SERVICE ORIENTED ARCHITECTURES

Lecture 5

LAST TIME

Business Logic Layers

- Domain driven
- Services

CONTENT

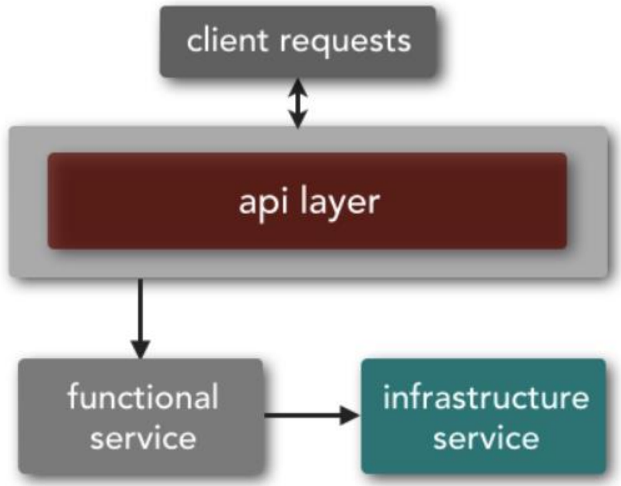
- Services and Microservices
- iDesign Architecture and Method
- Example

REFERENCES

- **Juval Lowy, Righting software, O'Reilly, 2020**
- **Mark Richards, Software Architecture Patterns, O'Reilly, 2015 [SAP]**
- **Mark Richards, Microservices vs. Service-Oriented Architecture O'Reilly, 2016**
- David Patterson, Armando Fox, Engineering Long-Lasting Software: An Agile Approach Using SaaS and Cloud Computing, Alpha Ed.[Patterson]
- Taylor, R., Medvidovic, N., Dashofy, E., Software Architecture: Foundations, Theory, and Practice, 2010, Wiley [Taylor]
- Ian Gorton. 2011. Essential Software Architecture. 2nd Edition, Springer-Verlag [Gorton]
- Microsoft Application Architecture Guide, 2009 [MAAG]
- Armando Fox, David Patterson, and Koushik Sen, SaaS Course Stanford, Spring 2012 [Fox]
- Jacques Roy, SOA and Web Services, IBM
- Mark Bailey, Principles of Service Oriented Architecture, 2008
- Erl, Thomas. Service-Oriented Architecture: Analysis and Design for Services and Microservices. Pearson Education. 2016
- Erl, Thomas. SOA Design Patterns, Prentice Hall, 2009.

  http://soapatterns.org

MICROSERVICES TOPOLOGY



Functional services

- support specific business operations or functions
- accessed externally and are generally not shared with any other service

Infrastructure services

- support nonfunctional tasks such as authentication, authorization, auditing, logging, and monitoring.
- not exposed to the outside world but rather are treated as private shared services only available internally to other services

GRANULARITY

SOA – [small application services; very large enterprise services]

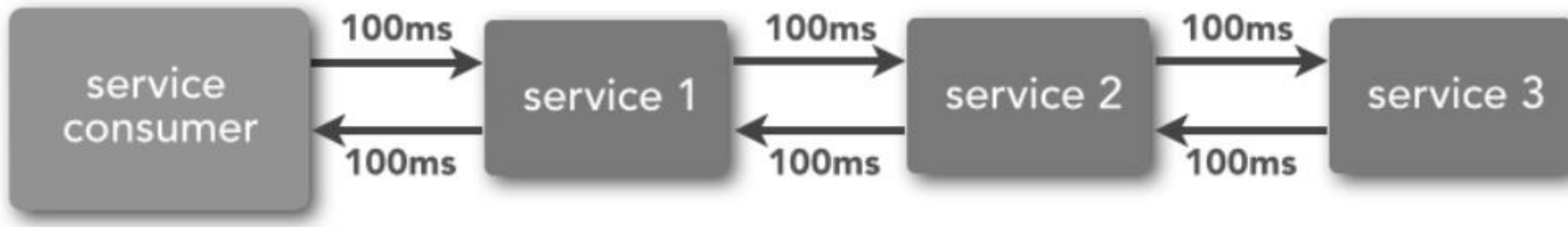
Ex. enterprise *Customer* service handles update and retrieval data views, delegating the lower-level getters and setters to application-level services that were not exposed remotely to the enterprise

Microservices - single-purpose services that do one thing really, really well

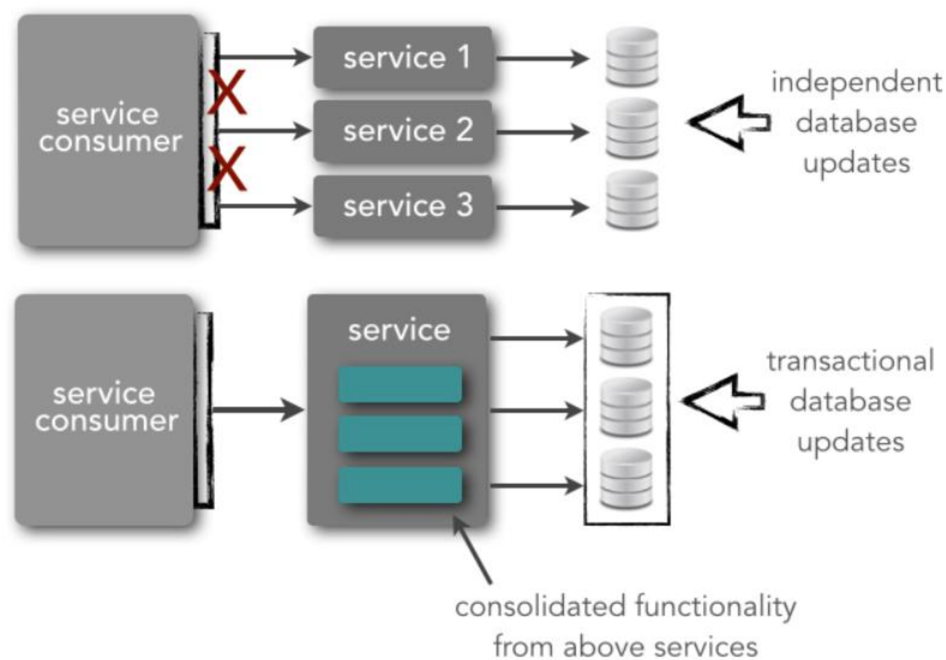
Ex. fine-grained getter and setter services like *GetCustomerAddress*, *GetCustomerName*, *UpdateCustomerName*

IMPACT OF GRANULARITY

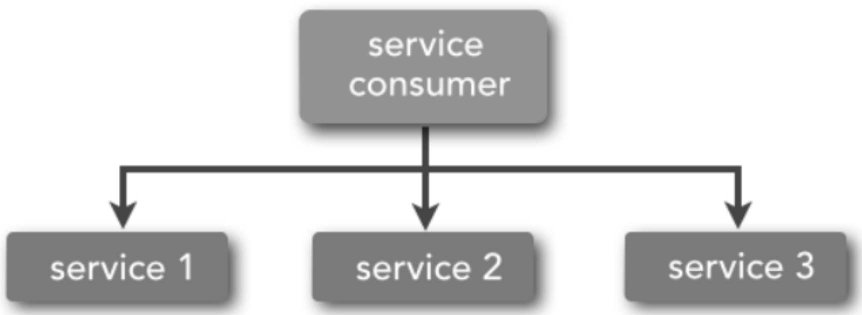
Performance

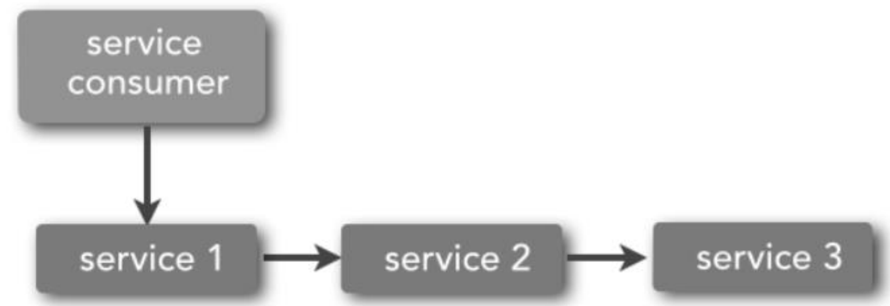


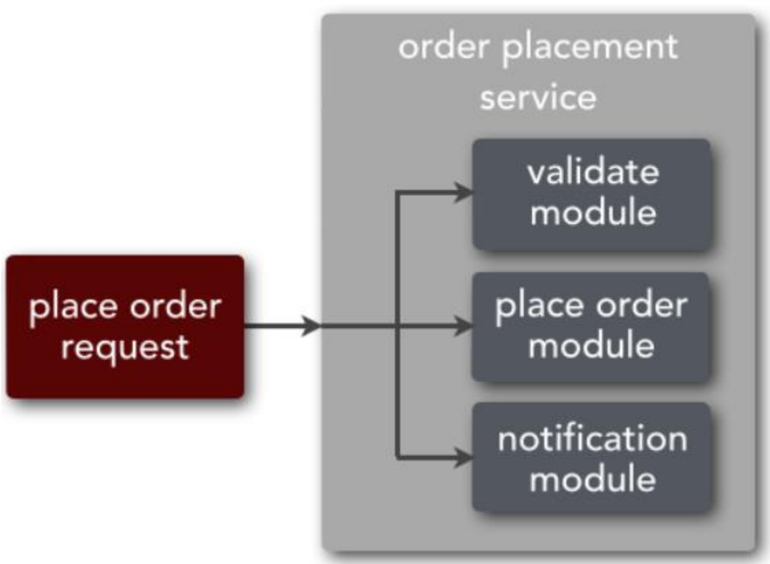
Transaction management

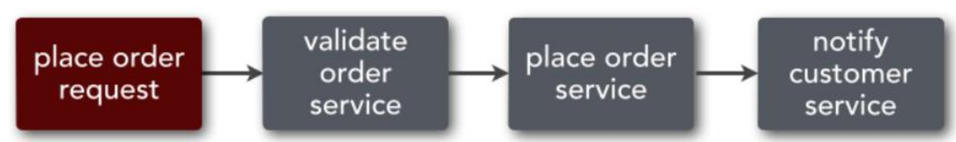


ORCHESTRATION AND CHOREOGRAPHY









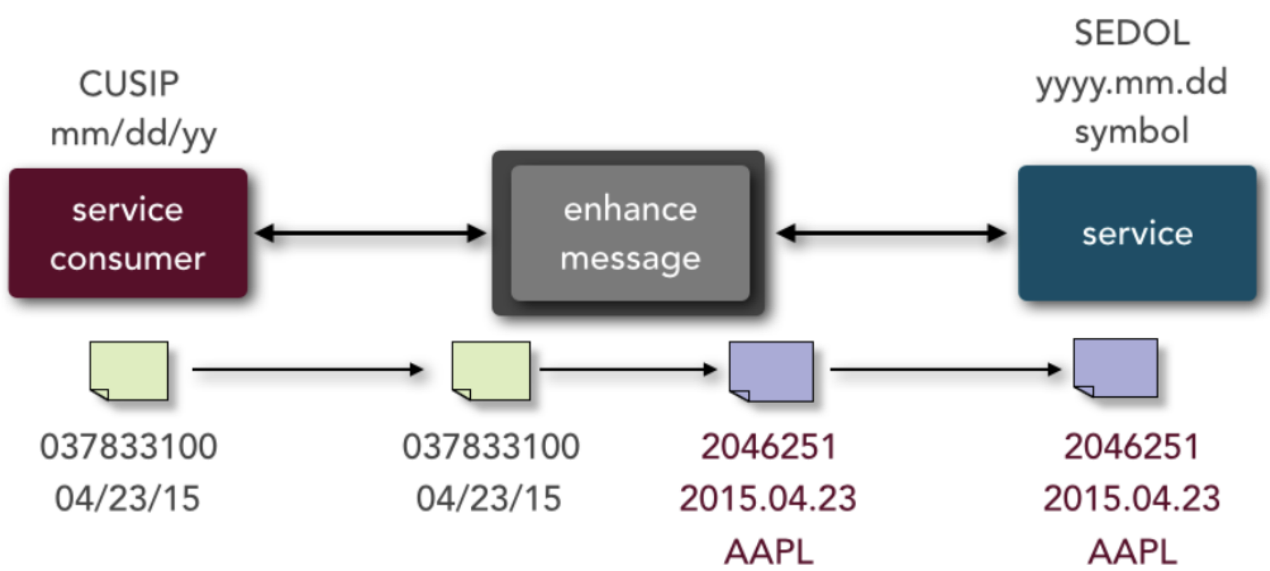
SOA MIDDLEWARE RESPONSIBILITIES

Mediation and routing - locate and invoke a service (or services) based on a specific business or user request

MORE

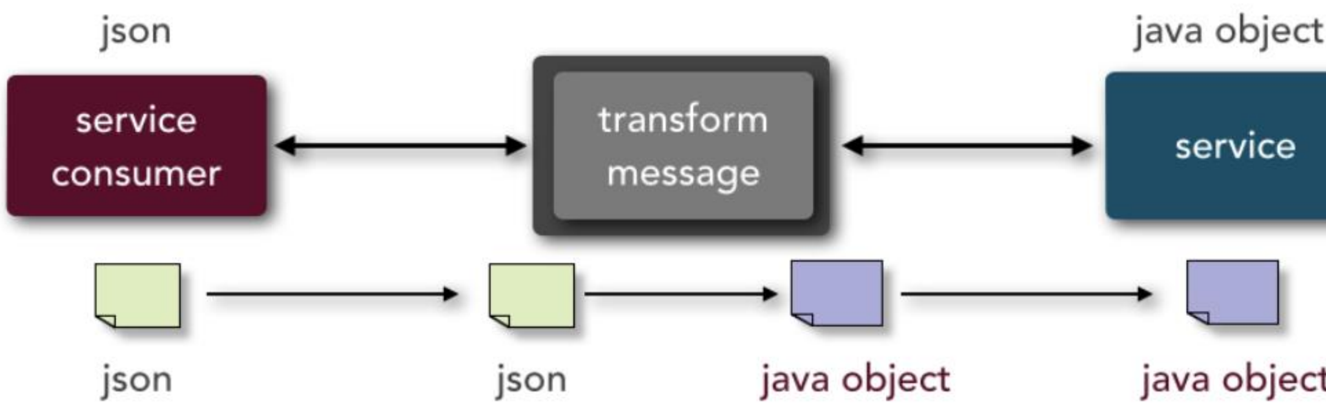
Message enhancement - modify, remove, or augment the data portion of a request before it reaches the service.

