


DePaul University

SE480 Week 8 – Patterns, Life Cycle

Steven Engelhardt

Add WeChat powcoder

DePaul University

Autumn 2020

Table of Contents

Summary of Last Week

- Discussed many patterns
  - Layers
  - Tiers
  - Pipe and Filter
  - Publish-Subscribe
  - Saga
  - Command-Query Responsibility Separation (CQRS)
  - Event Sourcing
  - UI Patterns (MVC, MVP, MVVM))
  - Peer-to-Peer
  - Map-Reduce

Table of Contents

Table of Contents

Reflection Pattern

- Provides a mechanism for changing structure and behavior of software systems dynamically. Supports the modification of fundamental aspects, such as type structures and functional call mechanisms.
- Example: A C++ application that needs to store and write arbitrary objects (not utilizing serialization).

Solution

- Make the software self-aware and make selected aspects of its structure and behavior accessible for adaptation and change.
- Split the architecture into two parts – meta level and base level.
  - Metalevel provides information about selected system properties and makes the software self-aware.
  - Base level defines application logic. Its implementation builds on the meta level to remain independent of those aspects that are likely to change.
  - Metaobjects define the way in which base-level components behave.

Example: Programming vs. Metaprogramming

```
class User {
  public String FirstName;
  public String LastName;
  public Address HomeAddress;
}

class UserManager {
  public void init() {
    // Do nothing
  }

  public void createUser() {
    User user = new User();
    user.FirstName = "John";
    user.LastName = "Doe";
    ...
  }
}
```

Listing 1: Programming

```
class CustomTypeManager { ... }

class UserManager {
  private CustomType _type;

  public void init() {
    _type = CustomTypeManager.createType("User");
    _type.addField("FirstName", String.class);
    _type.addField("LastName", String.class);
    _type.addField("Address", Address.class);
    ...
  }

  public void createUser() {
    CustomObject obj = CustomObjectManager.createObject(_type);
    obj.setField("FirstName", "John");
    obj.setField("LastName", "Doe");
    ...
  }
}
```

Listing 2: Metaprogramming

Metaobject Protocol (MOP)

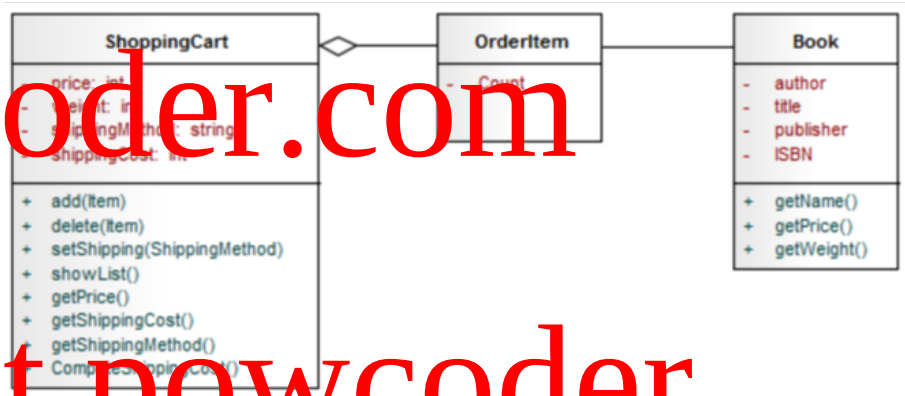
- The MOP is an interface for manipulating metaobjects.
- It allows clients to specify changes such as modifications to the function call mechanism, or to the way inheritance is implemented.
- MOP is responsible for:
  - Checking the correctness of the change specification
  - Performing the change

Meta Level

- Consists of a set of meta objects – each of which encapsulates selected information about a single aspect of the structure, behavior, or state of the base level.
- Three sources of such information:
  - Run-time environment of the system.
  - User defined (i.e. the user specifies the way in which the function call mechanism works).
  - Retrieved from the base level at runtime (i.e. current state of computation).

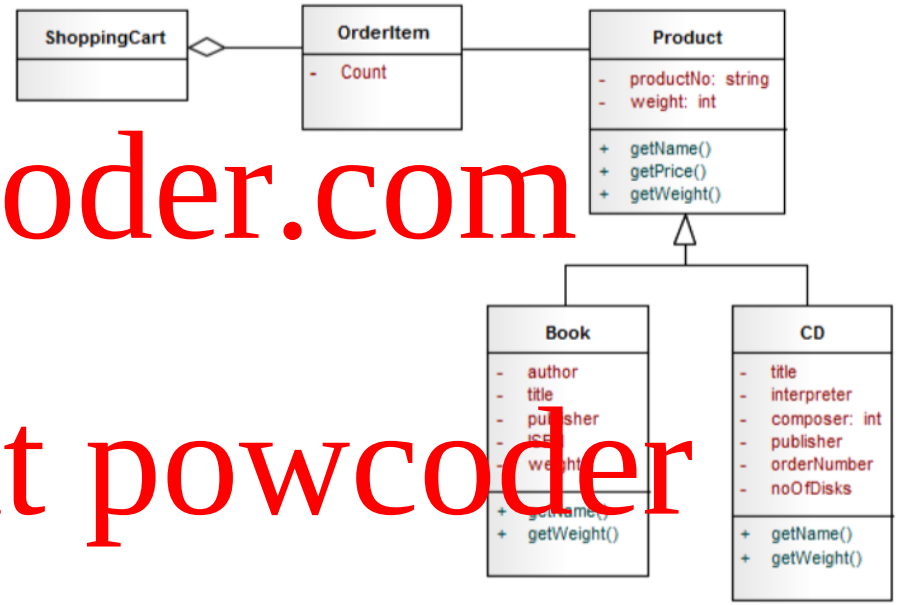
Example

- Susan, the manager of a local bookstore, wants to expand into the Internet. So she asks you to write a simple program for an Internet bookshop.



Example

- But changes come on the Web in Internet time: the bookshop is a success and Susan decides to sell CDs as well. So you have to change your program. With object orientation, you can do this quite easily and your modified class model will look like this...



Example

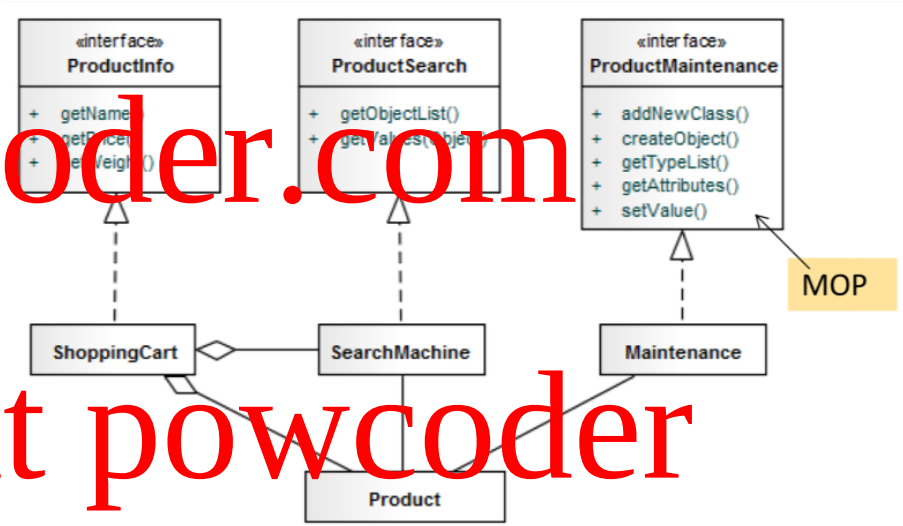
- Susan decides to start selling music accessories such as T-shirts, posters, etc. as well.
- She wants to make her system more adaptable so she can add different kinds of things with different attributes and functions at runtime.
- She realizes that she needs a new additional interface for a general search machine to answer:
  - What is the actual class of the object?
  - What attributes does the class have?
  - What are the values of these attributes?

Example

- This is a classic reflection interface that provides information about the properties of classes and objects.

Example

- We also need a third interface (MOP – Meta-Object Protocol) for product maintenance
- The MOP allows us to define new product classes, specify the attributes for them, create instances of these classes, and set the attribute values of these instances.



What is a MOP?

- Information about the attribute values of an object, ex: someBook.author, is information on the object level.
- In many OO languages information about classes themselves (properties, attributes) is captured in the class definition for the object, so the class is a meta-object and available at development time (but not runtime).

- You cannot manipulate classes like objects, you cannot add new classes at runtime.

