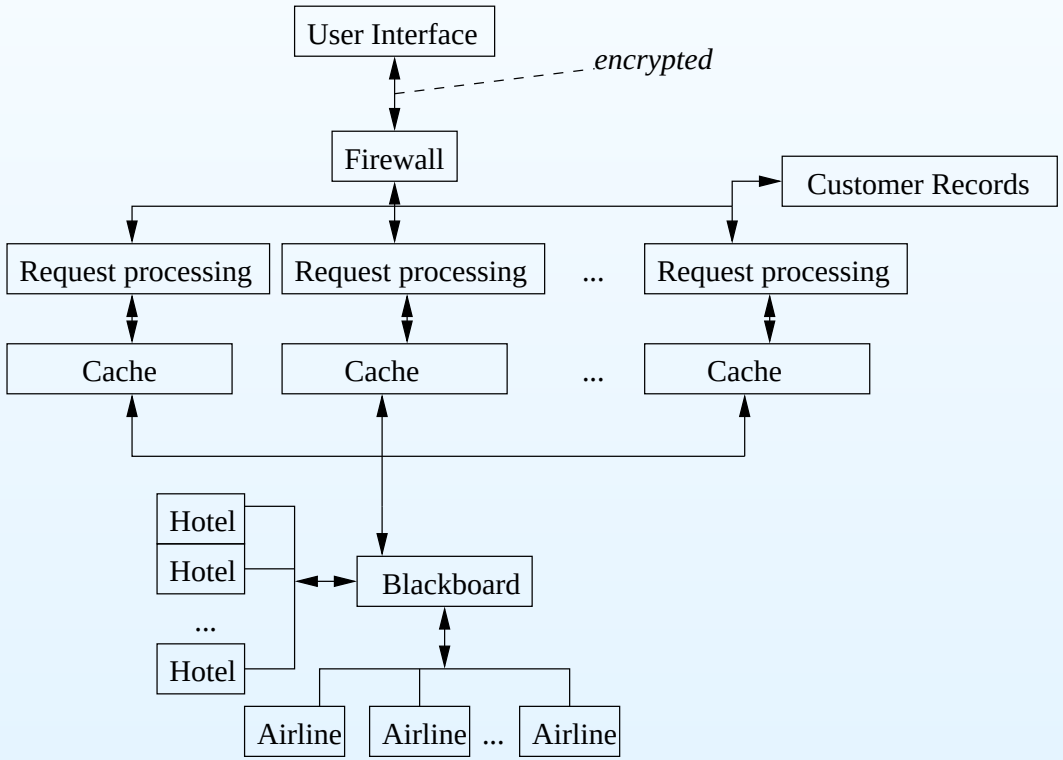
Example: Verify

- travel agents pay for access and get more functionality



Example: Refine

- Prevent access to customer data by external attacker becomes . . .
- Prevent access
  - to customer data in communications channel and
  - to internal system