Ex. changing a date format, adding additional derived or calculated values to the request, and performing a database lookup to transform one value into another



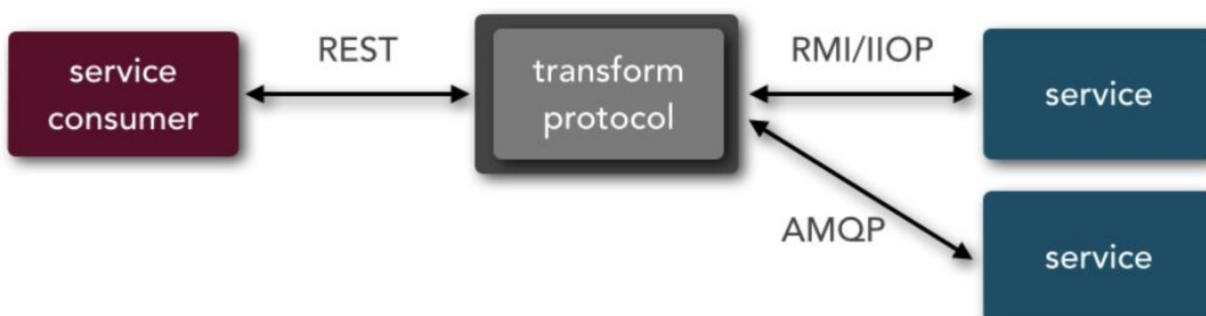
Message transformation - modify the format of the data from one type to other.

Ex. the service consumer is calling a service and sending the data in JSON format, whereas the service requires a Java object.



Protocol transformation - have a service consumer call a service with a protocol that differs from what the service is expecting.

Ex. the service consumer is communicating through REST, but the services invoked require an RMI/IIOP connection and an AMQP connection.



SOA VS. MICROSERVICES

- Share-as-much-as-possible
- Uses orchestration and choreography
- Uses Message middleware
- Coarse-grained services
- No pre-described limits as to which remote-access protocols can be used

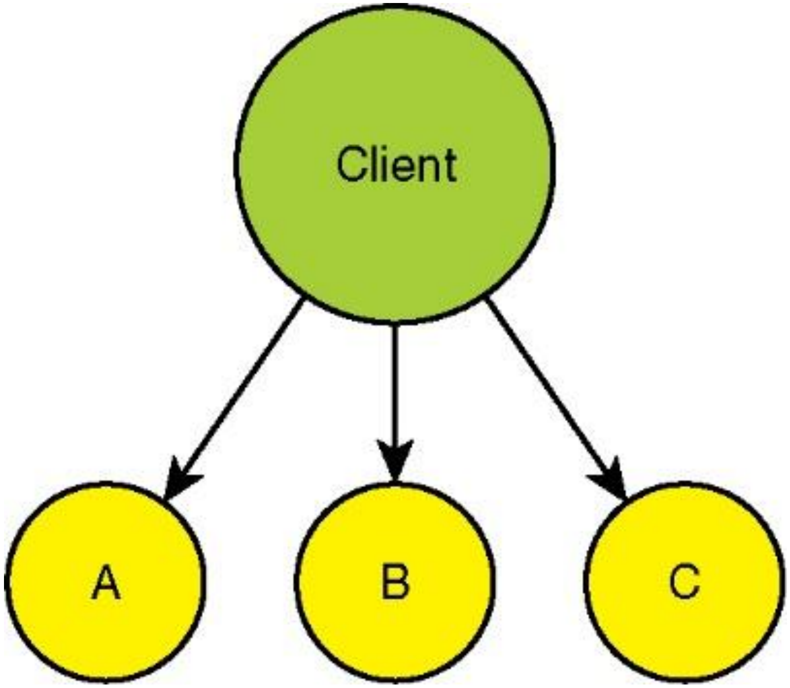
- Share-as-little-as-possible
- Favorizes choreography
- Uses API layer as service access façade
- Fine-grained services
- rely on 2 different remote-access protocols to access services: REST and simple messaging (JMS, MSMQ, AMQP, etc.)

IDESIGN METHOD FOR SOA

- Decomposition
- Structure
- Composition
- Example

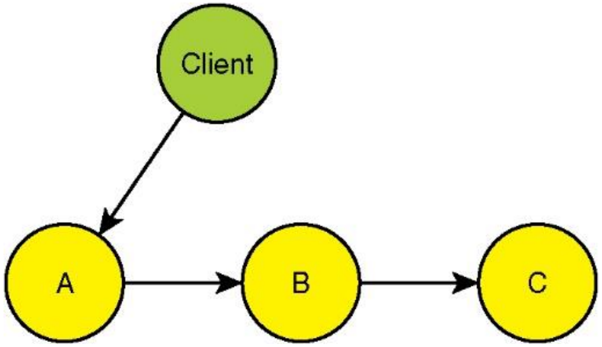
DECOMPOSITION

- Functional?
- Bloated client orchestrating functionality

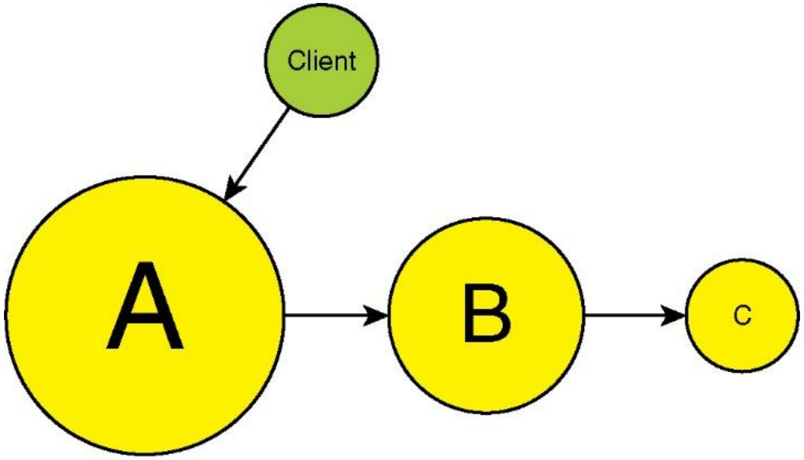


FUNCTIONAL DECOMPOSITION

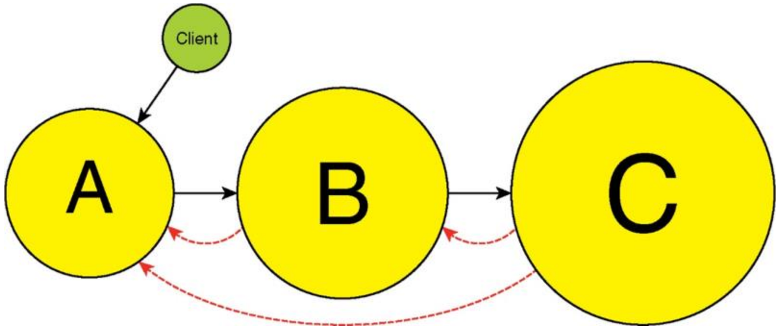
- Chaining functional services (choreography)



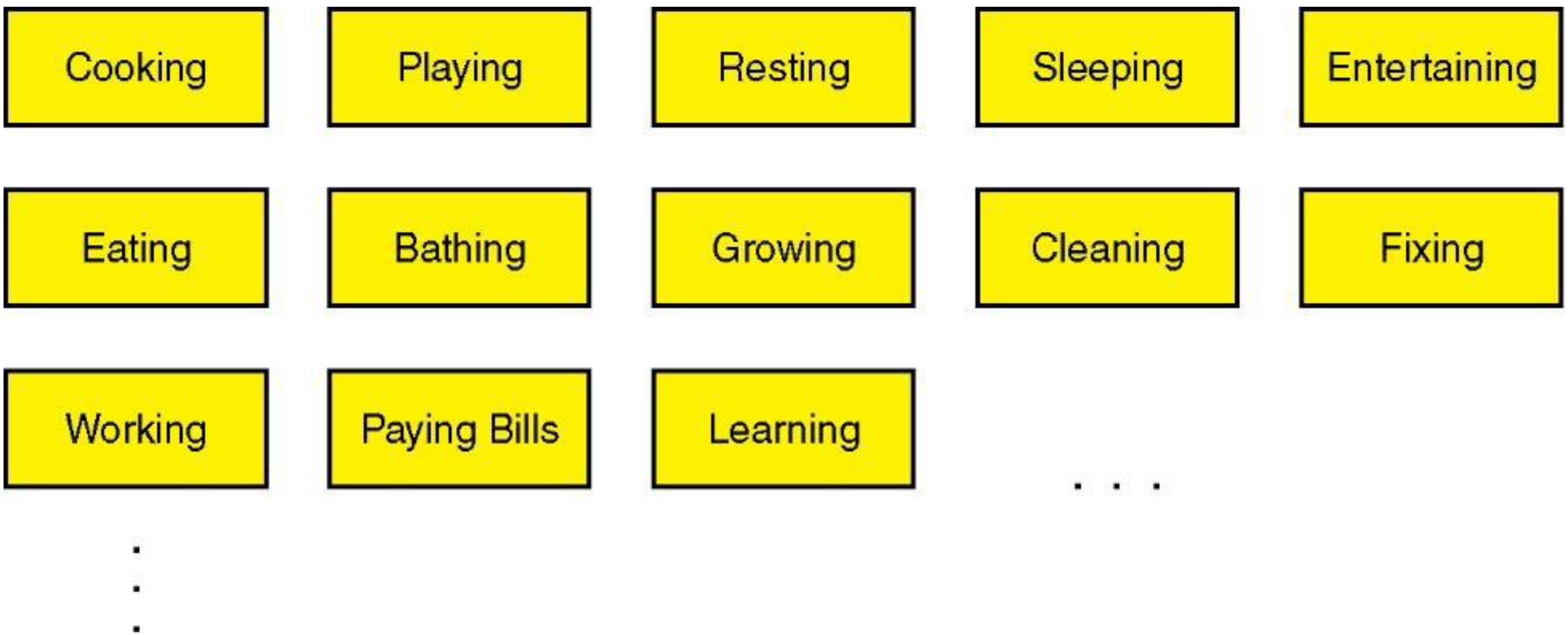
- Bloated services



- Additional bloating and coupling



FUNCTIONAL DECOMPOSITION OF A HOUSE



WHEN TO USE FUNCTIONAL DECOMPOSITION

- requirements discovery technique
- uncover requirements and their relationship,
- structure the requirements in a tree-like manner,
- identify redundancies or mutually exclusive functionalities

“The hallmark of a bad design is when any change to the system affects the client”

DOMAIN DRIVEN DECOMPOSITION OF A HOUSE

BUILDING A DOMAIN HOUSE

- start with a clean plot of land.
- dig a trench for the foundation for the kitchen, pour concrete for the foundation (just for the kitchen), and add bolts in the concrete.
- erect the kitchen walls (all have to be exterior walls); bolt them to the foundation;
- run electrical wires and plumbing in the walls; connect the kitchen to the water, power, and gas supplies; connect the kitchen to the sewer discharge; add heating and cooling ducts and vents; connect the kitchen to a furnace; add water, power, and gas meters;
- build a roof over the kitchen; screw drywall on the inside; hang cabinets; coat the outside walls (all walls) with stucco; and paint it.
- announce to the customer that the Kitchen is done and that milestone 1.0 is met.

VOLATILITY DRIVEN DECOMPOSITION

Change

CLIENT LAYER VOLATILITIES

- Client types: web application, rich desktop application, mobile app, holograms, APIs, etc.
- Different technologies, deployments, versions, life cycles, development teams
- Same entry points to the system, same access security, data types

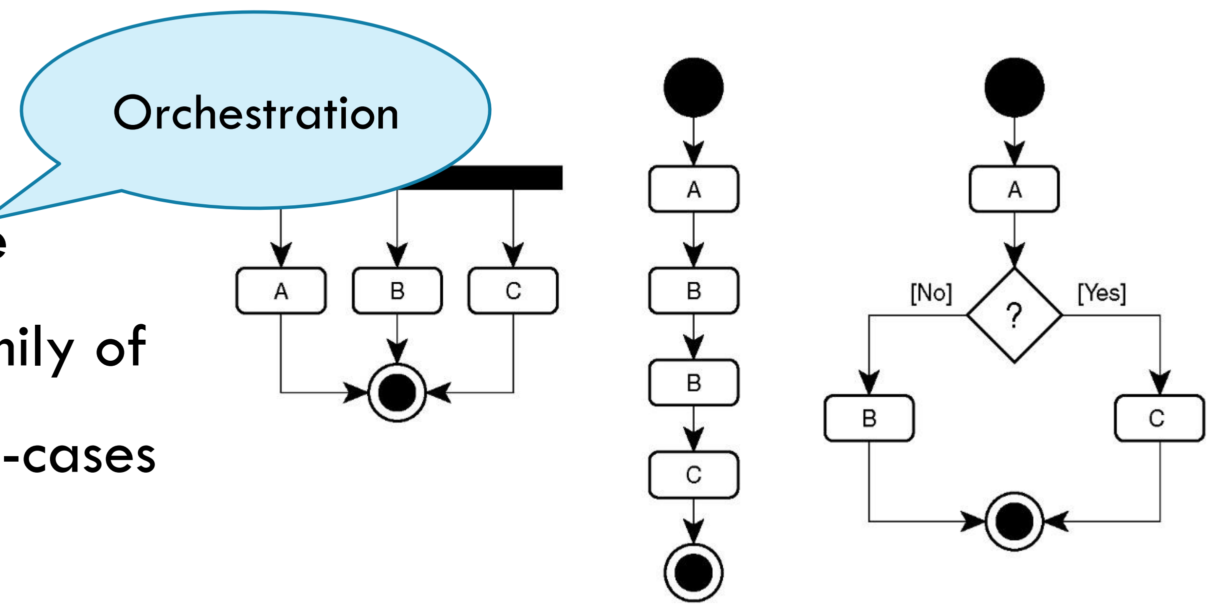
BUSINESS LOGIC LAYER VOLATILITIES

- Axes of volatility
  - Same client over time
  - Several clients at one point in time

