


DePaul University

SE480 Week 7 – Architectural Patterns

Steven Engelhardt

DePaul University

Autumn 2020

Table of Contents

1. Last Week
   - Quiz 3 Review
2. Architectural Patterns
   - Introduction to Architectural Patterns
   - Layered Pattern
   - Multi-Tier Pattern
   - Pipe and Filter
   - Publish-Subscribe
   - Saga Pattern
   - Command-Query Responsibility Separation (CQRS)
   - Event Sourcing
   - UI Patterns (MVC, MVP, MVVM)
     - Model-View-Controller
     - Model-View-Presenter
     - Model-View-ViewModel
   - Peer-to-Peer
   - Map-Reduce
   - Reflection
3. Wrap-Up

Summary of Last Week

- Most of the cost of the typical software system occurs after it has been initially released, so investing in *modifiability* makes a lot of economic sense
- Talked about many other architectural qualities such as *observability*, *deployability*, *portability*, etc.

Table of Contents

1. Last Week
   - Quiz 3 Review
2. Architectural Patterns
   - Introduction to Architectural Patterns
   - Layered Pattern
   - Multi-Tier Pattern
   - Pipe and Filter
   - Publish-Subscribe
   - Saga Pattern
   - Command-Query Responsibility Separation (CQRS)
   - Event Sourcing
   - UI Patterns (MVC, MVP, MVVM)
     - Model-View-Controller
     - Model-View-Presenter
     - Model-View-ViewModel
   - Peer-to-Peer
   - Map-Reduce
   - Reflection
3. Wrap-Up

Question 1

(1 point(s)) Which of the below terms best describes a series of events which arrive according to some probabilistic distribution?

1. Periodic
2. Stochastic
3. Sporadic
4. Random

Question 1

(1 point(s)) Which of the below terms best describes a series of events which arrive according to some probabilistic distribution?

1. Periodic
2. **Stochastic**
3. Sporadic
4. Random

Question 2

(1 point(s)) As architect, you decide that you want the response time of your system to be as *predictable* as possible. Which performance metric will you most focus upon?

1. Latency
2. Throughput
3. Jitter
4. Miss rate

Question 2

(1 point(s)) As architect, you decide that you want the response time of your system to be as *predictable* as possible. Which performance metric will you most focus upon?

1. Latency
2. Throughput
3. **Jitter**
4. Miss rate

Question 3

(3 point(s)) After careful profiling, you have determined that the performance of your application is completely bound by how fast it can retrieve data from memory and/or cache. Your application's data access pattern presently retrieves 20% of data from L1 cache, 30% from L2 cache, and 50% from main memory. After weeks of optimization, you manage to improve this to 50% of data from L1 cache, 40% from L2 cache, and 10% from main memory. How much faster is your application?
Use the latency measurements from the Latency Numbers Every Programmer Should Know slide.
Hint: If the system previously had an average latency of 500 ns, and the new system had an average latency of 100 ns, then the system is $500/100 = 5$ times faster.

Question 3

(3 point(s)) After careful profiling, you have determined that the performance of your application is completely bound by how fast it can retrieve data from memory and/or cache. Your application's data access pattern presently retrieves 20% of data from L1 cache, 30% from L2 cache, and 50% from main memory. After weeks of optimization, you manage to improve this to 50% of data from L1 cache, 40% from L2 cache, and 10% from main memory. How much faster is your application?
Use the latency measurements from the Latency Numbers Every Programmer Should Know slide.
Hint: If the system previously had an average latency of 500 ns, and the new system had an average latency of 100 ns, then the system is $500/100 = 5$ times faster.

Old average latency (ns): $0.2 \times 0.5 + 0.3 \times 7 + 0.5 \times 100 = 52.2$
New average latency (ns): $0.5 \times 0.5 + 0.4 \times 7 + 0.1 \times 100 = 2.8 + 0.25 + 10 = 13.05$
Speedup: $52.2/13.05 = \boxed{4}$

Question 4

(1 point(s)) A startup which is creating a video streaming application has hired you to consult on their architecture. They ask you for a recommendation as to how they should deal with mobile customers whose network quality varies as they travel. Which is the most appropriate tactic for you to recommend?

1. Manage sample rate
2. Limit event response
3. Prioritize events
4. Reduce overhead

Question 4

(1 point(s)) A startup which is creating a video streaming application has hired you to consult on their architecture. They ask you for a recommendation as to how they should deal with mobile customers whose network quality varies as they travel. Which is the most appropriate tactic for you to recommend?

1. **Manage sample rate**
2. Limit event response
3. Prioritize events
4. Reduce overhead

Question 5

(3 point(s)) You are working on an iPhone application which displays a table of data which can be sorted, filtered, and paged. This table has 1 million records in it, and each record is exactly 100 bytes.

You decide that you want customers to be able to perform all sorting, filtering, and paging on the client-side so that customers can still interact with the table even if they lose their Internet connection.

In Q1-Q2 2019, the average mobile download speed was 33.88 mega*bits* per second. Assuming that all sorting, paging, and filtering is performed on the client side, and ignoring the effects of compression, etc, how long will it take (in seconds) for the customers to view the first page's worth of data?

Remember that 1 kilobyte is 1024 bytes and that 1 megabyte is 1024 kilobytes.

Question 5

(3 point(s)) You are working on an iPhone application which displays a table of data which can be sorted, filtered, and paged. This table has 1 million records in it, and each record is exactly 100 bytes.

You decide that you want customers to be able to perform all sorting, filtering, and paging on the client-side so that customers can still interact with the table even if they lose their Internet connection.

In Q1-Q2 2019, the average mobile download speed was 33.88 mega*bits* per second. Assuming that all sorting, paging, and filtering is performed on the client side, and ignoring the effects of compression, etc, how long will it take (in seconds) for the customers to view the first page's worth of data?

Remember that 1 kilobyte is 1024 bytes and that 1 megabyte is 1024 kilobytes.

$$1000000\,\cancel{\text{records}} \times \frac{100\,\cancel{\text{bytes}}}{1\,\cancel{\text{record}}} \times \frac{1\,\cancel{\text{KiB}}}{1024\,\cancel{\text{bytes}}} \times \frac{1\,\cancel{\text{MiB}}}{1024\,\cancel{\text{KiB}}} \times \frac{8\,\cancel{\text{megabits}}}{1\,\cancel{\text{MiB}}} \times \frac{1\,\text{second}}{33.88\,\cancel{\text{megabits}}} = \boxed{22.52\,\text{seconds}}$$

Question 6

(1 point(s)) In order to improve the performance of a mobile application, you decide to introduce a cache to all data access. Which of the below problems are you *most likely to introduce* due to the addition of the cache?

1. Reduced portability of the application
2. Difficulty keeping data in the cache consistent and synchronized
3. Reduced scalability of your application due to contention
4. Increase consumption of battery power

Question 6

(1 point(s)) In order to improve the performance of a mobile application, you decide to introduce a cache to all data access. Which of the below problems are you *most likely to introduce* due to the addition of the cache?

1. Reduced portability of the application
2. **Difficulty keeping data in the cache consistent and synchronized**
3. Reduced scalability of your application due to contention
4. Increase consumption of battery power

Question 7

(1 point(s)) You are the architect of a job execution system. In an effort to keep the system simple and fair, you give each job 1 minute of execution time, and then you preempt the job to work on the next one. Which scheduling policy have you implemented?

1. Cyclic executive
2. Semantic importance
3. Rate monotonic
4. Static scheduling

Question 7

(1 point(s)) You are the architect of a job execution system. In an effort to keep the system simple and fair, you give each job 1 minute of execution time, and then you preempt the job to work on the next one. Which scheduling policy have you implemented?

1. **Cyclic executive**
2. Semantic importance
3. Rate monotonic
4. Static scheduling

Question 8

(1 point(s)) Which is often the cheapest and fastest way to make a performance improvement in a system?

1. Bound Execution Times
2. Manage Sampling Rate
3. Co-Locate Resources
4. Increase Resources

Question 8

(1 point(s)) Which is often the cheapest and fastest way to make a performance improvement in a system?

1. Bound Execution Times
2. Manage Sampling Rate
3. Co-Locate Resources
4. **Increase Resources**

Question 9

(1 point(s)) Which of the below forms of scalability best describes the type of scalability that Azure virtual machine scale sets provide? (You may need to research Azure virtual machine scale sets)

1. Vertical scalability
2. Elasticity
3. Sharding
4. Partition by function

Question 9

(1 point(s)) Which of the below forms of scalability best describes the type of scalability that Azure virtual machine scale sets provide? (You may need to research Azure virtual machine scale sets)

1. Vertical scalability
2. **Elasticity**
3. Sharding
4. Partition by function

Question 10

(1 point(s)) Which of the below is the *least accurate* description of a conclusion from the paper *The Free Lunch is Over* by Herb Sutter?

1. Applications will increasingly need to be concurrent if they want to fully exploit CPU throughput gains.
2. Efficiency and performance optimization will get less important over time.
3. Applications are likely to become increasingly CPU-bound over time.
4. Programming languages and systems will increasingly be forced to deal well with concurrency.

Question 10

(1 point(s)) Which of the below is the *least accurate* description of a conclusion from the paper *The Free Lunch is Over* by Herb Sutter?

1. Applications will increasingly need to be concurrent if they want to fully exploit CPU throughput gains.
2. **Efficiency and performance optimization will get less important over time.**
3. Applications are likely to become increasingly CPU-bound over time.
4. Programming languages and systems will increasingly be forced to deal well with concurrency.

Question 11

(1 point(s)) Your next-generation social website, which you created on a whim, has really taken off! Every day, thousands of new users are joining, and each user is posting hundreds of status updates, pictures, and videos a month. You're currently running on the fastest hardware that's commercially available to you and barely keeping up with the write load. What is the most appropriate strategy for you to continue to support your user growth?

1. Replication
2. Sharding
3. Distributed transactions
4. Peer-to-peer

Question 11

(1 point(s)) Your next-generation social website, which you created on a whim, has really taken off! Every day, thousands of new users are joining, and each user is posting hundreds of status updates, pictures, and videos a month. You're currently running on the fastest hardware that's commercially available to you and barely keeping up with the write load. What is the most appropriate strategy for you to continue to support your user growth?

1. Replication
2. **Sharding**
3. Distributed transactions
4. Peer-to-peer

Table of Contents

1 Last Week
Quiz 3 Review

2 Architectural Patterns
Introduction to Architectural Patterns
Layered Pattern
Multi-Tier Pattern
Pipe and Filter
Publish-Subscribe
Saga Pattern
Command-Query Responsibility Separation (CQRS)
Event Sourcing
UI Patterns (MVC, MVP, MVVM)
Model-View-Controller
Model-View-Presenter
Model-View-ViewModel
Peer-to-Peer
Map-Reduce
Reflection

3 Wrap-Up

Table of Contents

1. Last Week
   - Quiz 3 Review
2. Architectural Patterns
   - Introduction to Architectural Patterns
   - Layered Pattern
   - Multi-Tier Pattern
   - Pipe and Filter
   - Publish-Subscribe
   - Saga Pattern
   - Command-Query Responsibility Separation (CQRS)
   - Event Sourcing
   - UI Patterns (MVC, MVP, MVVM)
     - Model-View-Controller
     - Model-View-Presenter
     - Model-View-ViewModel
   - Peer-to-Peer
   - Map-Reduce
   - Reflection
3. Wrap-Up

What is a Pattern?

- "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" [Fow02]

- An architectural pattern establishes a relationship between:
  - A *context* – A recurring, common situation in the world that gives rise to a problem
  - A *problem*
  - A *solution*

- A *pattern definition* is therefore the specification of the three

Why are Patterns Important?

- Architectural patterns are ways of capturing proven good design structures, so they can be *reused*
- Patterns create a *language* which allow us to more efficiently communicate, understand tradeoffs, etc.

Sources of Patterns

- There are immense numbers of architectural patterns out there – far too many to cover them all.
- Here are some sources for architectural patterns that I've used in the past:
  - Patterns of Enterprise Application Architecture
  - Enterprise Integration Patterns
  - SOA Patterns
  - Catalog of Patterns of Enterprise Application Architecture
  - Cloud Design Patterns

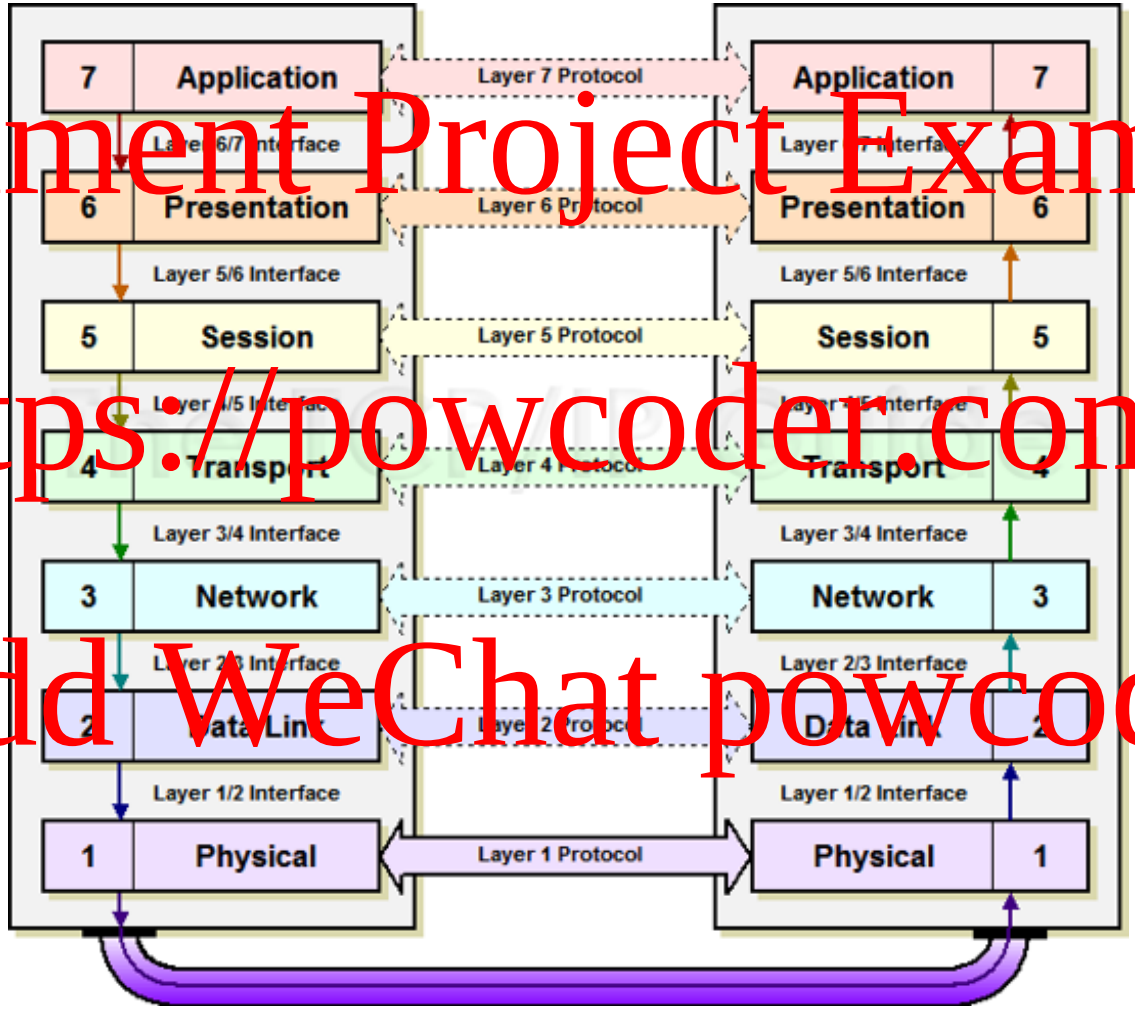
Table of Contents

1. Last Week
   - Quiz 3 Review
2. Architectural Patterns
   - Introduction to Architectural Patterns
   - Layered Pattern
   - Multi-Tier Pattern
   - Pipe and Filter
   - Publish-Subscribe
   - Saga Pattern
   - Command-Query Responsibility Separation (CQRS)
   - Event Sourcing
   - UI Patterns (MVC, MVP, MVVM)
     - Model-View-Controller
     - Model-View-Presenter
     - Model-View-ViewModel
   - Peer-to-Peer
   - Map-Reduce
   - Reflection
3. Wrap-Up

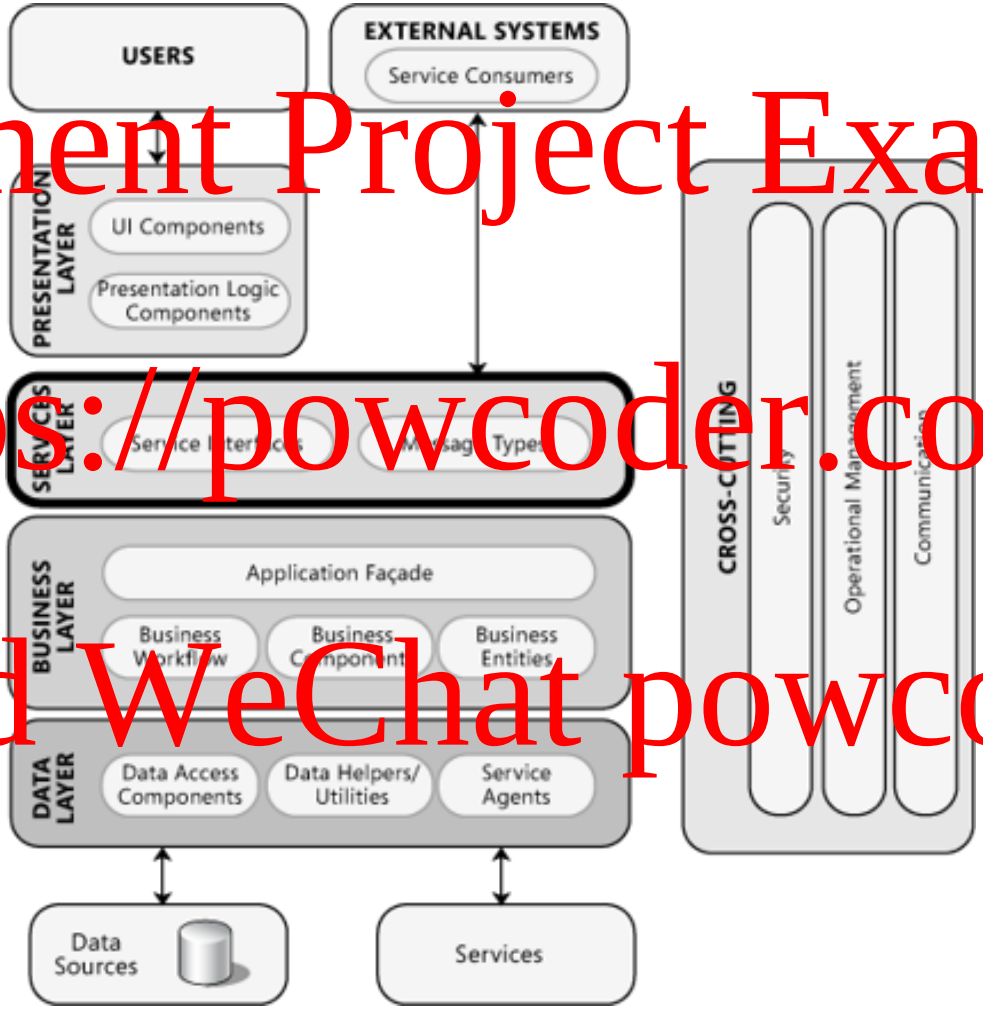
Layered Pattern

- *Context*: A large system that requires decomposition.
- *Problem*: The software needs to be segmented in such a way that the modules can be developed and evolved separately with little interaction among the parts supporting portability, modifiability, and reuse.
- *Solution*: Divide the software into units called layers. Each layer is a grouping of modules that offers a cohesive set of services. There are constraints on the allowed-to-use relationship among the layers: the relations must be unidirectional.