- A MOP makes the class definitions normal objects and the object properties normal attribute values of the class definitions that can be manipulated at runtime.

A MOP must provide

- Definition of new classes
- Adding attributes to classes
- Querying attributes of classes
- Creating objects

- Querying the class of an object
- Setting attribute values of an object
- Querying attribute values of an object

- Deleting objects

Table of Contents

Patterns Summary

Need	Pattern(s)
Application Logical Design	Layers
Application Physical Design	Tiers
UI Programming	MVC, MVP, MVVM
Building a data transformation pipeline	Pipe and Filter
Divide-and-conquer data transfer	Peer-to-peer
Divide-and-conquer calculation	Map-reduce (or replacement)
Application data model extensibility	Reflection
Decoupling reads and writes	CQRS
Modern application data flow	Event Sourcing
Decoupling producers from consumers	Publish-subscribe
Scalable, distributed transactions	Saga

Table of Contents

Table of Contents

- How does software architecture fit into the software development life cycle?
  - How do architecture concerns affect the gathering of requirements, the carrying out of design decisions, the validation & capturing of the design, and the transformation of design into implementation?

Engage Stakeholders

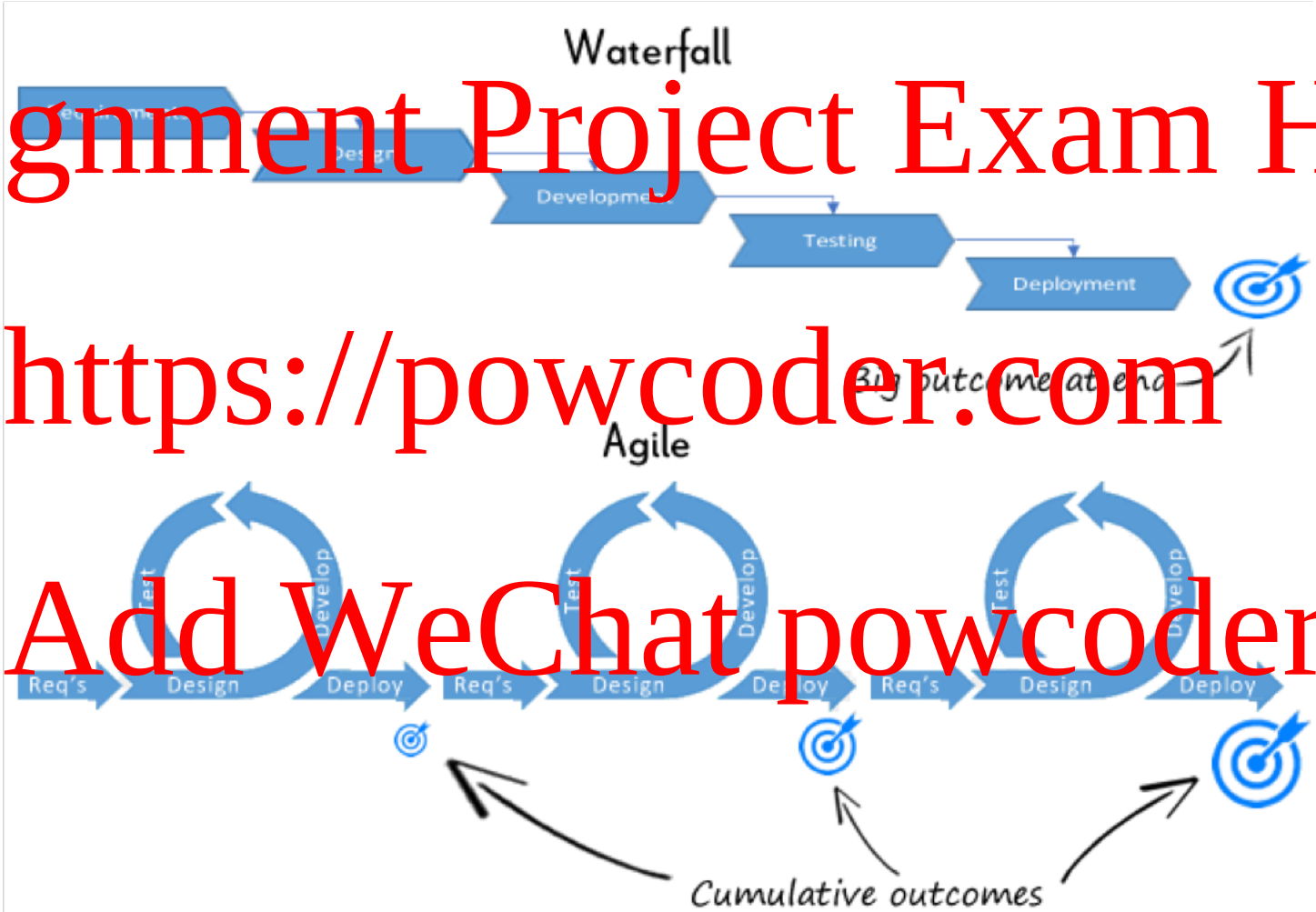
- In order to understand what an architecture under development should provide, you must *engage stakeholders*
- Engaging stakeholders is about *getting key knowledge* from them – everyone has a different, valuable perspective and for a product to be successful all this feedback must be considered
- This activity is fundamentally *social* and thus may not come naturally to everyone

Common Ways to Engage Stakeholders

- All stakeholders in the same room for the duration of the exercise (costliest, quickest, demonstrates its importance, hardest to schedule)
- Some stakeholders participate in exercise remotely
- Facilitators interviewing stakeholders individually or in small groups

Table of Contents

What is Agile?



Principles of Agile

1. Customer satisfaction by early and continuous delivery of valuable software
2. Welcome changing requirements, even in late development
3. Working software is delivered frequently (weeks rather than months)
4. Close, daily cooperation between business people and developers
5. Projects are built around motivated individuals, who should be trusted
6. Face-to-face conversation is the best form of communication (co-location)
7. Working software is the primary measure of progress
8. Sustainable development, able to maintain a constant pace
9. Continuous attention to technical excellence and good design
10. Simplicity—the art of maximizing the amount of work not done—is essential
11. Best architectures, requirements, and designs emerge from self-organizing teams
12. Regularly, the team reflects on how to become more effective, and adjusts accordingly

Common Agile Challenges

- *Long-term planning* – how to effectively project features, capabilities, etc. over long-term planning?
- *Agility vs. commitment* – how to balance between providing new capabilities quickly, vs. ensuring the long-term survival of the system & the company

Agile Architecture Principles

- Architecture in Agile is not *whether* but rather *how much*?
- Architecture is fundamentally about *reducing risk*, so the amount of up-front planning & analysis should be justified by the potential risks
- *Write for the reader* – if there is no audience, there is no need to produce the documentation
  - This doesn't mean "don't architect", but perhaps instead of documenting, you can organize small group whiteboarding sessions or the like
- Expect to *evolve* the architecture over time
- Expect substantial *ambiguity* and *uncertain strategic direction*
- Don't forget to *socialize* the architecture!

Walking Skeleton [Alistair Cockburn]

- *A walking skeleton* is a tiny implementation of the system that performs a small end-to-end function. It need not use the final architecture, but it should link together the main architectural components. The architecture and the functionality can then evolve in parallel.

Architectural Stories & Spikes

- A *user story* is used to capture a description of a software feature from an end-user perspective. The user story describes the type of user, what they want and why. A user story helps to create a simplified description of a requirement.

- A *spike* is a story that cannot be estimated until a development team runs stoa timeboxed investigation. The output of a spike is an estimate for the original story.
- Use architectural experiments (spikes) to *evaluate & resolve tradeoffs*

- *Technical debt* reflects the implied cost of additional rework caused by choosing an easy solution now instead of using a better approach that would take longer. Like any debt, it simply needs to be *managed well*.

Agile Architecture Advice [SAIP]

- If you are building a large and complex system with relatively stable and well-understood requirements, do a large amount of architecture up front.
- On big projects with vague or unstable requirements, start by quickly designing a complete candidate architecture even if it leaves out many details.
- On smaller projects with uncertain requirements, at least try to get agreement on the central patterns to be employed, but don't spend too much time on construction, documentation, or analysis up front.