- Sequence volatility

=> **Manager service**

⇒Encapsulates a family of logically related use-cases

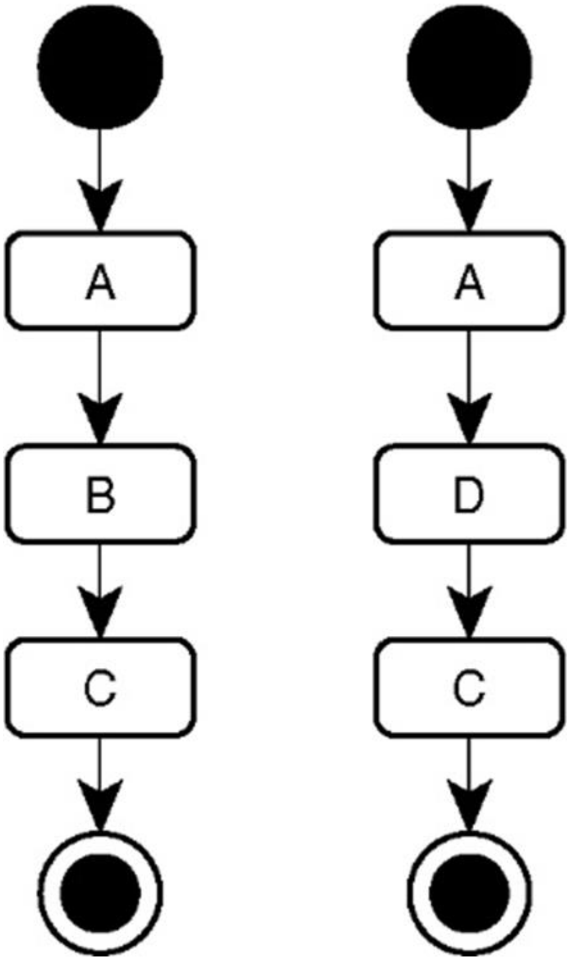


BLL VOLATILITIES - 2

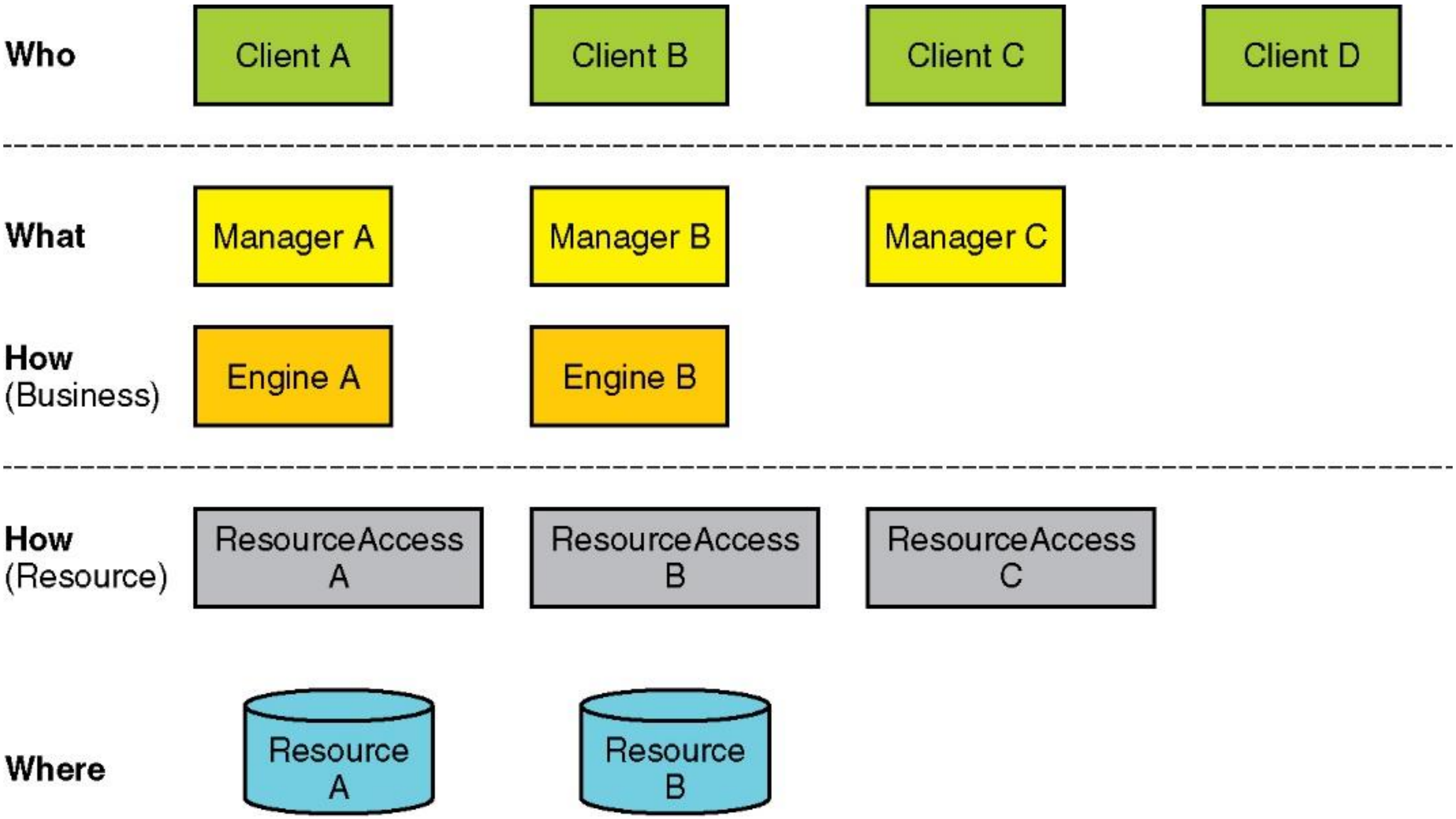
- Activity volatility

⇒**Engine service**

⇒Encapsulates business rules and activities



IDESIGN STRUCTURE



MANAGERS VS ENGINES

- *Managers* may use **zero or more** *Engines*
- *Engines* may be **shared** between *Managers*
- *Engines* should be **highly reusable**
- *Manager-Engines* = materializations of the **Strategy** Design Pattern

RESOURCE ACCESS AND RESOURCES

- *ResourceAccess* services encapsulate the volatility in **accessing** a *Resource*
- A *Resource* can be a relational database, file system, a cache, a message queue, a distributed cloud-based hash-table, etc.
- Well-designed *ResourceAccess* components expose in their contract the **atomic business verbs** around a resource.

Example: in a banking system, atomic business verbs are *debit* and *credit* operations on an Account.

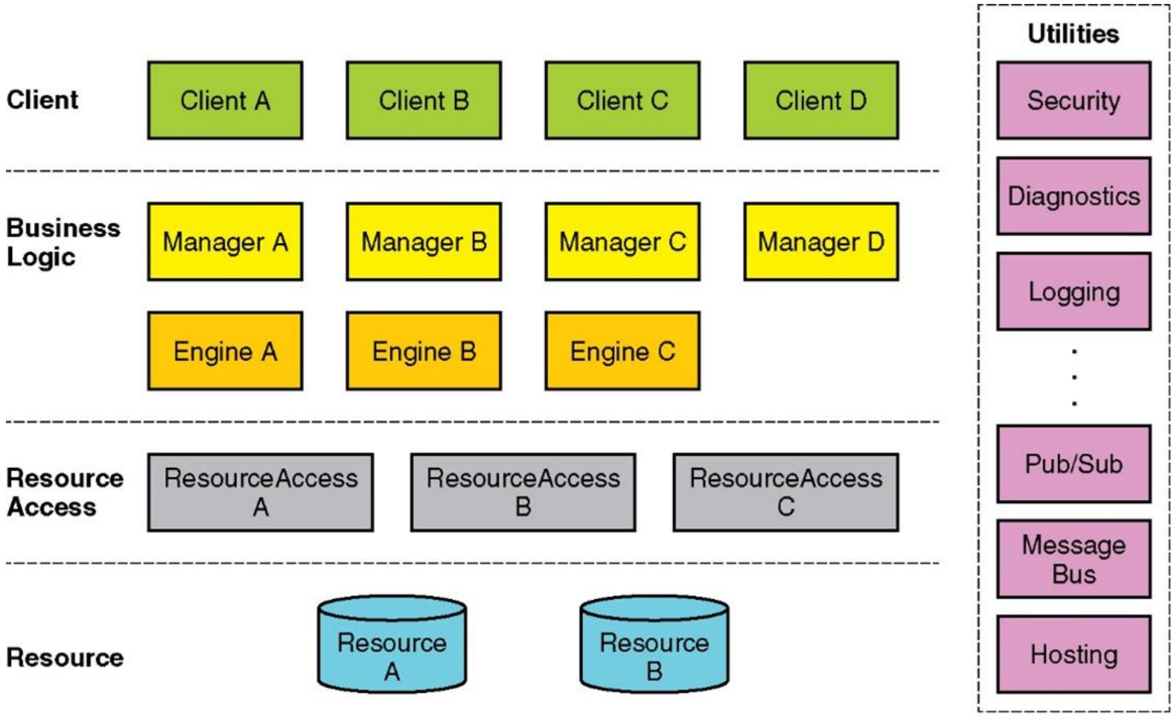
- The *ResourceAccess* component translates atomic business verbs (i.e. transactions) into CRUD operations on the specific *Resource*

RESOURCE ACCESS AND RESOURCES

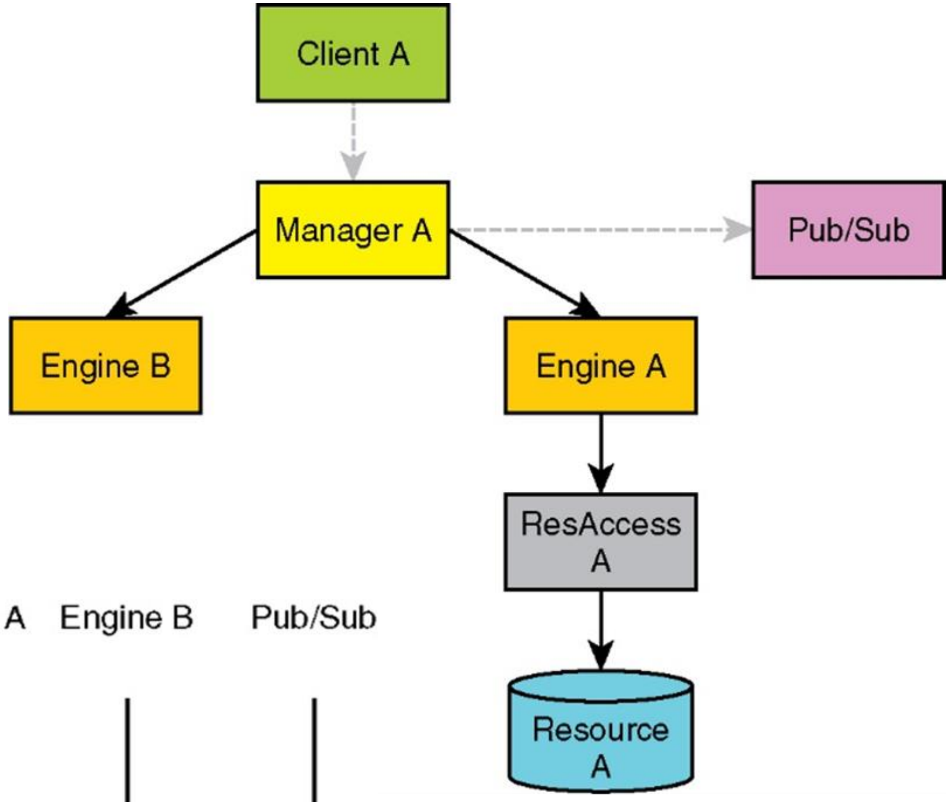
- *ResourceAccess* services can be **shared** between *Managers* and *Engines*.
- The *Resource* can be internal to the system or outside the system. Often, the *Resource* is a whole system in its own right (ex. a SQL DBMS) .
- *Resource* **changes** invariably change *ResourceAccess* as well.

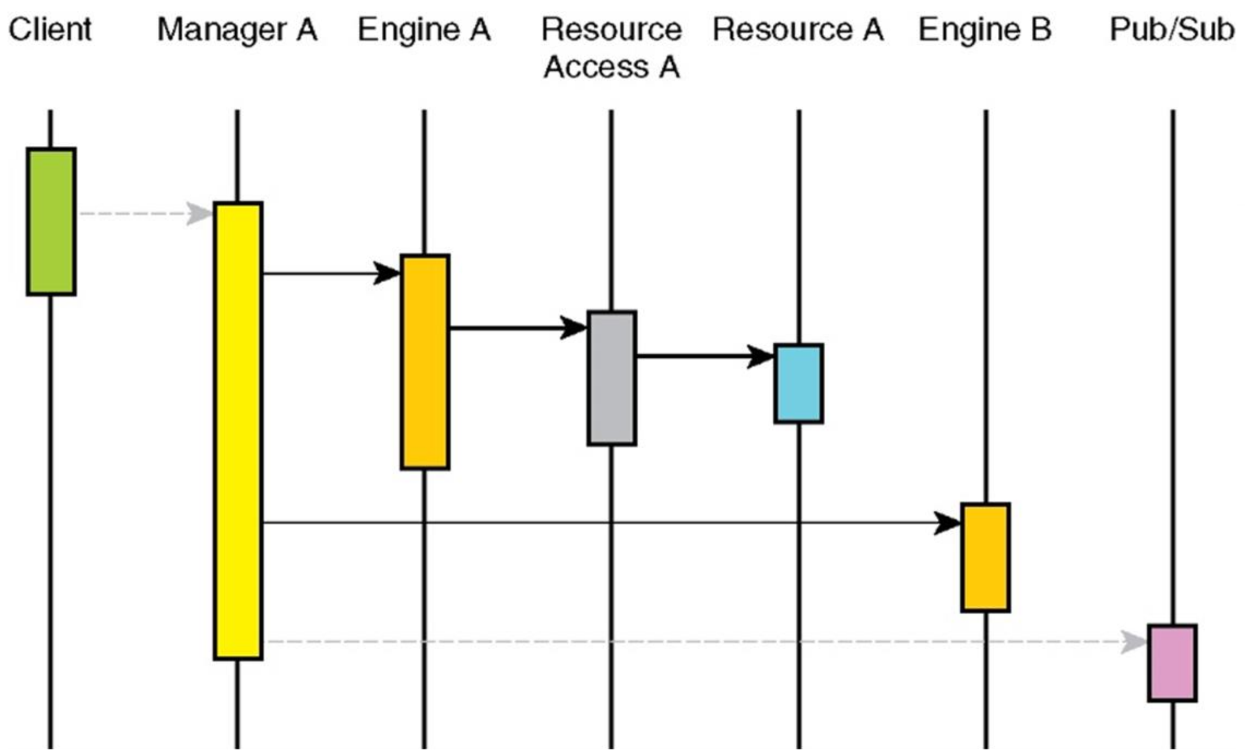
UTILITIES

- Services building a common infrastructure that nearly all systems require
- *Utilities* may include Security, Logging, Diagnostics, Instrumentation, Pub/Sub, Message Bus, Hosting, etc.