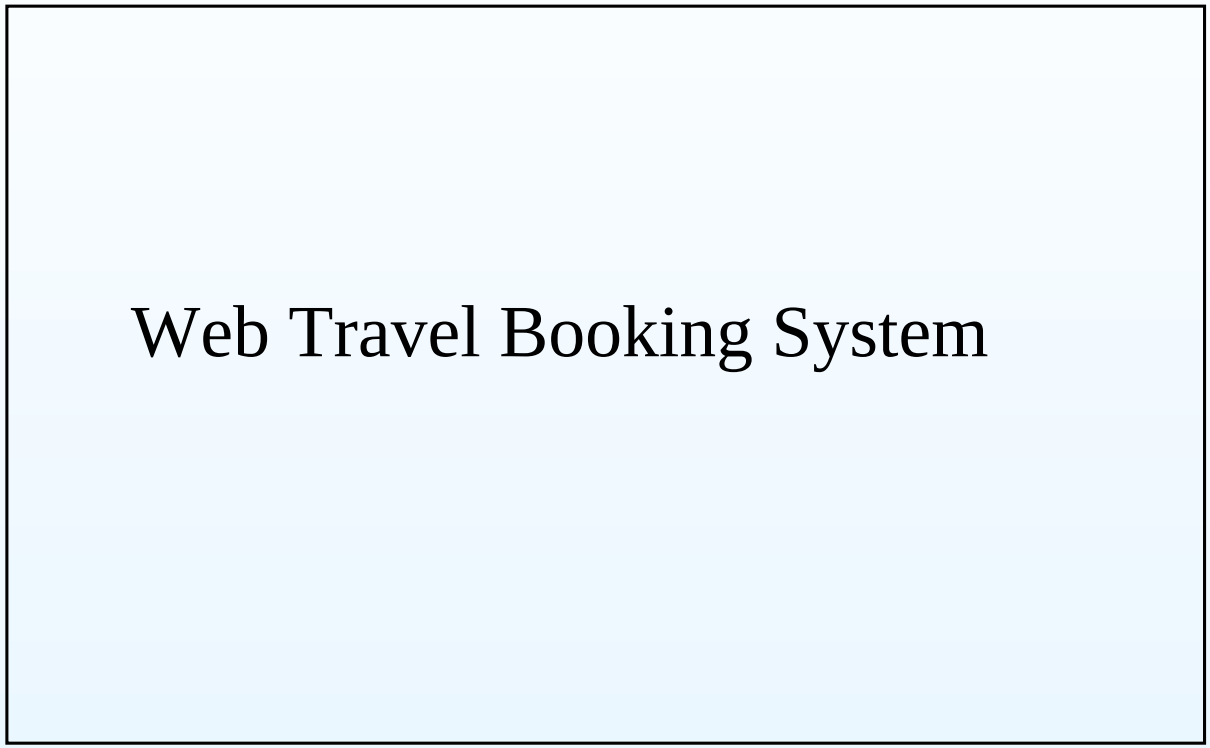


Example: Performance Assumptions

- What's the longest an ordinary user will wait in a normal situation?
- Assumptions: the Internet isn't broken or overloaded, the user's box is "current", the browser is as least IE6.0, a database transaction can be done in 0.001 seconds, the LAN runs at 1Gb/s

Case Study: Web Travel Booking System, Iteration 1

- Agenda
- Creating Architectures
- ADD
- Example
- Example ASRs
- Security Tactics
- Example Tactics
- Example Patterns
- ADD Next steps
- Define Interfaces
- Verify and Refine
- Example Verify
- Example Refine
- Example Verify
- WTBS



ASR QAS: user convenience

Source end user — Standard user at home

Stimulus use system conveniently — wants to make a travel booking right now from Auckland to Wellington for next tuesday evening

Artifact system — web travel booking system

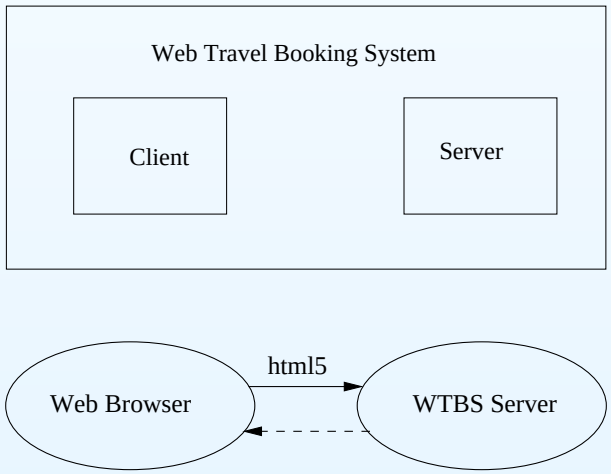
Environment at runtime, user device requirements — at runtime, user has modern web browser, adequate internet access and standard data plan, no internet problems, user knows address

Response access provided via user device — users can access web travel booking system via their web browser over the internet

Response Measure user satisfaction — users are able to access web travel booking web site.

Case Study, Iteration 1

- Step 1. Choose module — “WTBS”
- Step 2. Refine module
  - 2a. ASR — User Convenience
  - 2b. Architectural pattern — (Thin) Client/Server
  - 2c, 2d



- solid arrow — initiation of communication/request, dashed arrow — response to request
- client — everything developed for WTBS that runs in client browser, server — everything developed for WTBS that does not run in client browser

ASR QAS: easy to learn

Source end user — Standard user at home

Stimulus learn system features — wants to search for a travel booking for the first time for a known date and approximate time

Artifact system — web travel booking system

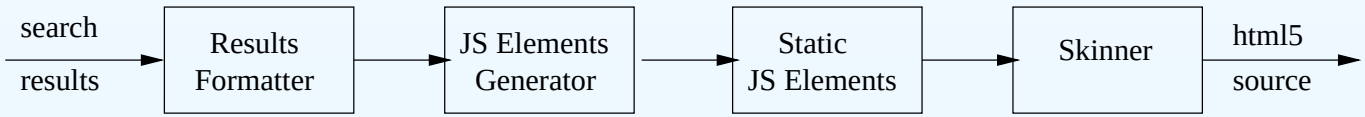
Environment at runtime, experienced user — user is comfortable using google-type searches, navigating links, choosing from drop-down menus and pop-up windows