Agile Architecture Thoughts [Steve]

- Agile is about being able to *quickly adapt to changing requirements*. If your architecture cannot evolve to support the changing requirements at the required rate, you are doomed.
- Consider the *cost to your reputation* if you convince your sponsors that you want invest substantial time into an initial architecture, and then your requirements change and your initial architecture is no longer valid
- If you're building a large, complex, well-defined system, *is Agile the right framework for you?*
- To what extent can you *break up a big project* into many, smaller, minimally-interdependent projects, and apply Agile to the individual pieces?
  - Akin to applying microservices to product management

Misc. Architecture Thoughts [Steve]

- *Working code* (e.g. POCs) is worth far, far more than a document, and the code can become a point of reference
- Be careful with your *messaging & delivery*.
  - Software is about delivering customer value. If your architecture work is not communicated in these terms (e.g. "this work will allow us to deliver future features faster", "this work will make the system more reliable / performant for our users") you risk losing the support of your business sponsors.
  - Pay careful attention to *body language*, and adjust your messaging accordingly.
  - Avoid laundry lists of all the work you need to do to build a feature. We all have a lot to do, that's why we have jobs.
- *Manage your reputation*
  - Once you've lost it, it's *incredibly hard to get back*.
  - Set yourself up to be an ally to, not a enemy of, "the business"
  - This does not mean being a sycophant, but you need to communicate using the *language & values of your counterpart*

Table of Contents

Architecturally Significant Requirements

- An *architecturally significant requirement (ASR)* is a requirement that will have a profound effect on the architecture
- ASRs often, but not always, take the form of *quality attribute requirements* – performance, security modifiability, availability, etc.
- Architects uncover candidate ASRs by *talking to important stakeholders*

Gathering ASRs from Requirements Document

- The first place to look for candidate ASRs is in the *requirements documents* or in *user stories*
- *Don't wait* for requirements to be finished before starting work!
- ASRs often derive from *business goals in the development organization* itself rather than from requirements

Gathering ASRs by Interviewing Stakeholders

- Don't expect to be given quality attribute requirements – *help set them*!
- Perform *stakeholder interviews* and create a list of architectural drivers and a set of QA scenarios that the stakeholders (as a group) prioritized

Quality Attribute Workshop

- The *quality attribute workshop (QAW)* is a facilitated, stakeholder-focused method to generate, prioritize, and refine quality attribute scenarios before the software architecture is completed.
- At the end of the QAW, you should have:
  - Broad consensus as to what is important to the system
  - Identification of conflicting assumptions about system requirements
  - A prioritized list of important scenarios which your architecture must support
  - A list of business/programmatic goals that are affected by these scenarios
  - A list of quality attributes & quality attribute scenarios that are affected by these scenarios, with quantitative measures of success, that can be turned into ASRs

Gathering ASRs by Understanding the Business Goals

- Business goals are the *raison d'être* for building a system and are worth *capturing explicitly*
- Given business goals, we can *assess their effect on the architecture*. A goal can
  - Lead to a quality attribute requirement
  - Directly affect the architecture without precipitating a quality attribute requirement at all
  - Have no influence at all
- Capturing business goals is well served by having a set of *candidate business goals* handy to use as conversation starters

Standard Business Goal Categories

- Contributing to the growth and continuity of the organization
- Meeting financial objectives
- Meeting personal objectives
- Meeting responsibility to employees
- Meeting responsibility to society
- Meeting responsibility to state
- Meeting responsibility to shareholders
- Managing market position
- Improving business processes
- Managing the quality and reputation of products
- Managing change in environmental factors

Expressing Business Goals

- For the system being developed, *goal subject* desires that *goal-object* achieve *goal* in the context of *environment* and will be satisfied if *goal-measure*
  - Rather similar to the commonly-used Agile user story template "As a *type of user*, I want *some goal* so that *some reason*."

- Example: "For MySys, the project manager has the goal that his family's stock in the company will rise by 5 percent (as a result of the success of MySys)."
  - Not all business goals need to be "noble". I have a goal that my company will succeed so that I can retire early.

Capturing ASRs in a Utility Tree

Quality Attribute	Attribute Refinement	ASR	Business Value	Architectural Impact
Performance	Transaction response time	A user updates a patient's account in response to a change-of-address notification while the system is under peak load, and the transaction completes in less than 0.75 seconds.	H	M
		A user updates a patient's account in response to a change-of-address notification while the system is under double the peak load, and the transaction completes in less than 4 seconds.	L	M
	Throughput	At peak load, the system is able to complete 150 normalized transactions per second.	M	M
Extensibility	Adding new product	A product that tracks blood bank donors is created within 2 person-months.	M	M

Table of Contents

Design Strategy

- To begin design, you need ASRs
- Three ideas that are key to architecture design methods:
  - *Decomposition*
  - *Designing to architecturally significant requirements*
  - *Generate and test*
- When decomposing, you can simultaneously *decompose quality attribute requirements*
- When designing to ASRs, you want to *use patterns* to *design for multiple at once*
- *Generate and test* refers to whether the design meets the ASRs, and require *collateral*. Types of collateral include:
  - Existing systems
  - Frameworks
  - Patterns & tactics
  - Domain decomposition
  - Design checklists

Attribute-Driven Design (ADD)

- *Attribute-Driven Design (ADD)* is a method which applies the previous strategy.
- ADD is *iterative* with the following five steps:
  1. Choose an element of the system to design
  2. Identify the ASRs for the chosen element
  3. Generate a design solution for the chosen element
  4. Inventory remaining requirements and select the input for the next iteration
  5. Repeat steps 1-4 until all the ASRs have been satisfied
- ADD may be refined *breadth-first* or *depth-first*, but breadth-first allows you to apportion the most work to the most teams soonest

ADD Example

- *Track Manager* provides a tracking service for two types of clients:
  - *Update clients*: These clients send track updates to the Track Manager periodically. The Track Manager can tolerate some occasional loss of updates, especially during transient conditions caused by equipment failure.
  - *Query clients*: These clients operate sporadically and must receive exactly one reply to their query.

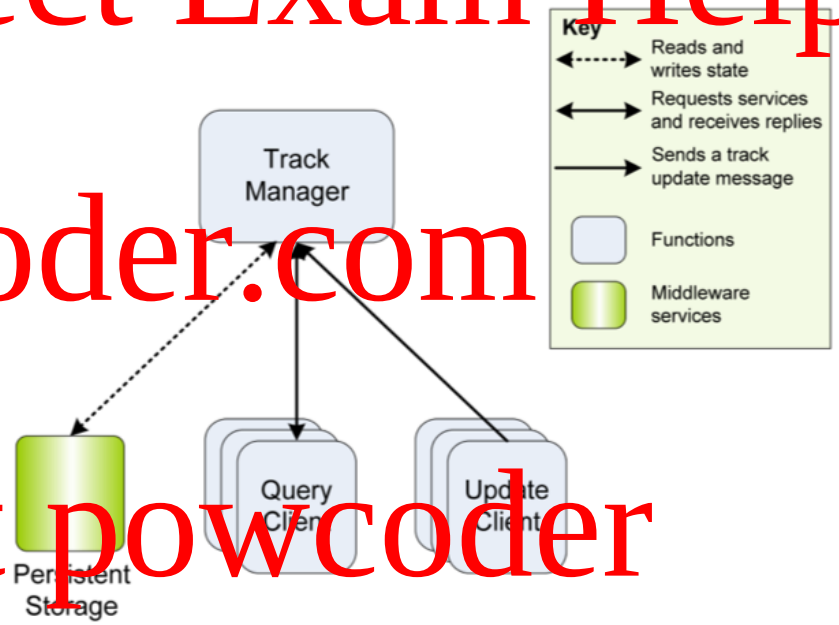


Figure: Track Manager Initial Design

Track Manager Design Constraints

- Three design constraints are required:
  1. *Capacity restrictions*: The provided processors shall have 50% spare processor and memory capacity on delivery, and the local area network (LAN) has 50% spare throughput capability. There are 100 update clients and 25 query clients. For the purposes of timing estimates, assume that there are 100 updates and 5 queries per second.
  2. *Persistent storage service*: This service will maintain a copy of state that is checked at least once per minute by the Track Manager. If all replicas of the Track Manager fail, a restart can begin from the checkpoint file.
  3. *Two replicas*: To satisfy the availability and reliability requirements, a Reliability, Availability, and Maintainability (RMA) study has been conducted, and the Track Manager and persistent storage elements shall all have two replicas operating during normal circumstances.

Track Manager Quality Attribute Scenario 1 – Quick Recovery

Element	Statement
Stimulus	A Track Manager software or hardware component fails.
Stimulus source	A fault occurs in a Track Manager software or hardware component.
Environment	Many software clients are using this service. At the time of failure, the component may be servicing a number of clients concurrently with other queued requests.
Artifact	Track Manager
Response	All query requests made by clients before and during the failure must be honored. Update service requests can be ignored for up to two seconds without noticeable loss of accuracy.
Response measure	The secondary replica must be promoted to primary and start processing update requests within two seconds of the occurrence of a fault. Any query responses that are underway (or made near the failure time) must be responded to within three seconds of additional time (on average).