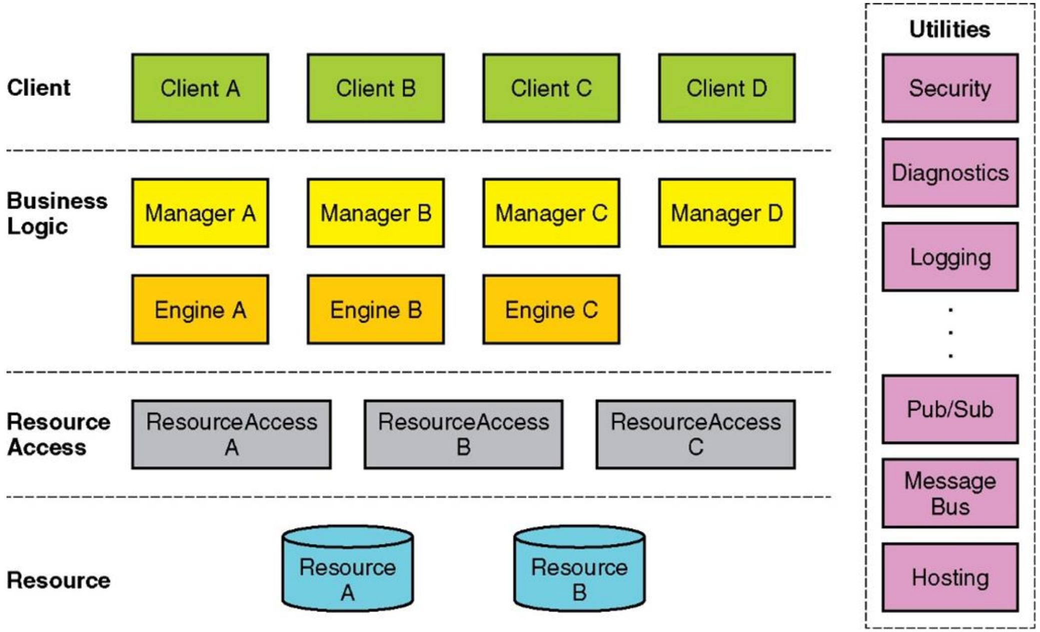
INTERACTIONS



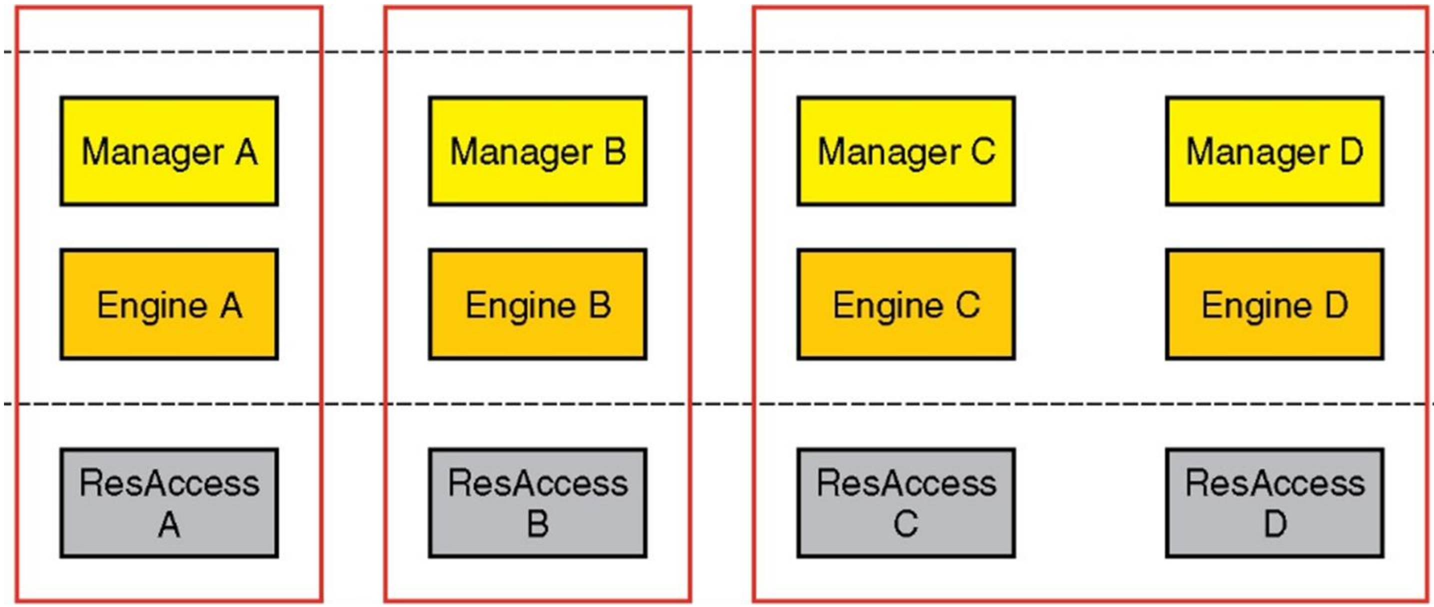


COMMENTS

- *Managers* to *Engines* ratio
  - 8 > #Managers > #Engines
- Volatility decreases top-down
- Reuse increases top-down



MICROSERVICES



"I fear that microservices will be the biggest failure in the history of software. Maintainable, reusable, extensible services are possible—just not in this way."

DON'TS!

- *Clients* do not call multiple *Managers* in the same use case
- *Clients* do not call *Engines*
- *Managers* do not queue calls to more than one *Manager* in the same use case
- *Engines* do not receive queued calls
- *ResourceAccess* services do not receive queued calls
- *Clients* do not publish events
- *Engines* do not publish events
- *ResourceAccess* services do not publish events
- *Resources* do not publish events
- *Engines*, *ResourceAccess*, and *Resources* do not subscribe to events.
- *Engines* never call each other
- *ResourceAccess* services never call each other.

COMPOSITION

- Objective:
  - Address the current requirements +
  - Withstand future requirements
  - Validate Design

$\Rightarrow$ ***Never design against (i.e. based on) the requirements***

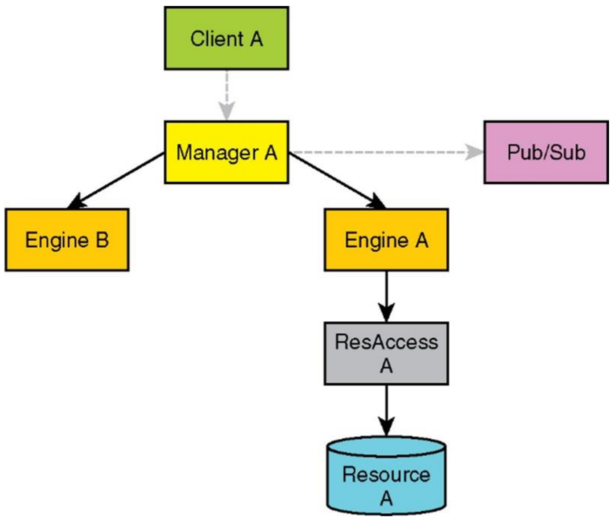
- Use-cases = behavior
- Identify Core Use-cases (usually <= 6)!
- Core use-cases represent the essence of the business
- Core use-cases are not explicitly stated
- Changing use-cases => different interaction between components, **not a different decomposition**!

COMPOSABLE DESIGN

- Architects mission is to identify **the smallest set of components** that he can, put together, **satisfy all the core use cases.**
- Example: Requirements specs have 300 use-cases. How many services?
  - 1 service => monolithic architecture 
  - 300 services => difficult to integrate 
  - Order of magnitude = 10 
  - Typically: 2 – 5 *Managers*, 2 -3 *Engines*, 3 - 8 *ResourceAccess* and *Resources*, 2 - 6 *Utilities.*

VALID DESIGN

- **Valid design** if you can produce an interaction between your services **for each core use case**
- Call chain diagram shows the interaction between components required to satisfy a particular use case
  - a solid black arrow for synchronous (request/response) calls,
  - a dashed gray arrow for a queued call



FROM DESIGN TO FEATURES

- ***Features are always and everywhere aspects of integration, not implementation.***
- Containing the change:
  - *Manager* implements the workflow executing the use case. The *Manager* may be gravely affected by a change.
  - The underlying components that the *Manager* integrates are not affected by the change

FROM DESIGN TO FEATURES

- Implementing *Engines* is expensive. Each *Engine* represents business activities vital to the system's workflows and encapsulates the associated volatility and complexity.
- Implementing a *ResourceAccess* is nontrivial. It involves:
  - Identifying the atomic business verbs,
  - translating them into the access methodologies for some *Resource*,
  - exposing them as a *Resource*-neutral interface.
- Designing and implementing *Resources* that are scalable, reliable, highly performant, and very reusable is time- and effort-consuming. These tasks may include:
  - designing data contracts, schemas, cache access policies, partitioning, replication, connection management, timeouts, lock management, indexing, normalization, message formats, transactions, delivery failures, poison messages, and much more.

FROM DESIGN TO FEATURES

- Implementing *Utilities* always requires top skills, and the result must be trustworthy. *Utilities* are the backbone of your system. World-class security, diagnostics, logging, message processing, instrumentation, and hosting do not happen accidentally.
- Designing a superior user experience or a convenient and reusable API for *Clients* is time and labor intensive. The *Clients* also have to interface and integrate with the *Managers*.

SYSTEM DESIGN EXAMPLE [IDESIGN CHAPTER 5]

- TradeMe, a system for matching tradesmen to contractors and projects.
- Each tradesman has a skill level, and some are certified by regulators to do certain tasks.
- The payment rate for the tradesman varies based on various factors such as discipline, skill level, years of experience, project type, location, and even weather.
- The contractors are general contractors, and they need tradesmen on an ad hoc basis, from as little as a day to as long as a few weeks.
- Tradesmen can come and go on a single project.
- Tradesmen can sign up, list their skills, their general geographic area of availability, and the rate they expect.
- Contractors can sign up, list their projects, the required trades and skills, the location of the projects, the rates they are willing to pay, the duration of engagement, and other attributes of the project. Contractors can even request specific tradesmen with whom they would like to work.

TRADEME BEHAVIOR

- The system lets market forces set the rate and find equilibrium.
- The projects are construction projects for buildings. The system may also be useful in newly emerging markets, such as oil fields or marine yards.
- The system processes the requests and dispatches the required tradesmen to the work sites. It also keeps track of the hours and wages, and the rest of the reporting to the authorities.
- The system isolates tradesmen from contractors. It collects funds from the contractors and pays the tradesmen. Contractors cannot bypass the system and hire the tradesmen directly
- TradeMe aims to find the best rate for the tradesmen and the most availability for the contractors.
- Both tradesmen and contractors are members in the system and pay an (annual) fee.

TRADEME LEGACY STATUS

- Presently, nine call centers handle the majority of the assignments. Each call center is specific to a particular locale, regulations, building codes, standards, and labor laws.
- Call centers are staffed with account representatives called reps.
- The legacy system, deployed in European call centers, has full-time users using a two-tier desktop application connected to a database
- Some rudimentary web portals for managing membership bypass the legacy system and work with the database directly
- Users are required to employ as many as five different applications to accomplish their tasks.
- The client applications are independent, each has its own repository, is full of business logic, and the UI and business logic are not separated.
- Vulnerable to security attacks. Was never designed at all, but rather grew organically.