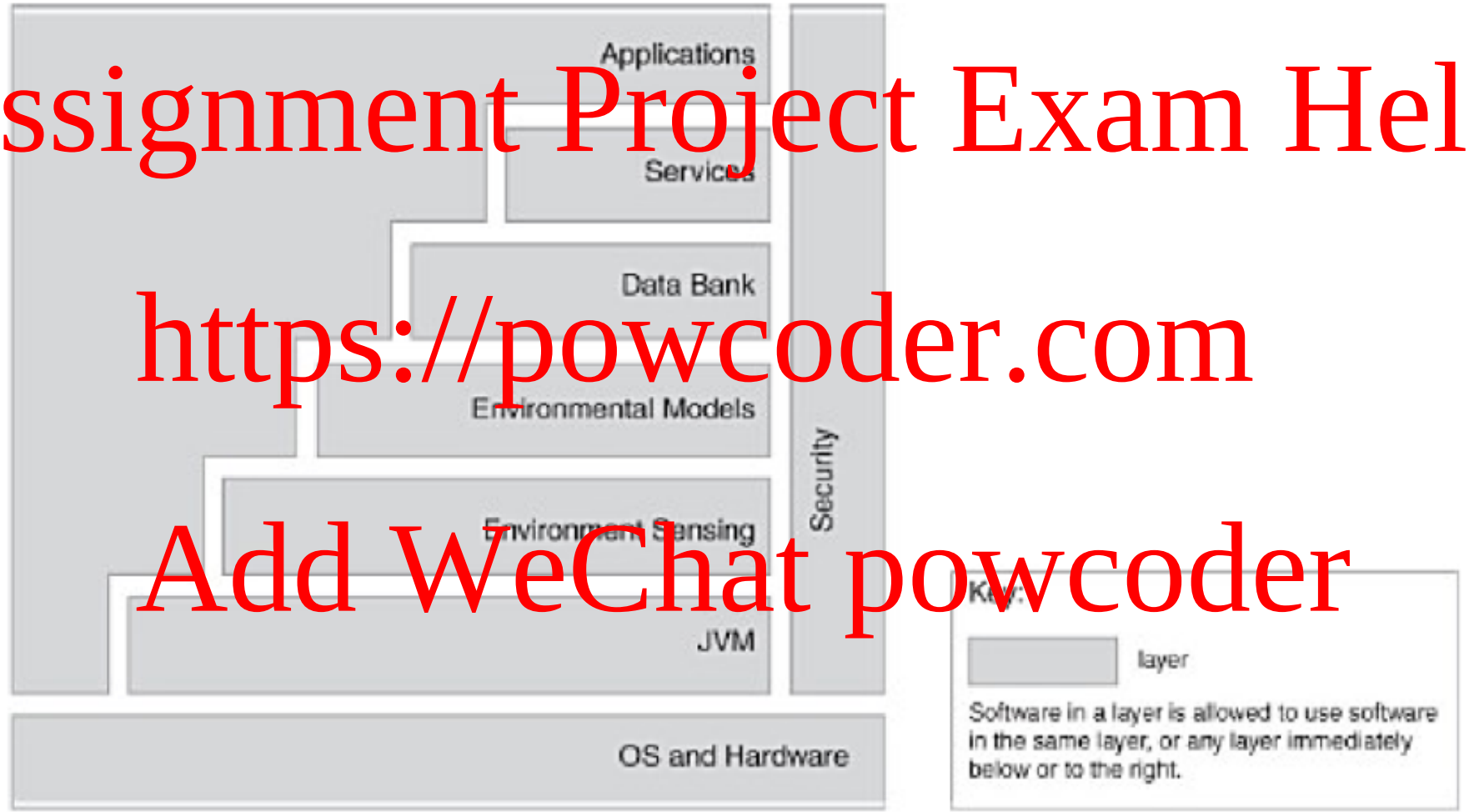
Layered Example – OSI Network Model



Layered Example – Application Layers



Layered Example –- Stairstep Diagram



The Three Principal Layers

1. *Presentation* logic is about how to handle the interaction between the user and the software
2. *Data source* logic is about communicating with other systems that carry out tasks on behalf of the application
3. *Domain* logic, also referred to as business logic, is the work that this application needs to do for the domain you're working with

[Fow02]

Layers – Forces

- Late source code changes should not ripple through system.
- Interfaces should be stable (perhaps prescribed by standards body).
- Parts of the system should be interchangeable (usually at require code changes and recompilation) but can sometimes be at runtime i.e. client switching to a different service.
- Reuse of lower-level layers in future systems.
- Similar responsibilities grouped together to help understandability and maintainability.
- Complex components need further decomposition.
- Crossing component boundaries may impede performance (i.e. when a substantial amount of data must be transferred over several boundaries).
- The system will be built by a team of programmers, and work needs to be subdivided along clear boundaries.

Layers – High Level Solution

- Structure system into an appropriate number of layers and place them on top of each other.
- Most services that Layer J provides are composed of services provided by Layer $J - 1$.
  - The services for each layer implement a strategy for combining the services of the layer below in a meaningful way.
  - Layer J's services may depend on other services in Layer J.

Layers – Dynamics

- Scenario 1: Top down communication
  - A client issues a request to Layer N. Layer N passes the request down to $N-1$, which passes to $N-2$ etc – until the task is finally performed and results are passed back up through the layers.

Layers – Dynamics

- Scenario 1: Top down communication
  - A client issues a request to Layer N. Layer N passes the request down to $N-1$, which passes to $N-2$ etc – until the task is finally performed and results are passed back up through the layers.
- Scenario 2: Bottom up communication
  - A chain of actions starts at Layer 1 (example: a device driver detects input). The input is interpreted and reported to the Layer $N+1$ etc. Referred to as notifications.

Layers – Dynamics

- Scenario 1: Top down communication
  - A client issues a request to Layer N. Layer N passes the request down to $N-1$, which passes to $N-2$ etc – until the task is finally performed and results are passed back up through the layers.
- Scenario 2: Bottom up communication
  - A chain of actions starts at Layer 1 (example: a device driver detects input). The input is interpreted and reported to the Layer $N+1$ etc. Referred to as notifications.
- Scenario 3: Requests travel through subset of layers
  - A cache at $N-1$ is able to handle the request without passing to lower layers. Note: This means higher levels might maintain state.

Layers – Dynamics

- Scenario 1: Top down communication
  - A client issues a request to Layer N. Layer N passes the request down to $N-1$, which passes to $N-2$ etc – until the task is finally performed and results are passed back up through the layers.
- Scenario 2: Bottom up communication
  - A chain of actions starts at Layer 1 (example: a device driver detects input). The input is interpreted and reported to the Layer $N+1$ etc. Referred to as notifications.
- Scenario 3: Requests travel through subset of layers
  - A cache at $N-1$ is able to handle the request without passing to lower layers. Note: This means higher levels might maintain state.
- Scenario 4: Intercept
  - Similar to Scenario 3. Request is intercepted at an intermediate layer. For example, an impatient human sends a repeat request, but the response from the previous request is already on its way back up.

Layers – Dynamics

- Scenario 1: Top down communication
  - A client issues a request to Layer N. Layer N passes the request down to $N-1$, which passes to $N-2$ etc – until the task is finally performed and results are passed back up through the layers.
- Scenario 2: Bottom up communication
  - A chain of actions starts at Layer 1 (example: a device driver detects input). The input is interpreted and reported to the Layer $N+1$ etc. Referred to as notifications.
- Scenario 3: Requests travel through subset of layers
  - A cache at $N-1$ is able to handle the request without passing to lower layers. Note: This means higher levels might maintain state.
- Scenario 4: Intercept
  - Similar to Scenario 3. Request is intercepted at an intermediate layer. For example, an impatient human sends a repeat request, but the response from the previous request is already on its way back up.
- Scenario 5: Two stacks of N Layers communicate with each other
  - Remember the OSI network model

Layers – Step-Wise Refinement

1. Define abstraction criterion for grouping tasks into layers. For example in a chess game application:
   - Elementary units (bishop)
   - Basic moves – such as castling
   - Medium term tactics such as Sicilian defense
   - Overall game strategies

Layers – Step-Wise Refinement

1. Define abstraction criterion for grouping tasks into layers. For example in a chess game application:
   - Elementary units (bishop)
   - Basic moves – such as castling
   - Medium term tactics such as Sicilian defense
   - Overall game strategies
2. Determine number of abstraction levels
   - Not always obvious – think about whether to combine two abstraction levels of separate them into layers.

Layers – Step-Wise Refinement

1. Define abstraction criterion for grouping tasks into layers. For example in a chess game application:
   - Elementary units (bishop)
   - Basic moves – such as castling
   - Medium term tactics such as Sicilian defense
   - Overall game strategies
2. Determine number of abstraction levels
   - Not always obvious – think about whether to combine two abstraction levels of separate them into layers.
3. Name layers and assign tasks to them

Layers – Step-Wise Refinement

1. Define abstraction criterion for grouping tasks into layers. For example in a chess game application:
   - Elementary units (bishop)
   - Basic moves – such as castling
   - Medium term tactics such as Sicilian defense
   - Overall game strategies
2. Determine number of abstraction levels
   - Not always obvious – think about whether to combine two abstraction levels of separate them into layers.
3. Name layers and assign tasks to them
4. Specify the services
   - Layers are strictly separated from each other (no component may spread over more than one layer)
   - Modules shared between layers relax the principles of strict layering.
   - Keep base layers slim and provide rich higher level services.

Layers – Step-Wise Refinement (Continued)

5. Iterate over the above steps.
   - Not a good idea to define components and services first. Think about layers EARLY.

Layers – Step-Wise Refinement (Continued)

5. Iterate over the above steps.
   - Not a good idea to define components and services first. Think about layers EARLY.
6. Specify an interface for each layer.
   - *Black box*: Design a flat interface that offers all of Layer J's services. Consider encapsulating this service in a Facade object.
   - *White box*: Layer $J+1$ sees the internals of Layer J.
   - *Gray box*: Compromise. Layer $J+1$ knows that Layer J contains three components and addresses them separately, but is unaware of the internal workings of such components.
   - Use the black box approach whenever possible because it supports system evolution. Make exceptions for purposes of efficiency.

Layers – Step-Wise Refinement (Continued)

5. Iterate over the above steps.
   - Not a good idea to define components and services first. Think about layers EARLY.
6. Specify an interface for each layer.
   - *Black box*: Design a flat interface that offers all of Layer J's services. Consider encapsulating this service in a Facade object.
   - *White box*: Layer $J+1$ sees the internals of Layer J.
   - *Gray box*: Compromise. Layer $J+1$ knows that Layer J contains three components and addresses them separately, but is unaware of the internal workings of such components.
   - Use the black box approach whenever possible because it supports system evolution. Make exceptions for purposes of efficiency.
7. Specify communication between adjacent layers.
   - *Push model*: When Layer J invokes a service of Layer $J-1$ it pushes required information as part of the service call. (However this can introduce additional dependencies between layers).
   - *Pull model*: Use call back mechanisms. (See next step).

8. Decouple Adjacent Layers
   - Layer $J+1$ is aware of layer J, but J is entirely unaware of Layer $J+1$.
   - For bottom up communication, use callbacks. Upper layer registers callback functions with lower layer. (Reactor pattern, command pattern)
   - Also decouple upper layer from lower layer to a certain extent by coding the upper layer against an interface, so that the lower level interface can be easily exchanged at runtime.

Layers – Advantages

- *Reduces the scope of attention* by allowing you to think about responsibilities independently
- Allows you to *substitute different implementations* of modules (e.g. multiple different presentation layers on top of common domain logic)
- Exposes *seams that are good affordance for testing*
  - UI code is often tricky to test; the thinner the presentation layer, the less there is to test

Layers – Disadvantages

- Up-front cost and complexity
- Poor layering can get in the way rather than helping
- Performance cost

Layers – Recommendations

- For complex applications, first split by *domain*, then split by *layer*
- Beware separating development teams by layers
  - The rich interplay between layers within a domain necessitates frequent swapping between them
  - Developers don't have to be full-stack (although that is laudable) but teams should be

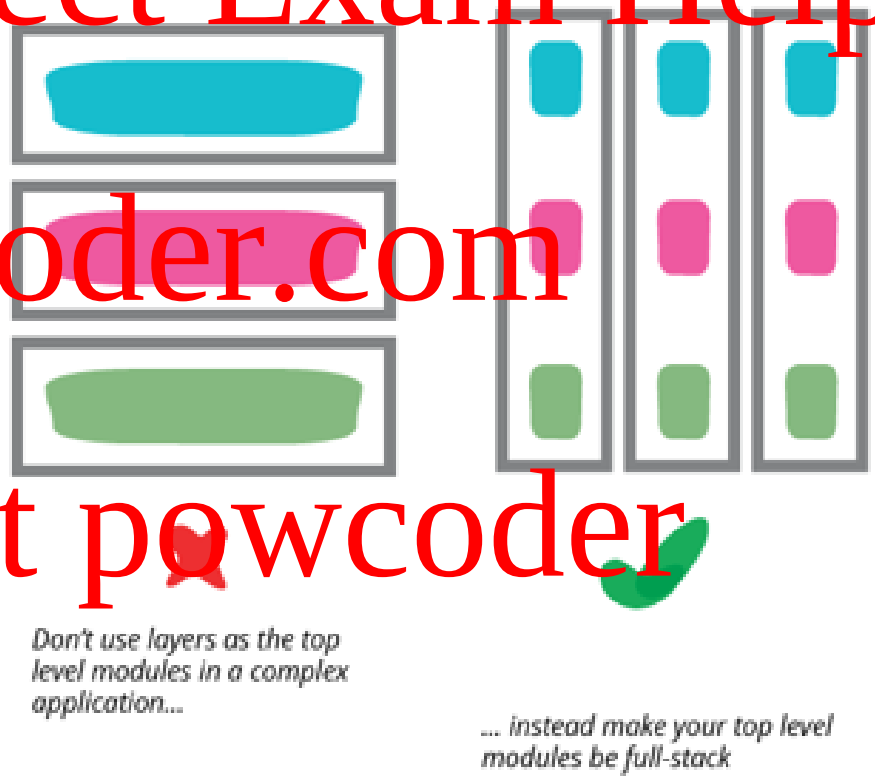


Table of Contents

1 Last Week
Quiz 3 Review

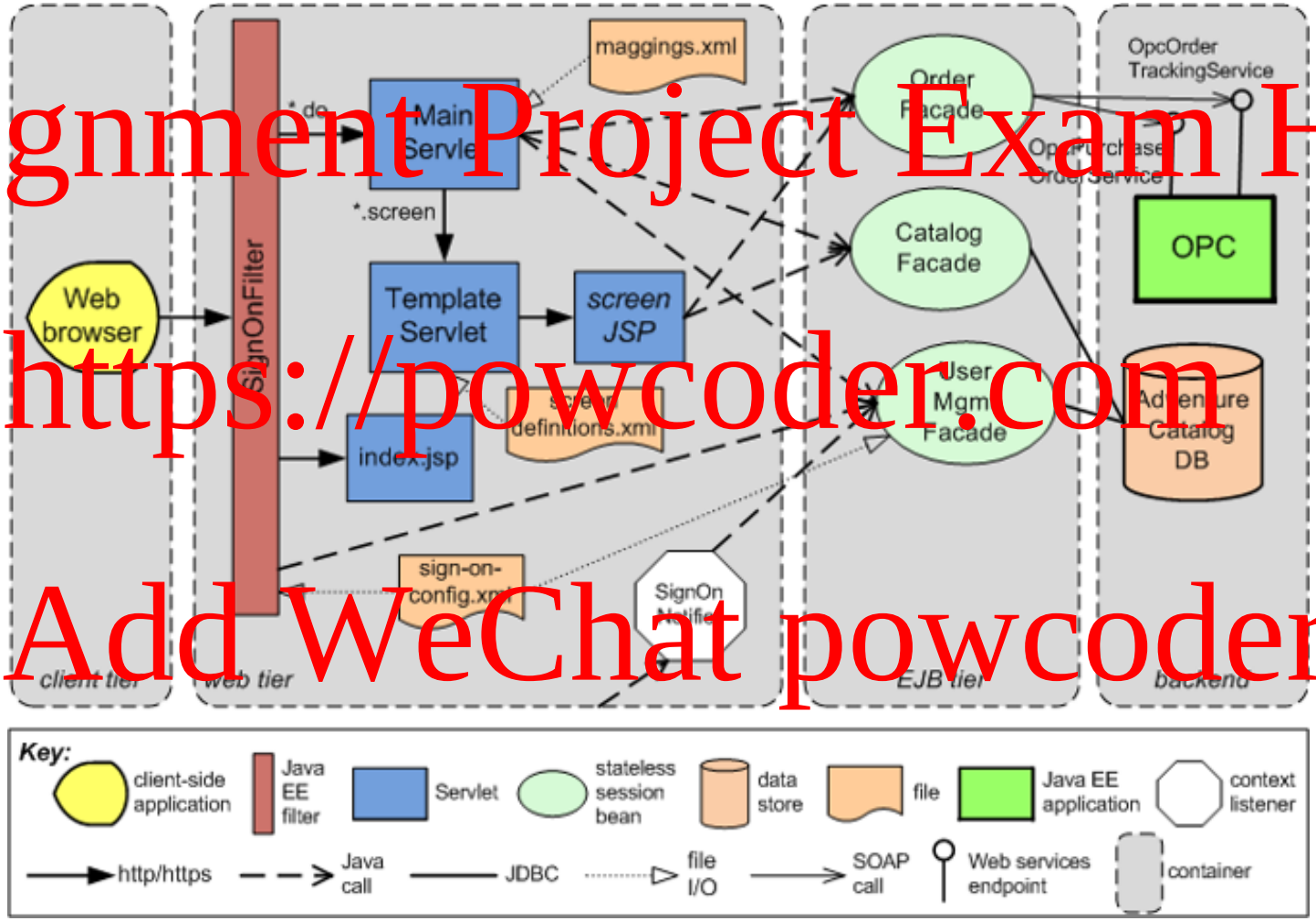
2 Architectural Patterns
Introduction to Architectural Patterns
Layered Pattern
Multi-Tier Pattern
Pipe and Filter
Publish-Subscribe
Saga Pattern
Command-Query Responsibility Separation (CQRS)
Event Sourcing
UI Patterns (MVC, MVP, MVVM)
Model-View-Controller
Model-View-Presenter
Model-View-ViewModel
Peer-to-Peer
Map-Reduce
Reflection

3 Wrap-Up

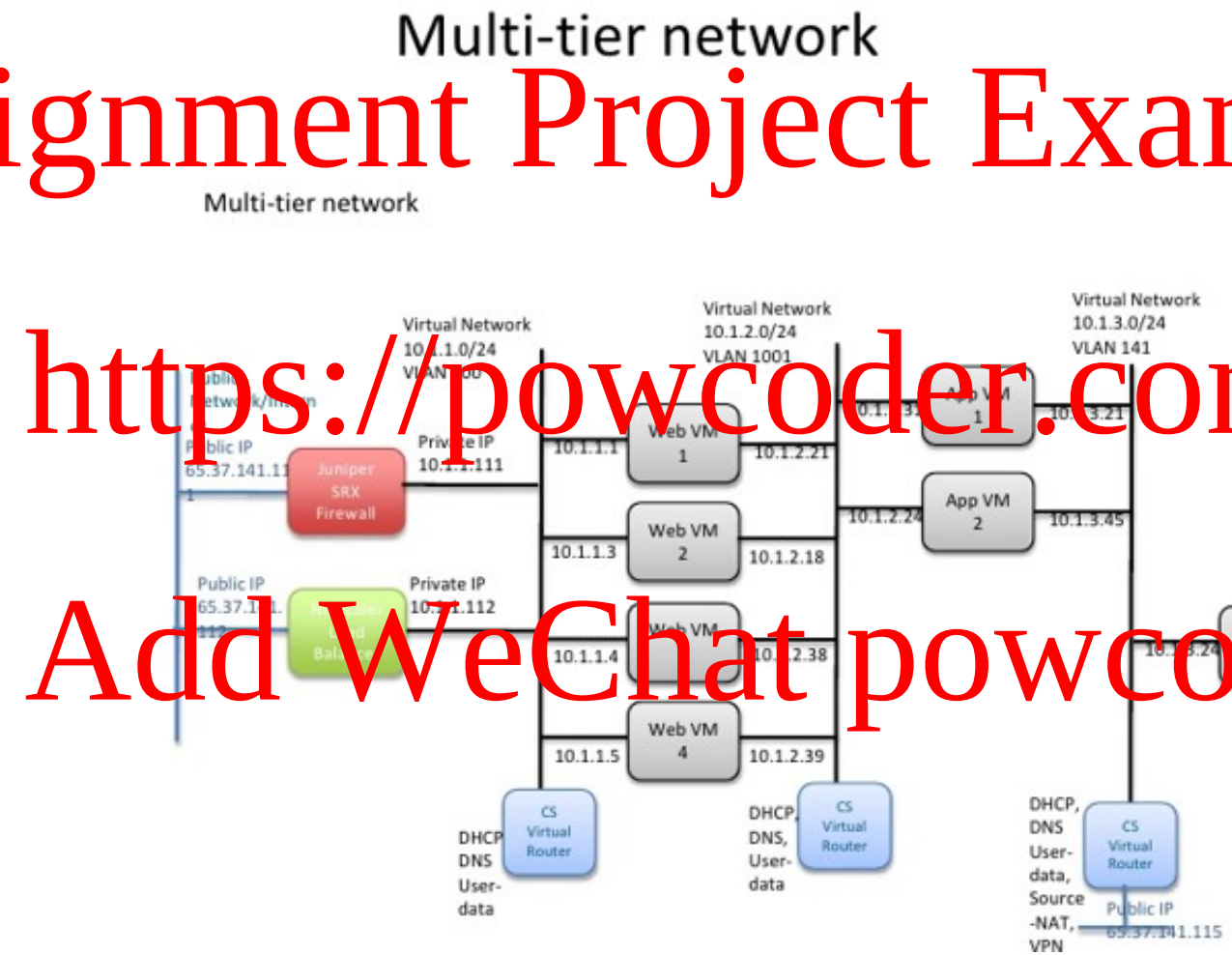
Multi-Tier Pattern

- *Context*: In a distributed deployment, there is often a need to distribute a system's infrastructure into distinct subsets.
- *Problem*: How can we split the system into a number of computationally independent execution structures – groups of software and hardware – connected by some communications media?
- *Solution*: Organize the execution structures into logical groupings called tiers.
- Tiers are not components, but rather *logical groupings of components*.
- Don't confuse tiers with layers! Tiers apply to *runtime entities*, but layering applies to *modules*.