Track Manager Quality Attribute Scenario 2 – Slow Recovery

Element	Statement
Stimulus	Track Manager hardware or software component fails when no backup service is available.
Stimulus source	An error occurs in a Track Manager software or hardware component.
Environment	A single copy of the Track Manager is providing services and it fails. A spare processor is available that does not contain a copy of this component. A copy of the component is available on persistent storage and can be transferred to the spare processor via the LAN.
Artifact	Track Manager
Response	The clients are informed that the service has become unavailable. A new copy of the service is started and becomes operational. The state of the component on restart may differ from that of the failed component but by no more than one minute. The clients are informed that the service is available to receive update signals. For some tracks, the new updates can be automatically correlated to the old tracks. For others, an administrator assists in this correlation. New tracks are started when necessary. The clients are then informed that the service is available for new queries.
Response measure	The new copy is available within three minutes.

Track Manager Quality Attribute Scenario 3 – Restart

Element	Statement
Stimulus	A new replica is started as the standby.
Stimulus source	The system resource manager starts the standby.
Environment	A single replica is servicing requests for service under normal conditions. No other replica is present.
Artifact	New replica of the Track Manager
Response	The initialization of the new replica has a transient impact on service requests that lasts for less than two seconds.
Response measure	The initialization of the new replica has a transient impact on service requests that lasts for less than two seconds.

Track Manager ADD Iteration 1

- The Track Manager has been broken into two elements: A and B. This decomposition allows two deployment strategies to be considered:
  - Strategy 1: Both elements (A and B) operate in a single processor, P1. A and B together consume 50
  - Strategy 2: Element A is in processor P1, and element B is in processor P2. Together, they can handle 150 update clients and 50 query clients. This strategy exceeds the system performance requirements.

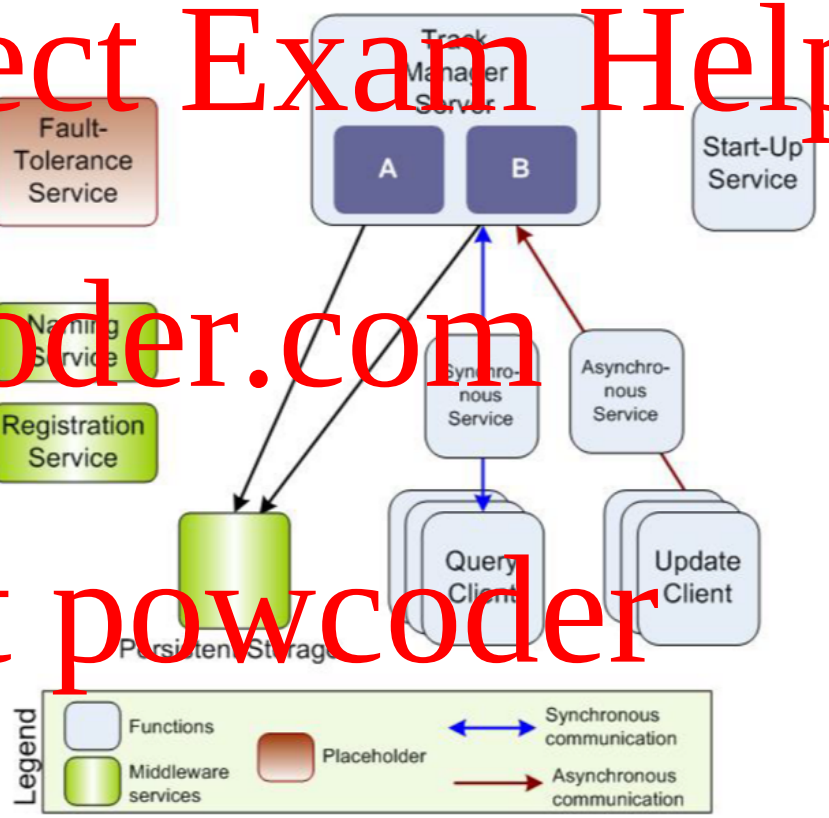


Figure: Track Manager ADD Iteration 1

Track Manager ADD Iteration 2

- After careful consideration of the quality scenarios, the fault tolerance expert has decided to use:
  - Two replicas
  - Warm standby
  - Checkpoints with bundled log changes
  - Heartbeat-based fault detection
  - A proxy to handle failures

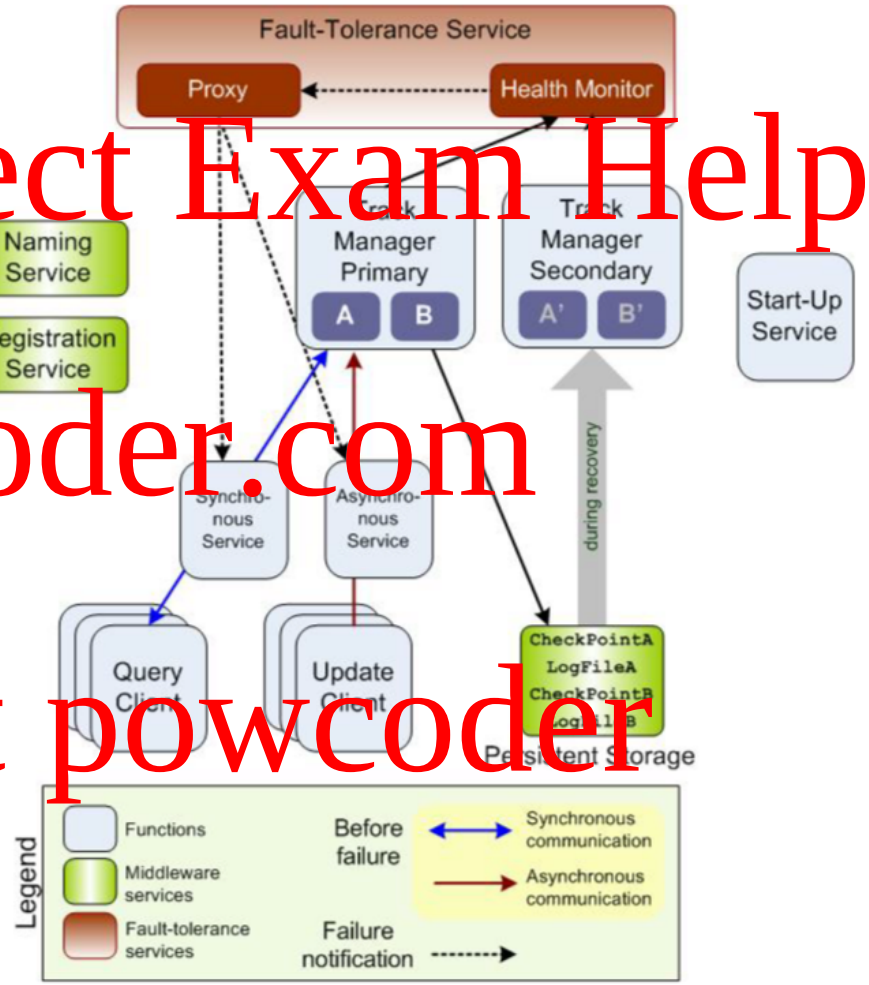


Figure: Track Manager ADD Iteration 2

Table of Contents

- Creating an architecture *isn't enough*. It has to be *communicated* in a way to let its stakeholders use it properly to do their jobs.
- Documentation should not be produced simply because it is required but because it is *essential to the matter at hand*.
- *Tailor your documentation* to your stakeholders' needs. If your organization is small enough, whiteboard discussions are probably sufficient.

Uses and Audiences for Architecture Documentation

- Documentation is both *prescriptive* (describing what should be true) and *descriptive* (describing what is true)
- Documentation has three uses:
  1. A *means of education*
  2. A *primary vehicle for communication among stakeholders*
  3. A *basis for system analysis and construction*

Notations

- Three main categories of notation:
  1. *Informal notations* (e.g. PowerPoint)
  2. *Semiformal notations* (e.g. UML)
  3. *Formal notations* (e.g. *architecture description languages*)
- Formal languages include precise (usually mathematically based) semantics which allows *formal analysis* of both syntax and semantics

Views

- A *view* is a representation of a set of system elements and relations among the
- Documenting an architecture is a matter of *documenting the relevant views* and then adding documentation that applies to more than one view
- Each view has a *cost and a benefit*, and you should ensure that the benefits of maintaining a particular view outweigh its costs

Module Views

- A *module* is an implementation unit that provides a coherent set of responsibilities.
- A module might take the form of a class, a collection of classes, a layer, an aspect, or any decomposition of the implementation unit.
- A module partitions the system and it should be clear how the functional requirements of a system are supported by module responsibilities.
- A module view is not typically used to make inferences about runtime behavior, so cannot be used for analysis of performance, reliability, etc.