TRADEME NEW DESIRED FEATURES

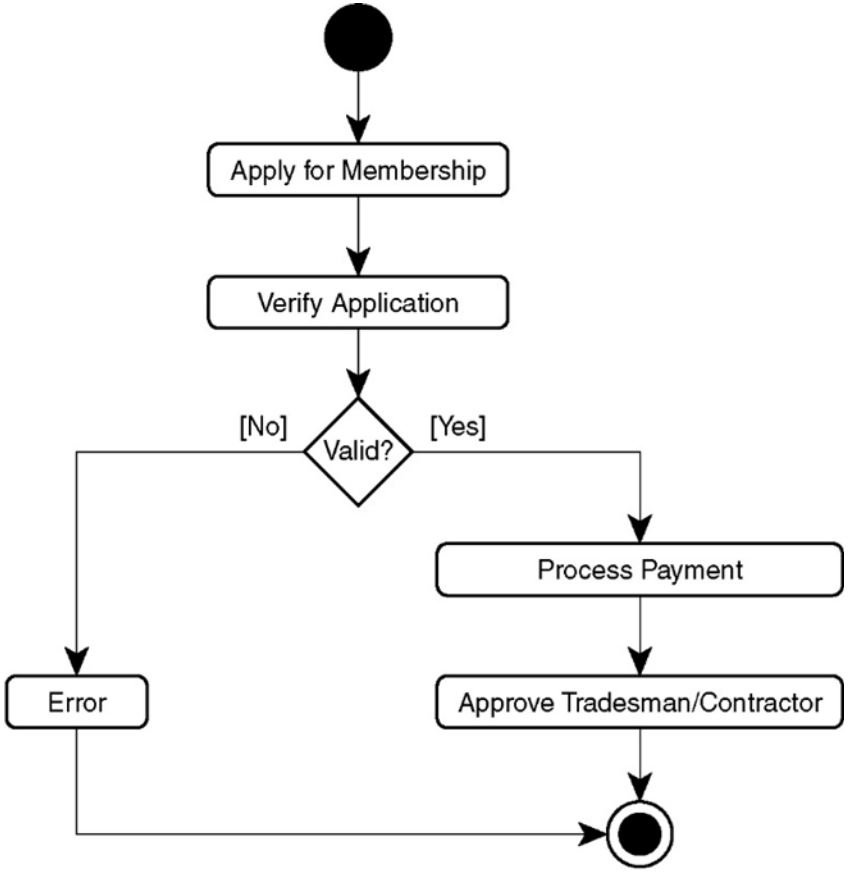
- Mobile device support
- Higher degree of automation of the workflow
- Some connectivity to other systems
- Migration to the cloud
- Fraud detection
- Quality of work surveys, including incorporating the tradesman's safety record in the rate and skill level
- Entering new markets (such as deployment at marine yards)

TRADEME CHALLENGES

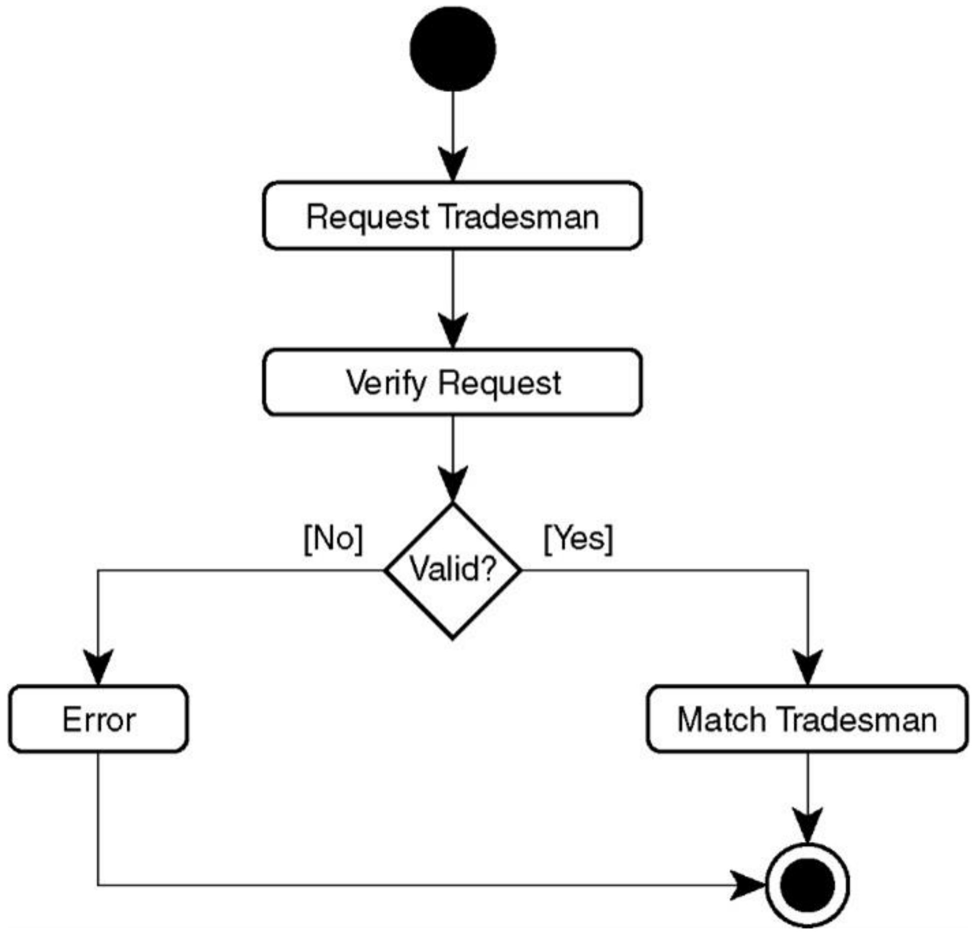
- inability to integrate new features (i.e. integration with external education centres for continuing education of tradesmen)
- adapting to new legislation across locales
- automated processing flows

TRADEME USE CASES

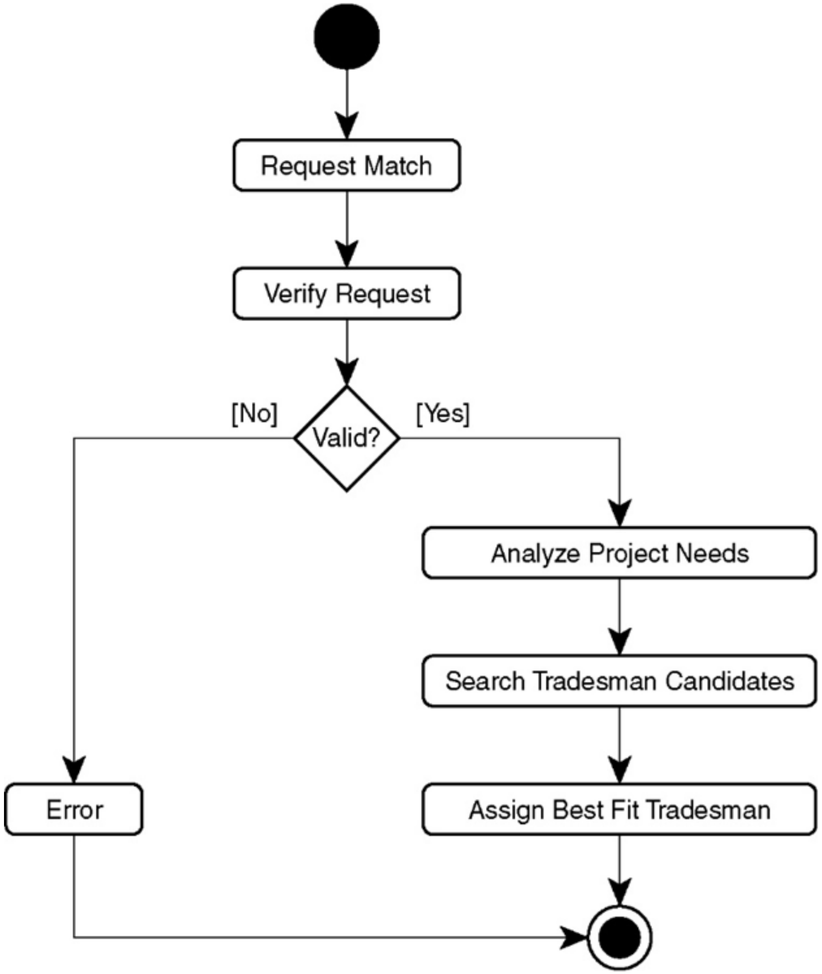
- Add Tradesman or Contractor use case



REQUEST TRADESMAN OR CONTRACTOR USE CASE



MATCH TRADESMAN USE CASE



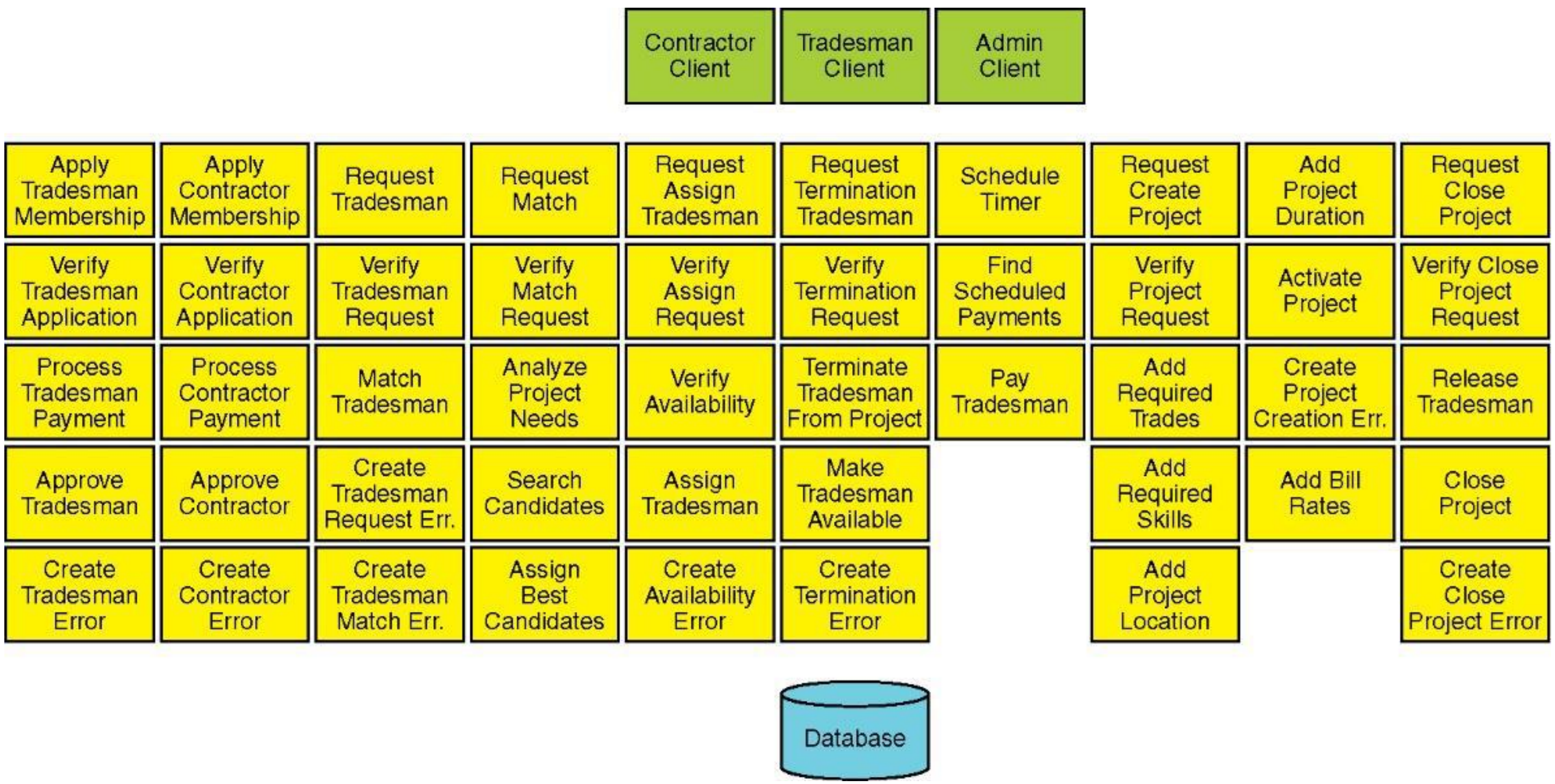
CORE USE CASE

TradeMe is a system for matching tradesmen to contractors and projects

- 3 types of roles:
  - the users (i.e. back-office data entry reps, sysadmins),
  - the market,
  - the members (tradesmen, contractors)