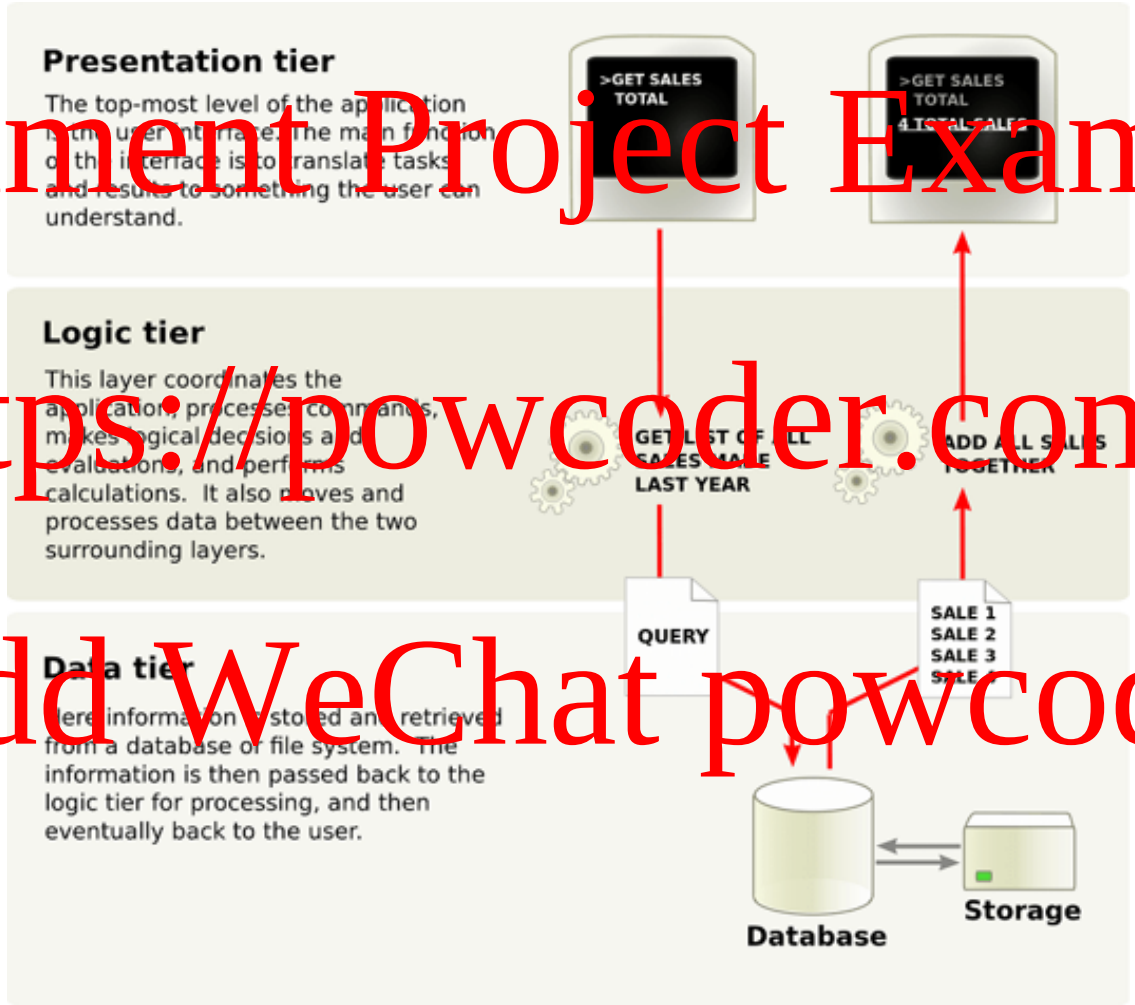
Multi-Tier Example 1



Multi-Tier Example 2



Multi-Tier Questionable Example



Multi-Tier Pattern Advantages and Disadvantages

- Advantages:
  - Makes it easier to ensure security
  - Can optimize performance and availability in specialized ways
- Disadvantages:
  - Cost
  - Complexity

Table of Contents

1 Last Week
Quiz 3 Review

2 Architectural Patterns
Introduction to Architectural Patterns
Layered Pattern
Multi-Tier Pattern
Pipe and Filter
Publish-Subscribe
Saga Pattern
Command-Query Responsibility Separation (CQRS)
Event Sourcing
UI Patterns (MVC, MVP, MVVM)
Model-View-Controller
Model-View-Presenter
Model-View-ViewModel
Peer-to-Peer
Map-Reduce
Reflection

3 Wrap-Up

Pipe-and-Filter Pattern

- *Context*: Designing a system which transforms streams of discrete data items
- *Problem*: How to divide the system into reusable, loosely coupled components with simple, generic interaction mechanisms
- *Solution*: Model the application as components which implement successive transformations of streams of data. Each processing step is a *filter component* and data is passed through *pipes* between adjacent filters.

Pipe-and-Filter Pattern

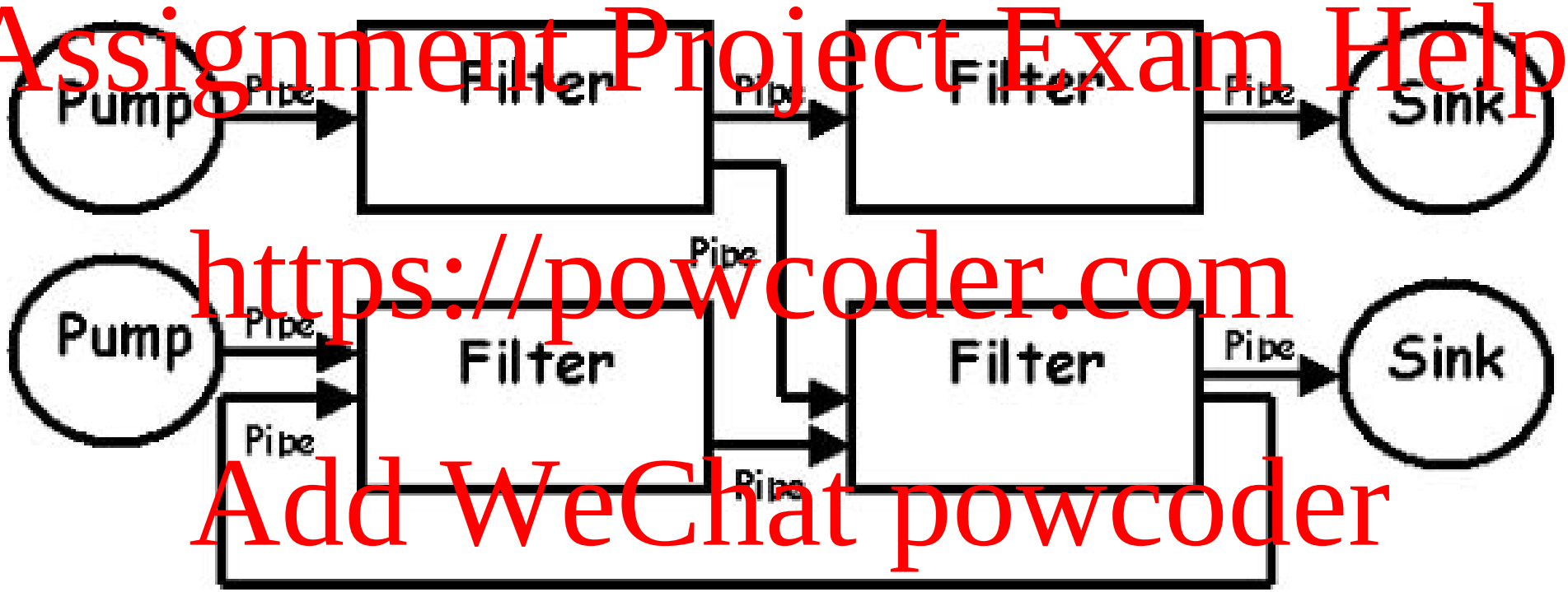


Figure: Pipe-and-filter pattern

Pipe-and-Filter Context

- Future system enhancements that involve exchanging, recombining, or adding new steps
- Small processing steps are easier to reuse in different contexts than larger components

- Non-adjacent processing steps do not share information
- Different sources of input data exists
- Present or store final results in various ways

- Explicit storage of intermediate results clutters directories and is error prone
- Multi-processing may be useful

Pipe-and-Filter Solution

- The Pipes and Filters Architectural pattern divides the task into several sequential processing steps
- Steps are connected by the data flow through the system
- Each processing step is implemented as a filter which *consumes and delivers data incrementally*
- Input to system provided by a *data source* (i.e. text file, sensor data, etc.)
- Output flows to a *data sink* (i.e. file, terminal, etc.)
- Data source, filters, and data sinks are connected sequentially by *pipes*
- The sequence is call a *processing pipeline*

- The conceptual basis for nearly every data processing system out there, including compilers; traditional ETL tools like Informatica, SSIS, and Pentaho; and modern stream-processing frameworks like Apache Beam & Apache Flink

Pipe-and-Filter Example: Compiler

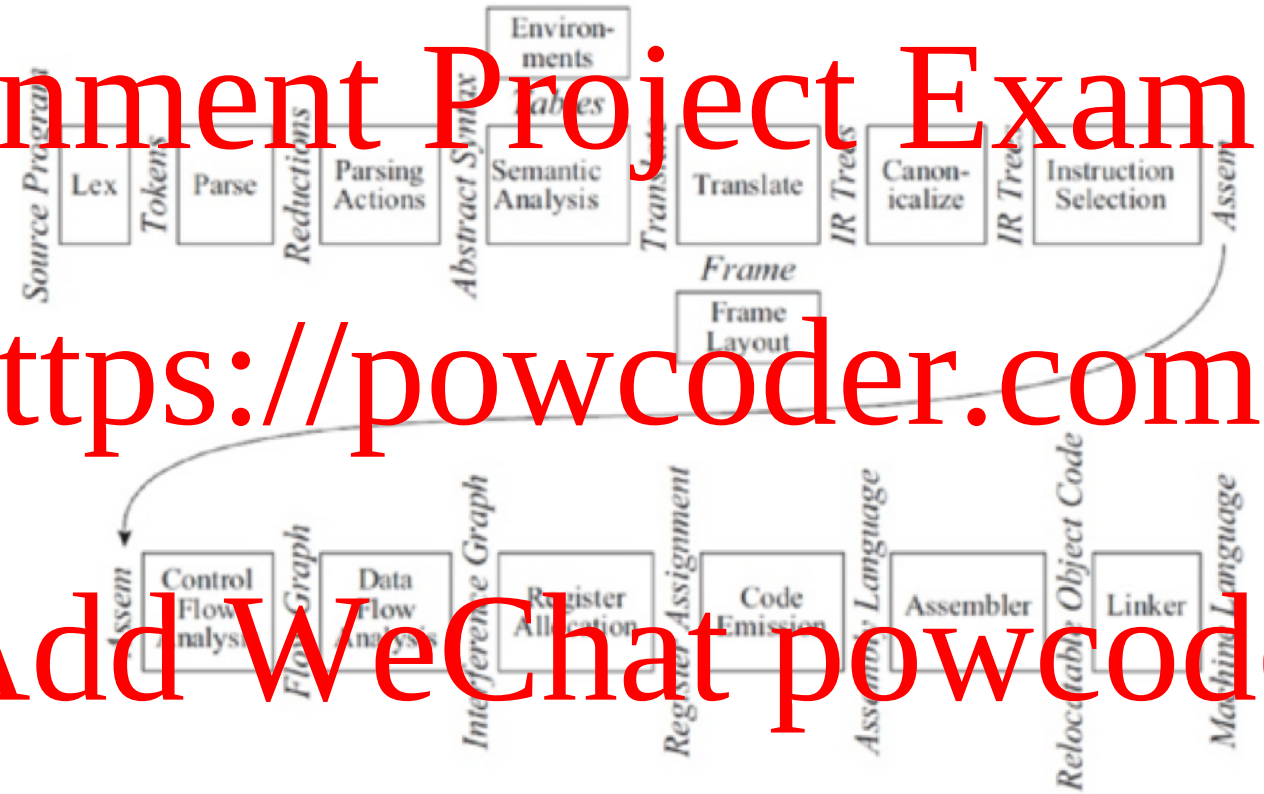
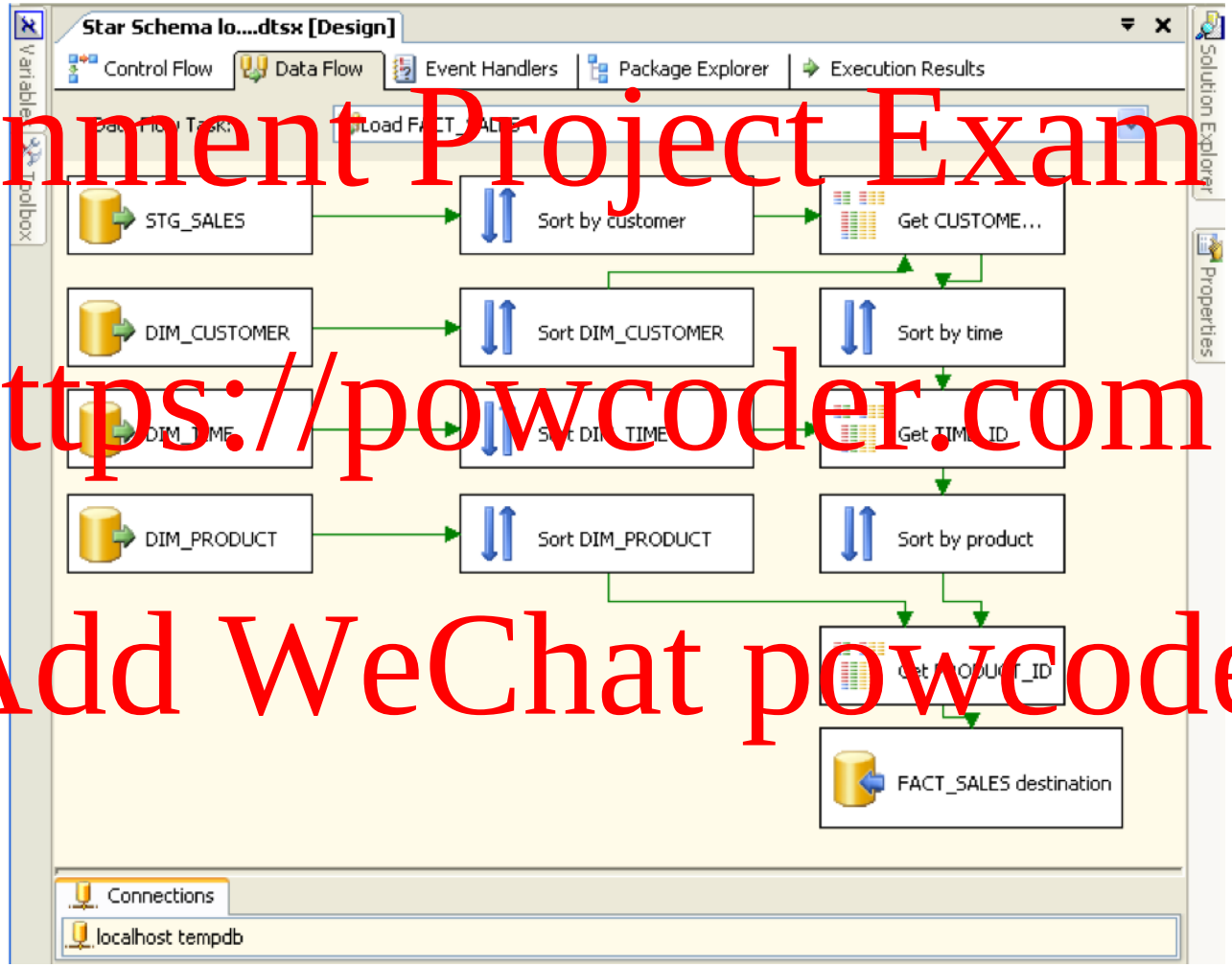
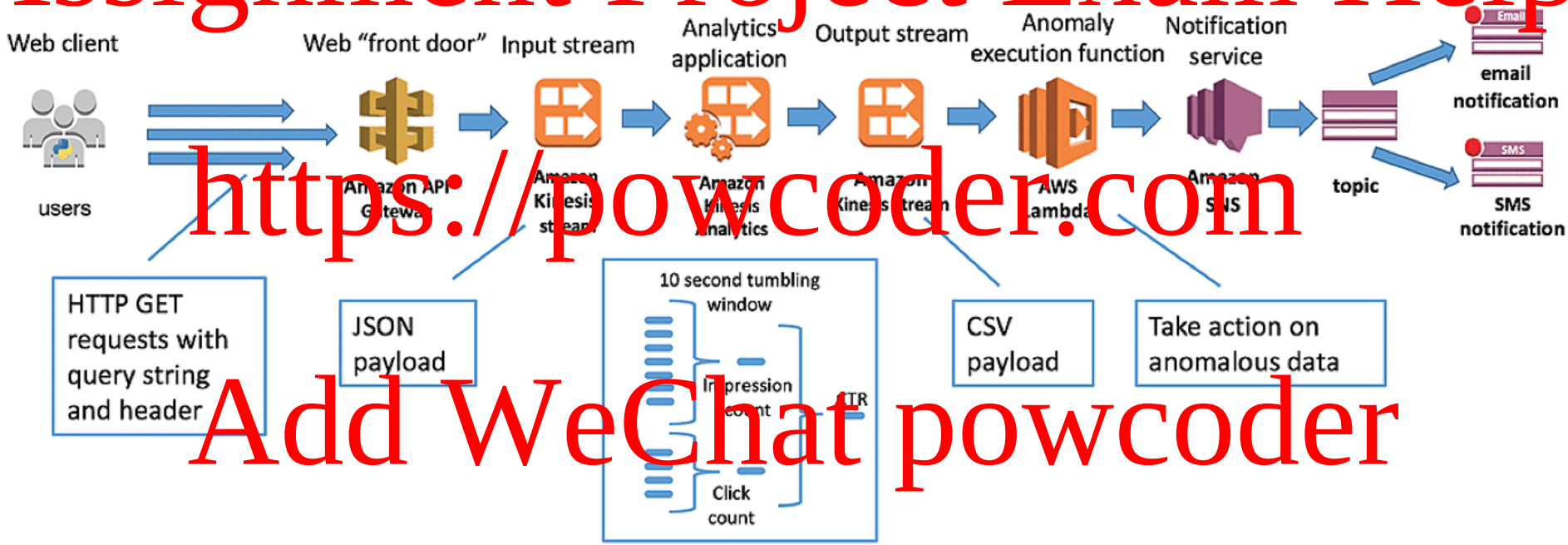