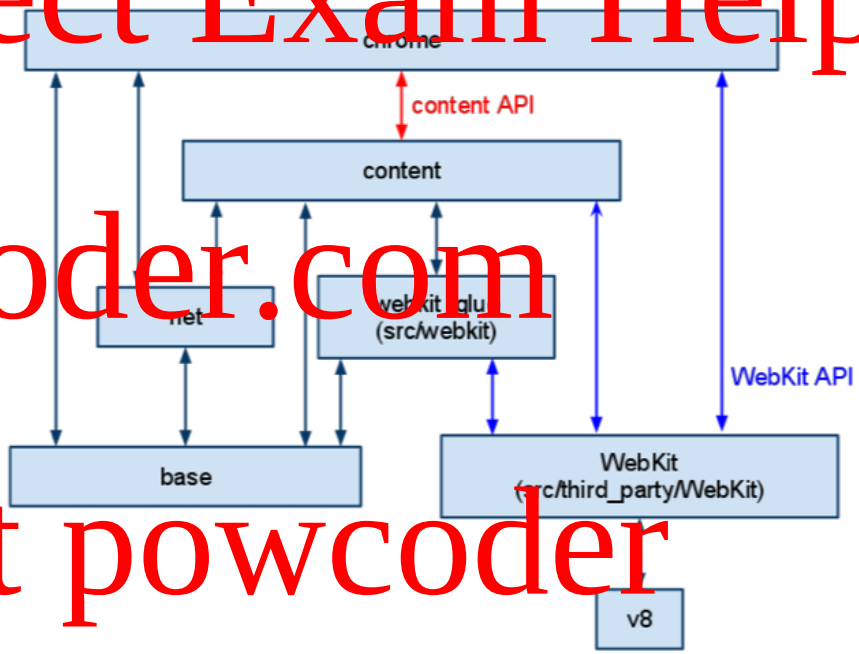


Figure: Chromium Module View

Component-and-Connector Views

- A *component-and-connector (C&C) view* shows elements that have runtime presence such as processes, objects, clients, servers, and data stores
- Components have interfaces called *ports*, which are points of potential interaction of a component with its environment
- The primary relation within a C&C view is an *attachment*, which indicates which connectors are attached to which components
- C&C views show how the system *works*, which can be used to reason about runtime system quality attributes

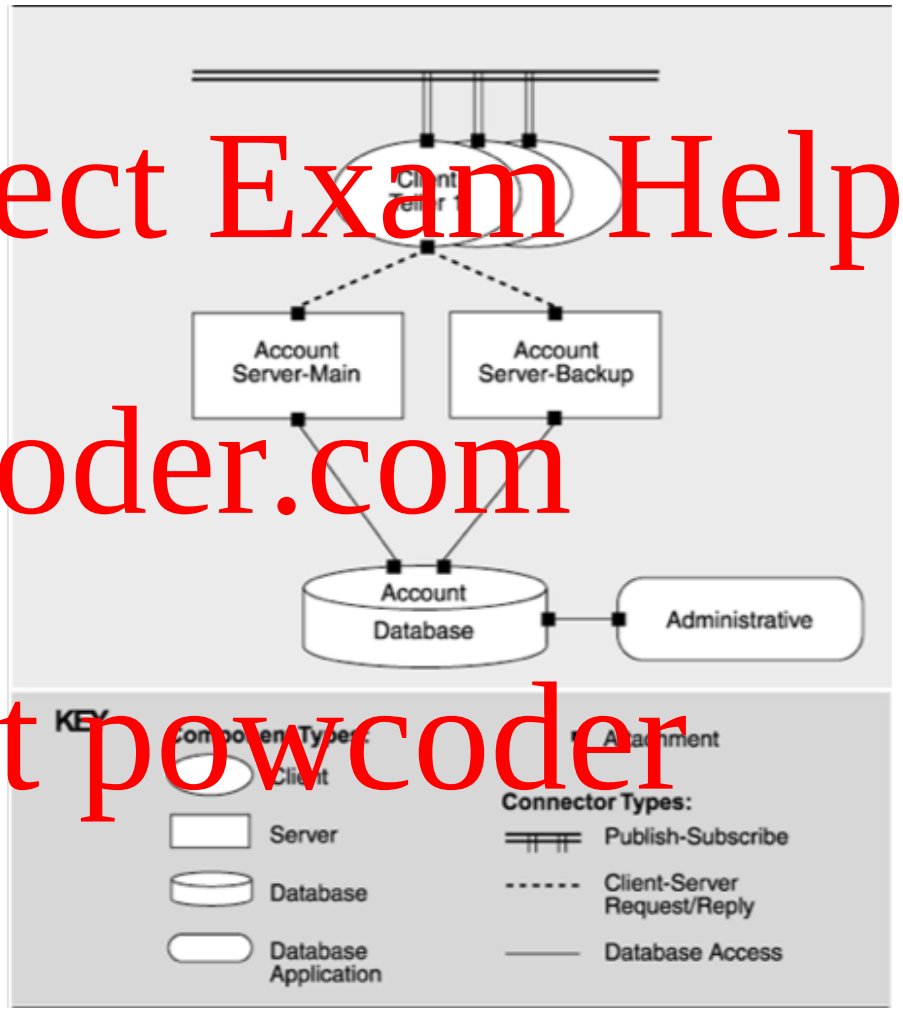


Figure: Teller-Account C&C View

Allocation Views

- *Allocation views* describe the mapping of software units to elements of an environment in which the software is developed or in which it executes
- The usual goal of an allocation view is to compare the properties *required* by the software element with the properties *provided* by the environmental elements to determine whether the allocation will be successful or not

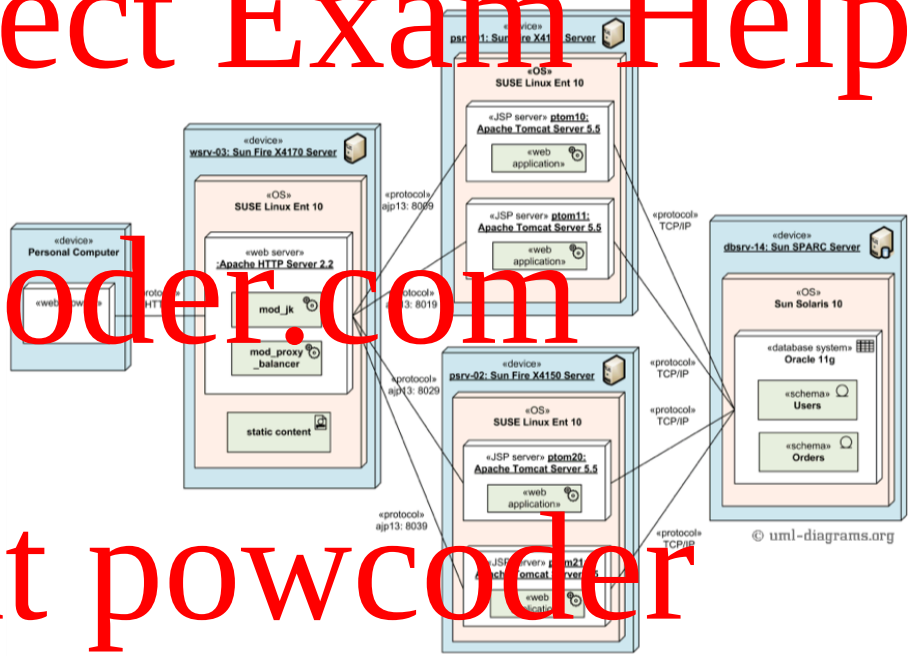


Figure: Allocation View

Choosing Views

- Three-step method for choosing the views:
  1. Build a stakeholder/view table
  2. Combine views
  3. Prioritize & stage
- At a minimum, expect to have at least one module view, at least one C&C view, and for larger systems, at least one allocation view in your architecture document.