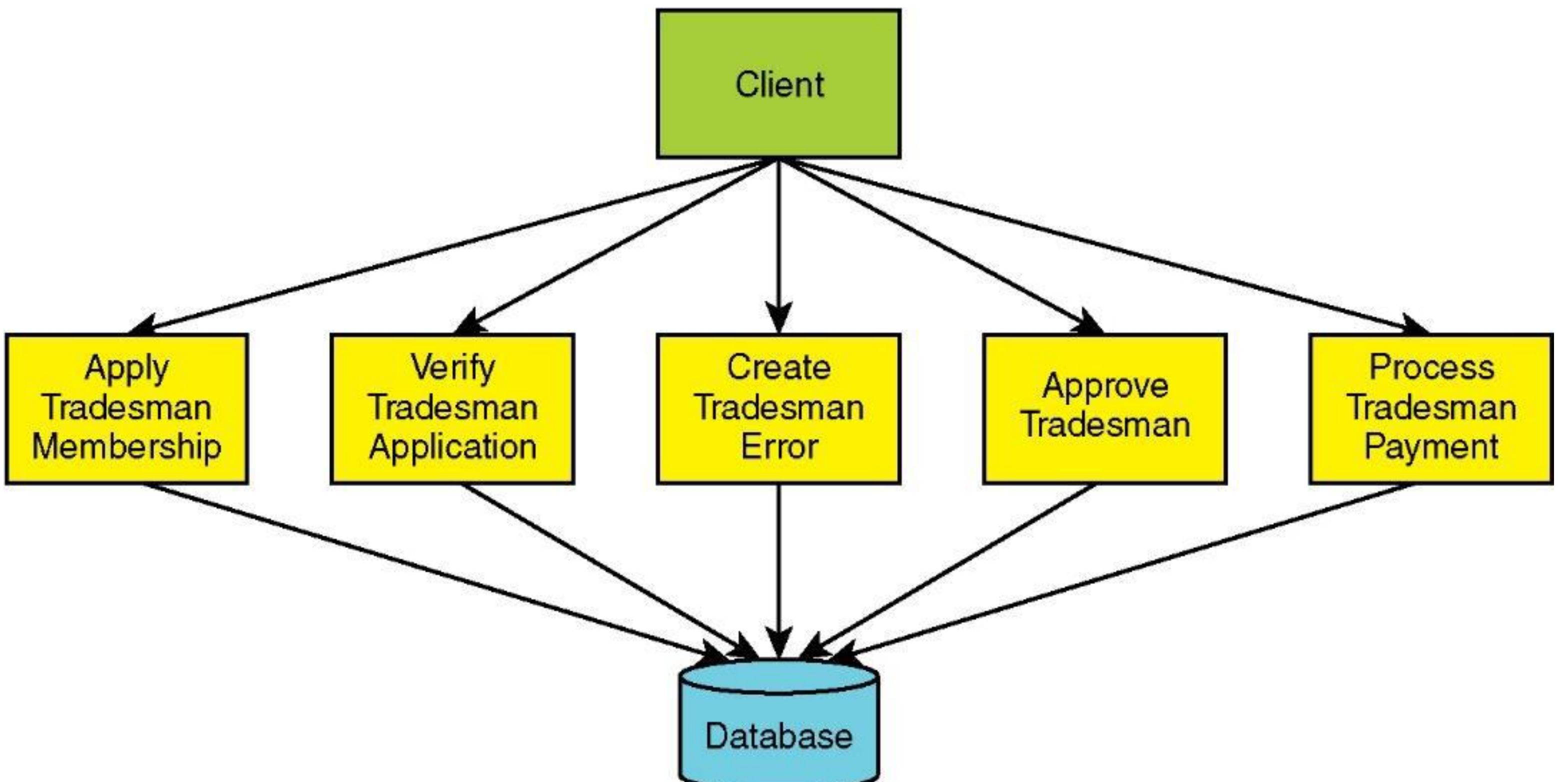
ANTI-PATTERN APPROACHES

- Monolith – one big service
- Granular components – one component per activity



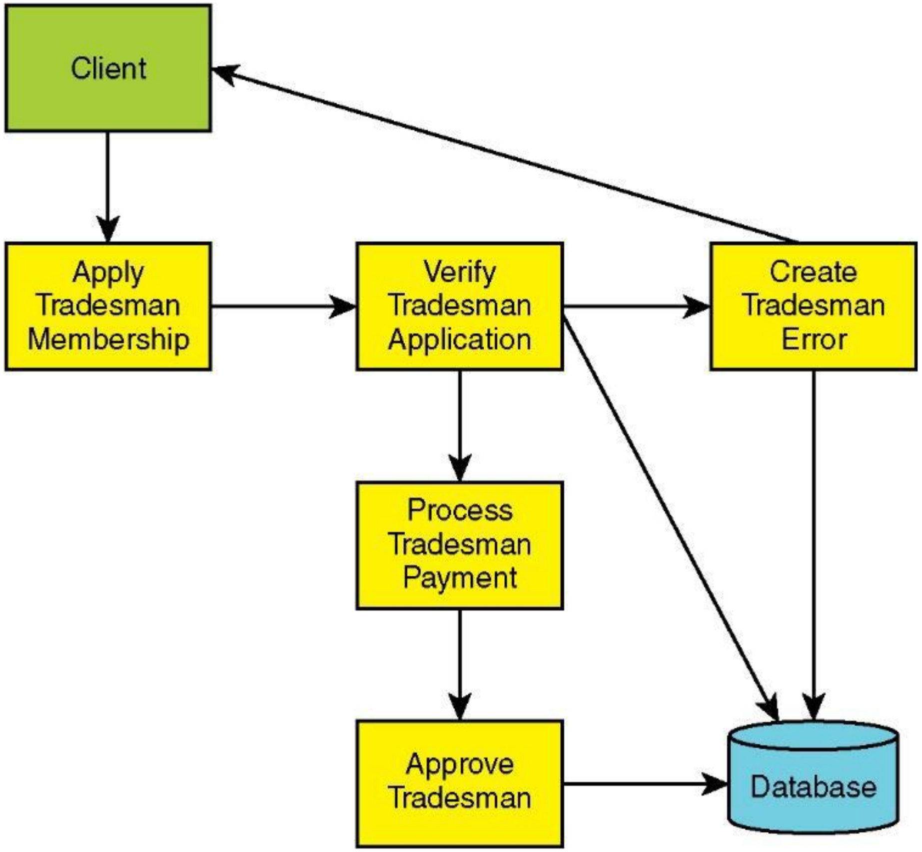
CLIENT AS ORCHESTRATOR

- Granular components – one component per activity



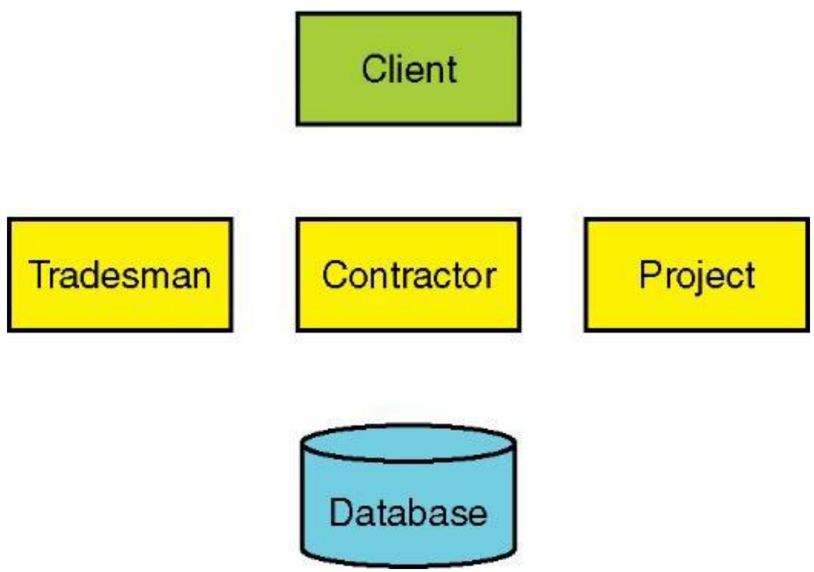
CHOREOGRAPHY – BASED APPROACH

- Granular components – one component per activity



DOMAIN DECOMPOSITION

- Large number of decomposition alternatives
- Difficult to validate the design
- Example: who handles a request for a tradesman? The Project? The Tradesman?



IDESIGN METHOD

- The Who
  - Tradesmen
  - Contractors
  - TradeMe reps
  - Education centers
  - Background processes (i.e., scheduler for payment)
- The What
  - Membership of tradesmen and contractors
  - Marketplace of construction projects
  - Certificates and training for continuing education

IDESIGN METHOD

- The How
  - Searching
  - Complying with regulations
  - Accessing resources
- The Where
  - Local database
  - Cloud
  - Other systems

AREAS OF VOLATILITY - 1

- Client applications
  - different users (i.e. tradesmen, contractors etc. background processes)
  - different UI technologies, devices, APIs
  - different access (local, net)
- Membership management (Membership Manager)
  - different benefits/discounts
  - different local rules
- Fees (Market Manager)
  - all the ways TradeMe makes money

AREAS OF VOLATILITY - 2

- Projects (Market manager)
  - different requirements and sizes
- Disputes (Membership Manager)
  - different dispute resolution methods
- Matching and approvals
  - searching criteria and their definition (Market Manager)
  - searching methods for a tradesman (Search Engine)
- Education
  - matching a training class to a tradesman (Education Manager)
  - searching for classes and certifications (Search Engine)
  - compliance with certification regulations (Regulation Engine)

AREAS OF VOLATILITY - 3

- Regulations (Regulation Engine)
  - different internal, external regulations
- Reports (Regulation Engine)
  - different types of reports
- Localization (Regulation Engine)
  - language and culture
  - local specific regulations
  - may affect the design of Resources
- Resources
  - portals to external systems
  - cloud-based database for tradesmen, projects, contractors
  - local database

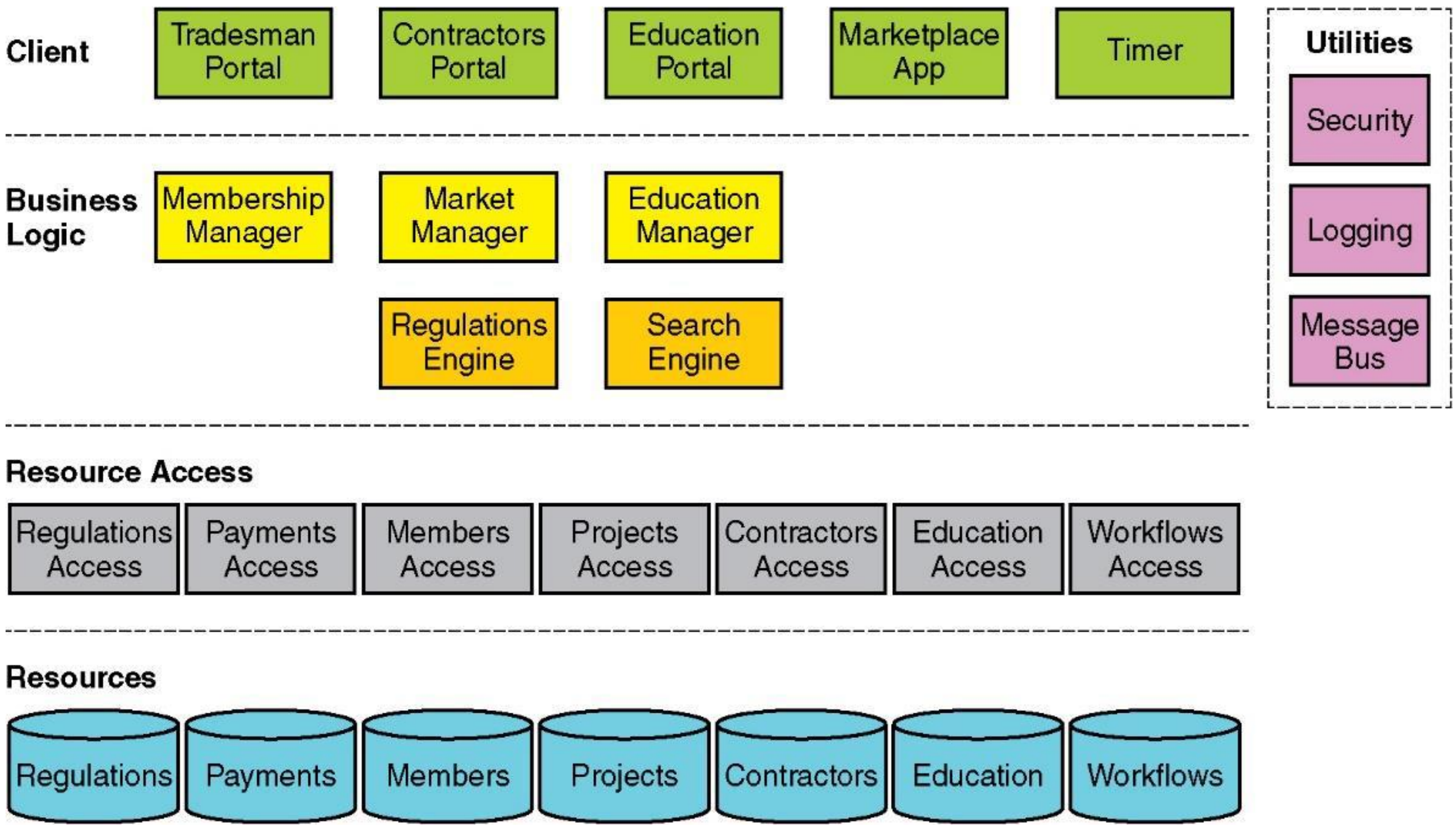
AREAS OF VOLATILITY - 4

- Resource Access (Regulation Engine)
  - different internal, external regulations
- Deployment model (Message Bus Utility)
  - Sometimes data cannot leave a geographic area, or
  - the company may wish to deploy parts or whole systems in the cloud.
- Authentication and authorization (Security Utility)
  - multiple options for representing credentials and identities.
  - many ways of storing roles or representing claims.

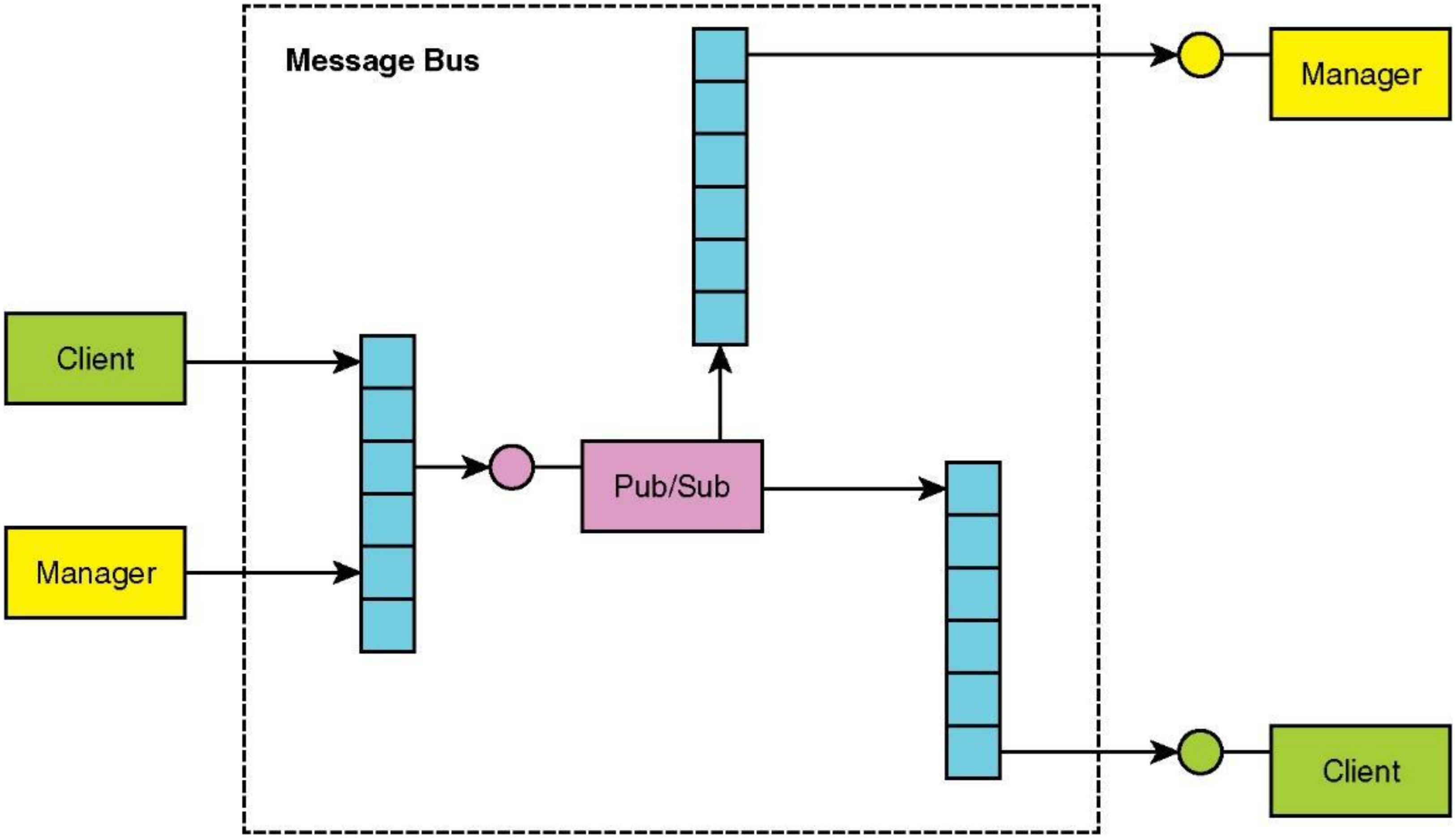
WEAK VOLATILITIES

- Notification (Regulation Engine)
  - different internal, external regulations
- Analysis (Regulation Engine)
  - different types of reports