Response interface is familiar to user — user is presented with user interface with familiar controls and page layout

Response Measure task time — user is able to complete first search in under 2 minutes

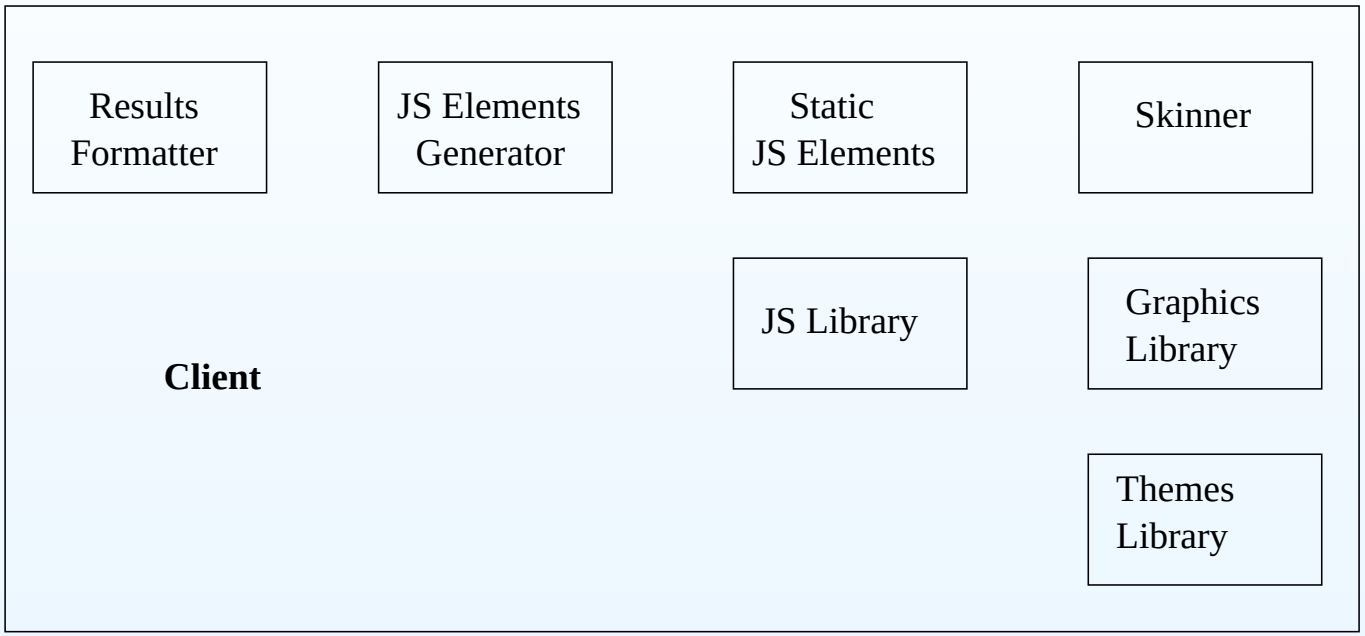
Case Study, Iteration 2

- Setp 1. Choose module — “Client”
- Step 2. Refine module
  - 2a. ASR — learn system features
  - 2b. Architecture Pattern — Pipe & Filter
  - 2c, 2d