Documenting Behavior

- Two kinds of notations available for documenting behavior
  1. *Traces* are sequences of activities or interactions that describe the system's response to a specific stimulus when the system is in a specific state. Includes use cases, sequence diagrams, communication diagrams, and activity diagrams.
  2. *Comprehensive models* show the complete behavior of structural elements. Includes state machine diagrams.

- What to do if your system changes faster than you can document them?
  - Document what it is true about all versions of your system
  - Document the ways the architecture is allowed to change

Documenting in Agile Projects

- Adopt a template or standard organization to capture your design decisions.
- Plan to document a view only if it has a strongly identified stakeholder consistency.
  - If there is no audience, there is no need to produce the documentation.
- Do not hold up all other progress until documentation is complete.
- Maybe just take a picture of the whiteboard!

Table of Contents

What is UML?

- *UML*, short for *Unified Modeling Language*, is a standardized modeling language consisting of an integrated set of diagrams, developed to help system and software developers for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems.

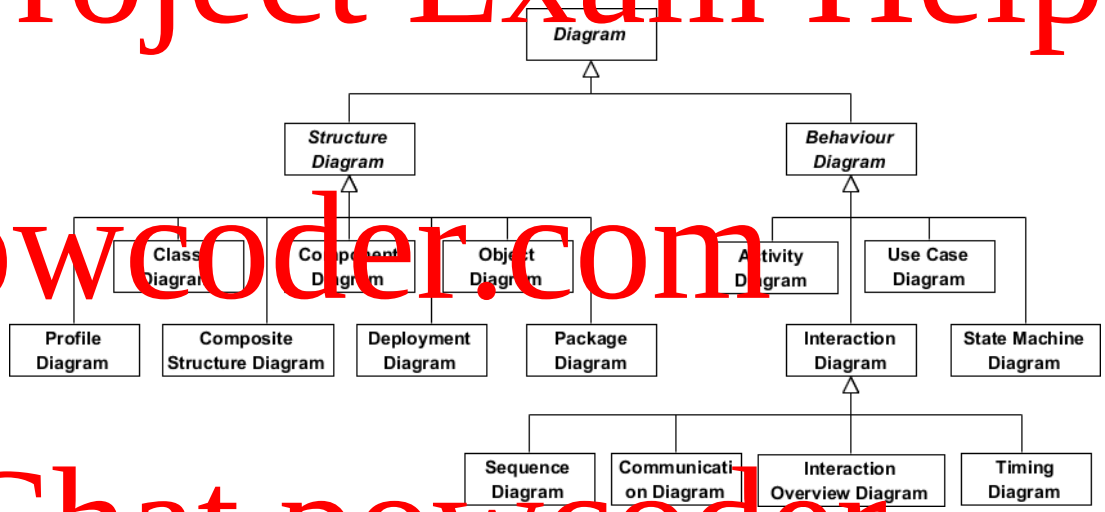
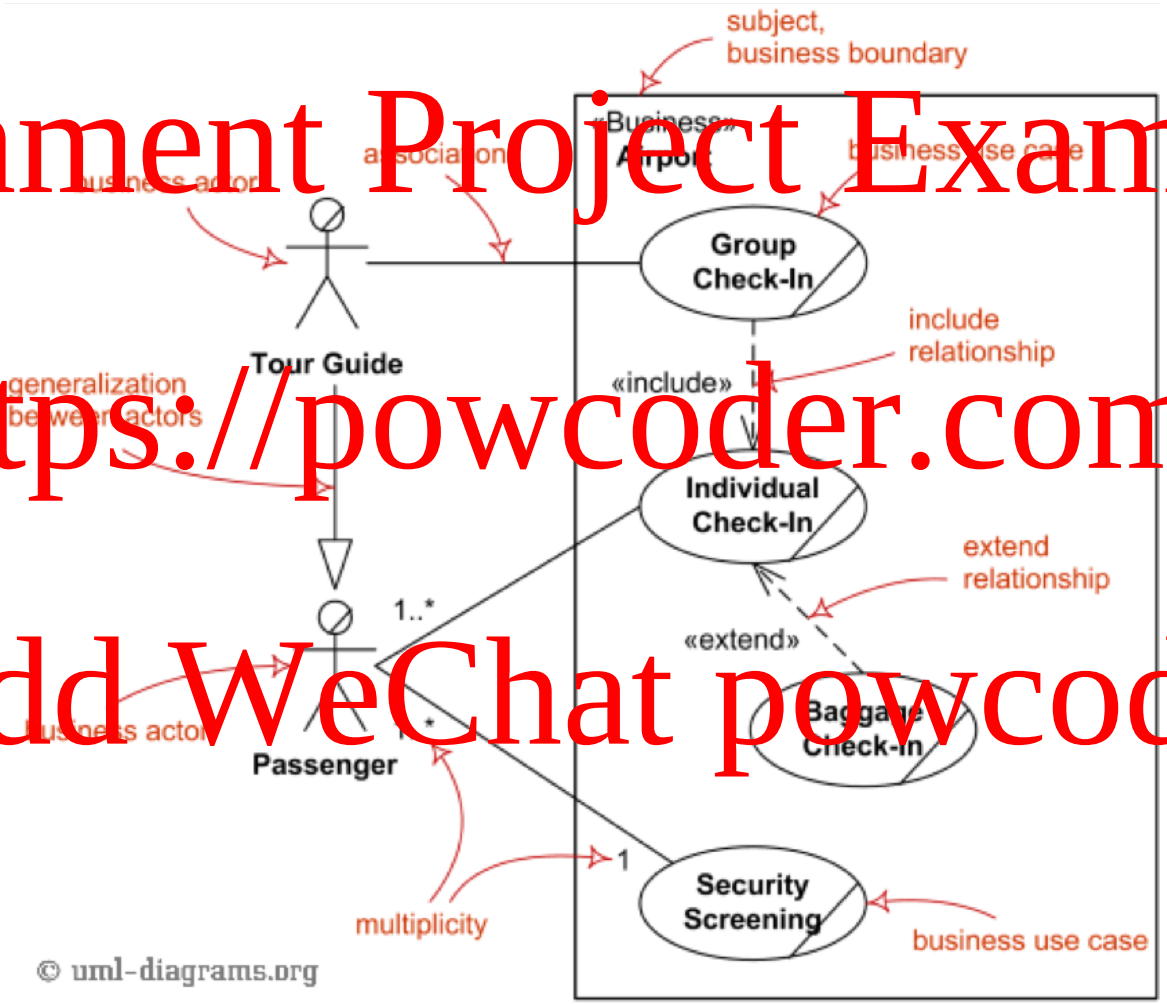
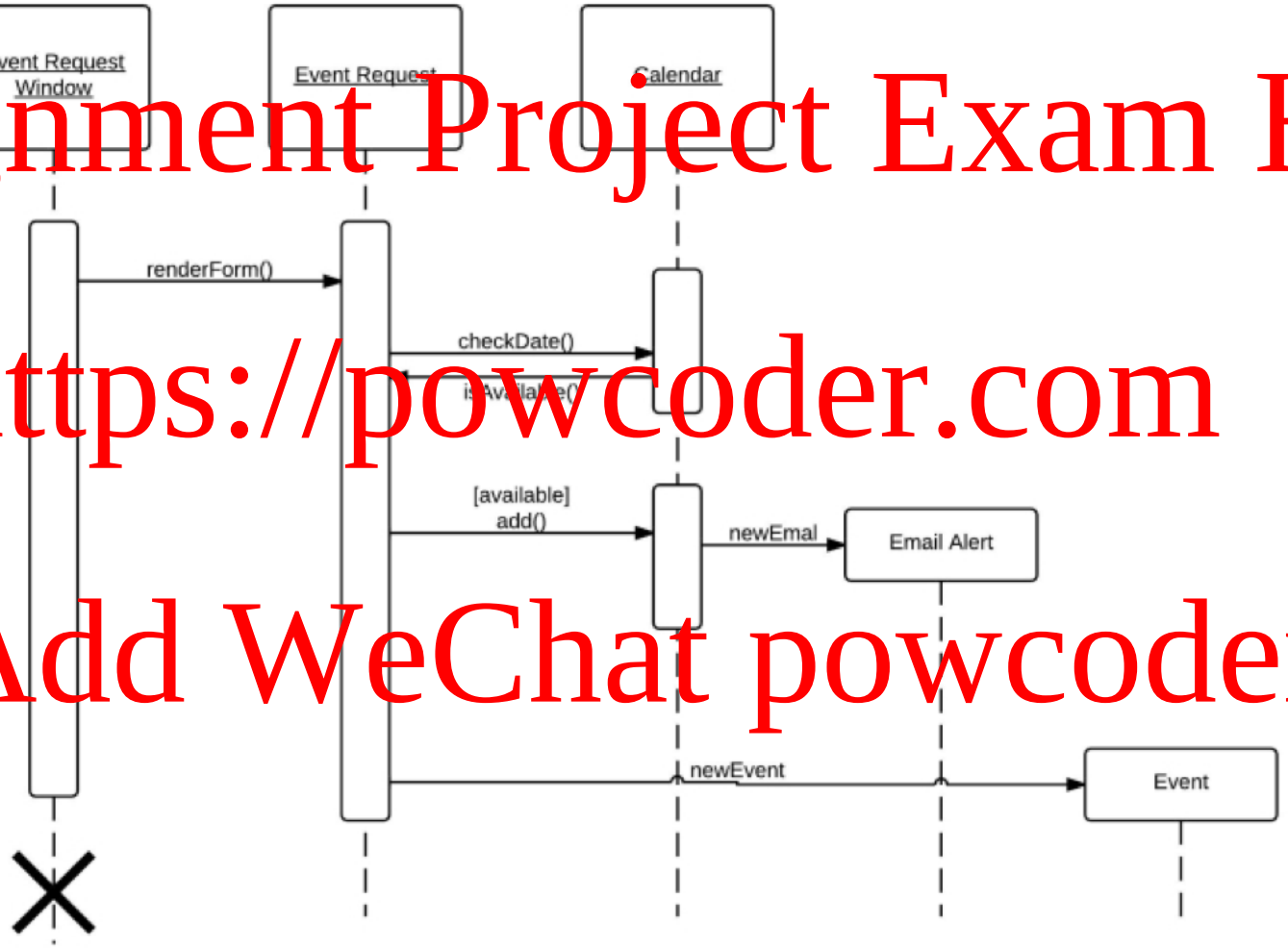