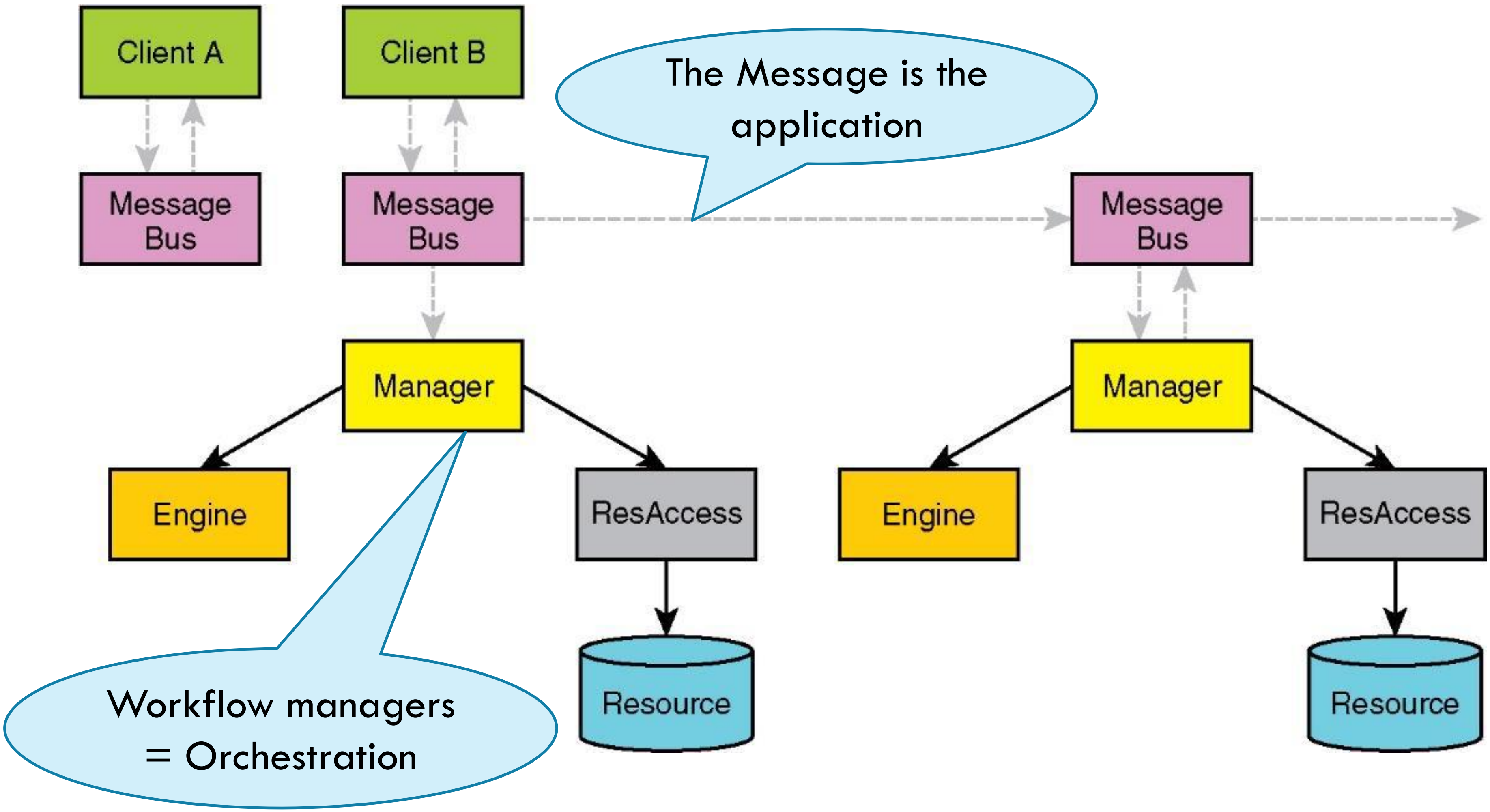
STATIC ARCHITECTURE



MESSAGE BUS



OPERATIONAL INTERACTION



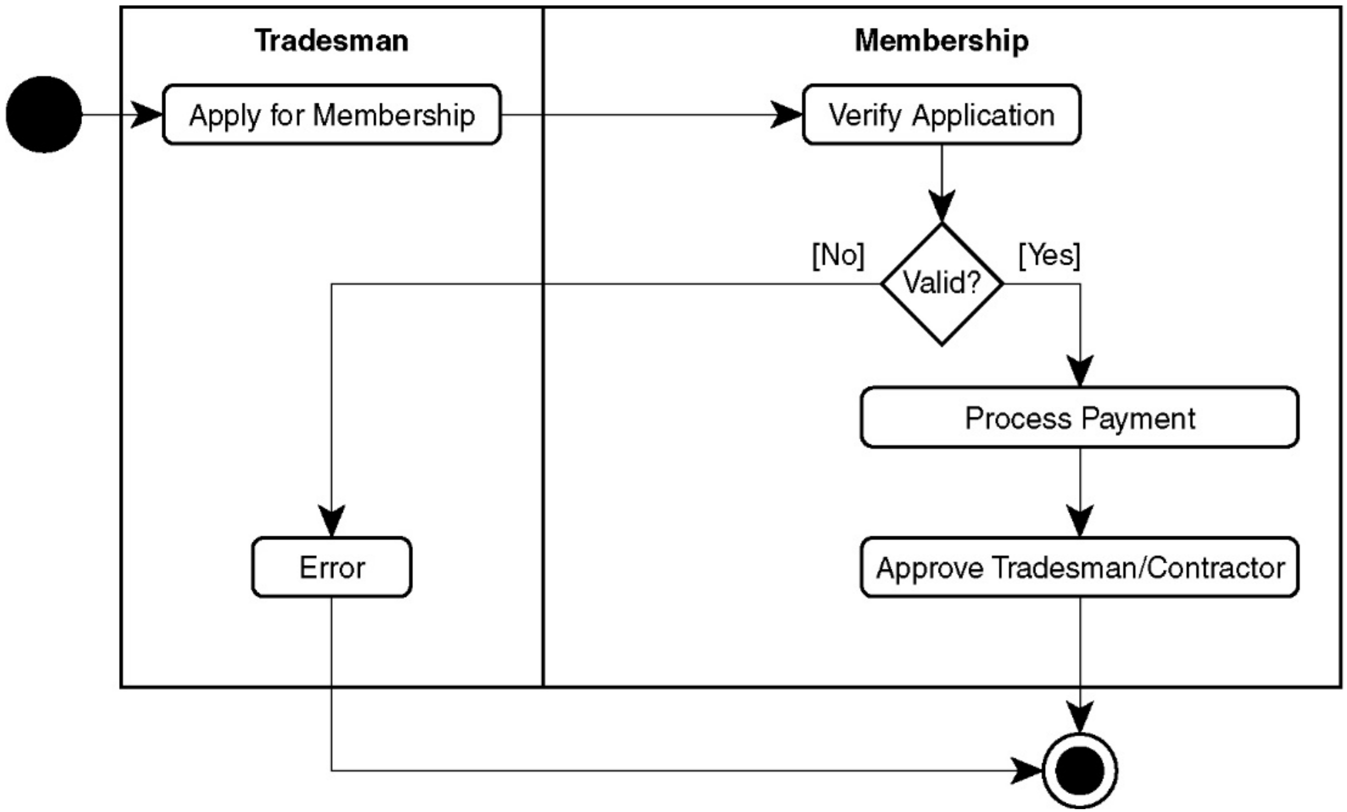
OPERATIONAL INTERACTION

- The *Clients* and the business logic in the subsystems are **decoupled** from each other by the *Message Bus*
- The ***Message Bus* encapsulates** the nature of the messages, the location of the parties, and the communication protocol
- No use case initiator *(i.e. Clients)* and use case executioner *(i.e. Managers)* ever interact directly.
- A multiplicity of **concurrent Clients can interact in the same use case**, with each performing its part of the use case.
- **High throughput** is possible because the queues underneath the Message Bus can accept a very large number of messages per second.

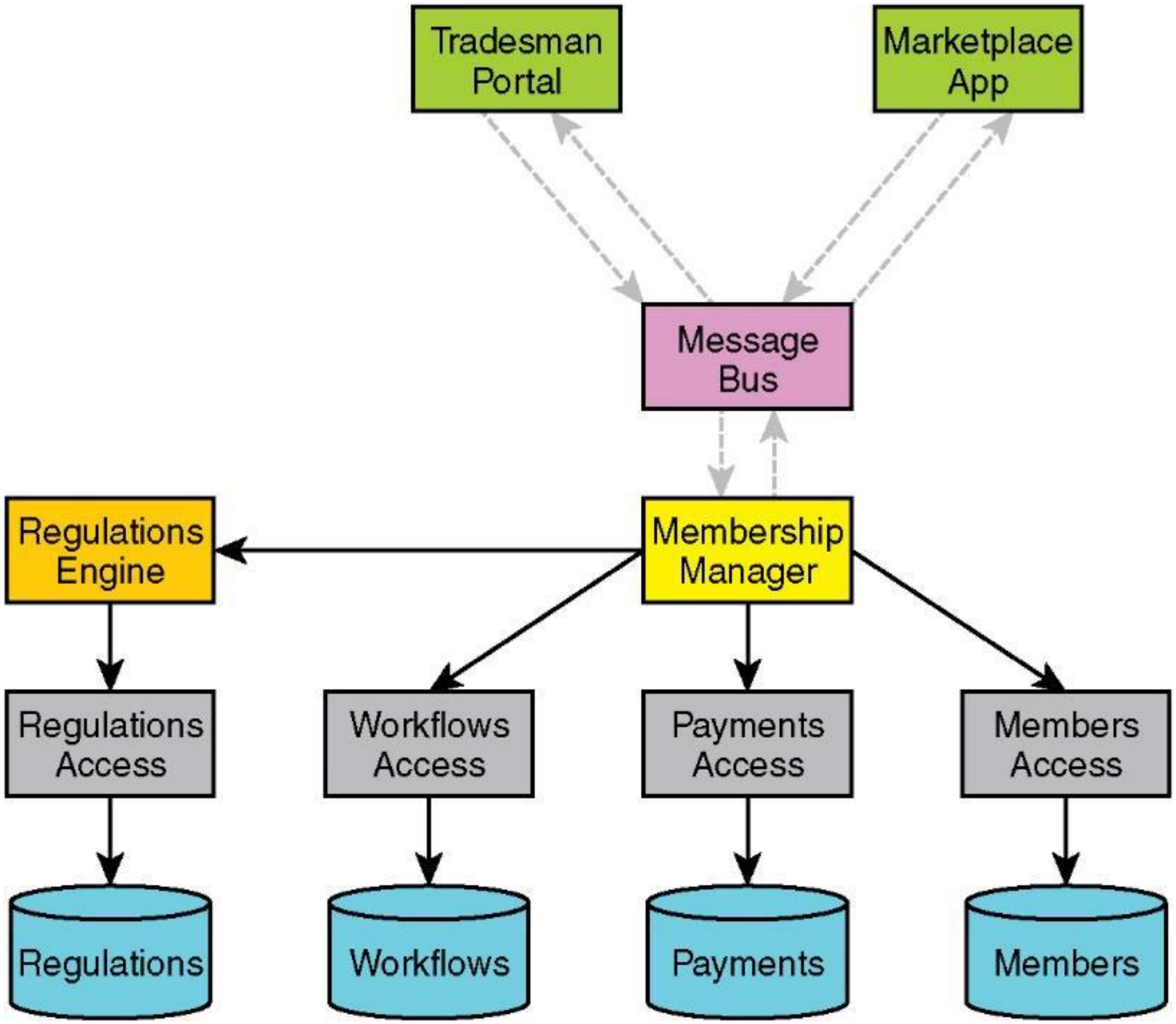
ARCHITECTURE VALIDATION

Take each use-case and show how the components integrate to support it.