FIGURE 1.1. Phases of a compiler, and interfaces between them

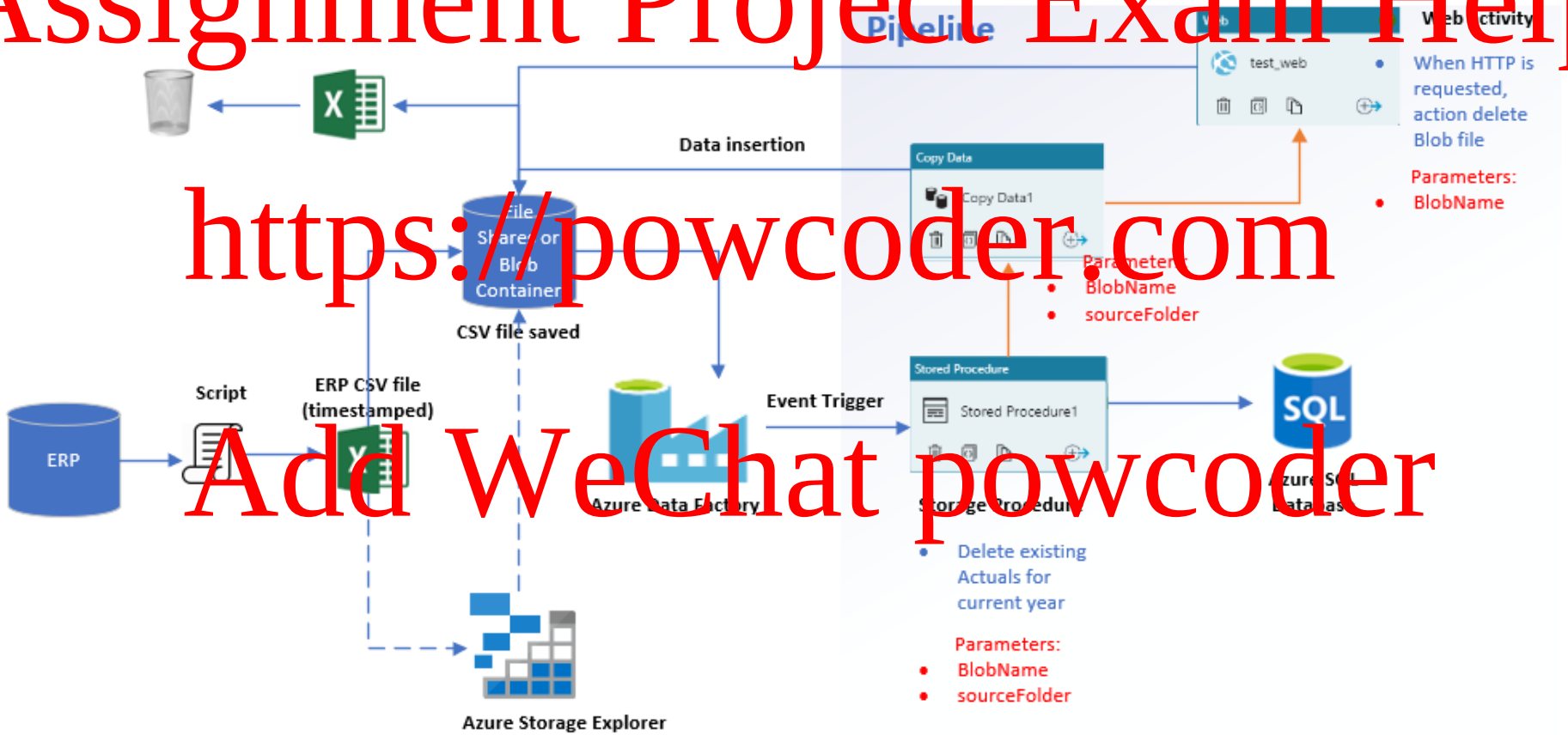
Pipe-and-Filter Example: ETL



Pipe-and-Filter Example: Clickstream Processing



Pipe-and-Filter Example: Azure Data Factory

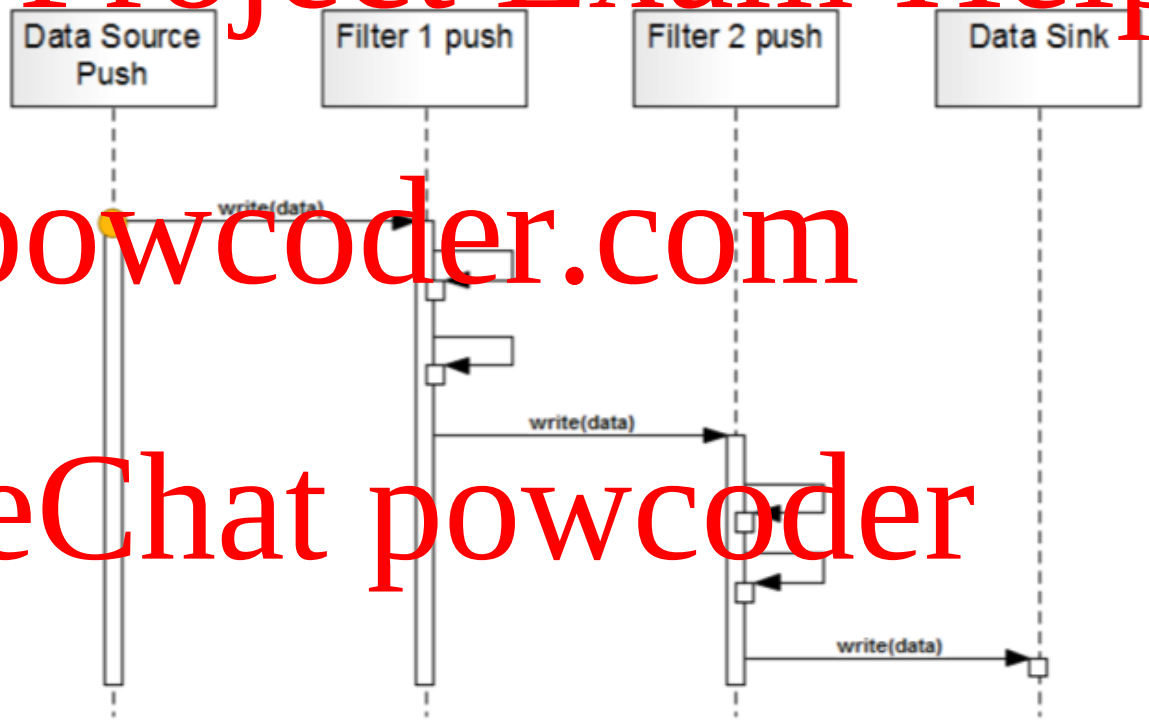


Filters

- *Passive filters*
  - Subsequent pipeline element *pulls* output data from filter.
  - Previous pipeline element *pushes* output data to filter.
- *Active filters*
  - Starts processing on its own as a separate program or thread.
  - Most common: filter is active in a loop, pulling its input from the pipeline and pushing it down the pipeline.

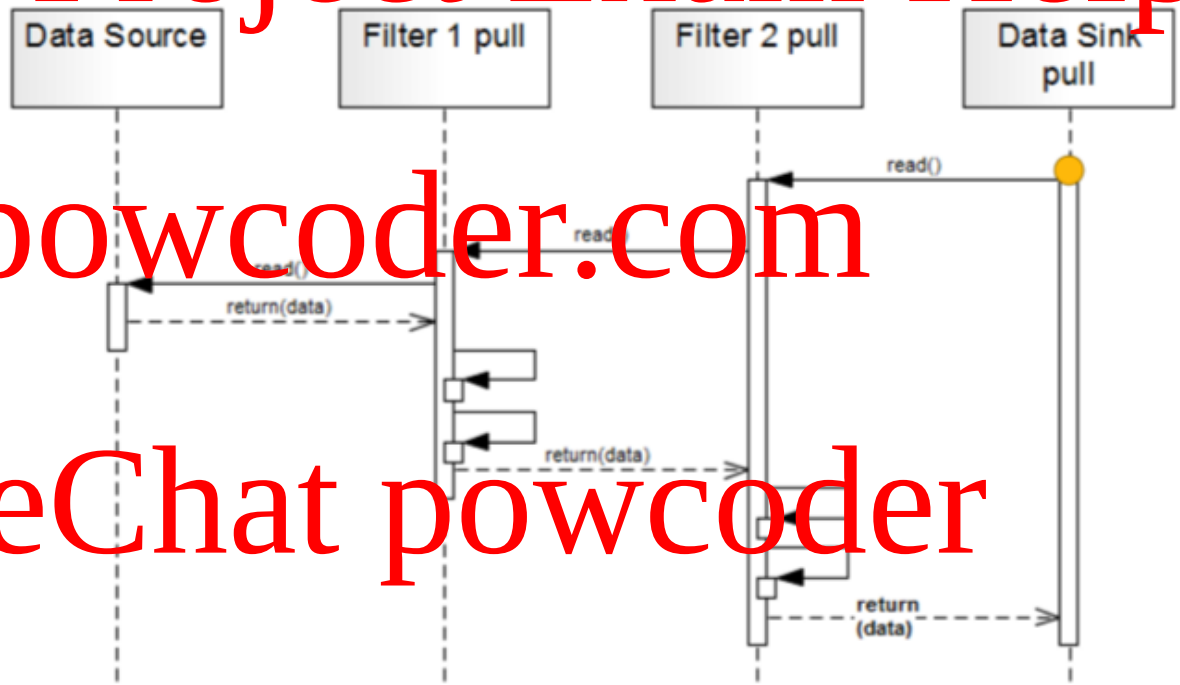
Pipe-and-Filter Scenario 1

- Push pipeline in which activity starts with the data source. Filter activity is triggered when data is written to passive filters.



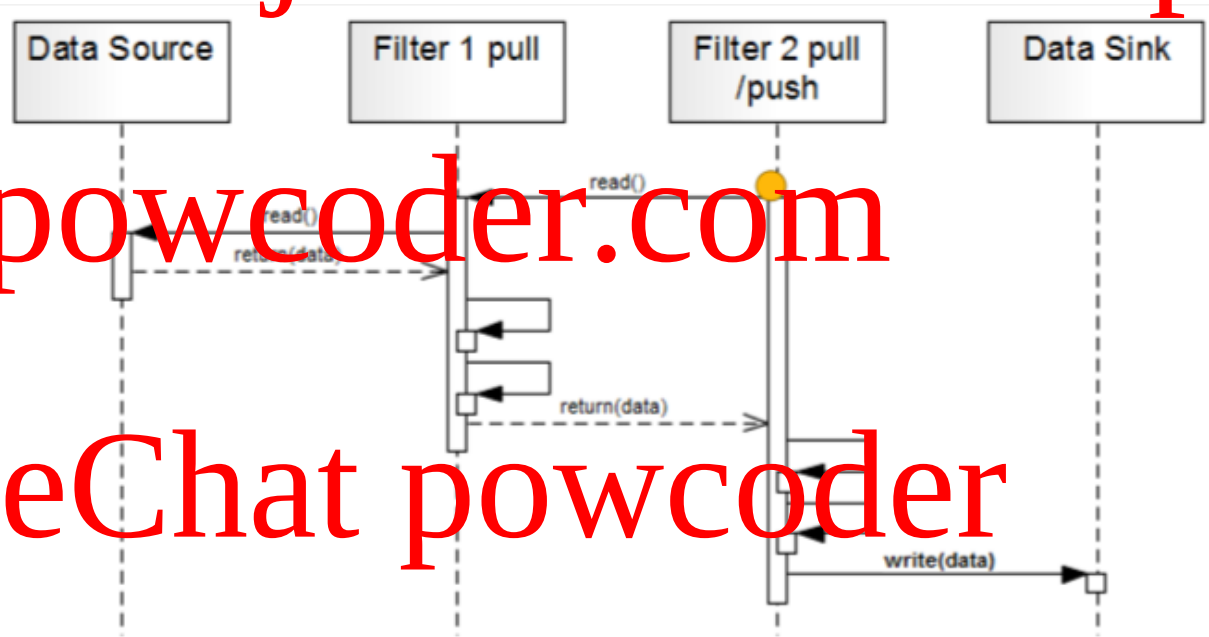
Pipe-and-Filter Scenario 2

- In the pull pipeline control flow is started by the datasink calling for data.



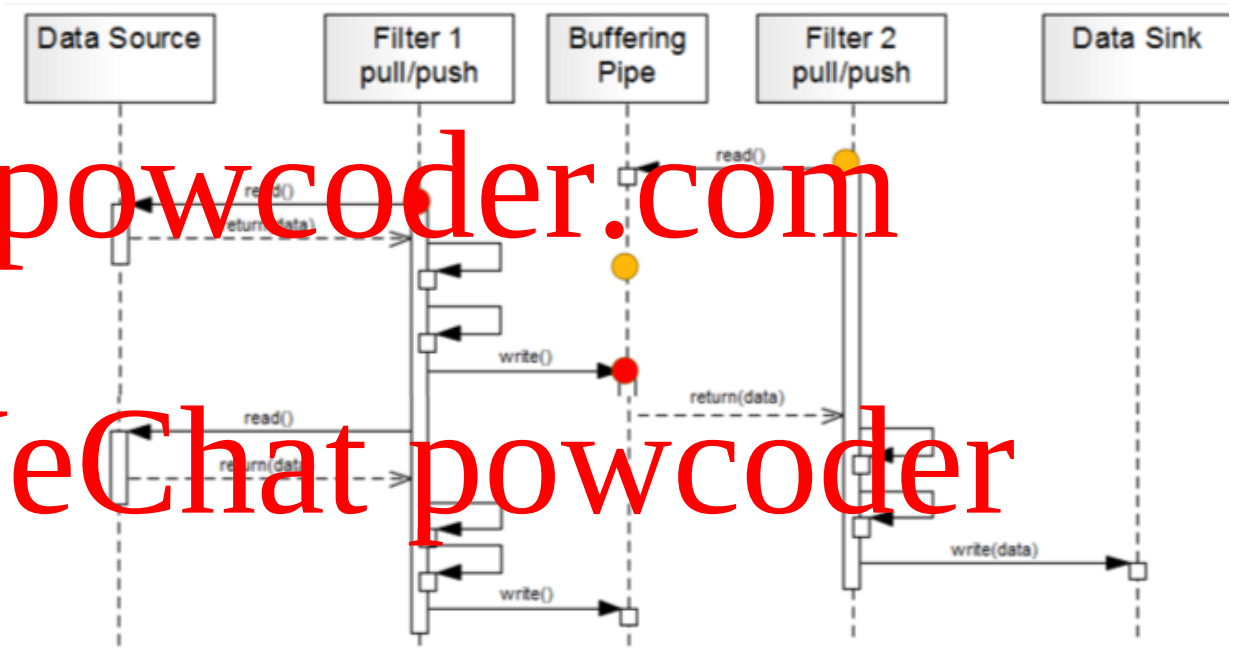
Pipe-and-Filter Scenario 3

- A mixed push-pull pipeline with passive data source and sink. The second filter plays the active role and starts the processing (in this example).



Pipe-and-Filter Scenario 4

- All filters actively pull, compute, and push data in a loop. Each runs its separate thread of control. Filters are synchronized by a buffering pipe between them. (In this example – the pipe buffers a single value only).



Pipe-and-Filter Implementation

1. Divide the system's tasks into a sequence of processing stages.
2. Define the data format to be passed along each pipe.
3. Decide how to implement each pipe connection (active, passive etc).
4. Design and implement the filters.
5. Design error handling.
6. Set up the processing pipeline (if the system handles a single task you can set this up in main(). Flexibility can be increased by setting up a shell so that customers/users can configure the pipeline.

Pipe-and-Filter Benefits

- No intermediate files necessary (although certainly possible)
- Flexibility by filter exchange.
- Flexibility by recombination.
- Reuse of filter components.
- Rapid prototyping of pipelines
- Efficiency by parallel processing.

Pipe-and-Filter Disadvantages

- Sharing state information is expensive (or inflexible).
- Efficiency gain by parallel process may be illusive.
  - Need to balance costs of transferring data between filters.
  - Filters may need to consume all their data before producing any output (loss of real pipeline behavior)
  - Context switching is expensive between threads or processes (but counteracted by multi-processors available today).
  - Synchronization of filters via pipes might create stop-and-go traffic.
- Data transformation overhead (if a single data type is used for all filter inputs and outputs)
- Error handling is the Achilles heel of the pipes and filters pattern. Define a common strategy for error reporting.

Table of Contents

1 Last Week
Quiz 3 Review
2 Architectural Patterns
Introduction to Architectural Patterns
Layered Pattern
Multi-Tier Pattern
Pipe and Filter
Publish-Subscribe
Saga Pattern
Command-Query Responsibility Separation (CQRS)
Event Sourcing
UI Patterns (MVC, MVP, MVVM)
Model-View-Controller
Model-View-Presenter
Model-View-ViewModel
Peer-to-Peer
Map-Reduce
Reflection
3 Wrap-Up

Publish-Subscribe Pattern

- *Context*: There are a number of independent producers and consumers of data that must interact. The precise number and nature of the data producers & consumers are not predetermined or fixed, nor is the data they share.

- *Problem*: How can we create integration mechanisms that support the ability to transmit messages among the producers and consumers in such a way that they are unaware of each other's identity, or potentially even their existence?

- *Solution*: Use the publish-subscribe pattern, often with an intermediary *message broker* or *event bus*.

Publish-Subscribe Concepts

- *Publish–subscribe* is a messaging pattern where senders of messages, called *publishers*, do not program the messages to be sent directly to specific receivers, called *subscribers*, but instead categorize published messages into classes without knowledge of which subscribers, if any, there may be.

- Similarly, subscribers express interest in one or more classes and only receive messages that are of interest, without knowledge of which publishers, if any, there are.

- The *message broker*, when present, normally performs filtering and store and forward to route messages from publishers to subscribers

[Wik20b]

Publish-Subscribe Filtering

- In the publish-subscribe model, subscribers typically receive only a subset of the total messages published.
- The process of selecting messages for reception and processing is called *filtering*.
- There are two common forms of filtering:
  - In a *topic-based* system, messages are published to "topics" or named logical channels. Subscribers in a topic-based system will receive all messages published to the topics to which they subscribe. The publisher is responsible for defining the topics to which subscribers can subscribe.
  - In a *content-based* system, messages are only delivered to a subscriber if the attributes or content of those messages matches constraints defined by the subscriber. The subscriber is responsible for classifying the messages.

[Wik20b]

Publish-Subscribe Pattern

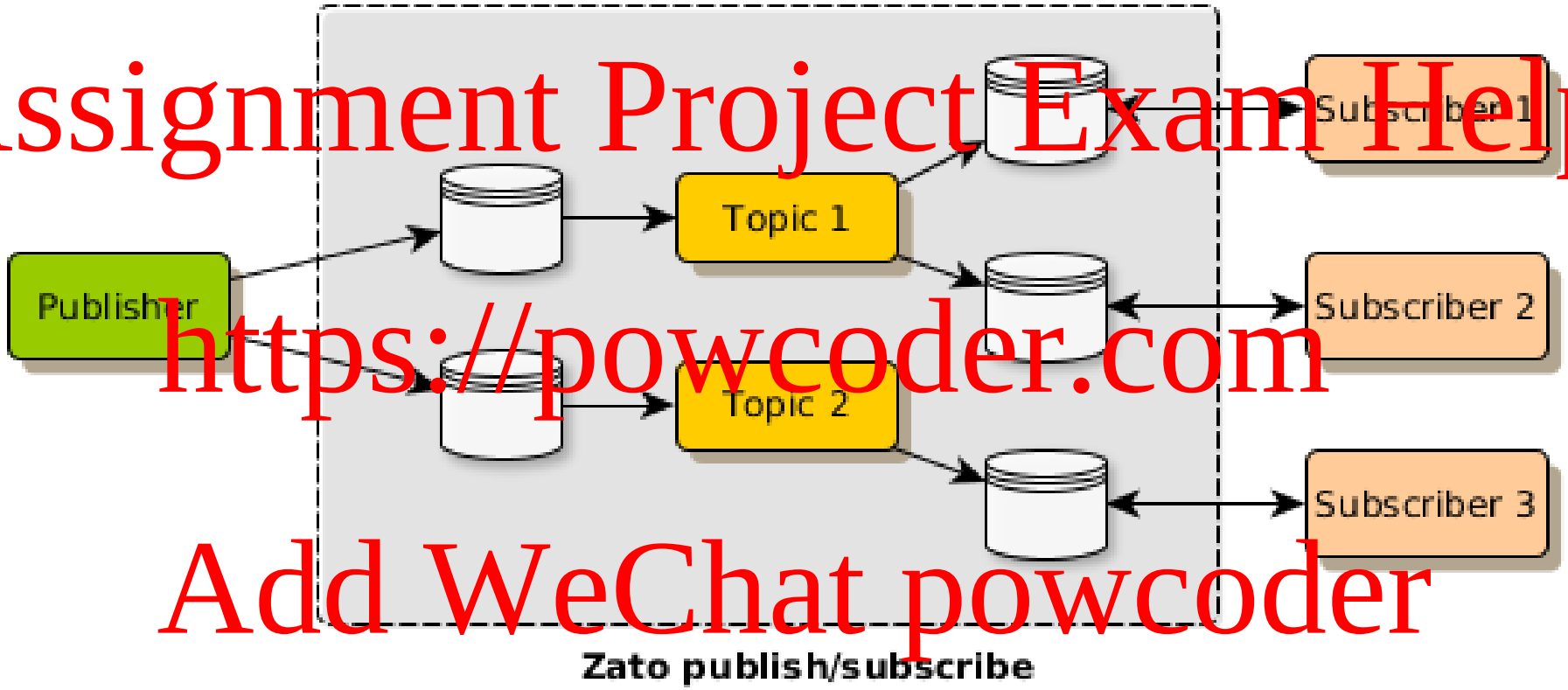


Figure: Example Publish-Subscribe System

Publish-Subscribe Advantages and Disadvantages

- Advantages:
  - Publishers are loosely coupled to subscribers
  - Scalability relative to traditional client-server
- Disadvantages:
  - Runtime performance, because indirection adds latency
  - Potential single point of failure
  - Difficult to achieve assured delivery

Publish-Subscribe Examples

- Event-based GUI systems (e.g. Win32)
- Event-based programming environments (e.g. Node.JS)
- MVC-based applications

- Enterprise resource planning (ERP) systems
- Extensible programming environments
- Mailing lists

- Social networks

Other Types of Publish-Subscribe

- *List-based publish-subscribe* – every publisher maintains a subscription list
  - Examples: mailing list
- *Broadcast-based publish subscribe* – events are broadcasted by publishers
  - Examples: ARP

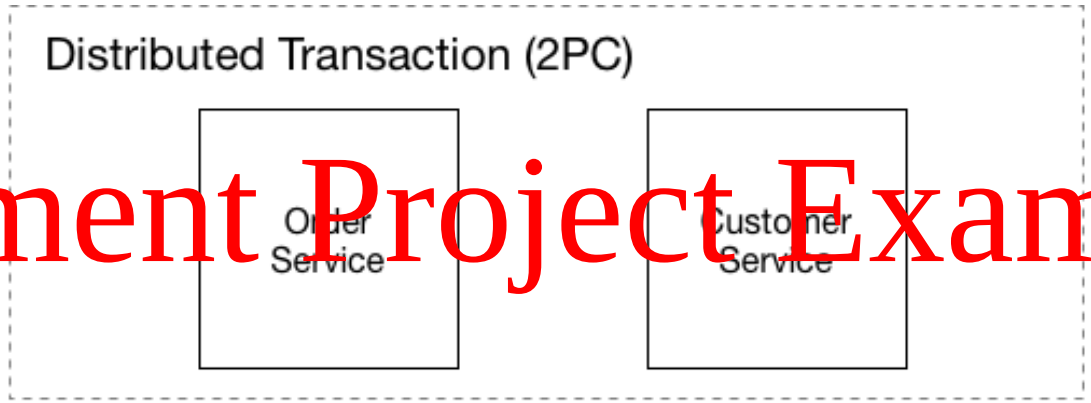
Table of Contents

1 Last Week
Quiz 3 Review
2 Architectural Patterns
Introduction to Architectural Patterns
Layered Pattern
Multi-Tier Pattern
Pipe and Filter
Publish-Subscribe
Saga Pattern
Command-Query Responsibility Separation (CQRS)
Event Sourcing
UI Patterns (MVC, MVP, MVVM)
Model-View-Controller
Model-View-Presenter
Model-View-ViewModel
Peer-to-Peer
Map-Reduce
Reflection
3 Wrap-Up

Saga Pattern

- *Context*: A service-oriented-architecture where each service has its own database
- *Problem*: How to implement transactions that span services?
- *Forces*: 2-phase commit is not an option
- *Solution*: Implement the *saga pattern*

Saga Diagram



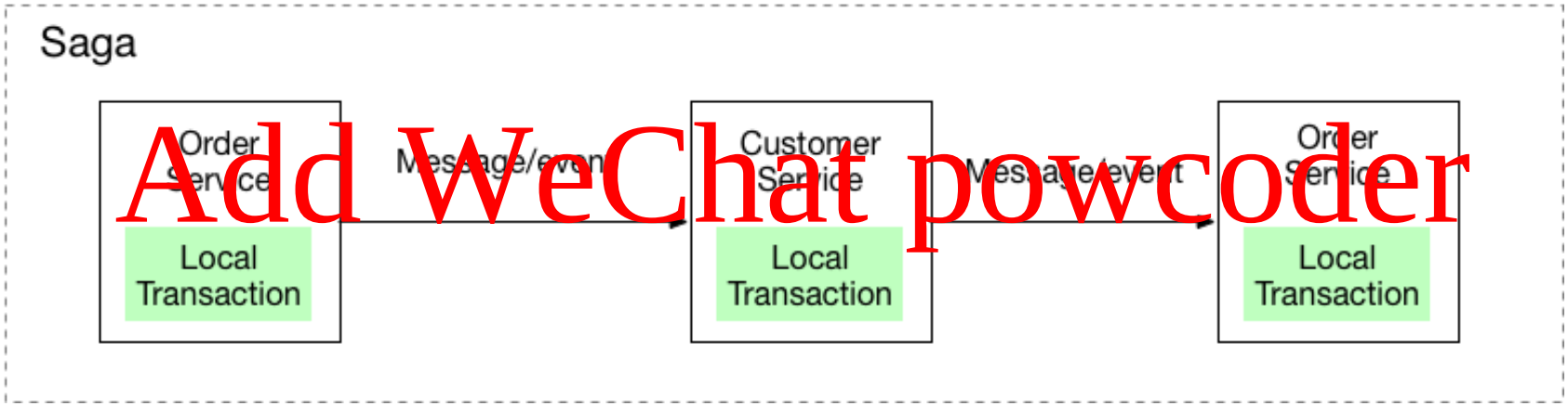


Figure: 2PC vs. Saga [Ric19]

Saga Concepts

- A *saga* is a sequence of local transactions in each of the participating services
- Unlike ACID transactions, sagas cannot automatically undo changes made by previous steps since those changes are already committed. Instead, you must write compensating transactions that explicitly undo those change.
- Unlike ACID transactions, intermediate (partially-committed) states are observable by external systems
- Each step of a saga that is followed by a step that can fail (for business reasons) must have a corresponding compensating transaction.