Figure: UML Diagram Types

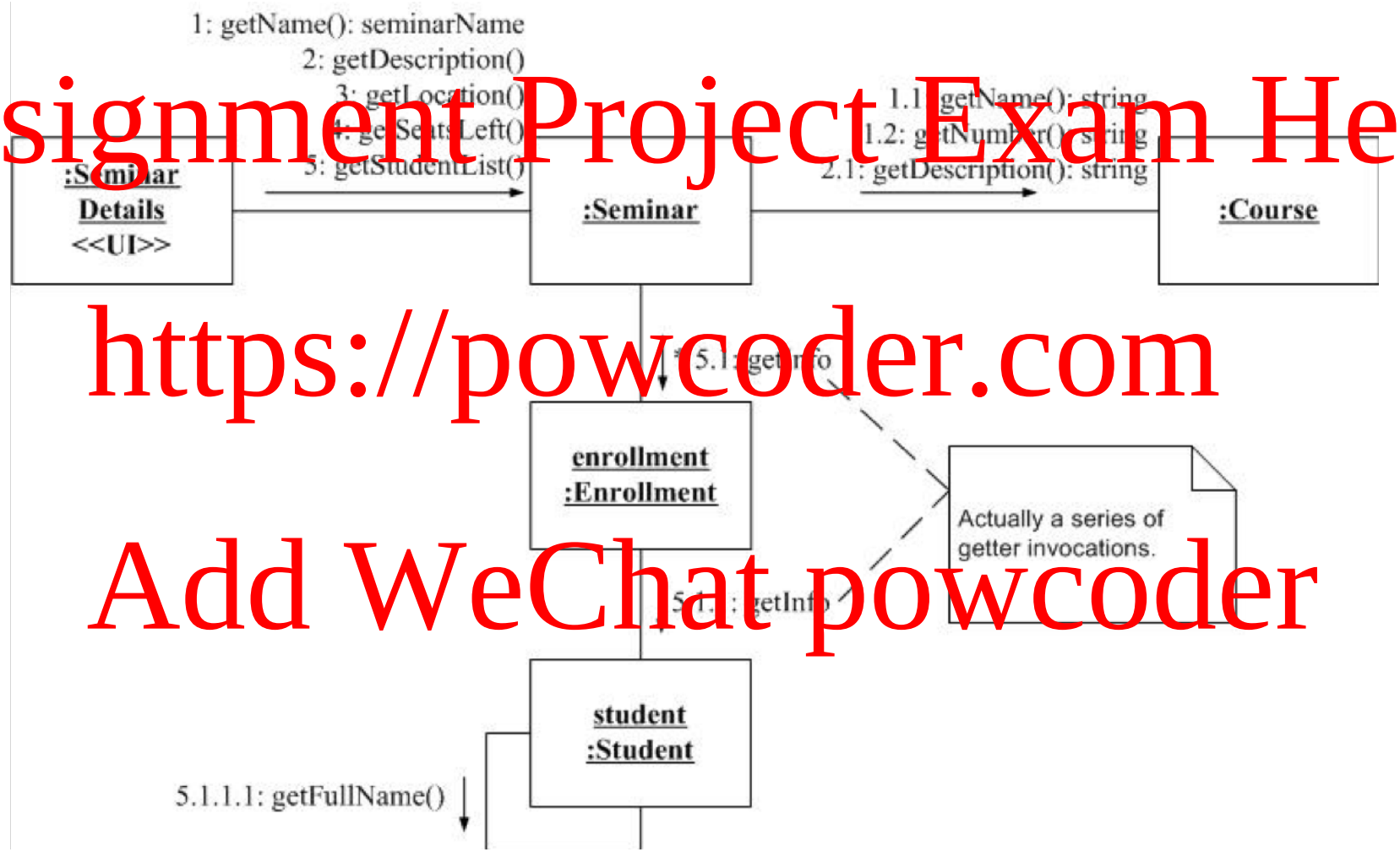
UML Use Case Diagram



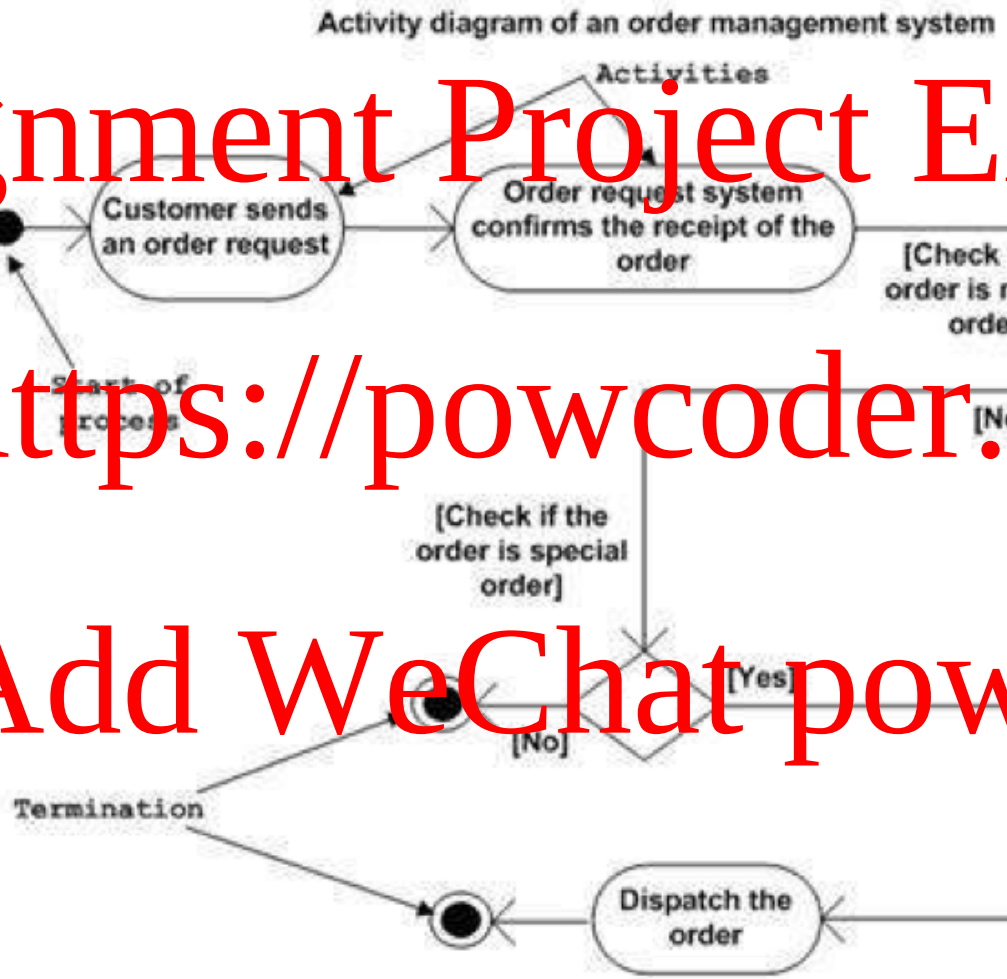
UML Sequence Diagram



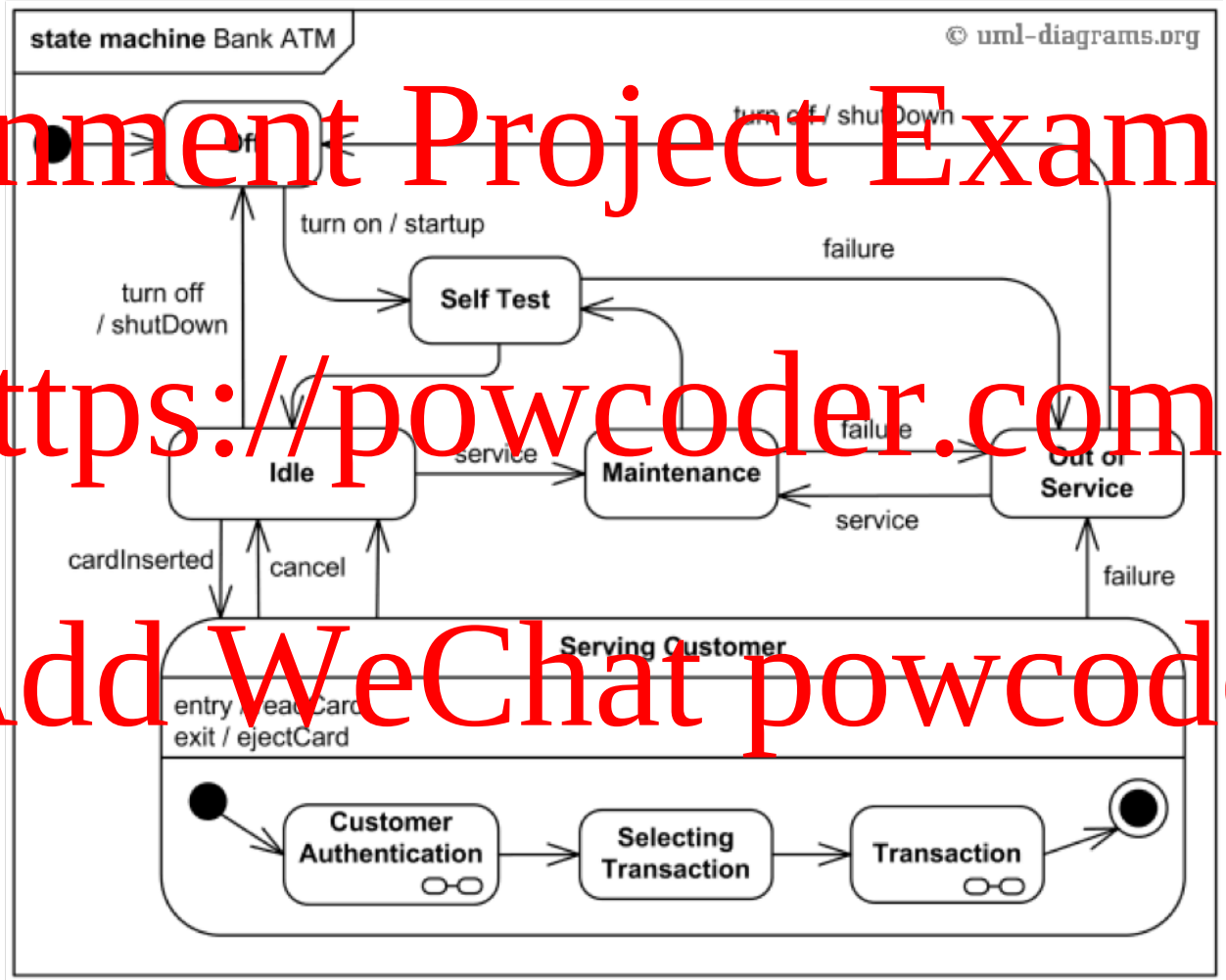
UML Communication Diagram



UML Activity Diagram



UML State Machine Diagram



Other UML Notes

- http://plantuml.com is a useful tool for creating UML diagrams from text

Table of Contents

Architectural Decision Records (ADRs)

- An *architectural decision (AD)* is a software design choice that addresses a significant requirement.
- An *architectural decision record (ADR)* is a way to track an AD, such as by writing notes, or logging information.
  - A *lightweight technique* for capturing important architectural decisions
- Easy to maintain inside your source code

ADR Example Template

- Title
- Context
- Decision
- Status
- Consequences

ADR Example (GovUK: Hosting Platforms)

- Date: 2017-06-30
- Status: Accepted
- Context: We need to decide upon a platform to host the future GOV.UK infrastructure. Long term, this will be primarily the GOV.UK PaaS but in the interim, we need to converge with that plan and also upgrade and modernise the current infrastructure. GDS policy for hosting of GDS internal services is PaaS first and AWS for anything that can not be run on the PaaS.
- Decision: We are using Amazon Web Services as our hosting provider of choice. This conforms to the GDS Tech Forum Hosting Guide. We will initially be using the eu-west-1 region, Ireland. This region has 3 availability zones and also contains the GDS PaaS which will allow easier sharing and peering.
- Consequences: We will have to migrate our infrastructure from our current hosting provider to Amazon in the near-term "as-is" and then iterate on that infrastructure in the medium term to deliver improvements to the platform. This will also make a subsequent migration of applications to the PaaS more straightforward.
- https://github.com/alphagov/govuk-aws/blob/master/doc/architecture/decisions/0002-hosting-platforms.md

Food for Thought

- What if your complete architectural documentation can live alongside your source code, rather than in a separate Wiki or Word document?
  - Markdown can do a lot to make it pretty, and GitHub and other systems have great support for it

Table of Contents

Table of Contents

Project Manager and Architect

- The *project manager* owns:
  - Creating & managing the budget & schedule
  - Providing resources
  - Negotiating with marketing
  - Ensuring quality
  - External-facing aspects of the project (e.g. reporting progress & risks to management)
- The *architect* owns:
  - Achieving quality
  - Determining measures to be used