Add Tradesman/Contractor Use Case



ADD TRADESMAN/CONTRACTOR USE CASE

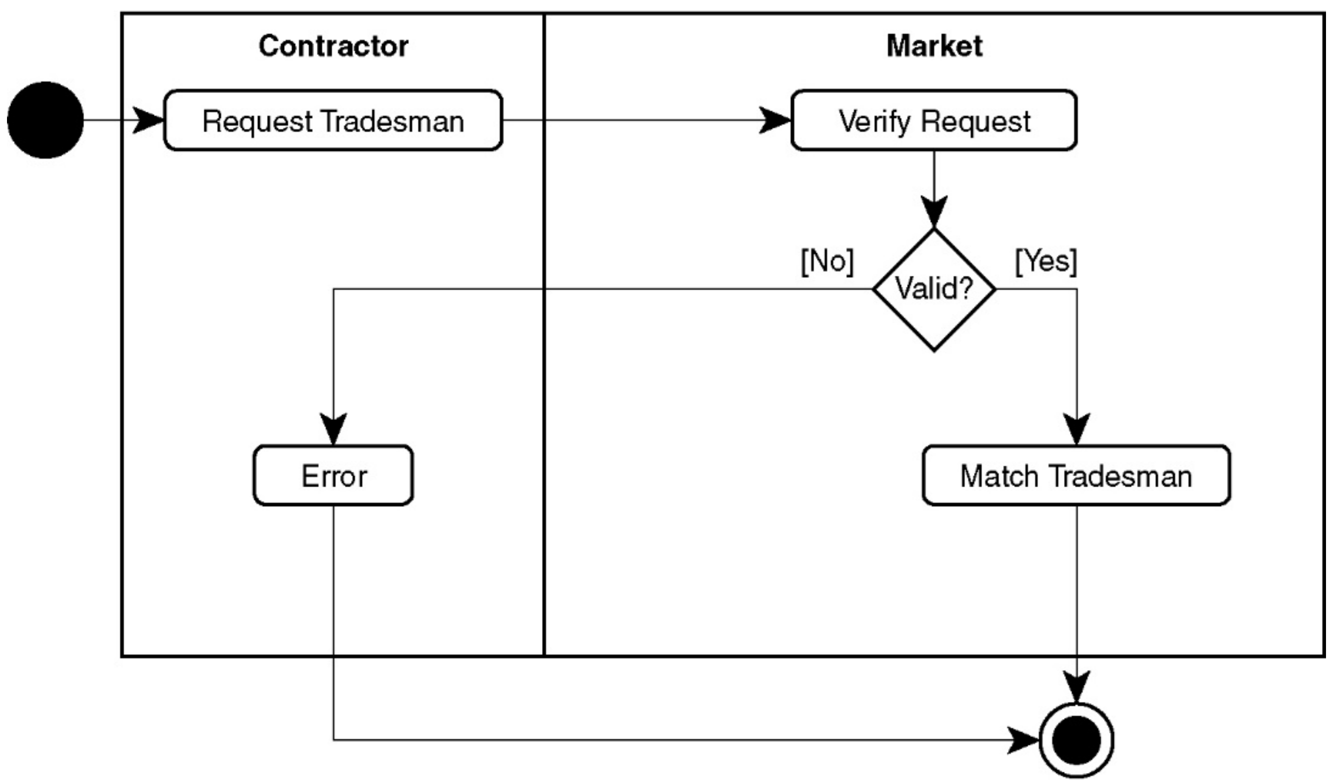


ADD TRADESMAN/CONTRACTOR USE CASE

- Upon receiving the message, the Membership Manager loads the appropriate workflow from the workflow storage.
- This either kicks off a new workflow or rehydrates an existing one to carry on with the workflow execution.
- Once the workflow has finished executing the request, the Membership Manager posts a message back into the Message Bus indicating the new state of the workflow, such as its completion,
- Clients can monitor the Message Bus as well and update the users about their requests.
- The Membership Manager
  - consults the Regulation Engine that is verifying the tradesman or contractor,
  - adds the tradesman or contractor to the Members store,
  - updates the Clients via the Message Bus.

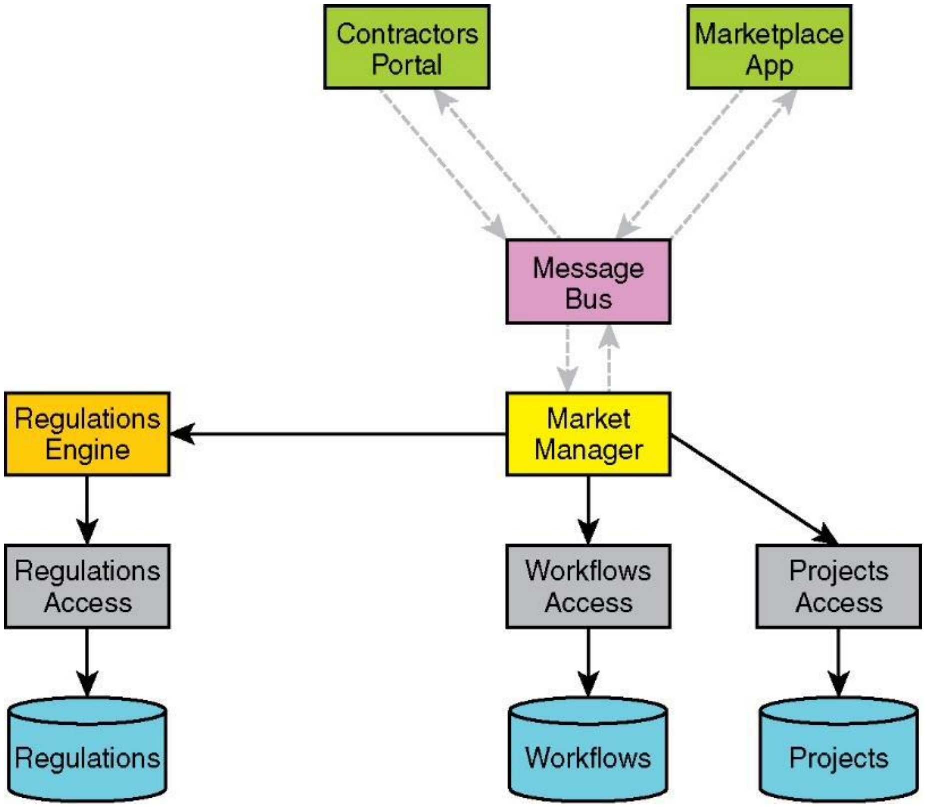
REQUEST TRADESMAN USE CASE

After initial verification of the request, this use case triggers another use case, Match Tradesman



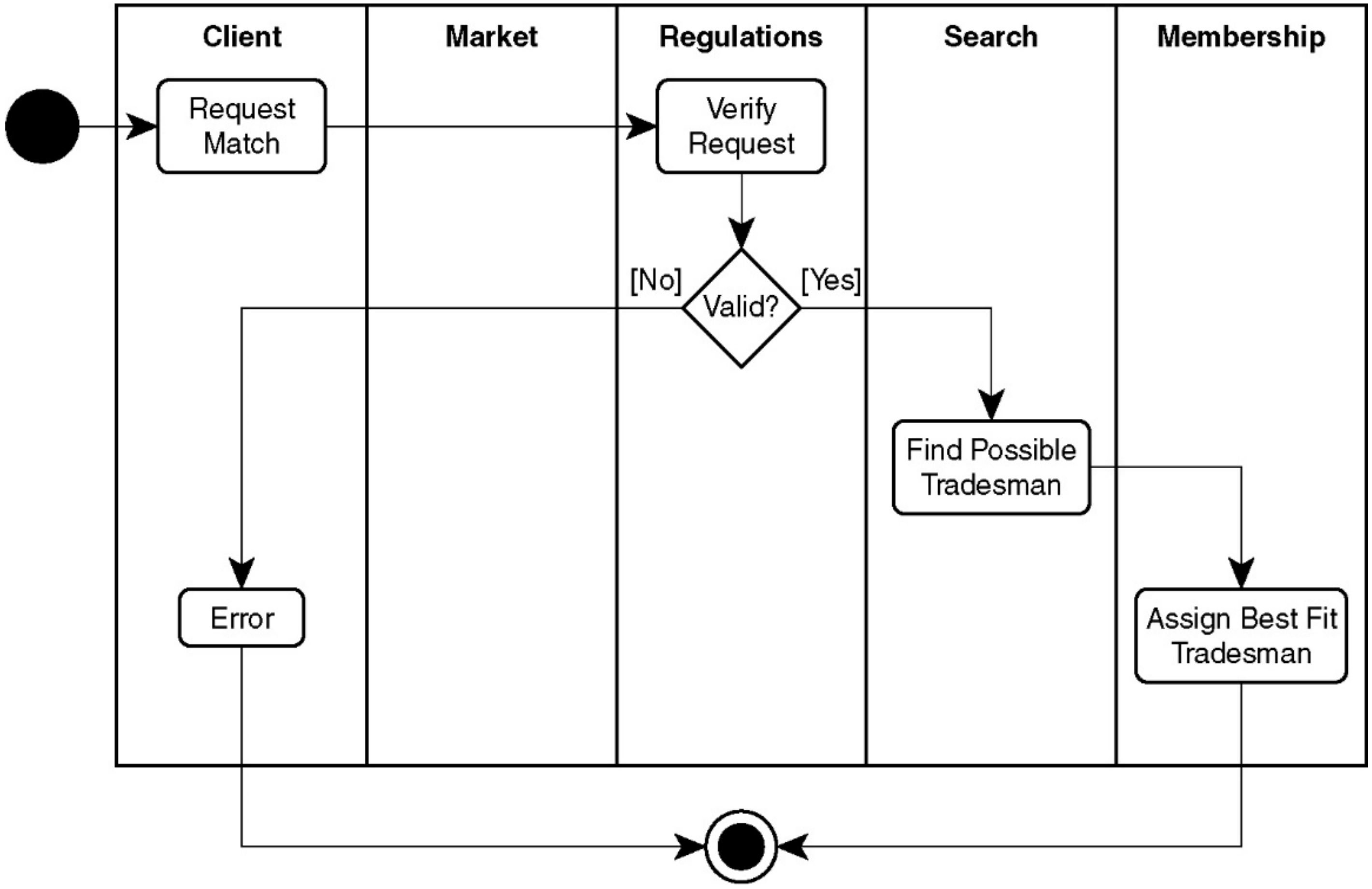
REQUEST TRADESMAN USE CASE

- Clients post a message to the bus requesting a tradesman.
- Market Manager receives that message and loads the workflow corresponding to this request
- Market Manager consults the Regulation Engine about what may be valid for this request or updates the project with the request for a tradesman.
- The Market Manager can then post back to the Message Bus that someone is requesting a tradesman =>trigger the matching and assignment workflows.



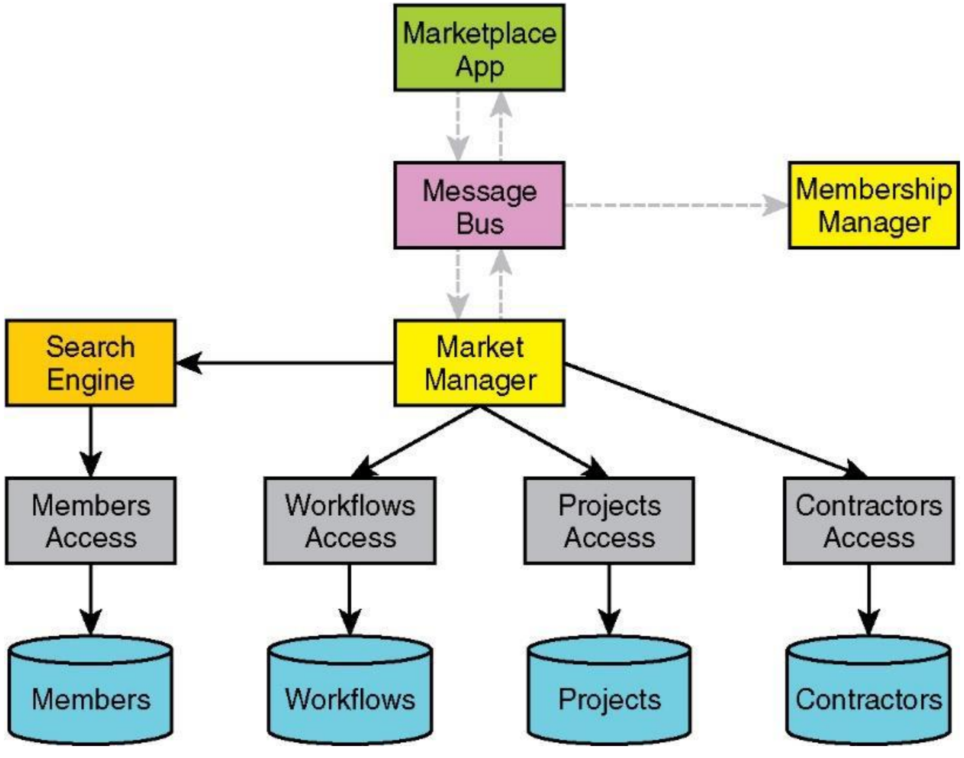
MATCH TRADESMAN USE CASE

- involves multiple areas of interest



MATCH TRADESMAN USE CASE

- Market Manager loads the workflow corresponding to this request
- Market Manager executes the workflow ..
- The Market Manager posts back to the Message Bus
- Message Bus sends message to Membership Manager



IDESIGN DISCUSSION

- A **directive** is a rule that you should never violate, since doing so is certain to cause the project to fail.
- A **guideline** is a piece of advice that you should follow unless you have a strong and unusual justification for going against it. Violating a guideline alone is not certain to cause the project to fail, but too many violations will tip the project into failure.

DIRECTIVES

THE PRIME DIRECTIVE

Never design against (i.e. based on) the requirements.

DIRECTIVES

Avoid functional decomposition.

Decompose based on volatility.

Provide a composable design.

Offer features as aspects of integration, not implementation.

Design iteratively, build incrementally.

GUIDELINES - 1

Requirements

- Capture required behavior, not required functionality.
- Describe required behavior with use cases.
- Document all use cases that contain nested conditions with activity diagrams.
- Eliminate solutions masquerading as requirements.
- Validate the system design by ensuring it supports all core use cases.

Cardinality

- Avoid more than five *Managers* in a system without subsystems.
- Avoid more than a handful of subsystems.
- Avoid more than three *Managers* per subsystem.
- Strive for a golden ratio of *Engines* to *Managers*.
- Allow *ResourceAccess* components to access more than one *Resource* if necessary.

GUIDELINES - 2

Attributes

- Volatility should decrease top-down.
- Reuse should increase top-down.
- Do not encapsulate changes to the nature of the business.
- *Managers* should be **almost expendable**.
- Architecture should be symmetric.
- Never use public communication channels for internal system interactions.

Layers

- Avoid open architecture.
- Avoid semi-closed/semi-open architecture.
- Prefer a closed architecture.
  - Do not call up.
  - Do not call sideways (except queued calls between *Managers*).
  - Do not call more than one layer down.
  - Resolve attempts at opening the architecture by using queued calls or asynchronous event publishing.
- Extend the system by implementing subsystems.

GUIDELINES - 3

Interaction rules

- All components can call *Utilities*.
- *Managers* and *Engines* can call *ResourceAccess*.
- *Managers* can call *Engines*.
- *Managers* can queue calls to another *Manager*.

Interaction don'ts

- *Clients* do not call multiple *Managers* in the same use case.
- *Managers* do not queue calls to more than one *Manager* in the same use case.
- *Engines* do not receive queued calls.
- *ResourceAccess* components do not receive queued calls.
- *Clients* do not publish events.
- *Engines* do not publish events.
- *ResourceAccess* components do not publish events.
- *Resources* do not publish events.
- *Engines*, *ResourceAccess*, and *Resources* do not subscribe to events.

WRAP-UP

- Modern, proven approach for designing service based architectures.
- Needs experience to correctly identify and design services