[Ric19]

Saga Example 1: Reserving Travel

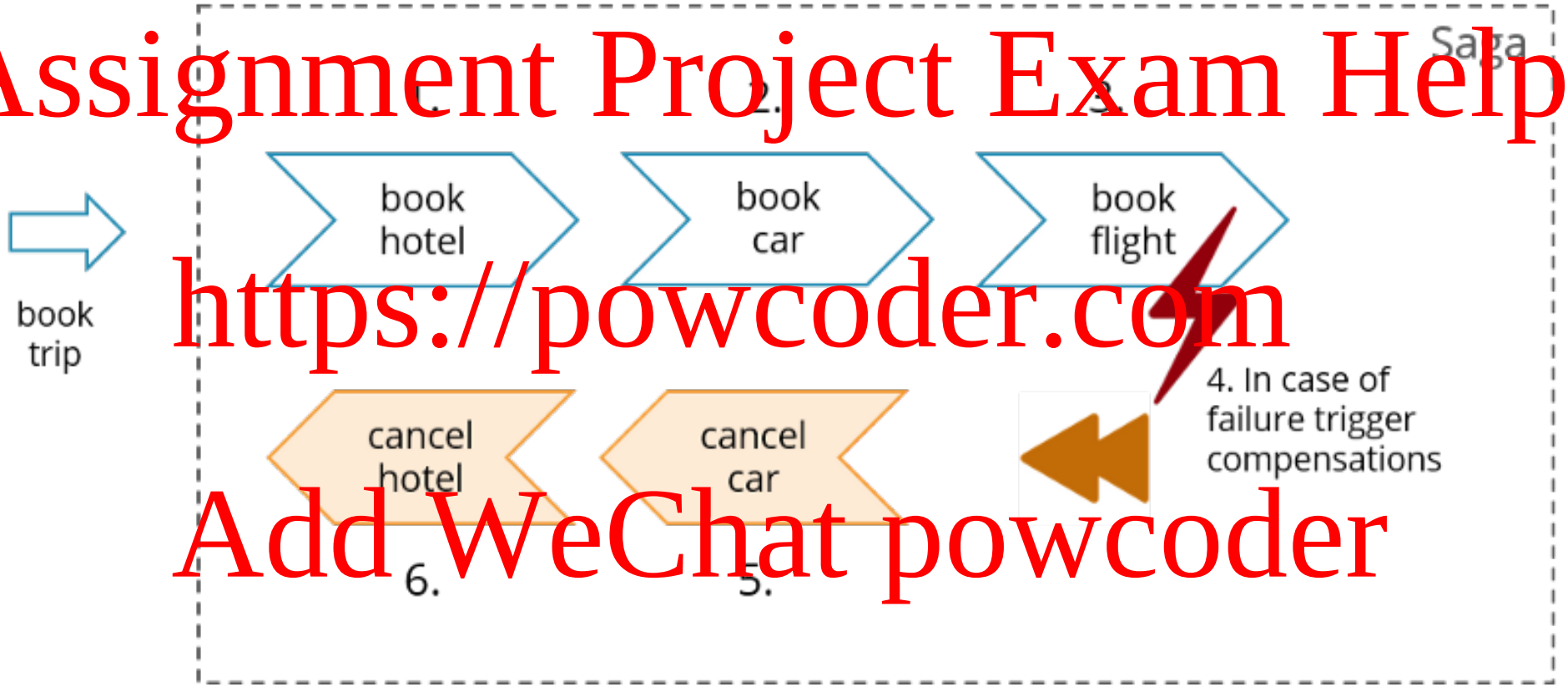


Figure: Example Saga

Saga Example 2: Provisioning Cloud Resources

Step	Transaction	Compensating Transaction
1	`New-AzResourceGroup -Name $resourceGroup -Location $location`	`Remove-AzResourceGroup -Name $resourceGroup`
2	`New-AzStorageAccount -ResourceGroupName $resourceGroup -Name $storageAccount ...`	`Remove-AzStorageAccount -Name $storageAccount -ResourceGroupName $resourceGroup`
3	`New-AzStorageContainer -Name $storageContainer ...`	`Remove-AzStorageContainer -Name $storageContainer ...`
...	...	...

- There are two ways of coordination sagas:
  1. *Choreography*: each local transaction publishes domain events that trigger local transactions in other services
  2. *Orchestration*: an orchestrator (object) tells the participants what local transactions to execute

Choreography-based saga

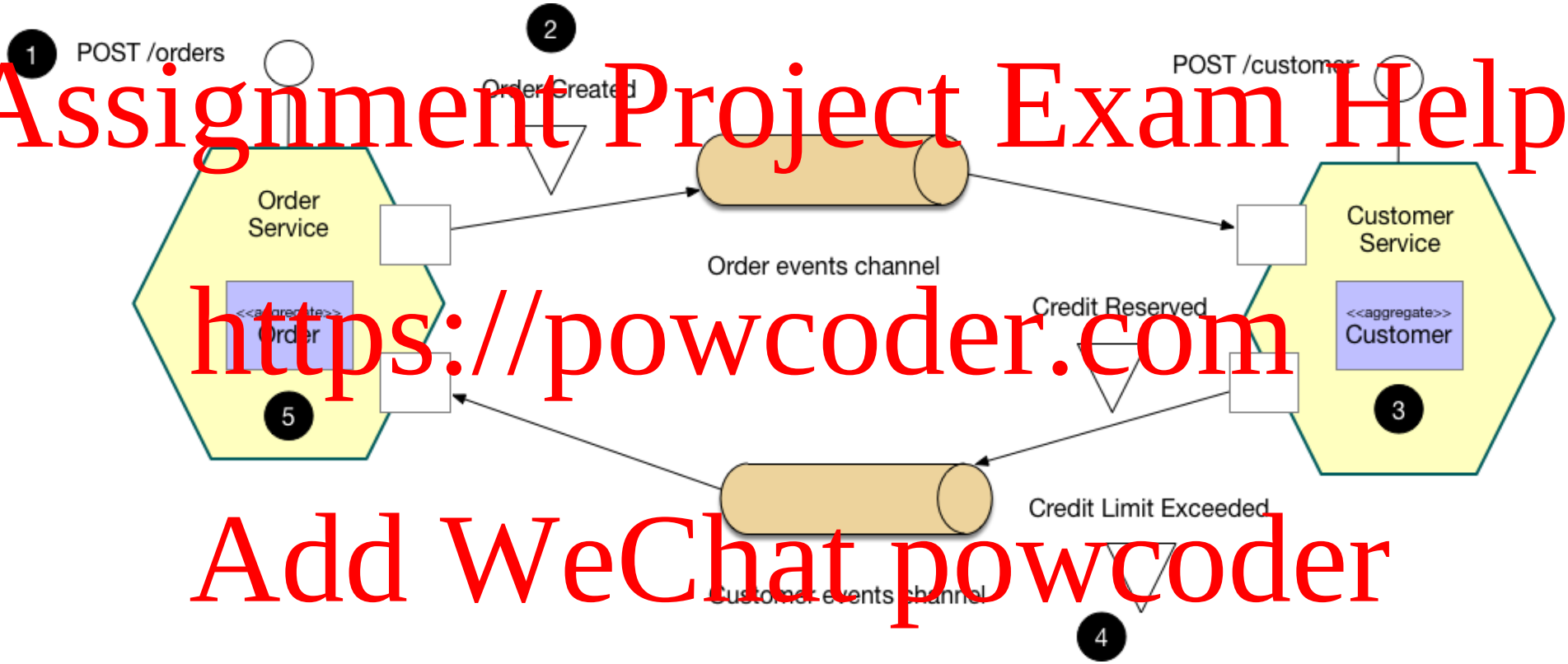


Figure: Choreography-based saga [Ric20]

Orchestration-based saga

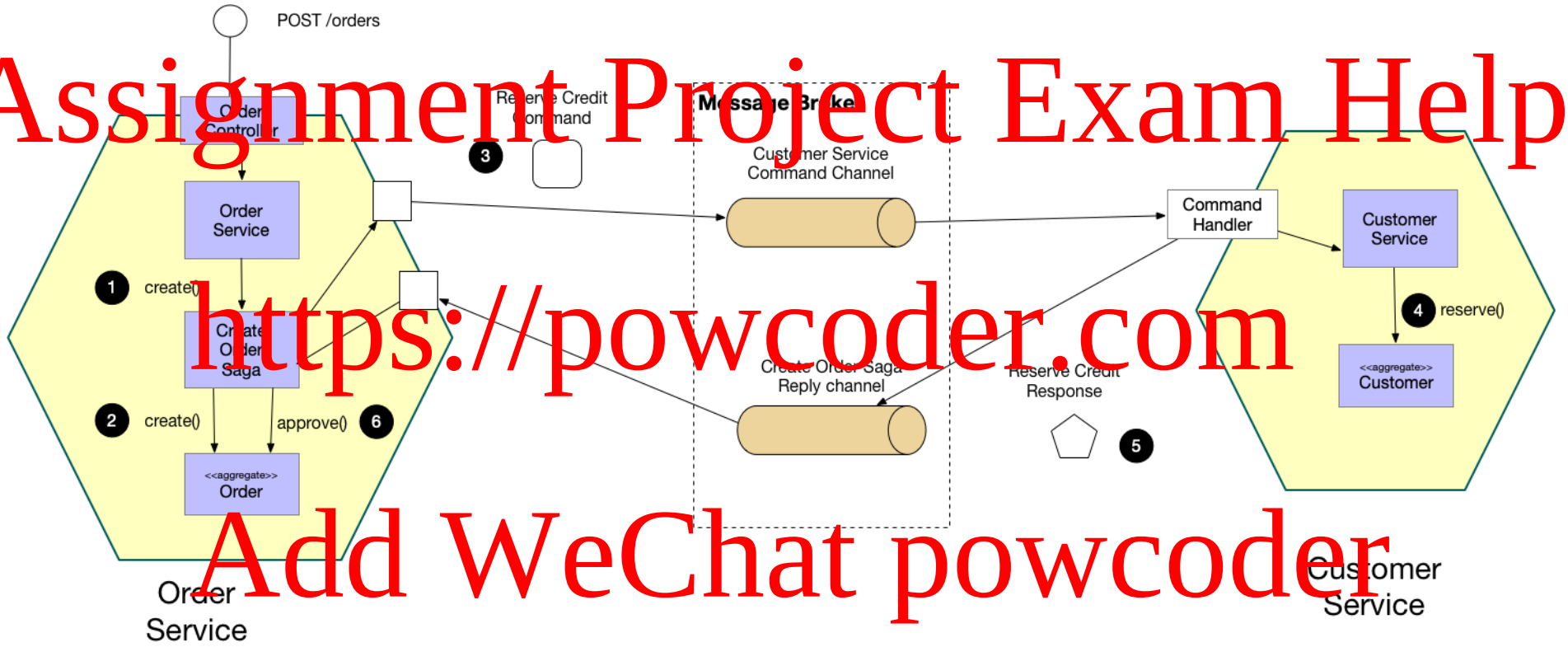


Figure: Orchestration-based saga [Ric20]

Table of Contents

1 Last Week
Quiz 3 Review

2 Architectural Patterns
Introduction to Architectural Patterns
Layered Pattern
Multi-Tier Pattern
Pipe and Filter
Publish-Subscribe
Saga Pattern
Command-Query Responsibility Separation (CQRS)
Event Sourcing
UI Patterns (MVC, MVP, MVVM)
Model-View-Controller
Model-View-Presenter
Model-View-ViewModel
Peer-to-Peer
Map-Reduce
Reflection

3 Wrap-Up

Command-Query Responsibility Separation (CQRS)

- *Context*: An application which is reading from and writing to a data store
- *Problem*: Mismatch, performance, or other issues when using the same representation of data for both reading and writing
- *Solution*: Implement the *CQRS pattern*

CQRS Background

- The mainstream approach people use for interacting with an information system is to treat it as a CRUD datastore.
  - *C*reate records
  - *R*ead records
  - *U*pdate records
  - *D*elete records
- As our needs become more sophisticated we steadily move away from that model
- As this occurs we begin to see multiple representations of information

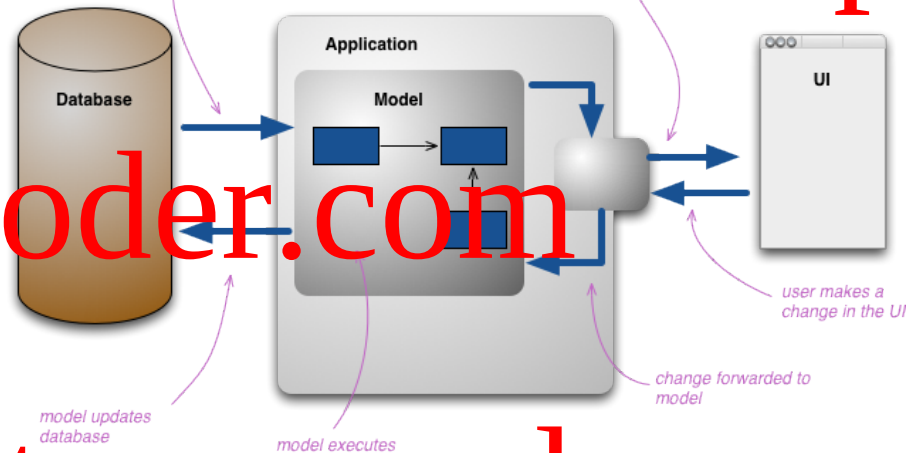


Figure: Single model for both reading and writing data [Fow20]

CQRS Pattern

- CQRS splits the conceptual model into separate models for update and display, which it refers to as *Command* and *Query* respectively
- *Commands* should be task based, rather than data centric. ("Book hotel room", not "set ReservationStatus to Reserved").
- Commands may be placed on a queue for asynchronous processing, rather than being processed synchronously.
- *Queries* never modify the database. A query returns a DTO that does not encapsulate any domain knowledge.

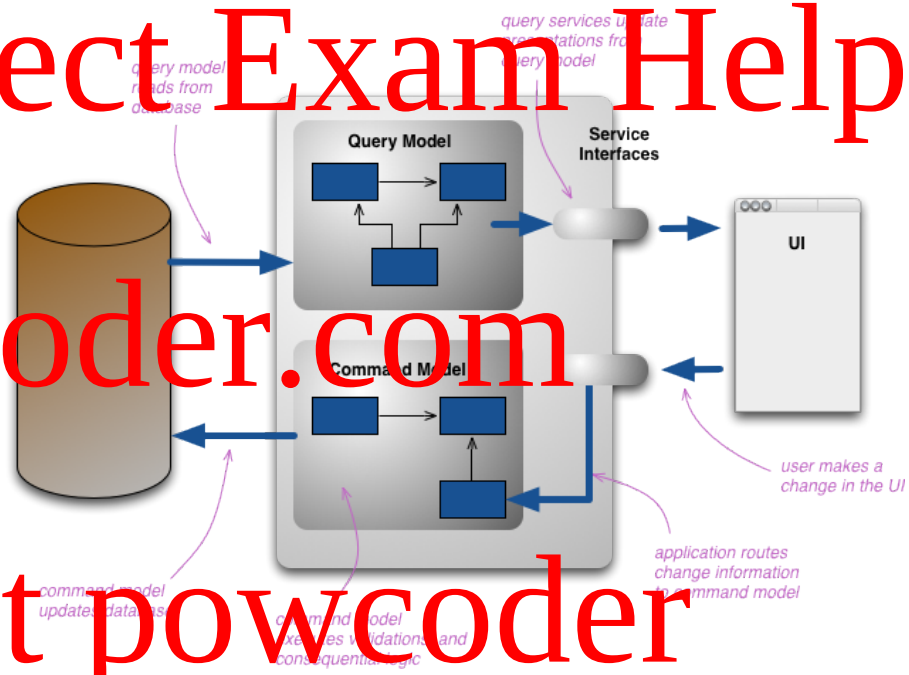


Figure: CQRS model for reading and writing data [Fow20]

CQRS Example

```
public interface IOrderCommandService {
    void Ship(OrderId orderId);
    void ChangeOrderShipmentAddress(OrderId orderId,
    ↪   Address newAddress);
    void CreateOrder(OrderCreateCommand order);
    void ChangeOrderPaymentMethod(OrderId orderId,
    ↪   PaymentMethod paymentMethod);
}

public interface IOrderQueryService {
    Order GetOrder(OrderId orderId);
}
```

CQRS Advantages

- *Independent scaling*: Can scale the read and write workloads independently
- *Optimized data schemas*: The read side can use a schema that is optimized for queries, while the write side uses a schema that is optimized for updates.
- *Security*: It's easier to ensure that only the right domain entities are performing writes on the data.
- *Separation of concerns*: Segregating the read and write sides can result in models that are more maintainable and flexible. Most of the complex business logic goes into the write model. The read model can be relatively simple.
- *Simpler queries*: By storing a materialized view in the read database, the application can avoid complex joins when querying.

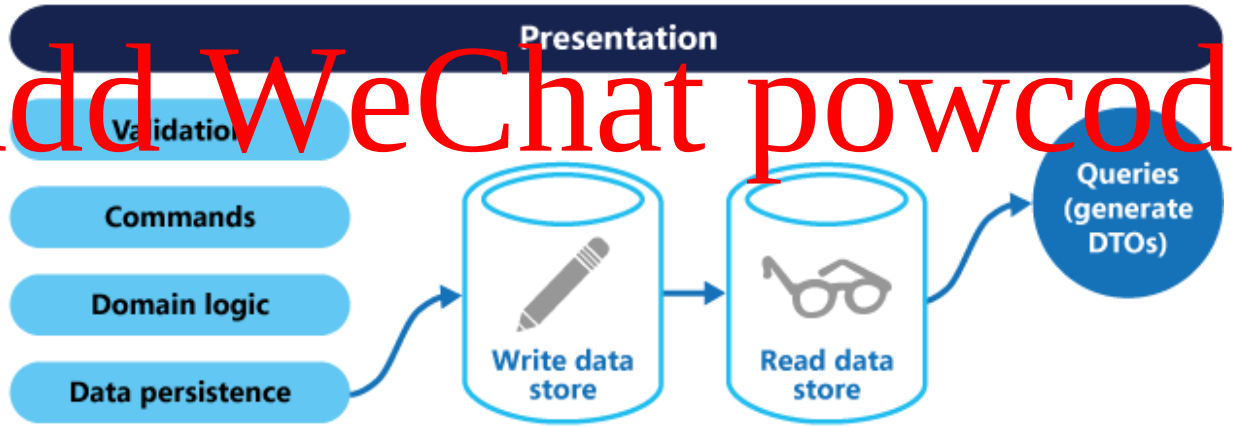


Figure: Separate read and write stores [Mic20]

CQRS Disadvantages

- *Complexity*: The basic idea of CQRS is simple. But it can lead to a more complex application design, especially if they include the Event Sourcing pattern.
- *Messaging*: Although CQRS does not require messaging, it's common to use messaging to process commands and publish update events. In that case, the application must handle message failures or duplicate messages.