  - Reviewing requirements for feasibility
  - Generating develop-time requirements
  - Leading the development team

Project Step Outline

- Create initial top-down plan with rough estimates on cost & schedule to get upper management budget approval
- Form architecture team. Produce an initial architecture design & release plans.
- Form development teams, assign project pieces, build cost & schedule estimates, and form bottom-up schedule.
- Reconcile bottom-up and top-down schedule.

Organization

- Remember: Architecture frequently defines organization (they should never be completely isolated)
- Typical roles in a development team:
  - Team leader
  - Developer
  - Configuration manager
  - System test manager
  - Product manager

Distributed & Global Development

- Much development is distributed & global due to cost, skill sets & labor availability, and local knowledge of markets.
- Coordination is crucial. Methods include:
  - Informal contacts
  - Documentation
  - Meetings
  - Electronic communication
- Documentation is especially important in distributed development

Implementing

- Project manager is responsible for resisting creeping functionality
- Typical release cycle is 2 weeks to three months
- Releases are in one of three states:
  1. Planning
  2. Development
  3. Test & repair
- Meetings are expensive!

Measuring

- "You can't manage what you can't measure"
- Global Metrics
  - Project size
  - Schedule deviation
  - Developer productivity
  - Defects
- Phase Metrics
  - Open issues
  - Unmitigated risks
  - Costs to complete

Table of Contents

Table of Contents

- *Architecture governance* is the practice and orientation by which enterprise architectures and other architectures are managed and controlled

Governance Board

Responsibilities:

- Implementing a system of controls over the creation and monitoring of all architectural components & activities, to ensure the effective introduction, implementation, and evolution of architecture within the organization
- Implementing a system to ensure compliance with internal and external standards and regulatory obligations
- Establishing processes that support effective management of the above processes within agreed parameters
- Developing practices that ensure accountability to a clearly identified stakeholder community, both inside and outside the organization

Governance Challenges

- How do you avoid excessive overhead?
- Each system has its own stakeholders and its own internal governance processes. Who is in control?
- Organizational politics

Table of Contents

- Homework 4 Due Thursday, November 5 5:30PM
- Quiz 4 Due Thursday, November 5 5:30PM