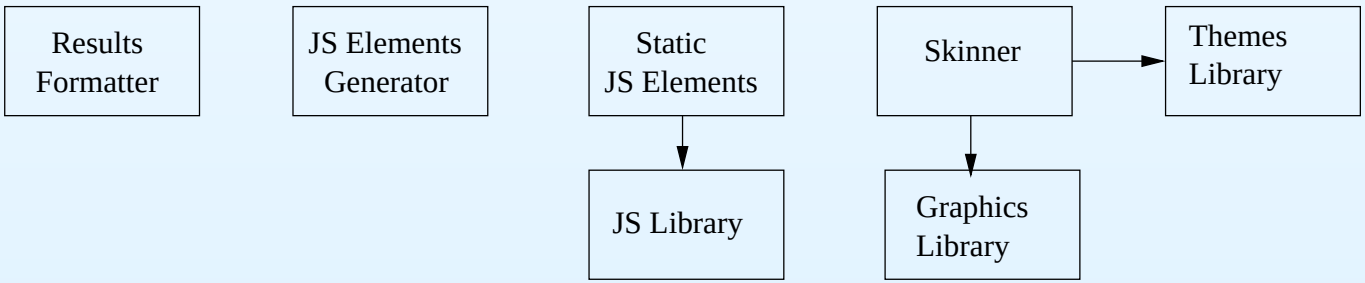


Case Study, Iteration 2

Decomposition



Uses



Usability Argument

- Use text field to enter search for journey start and end locations
- Use standard Javascript elements to support suggested results based on partial inputs
- Use standard Javascript elements to provide easy specification of dates and times
- Use standard Javascript elements to validate any values added
- Use standard layout conventions to clearly indicate where search elements are and submit options
- Time taken:
  - recognise where the search elements are to specify start and end locations
  - type in start and end locations
  - recognise where the search elements are to specify time and date
  - choose desired time and date
  - recognise where submit element is
- $\Rightarrow$ less than 2 minutes?

Quality attributes and Quality Attribute Scenarios

Ewan Tempero
Department of Computer Science

Part 1: Non-functional requirements

Scalable Web Architecture and Distributed Systems Kate Matsudaira, Chapter 1 of *The architecture of open source applications, Volume II* Amy Brown & Greg Wilson (editors)

- Section 1.1 lists several “key principles that influence the design of large-scale web systems”.
  - Which of these principles correspond to quality attributes?
  - For those that are quality attributes, which of the system quality attributes do they refer too?

Principles

- Availability $\rightarrow$ availability
- Performance $\rightarrow$ performance
- Reliability $\rightarrow$ security/availability
- Scalability $\rightarrow$ modifiability
- Manageability $\rightarrow$ usability
- Cost $\rightarrow$ QA but not a system QA

“Manageability”

- Matsudaira says of what she calls “Manageability”:
  Things to consider for manageability are the ease of diagnosing and understanding problems when they occur, ease of making updates or modifications, and how simple the system is to operate.
  - What kinds of (concrete) “stimulii” (and source) might be relevant for describing “manageability” requirements?
  - What kinds of “measures” might be appropriate for determining whether this requirement has been met?

Manageability

Stimulus	Source	Measure

Usability General Scenarios

Part	Details
Source	End user (possibly in specialised role)
Stimulus	Wants to learn system features; use system efficiently; minimize impact of errors; adapt system; feel comfortable
Artifact	System or part of system user interacts with
Environment	At runtime or configuration time
Response	System provides one or more of the following responses: • to support “learn system features” help system is sensitive to context; interface is familiar to user; interface is usable in an unfamiliar context • to support “use system efficiently” aggregation of data and/or commands; re-use of already entered data and/or commands; support for efficient navigation within a screen; distinct views with consistent operations; comprehensive searching; multiple simultaneous activities • to minimize “impact of errors” undo, cancel, recover from system failure, recognize and correct user error, retrieve forgotten password, verify system resources • to “adapt system” customizability; internationalization • to “feel comfortable” display system state; work at the users pace
Measure	Task time, number of errors, number of problems solved, user satisfaction, gain of user knowledge, ration of successful operations to total operations; amount of time/data lost

Part 2: QAS 1

Matsudaira says that one of the requirements for the Image Hosting Application (IHA) is: If a user uploads an image, the image should always be there (data reliability for images).
Consider the following statement:

> *A server crashes due to a random failure event with no other system or networking problems. The failure is recorded within 10 minutes, and IHA continues to provide full functionality, including access to any images stored previously to the crash.*

Is this statement relevant to the stated requirement?

- Restate the above statement as a Quality Attribute Scenario (QAS)

Model Solution

Part	Details
Stimulus	A crash occurs
Source	. . . due to a random event
Artifact	. . . to one of the servers
Environment	. . . with no other system or networking problems.
Response	The failure is recorded
Measure	. . . within 10 minutes and there is no loss of service.

SOFTENG 325:
Software Architecture

Part II, Tutorial: Structures and Tactics

Ewan Tempero
Department of Computer Science

Kinds of Structures

Scalable Web Architecture and Distributed Systems Kate Matsudaira, Chapter 1 of *The architecture of open source applications, Volume II* Amy Brown & Greg Wilson (editors)

- There are a number of figures shown. Which of these show some kind of architectural information?
- Of those figures that show architectural information, which of them show views of structures?
- Of those figures that are views of structures, what kind(s) of structure(s) are they?

Structure and Quality Attributes

- Consider Figure 1.4. Which quality attributes might you use this architectural view to reason about?
- What about Figure 1.11?
- Figure 1.14?

Tactics

- What is the *tactic* illustrated in Figures 1.8–1.12? (Hint: What quality attribute are these figures addressing?)
- What is the *tactic* illustrated in Figures 1.13–1.16? (Hint: What quality attribute are these figures addressing?)