- *Eventual consistency*: If you separate the read and write databases, the read data may be stale. The read model store must be updated to reflect changes to the write model store, and it can be difficult to detect when a user has issued a request based on stale read data.

Table of Contents

1 Last Week
Quiz 3 Review
Command-Query Responsibility Separation (CQRS)
Event Sourcing
UI Patterns (MVC, MVP, MVVM)
Model-View-Controller
Model-View-Presenter
Model-View-ViewModel
Peer-to-Peer
Map-Reduce
Reflection
2 Architectural Patterns
Introduction to Architectural Patterns
Layered Pattern
Multi-Tier Pattern
Pipe and Filter
Publish-Subscribe
Saga Pattern
3 Wrap-Up

Event Sourcing Pattern

- *Context*: An application which is reading from and writing to a data store
- *Problem*: Issues with writes such as performance slowdowns, data update conflicts, and loss of history
- *Solution*: Implement the *event sourcing pattern*

Event Sourcing Pattern

- Approach handling changes to data by modelling them as a sequence of events, each of which is recorded in an append-only store
- Persist the events to an *event store* that acts as the *system of record* (the authoritative data source) about the current state of the data
- Allow applications to read the history of events and use it to materialize the current state of an entity by playing back and consuming all the events related to that entity

[Mic17]

Event Sourcing Diagram

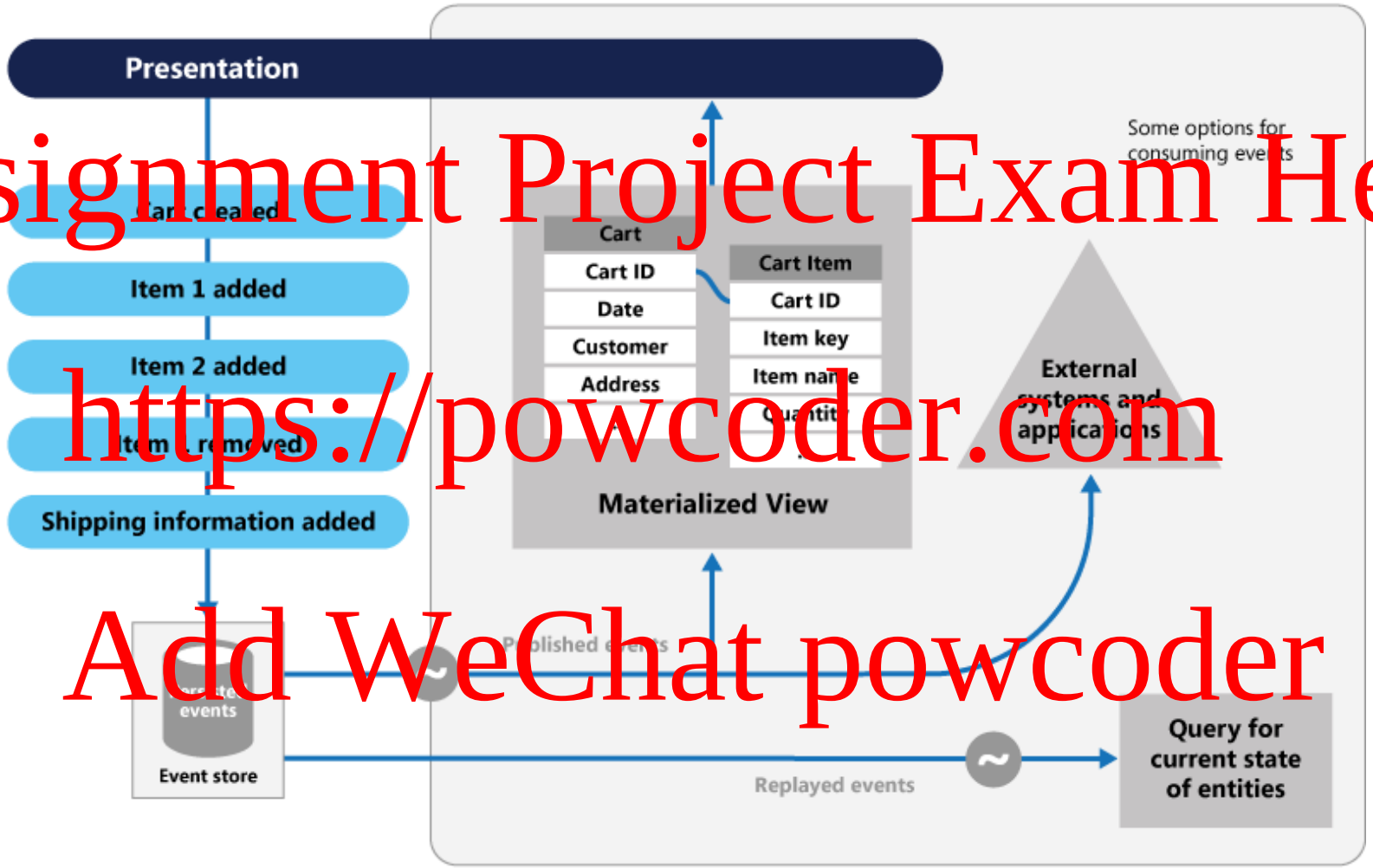


Figure: Event Sourcing [Mic17]

Event Sourcing Example

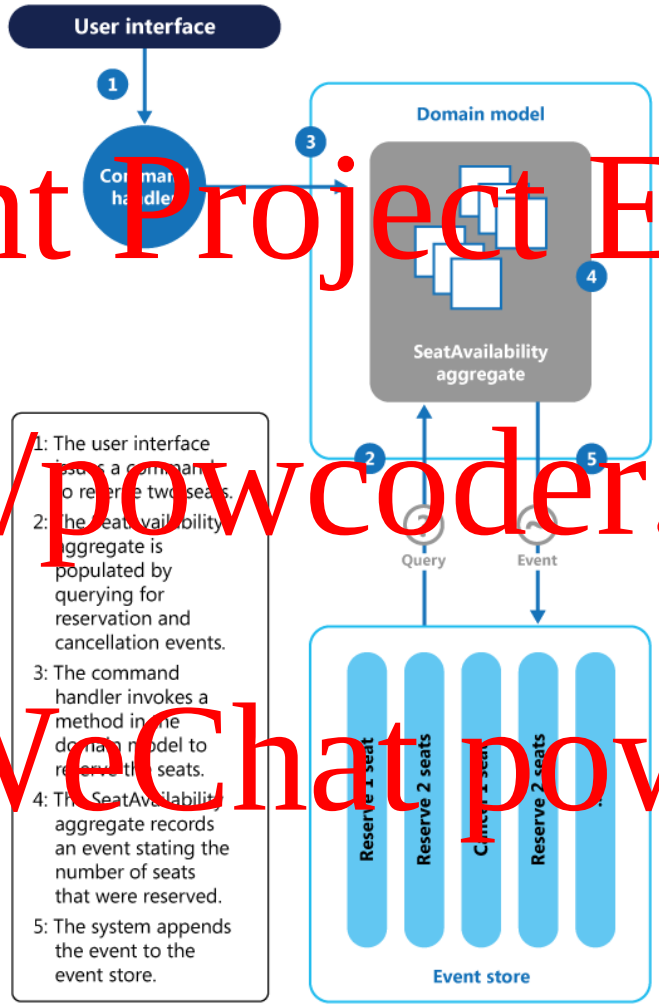


Figure: Conference Management System Event Sourcing Example [Mic17]

Event Sourcing Advantages

- Events are immutable and can be stored using an append-only operation. The user interface, workflow, or process that initiated an event can continue, and tasks that handle the events can run in the background. This, combined with the fact that there's no contention during the processing of transactions, can vastly improve performance and scalability for applications, especially for the presentation level or user interface.
- Events are simple objects that describe some action that occurred, together with any associated data required to describe the action represented by the event. Events don't directly update a data store. They're simply recorded for handling at the appropriate time. This can simplify implementation and management.
- Event sourcing can help prevent concurrent updates from causing conflicts because it avoids the requirement to directly update objects in the data store
- The append-only storage of events provides an audit trail that can be used to monitor actions taken against a data store, regenerate the current state as materialized views or projections by replaying the events at any time, and assist in testing and debugging the system
- The event store raises events, and tasks perform operations in response to those events. This decoupling of the tasks from the events provides flexibility and extensibility.

Event Sourcing Disadvantages

- *Eventual consistency*: An event sourcing based system is naturally eventually consistent, which means you can't guarantee the data in your projections is immediately up-to-date
- *Event upgrading*: Events will change shape over time, and this can be a bit tricky to handle if you don't plan for it in advance
- *Developers need deprogramming*: Most developers immediately jump to thinking in tables, and need to unlearn all the bad habits they've picked up

[O'S17]

Table of Contents

1. Last Week
   - Quiz 3 Review
2. Architectural Patterns
   - Introduction to Architectural Patterns
   - Layered Pattern
   - Multi-Tier Pattern
   - Pipe and Filter
   - Publish-Subscribe
   - Saga Pattern
   - Command-Query Responsibility Separation (CQRS)
   - Event Sourcing
   - UI Patterns (MVC, MVP, MVVM)
     - Model-View-Controller
     - Model-View-Presenter
     - Model-View-ViewModel
   - Peer-to-Peer
   - Map-Reduce
   - Reflection
3. Wrap-Up

UI Development

- In most UI development, there are three copies of the data involved:
  1. *Record state* – the data that lives in the database
  2. *Session state* – a copy of the data that lives in-memory within the application. This data may diverge from record state as users make changes before they commit them.
  3. *Screen state* – a copy of the data that lives in the GUI components themselves.

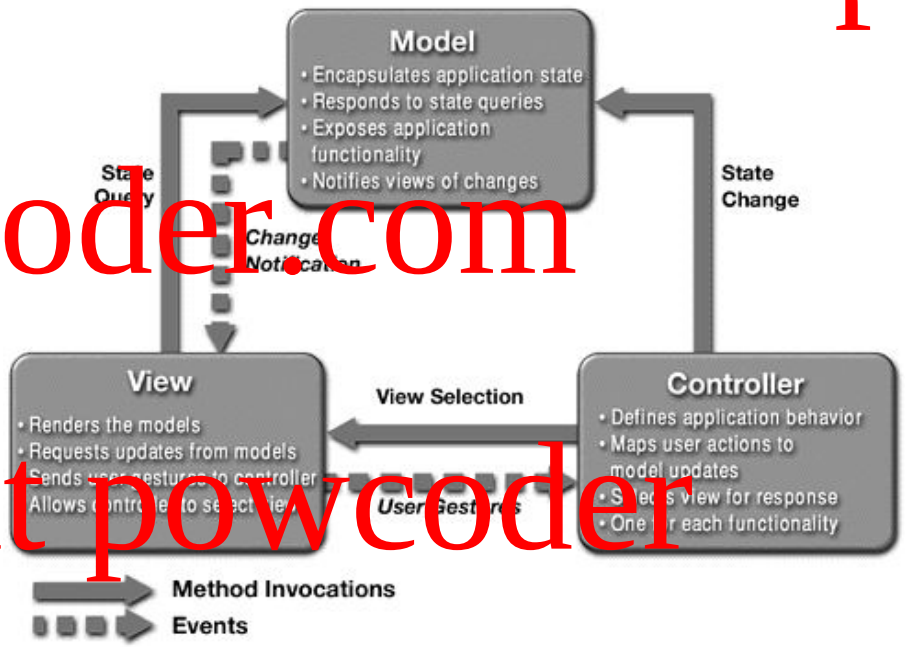
- UI development is deeply concerned with how to keep screen state, session state, and record state synchronized.

- There are many different techniques to solve this problem. One of the most popular is the *model-view-controller pattern*.

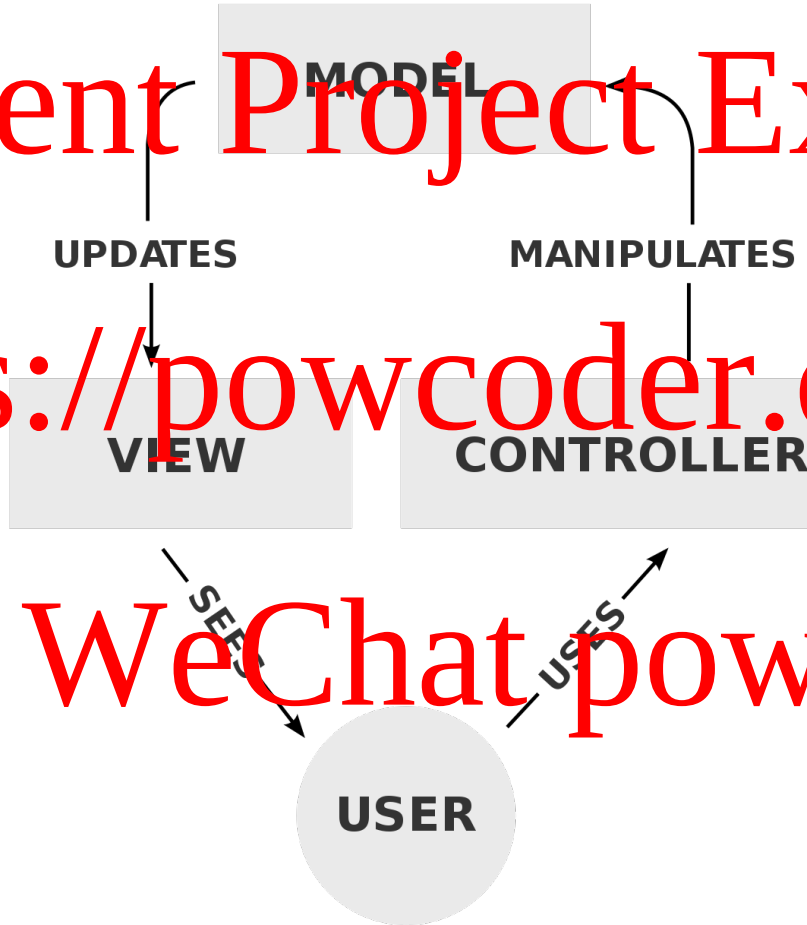
- *Context* – Writing user interface code, which is typically the most frequently modified portion of an interactive application
- *Problem* – How do keep user interface functionality separate from application functionality (*separated presentation*)? How to support multiple views on the same underlying application data & logic?

Model-View-Controller Pattern

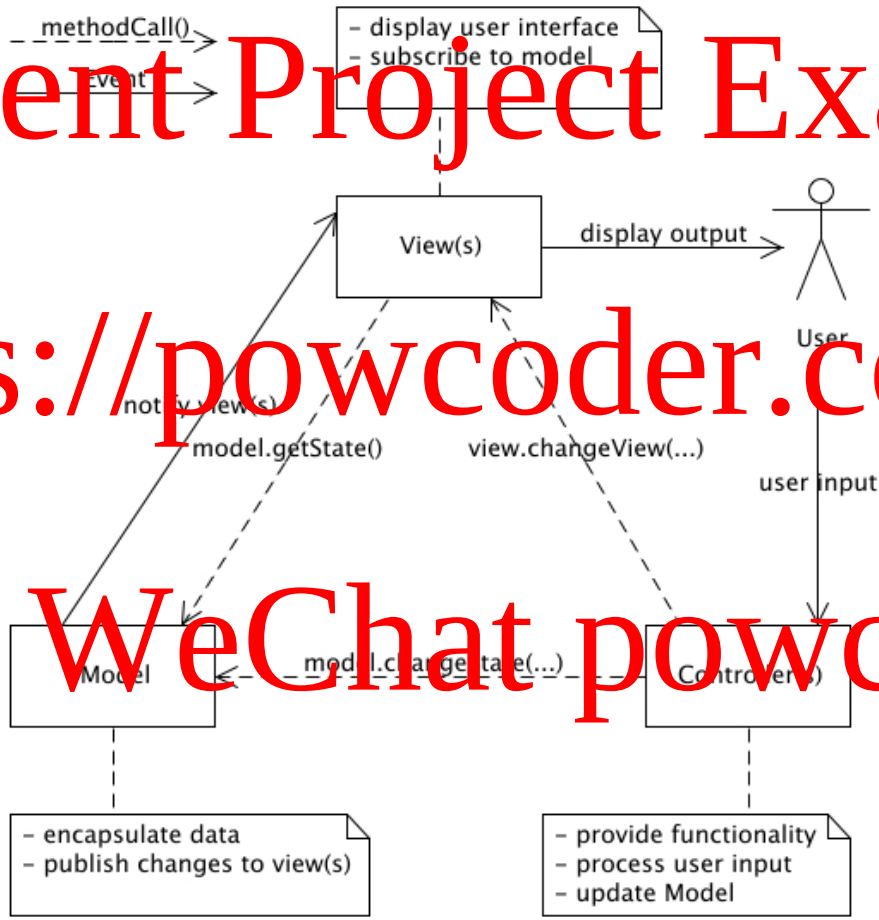
- Probably the most common UI architecture
- Decompose the application functionality into three types of components:
  1. *Model* – the application's state and functionality
  2. *View* – displays some portion of the underlying data and interacts with the user
  3. *Controller* – maps user actions to model updates
- Notice the importance of a change notification system



Model-View-Controller Alternative Diagram 1



Model-View-Controller Alternative Diagram 2



Model-View-Controller Pattern

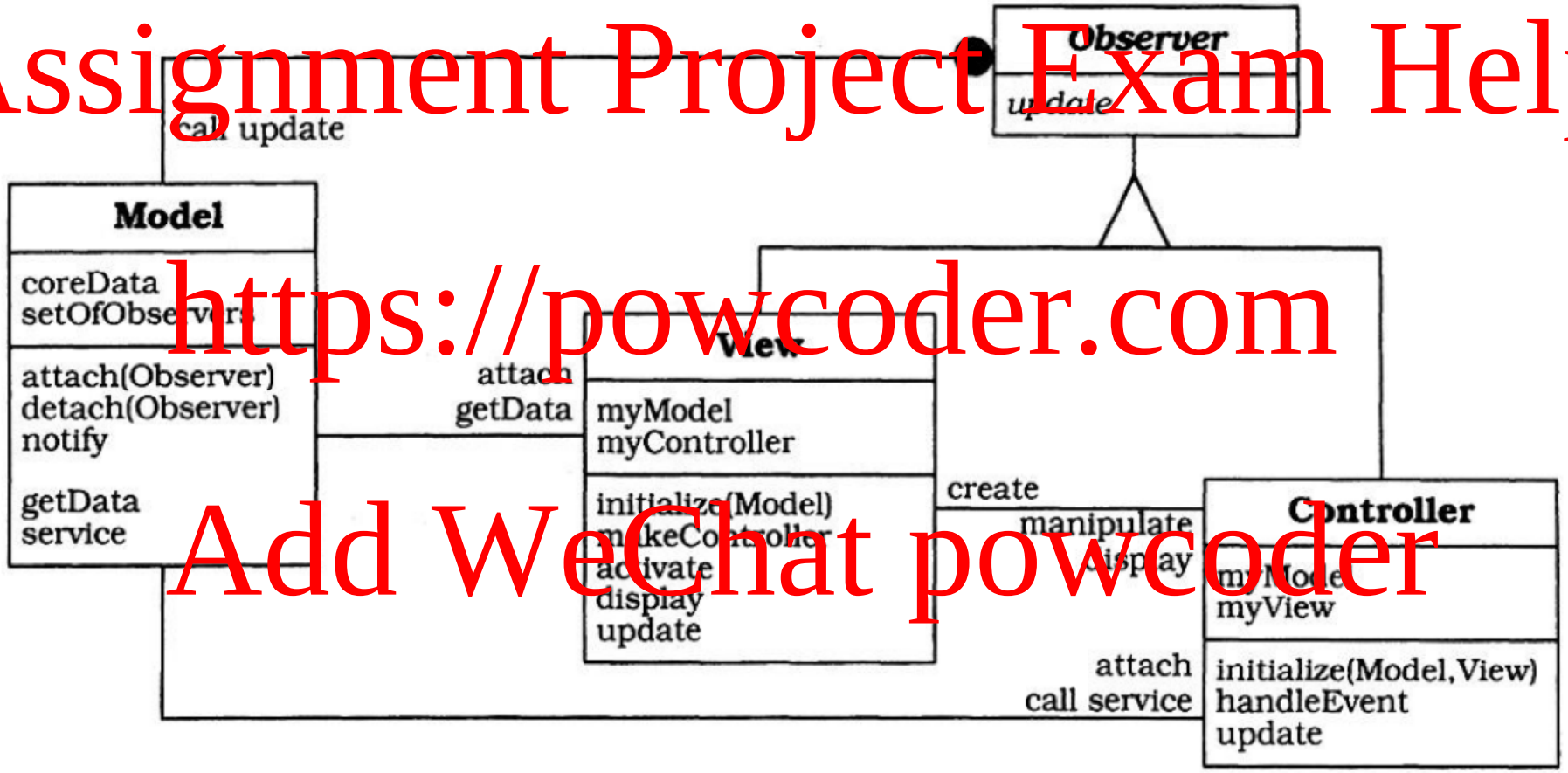
- The separation of the model from view and controller components allows multiple views of the same model.
- The user changes the model via the controller of one view, all other views dependent on this data should reflect the changes.
- The model therefore notifies all views whenever its data changes.
- The views in turn retrieve new data from the model and update the displayed information.
- Note that controllers never set the values in the view; they always just update the model and then rely on events to propagate the changes to the views (contrast with *two-way data binding*)
  - This implies that a controller does not know what widgets will change when the user manipulates a particular widget – this is very useful to support multiple screens showing different views on the same data

MVC Misconceptions

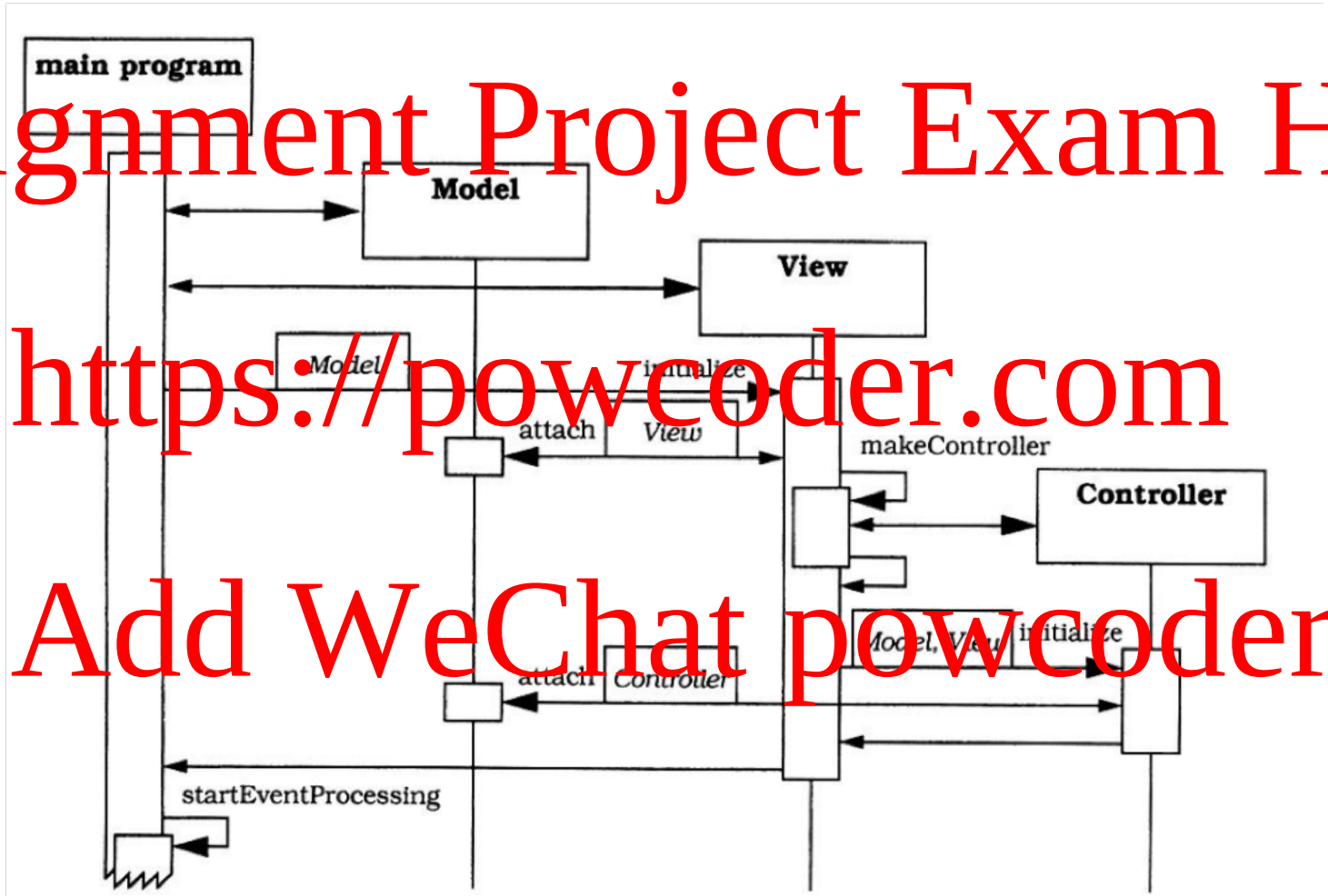
- One common misconception about the relationship between the MVC components is that the purpose of the Controller is to separate the View from the Model.
- While the MVC pattern does decouple the application's domain layer from presentation concerns, this is achieved through the Observer Pattern, not through the Controller.
- The Controller was conceived as a mediator between the end user and the application, not between the View and the Model.

[Gre07]

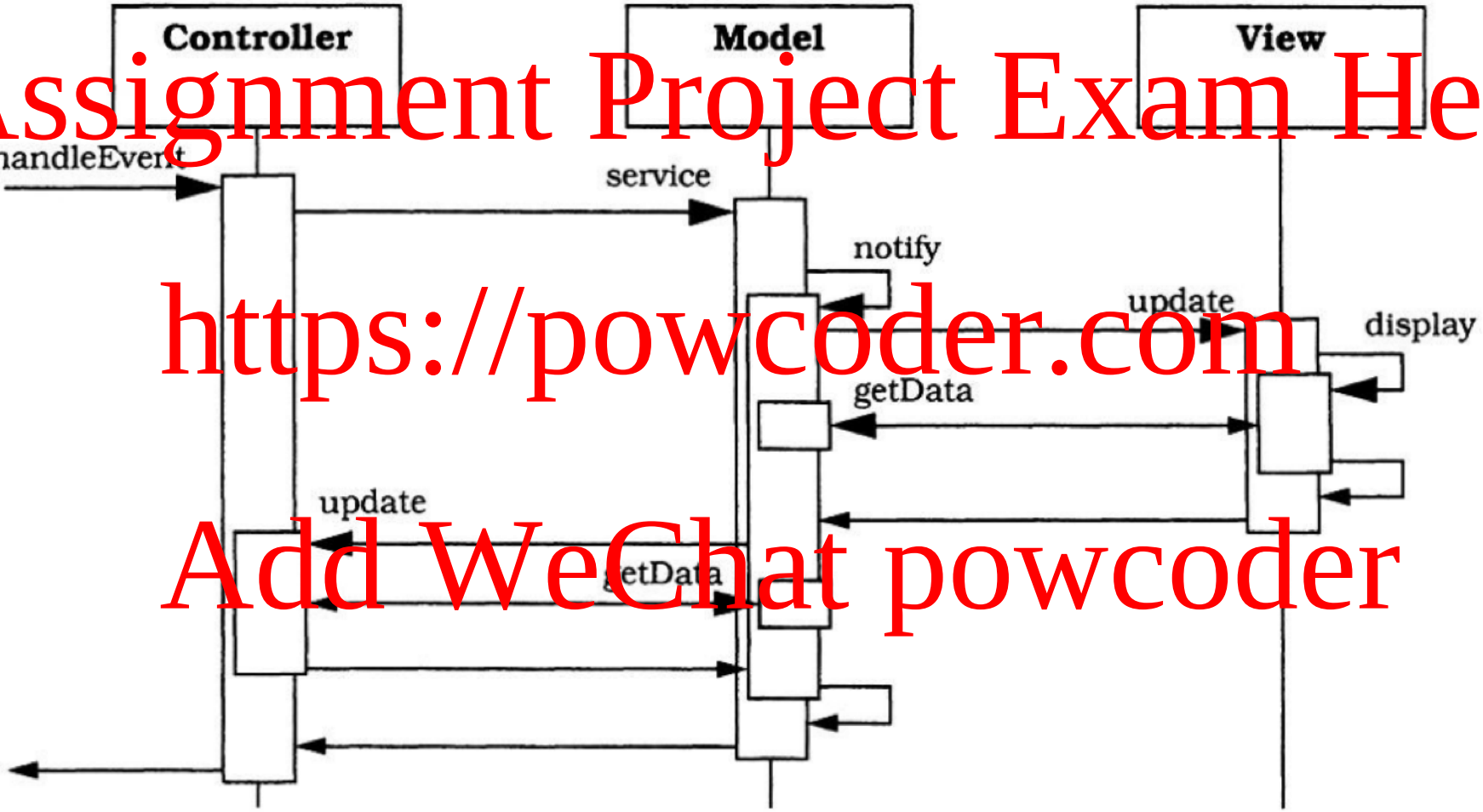
MVC Class Diagram



MVC – Initialization



MVC – UI Event Processing



MVC – How to Implement From Scratch

1. Separate human-computer interaction from core functionality.
   - Design the model to encapsulate the data and functionality needed for the core.
   - Provide functions for accessing the data to be displayed.
   - Decide which parts of the model's functionality are to be exposed to the user via the controller, and add a corresponding interface to the model.
2. Implement change-propagation mechanism. Assign role of publisher to the model.
3. Design and implement views
   - Design the appearance of each view, and also
   - Specify and implement a draw procedure to display the view on the screen (which acquires data from model).
4. Design and implement the controllers.
   - For each view, specify the behavior of the system in response to user actions (treated as an event).
   - A controller receives and interprets these events using a dedicated procedure (in non-trivial cases this can depend on the state of the model).

MVC – How to Implement (Continued)

5. Design and implement the view-controller relationship.
   - A view typically creates its associated controller during its initialization.
   - When you build a class hierarchy of views and controllers, apply the Factory Method design pattern [GHJV95] and define a method makeController() in the view classes.
6. Implement the set-up of MVC.
   - The set-up code first initializes the model, then creates and initializes the views.
   - After initialization, event processing is started.
   - Because the model should remain independent of specific views and controllers, this set-up code should be placed externally, for example, in a main program.
7. Dynamic Views
   - If the application allows dynamic opening and closing of views, it is a good idea to provide a component for managing open views.

MVC – How to Implement (Continued)

8. "Pluggable" controllers.
   - The separation of control aspects from views supports the combination of different controllers with a view.
   - Implement different modes of operation, such as casual user versus expert.
   - Construct read-only views using a controller that ignores any input.
   - Integrate new input and output devices.

9. Further decoupling from system dependencies.
   - Building a framework with an elaborate collection of view and controller classes is expensive.
   - Make these classes platform independent by providing the system with another level of indirection between it and the underlying platform by applying the Bridge pattern.

MVC – When to Use?

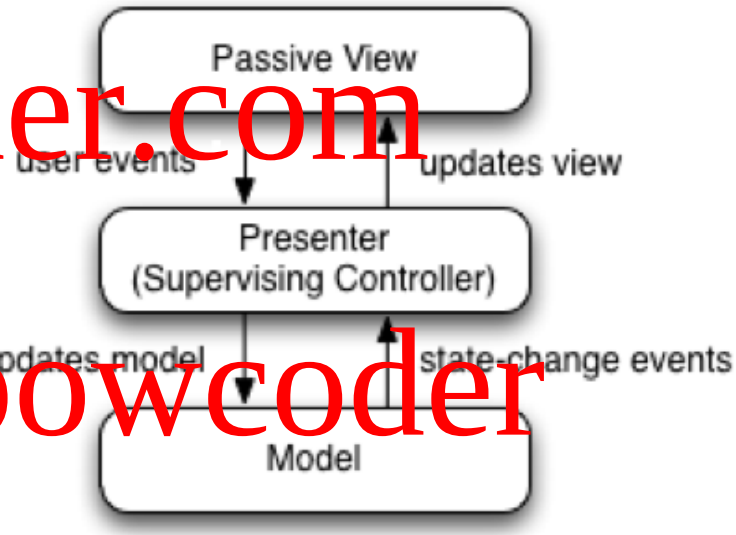
- The same information is presented differently in different windows, for example, in a bar or pie chart.
- The display and behavior of the application must reflect data manipulations immediately.
- Changes to the user interface should be easy, and even possible at run-time.
- Supporting different 'look and feel' standards or porting the user interface should not affect code in the core of the application.

MVC – Disadvantages

- Increased complexity
- Potential for excessive number of updates
- Hard to tell what will happen from reading the code
- Where to put view state like "which city is selected in a city dropdown?"
- Where to put view logic like "only enable save buttons if data has changed?" or "color values over 5% as red"?
  - For a lot of the above, you might say "the model", but the general idea of MVC is that the model should be ignorant of display concerns. This lead to the creation of *presentation models*.

Model-View-Presenter Pattern

- The *model* is an interface defining the data to be displayed or otherwise acted upon in the user interface.
- The *view* is a passive interface that displays data (the model) and routes user commands (events) to the presenter to act upon that data.
- The *presenter* acts upon the model and the view. It retrieves data from repositories (the model), and formats it for display in the view.



[Wik20a]

Taligent Model-View-Presenter Pattern

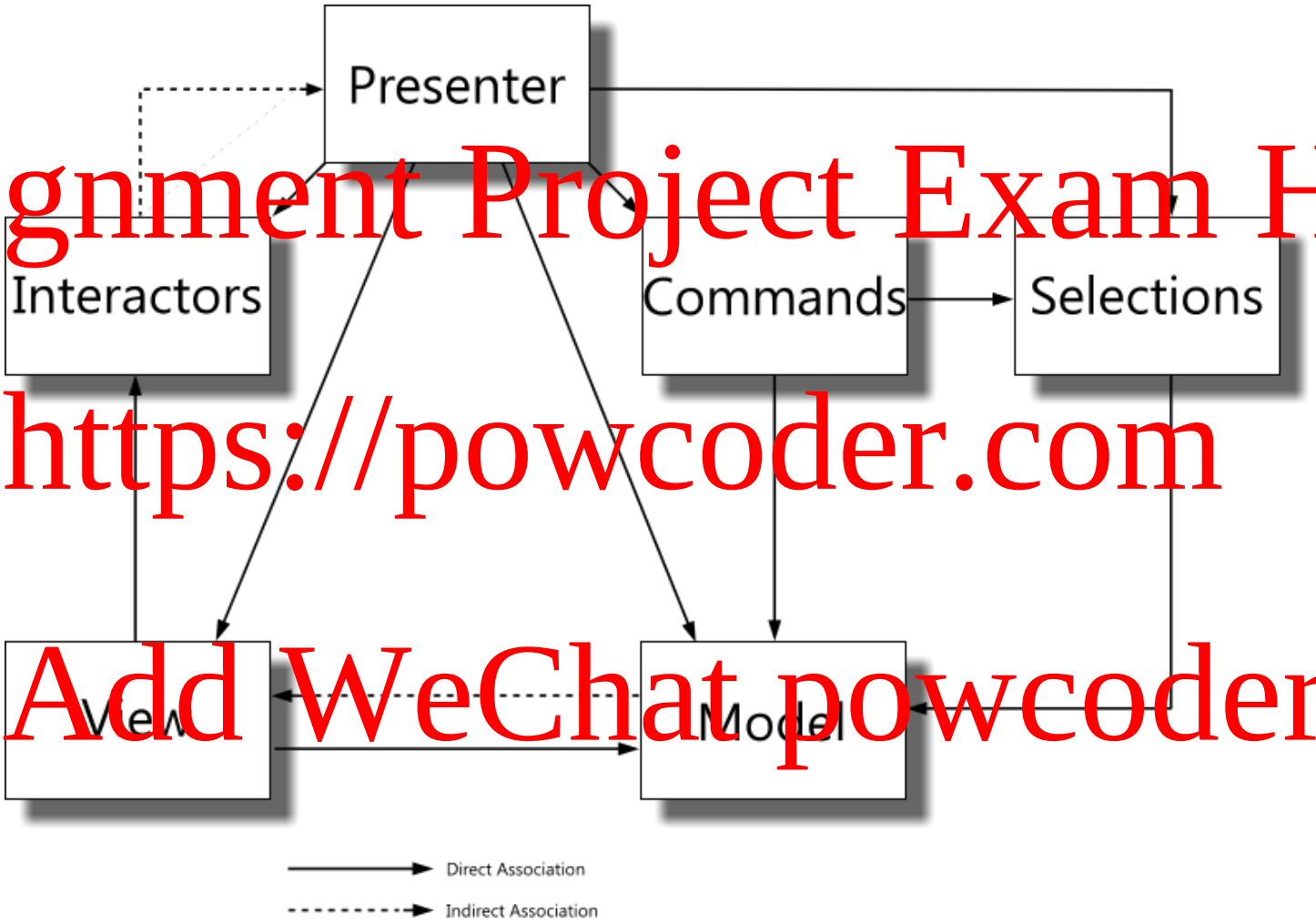


Figure: Taligent Model-View-Presenter Pattern [Gre07]

MVP vs. MVC

- Within the MVP pattern, the role of intercepting the user's input was moved to the View.
- Within the original Model-View-Controller pattern, the primary purpose of the Controller was to intercept user input. The Controller's role of updating the Model was largely a byproduct of this function rather than an inherent part of its purpose.
- Conversely, within the Model-View-Presenter pattern, the primary purpose of the Presenter was to update the Model. The Presenter's role of intercepting events delegated by the View was largely a byproduct of this function rather than an inherent part of its purpose.

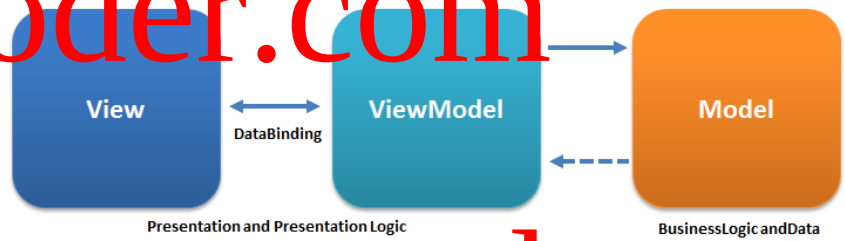
[Gre07]

MVP Advantages

- Provides a natural place for view state and view logic
- Designed to facilitate automated unit testing and improve the separation of concerns in presentation logic
- Many MVC frameworks might actually be MVP in disguise
- More on MVC-MVVM:
  https://www.martinfowler.com/eaaDev/uiArchs.html

Model-View-ViewModel Pattern

- Also known as Model-View-Binder
- *Model* – unchanged
- *View* – same as MVP
- *ViewModel* – an abstraction of the view exposing public properties and commands. Instead of the controller of the MVC pattern, or the presenter of the MVP pattern, MVVM has a binder, which automates communication between the view and its bound properties in the view model.



MVVM WPF Example

```
<UserControl ...>
  <UserControl.InputBindings>
    <KeyBinding Key="Enter" Command="{Binding ConvertTextCommand}"/>
  </UserControl.InputBindings>

  <StackPanel Height="336">
    <Label Foreground="Blue" Margin="5,5,5,0">Text To Convert</Label>
    <TextBox Text="{Binding SomeText, UpdateSourceTrigger=
    PropertyChanged}" Margin="5"/>
    <Label Foreground="Blue" Margin="5,5,5,0">History</Label>
    <ListBox ItemsSource="{Binding History}" Height="200" Margin="5"/>
    <Button Command="{Binding ConvertTextCommand}" Margin="5">Convert</
    Button>
  </StackPanel>
</UserControl>
```

MVVM Advantages

- Removes virtually all GUI code ("code-behind") from the view layer, using two-way data binding instead

MVVM Disadvantages

- MVVM (and MVC, MVP) is overkill for simple UI operations
- For larger applications, generalizing the view model becomes more difficult
- Data binding in very large applications can result in considerable memory consumption

UI Architecture Wrap-Up

- There are other UI architectures, such as:
  - Presentation-Abstraction-Control (PAC)
  - Model-View-Adapter
- UI architectures are all built around the idea of *separated presentation*
- The main difference is how they perform *separation of concerns* and on which architectural qualities they focus (e.g. MVP emphasizes *testability*)

Table of Contents

1 Last Week
Quiz 3 Review

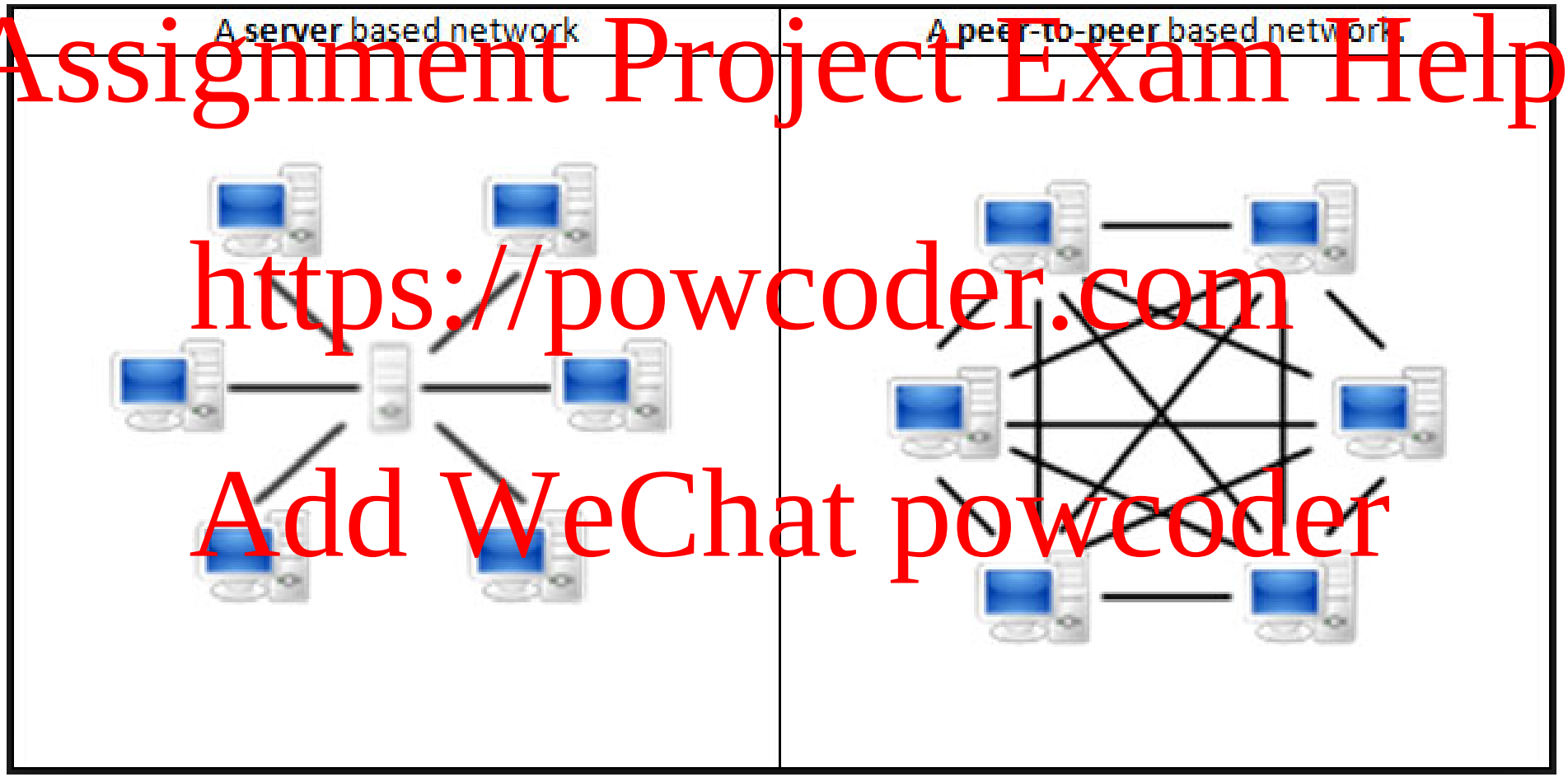
2 Architectural Patterns
Introduction to Architectural Patterns
Layered Pattern
Multi-Tier Pattern
Pipe and Filter
Publish-Subscribe
Saga Pattern
Command-Query Responsibility Separation (CQRS)
Event Sourcing
UI Patterns (MVC, MVP, MVVM)
Model-View-Controller
Model-View-Presenter
Model-View-ViewModel
Peer-to-Peer
Map-Reduce
Reflection

3 Wrap-Up

Peer-to-Peer Pattern

- *Context*: Equally-important distributed computational entities need to cooperate and collaborate to provide a service
- *Problem*: How can a set of "equal" distributed computational entities be connected to each other via a common protocol so that they can organize and share their services with high availability and scalability?
- *Solution*: Use the peer to peer (P2P) pattern

Peer-to-Peer Pattern



Peer-to-Peer Examples

- Skype: Uses P2P networking to implement Internet telephony
- BitTorrent: The dominant peer-to-peer file sharing protocol
  - In November 2004, BitTorrent was responsible for 25% of all Internet traffic.
  - Even Amazon S3 supports BitTorrent

- Battle.net: Used to use peer-to-peer technology for distributing games such as World of Warcraft (later removed)
- Netflix: As of 2014-2015, was researching the possibility of using P2P networking to reduce bandwidth costs

- Gnutella: The first decentralized peer-to-peer file sharing networks ever created

BitTorrent

- A small file, called a *descriptor*, contains basic information about a file and is distributed via traditional means (e.g. email, HTTP)
- The file to be distributed is divided into segments called *pieces*. Each piece is protected via a *cryptographic hash* contained in the descriptor.
- *Tracker servers* are used to keep track of where file copies reside on peer machines. Clients register with tracker servers to discover peers.
  - Alternatively, there is a distributed tracker mechanism which uses *distributed hash tables*
- BitTorrent clients download pieces in a random or *rarest-first* approach
- A BitTorrent client may connect to many different peers to download multiple pieces at the same time

Peer-to-Peer Challenges

- *Discovery*: How do you find potential peers?
  - Some architectures introduce specialized peer nodes (called supernodes) that have indexing or routing capabilities
- *Connectivity*: How do you deal with network devices that aren't externally accessible (e.g. behind firewalls or NATs)?
- *Data and Service Availability*: How do you ensure that the data & service will be available if peers can come and go at any time?
  - Likely need to maintain a minimum network size to ensure viability
  - Do you need to maintain *seed nodes*?
- *Security*: How do you protect yourself from attackers who want to poison your P2P network?
  - Content providers (e.g. HBO) have been known to deliberately poison BitTorrent networks to try to prevent piracy

Table of Contents

1 Last Week
Quiz 3 Review

2 Architectural Patterns
Introduction to Architectural Patterns
Layered Pattern
Multi-Tier Pattern
Pipe and Filter
Publish-Subscribe
Saga Pattern
Command-Query Responsibility Separation (CQRS)
Event Sourcing
UI Patterns (MVC, MVP, MVVM)
Model-View-Controller
Model-View-Presenter
Model-View-ViewModel
Peer-to-Peer
Map-Reduce
Reflection

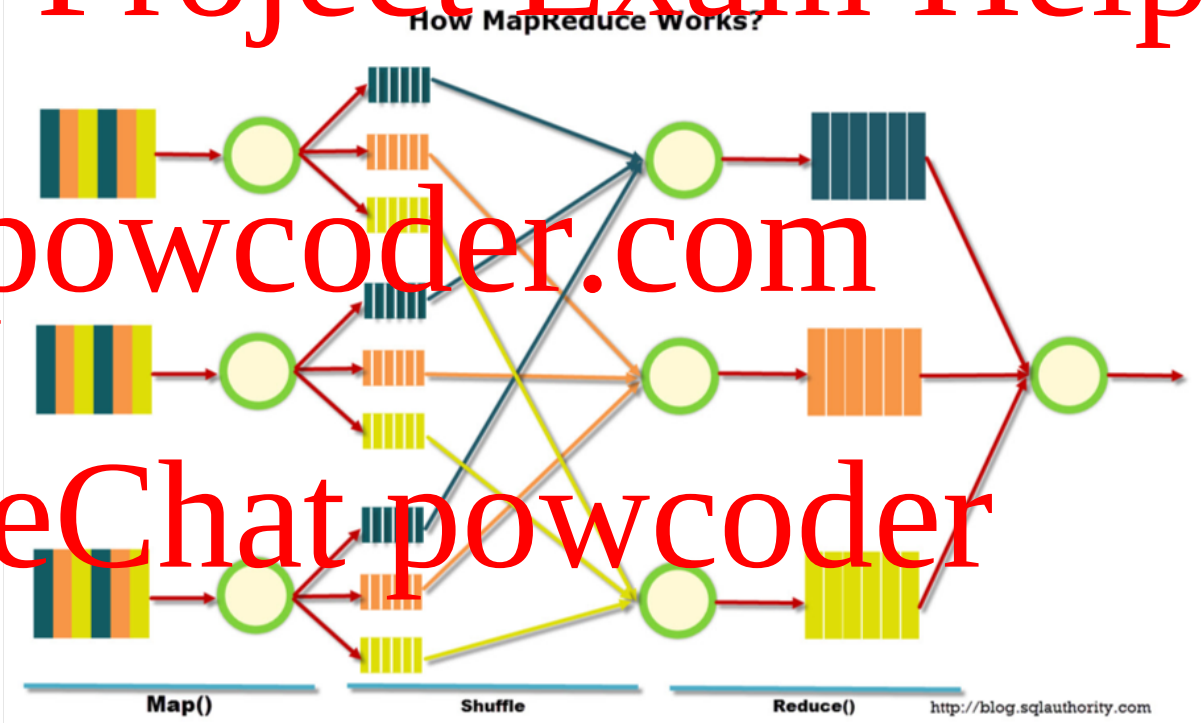
3 Wrap-Up

Map-Reduce Pattern

- *Context*: Businesses require the ability to quickly analyze enormous volumes of data they generate or access, at petabyte scale.
- *Problem*: How to efficiently apply a distributed and parallel algorithm to a large data set?
- *Solution*: Apply the map-reduce pattern.

Map-Reduce Pattern

- *Map*: Apply a single operation to a single element in a collection, potentially in parallel.
- *Shuffle*: Worker nodes redistribute data based on the output keys (produced by the "map()" function), such that all data belonging to one key is located on the same worker node.
- *Reduce*: Combine the results from map together to produce a final result.



Map-Reduce Example 1

```
function map(String name, String document):
  // name: document name
  // document: document contents
  for each word w in document:
    emit (w, 1)

function reduce(String word, Iterator partialCounts):
  // word: a word
  // partialCounts: a list of aggregated partial counts
  sum = 0
  for each pc in partialCounts:
    sum += pc
  emit (word, sum)
```

Why Map-Reduce?

- The map-reduce programming model is *trivial to parallelize* across many, many machines
  - Applying parallelization & distribution across many machines is the only way we can even think about processing a truly enormous data set
- Because the programming model is so restricted, it's fairly hard to write a bad (inefficient) map-reduce program
  - Compare that to how easy it is to write inefficient SQL
- Google famously created a very large-scale map-reduce system which powered their search index creation
  - It has since been replaced by other technologies that support streaming operation

Map-Reduce Uses

- Distributed pattern-based searching
- Distributed sorting
- Web link-graph traversal
- Web access log stats

- Inverted index construction
- Document clustering
- (more)

- Note: The book implies that map-reduce is only applicable to distributed sorting. *This is wrong!*

Map-Reduce Pattern

- Advantages:
  - Trivial to parallelize
- Disadvantages:
  - Extremely high overhead, not justified for "small" data sets
  - You must be able to divide your data set into similar sized subsets
  - Limited to batch processing
  - Restrictive, hard-to-use programming model

Map-Reduce Thoughts

- The traditional map-reduce execution frameworks are out of favor, but the concept remains. Most people have moved onto different distributed execution frameworks.
- For batch processing, Apache Spark is very popular. For real-time (streaming processing), see Spark Streaming, Apache Storm, Apache Flink, and Apache Beam for a few examples.

- The most important thing is to be able to scale your algorithm out to run on a cluster of machines so that you can handle arbitrarily large data.
  - For this to work well, the system must support moving *compute-to-data*, not *data-to-compute*

- If your data isn't that large, don't distribute!

Table of Contents

1 Last Week
Quiz 3 Review

2 Architectural Patterns
Introduction to Architectural Patterns
Layered Pattern
Multi-Tier Pattern
Pipe and Filter
Publish-Subscribe
Saga Pattern
Command-Query Responsibility Separation (CQRS)
Event Sourcing
UI Patterns (MVC, MVP, MVVM)
Model-View-Controller
Model-View-Presenter
Model-View-ViewModel
Peer-to-Peer
Map-Reduce
Reflection

3 Wrap-Up

- 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.

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

Example: Persistence

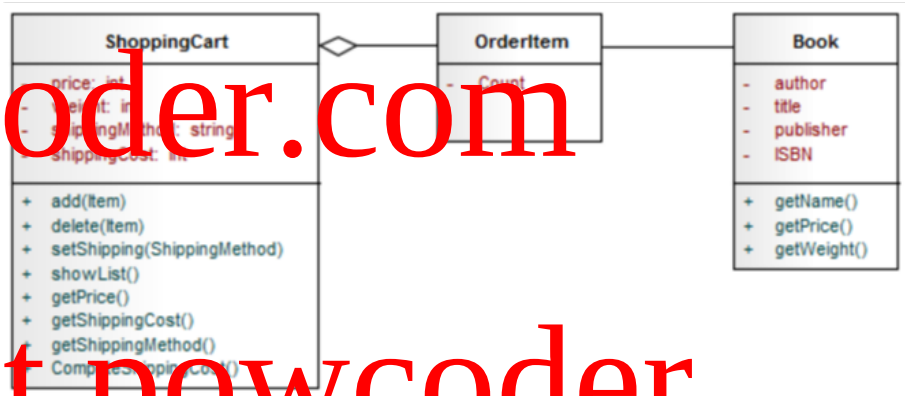
- We can create a persistence component that is able to read and store arbitrary data structures.
- A metaobject provides run-time type information – capable of understanding the internal structure of data members to be persisted.

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).

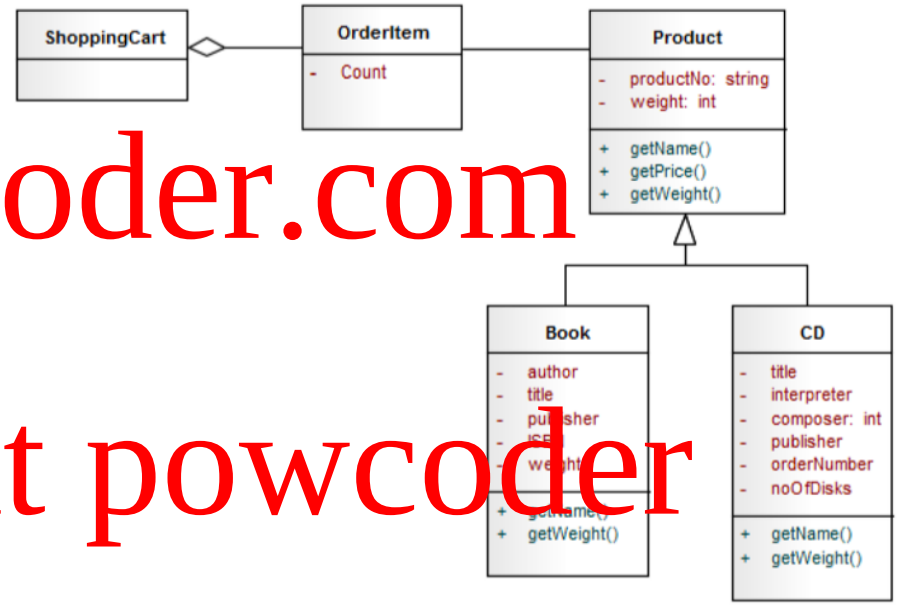
Example

- 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

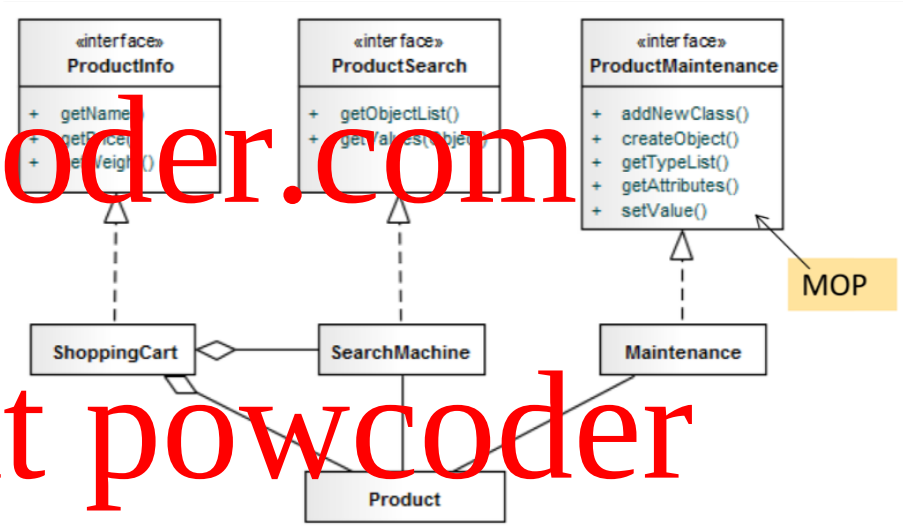
- Susan decides to start selling music accessories such as T-shirts, posters, etc. as well.
- She wants to make her system more adaptable so she can add different kinds of things with different attributes and functions at runtime.
- She realizes that she needs a new additional interface for a general 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

Reading Objects Stored from Disk

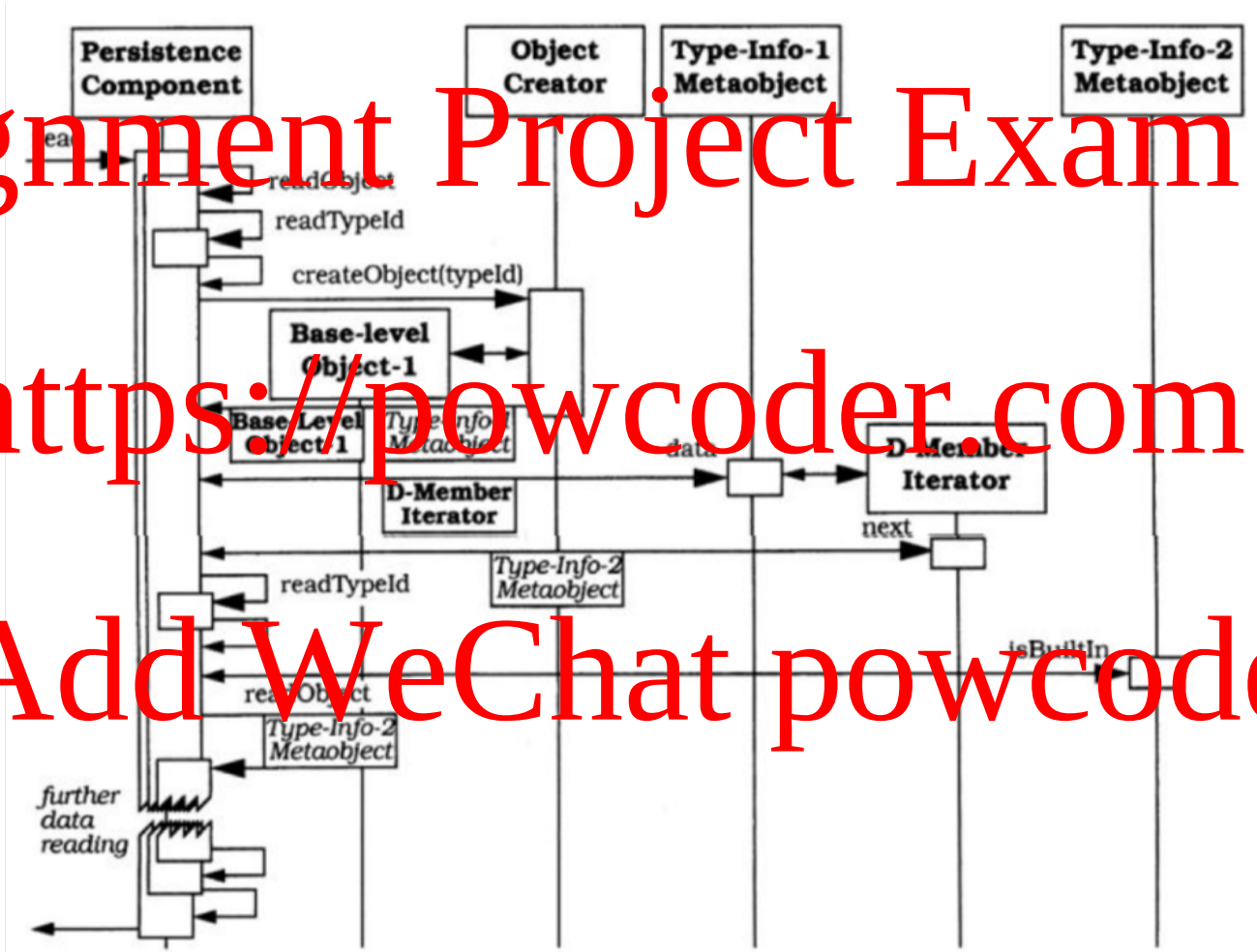


Table of Contents

1. Last Week
   - Quiz 3 Review
2. Architectural Patterns
   - Introduction to Architectural Patterns
   - Layered Pattern
   - Multi-Tier Pattern
   - Pipe and Filter
   - Publish-Subscribe
   - Saga Pattern
   - Command-Query Responsibility Separation (CQRS)
   - Event Sourcing
   - UI Patterns (MVC, MVP, MVVM)
   - Model-View-Controller
   - Model-View-Presenter
   - Model-View-ViewModel
   - Peer-to-Peer
   - Map-Reduce
   - Reflection
3. Wrap-Up

Wrap-Up

- This was Chapter 13 in SAIP, as well as numerous external sources
- Next week, we will cover chapters 15-22 in SAIP
- Homework 4 is due Thursday, November 5 at 5:30PM
- Quiz 4 will be made available next week. It will be due Thursday, November 5 at 5:30PM

References

[Fow02] Martin Fowler.
Patterns of Enterprise Application Architecture.
Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[Fow20] Martin Fowler.
Cqrs.
2020.

[Gre07] Derek Greer.
Interactive application architecture patterns.
2007.

[Mic17] Microsoft.
Event sourcing pattern.
2017.

[Mic20] Microsoft.
Command and query responsibility segregation (cqrs) pattern.
2020.

[O'S17] Barry O'Sullivan.
Event sourcing: What it is and why it's awesome.
2017.

[Ric19] Chris Richardson.
Managing data consistency in a microservice architecture using sagas - part 1.
2019.

[Ric20] Chris Richardson.
Pattern: Saga.
2020.

[Wik20a] Wikipedia.
Model-view-presenter.
2020.

[Wik20b] Wikipedia.
Publish–subscribe pattern.
2020.