SFWRENG / MECHTRON 3K04
Software Development

Today: Model-Driven Engineering

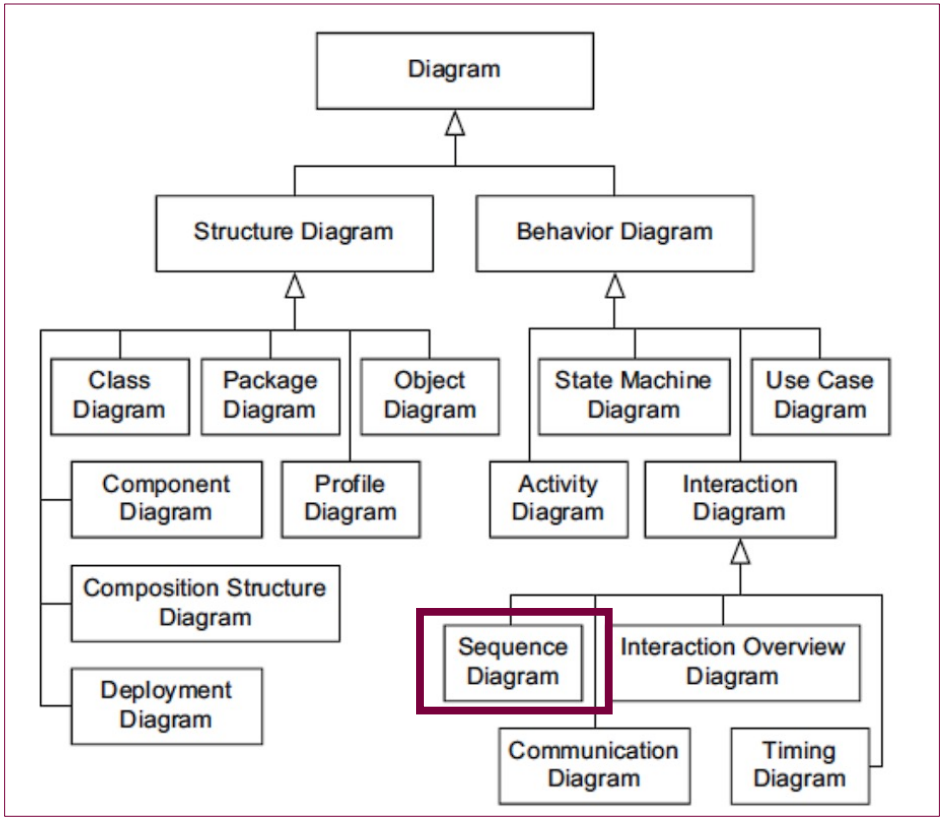
November 2nd, 2025

Dr. Angela Zavaleta Bernuy

McMaster University | Engineering
Computing & Software

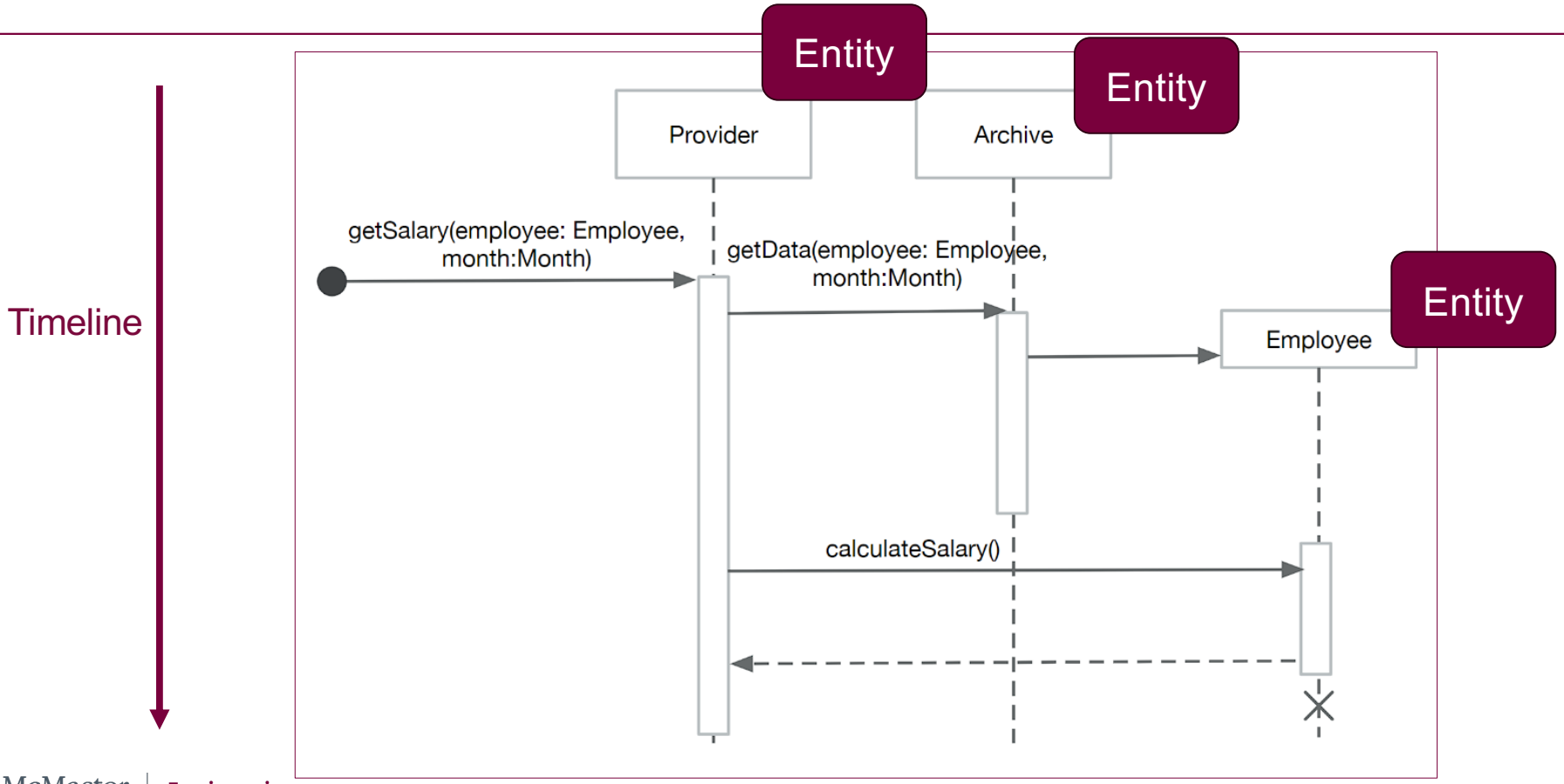
Brighter World

UML: Sequence Diagrams

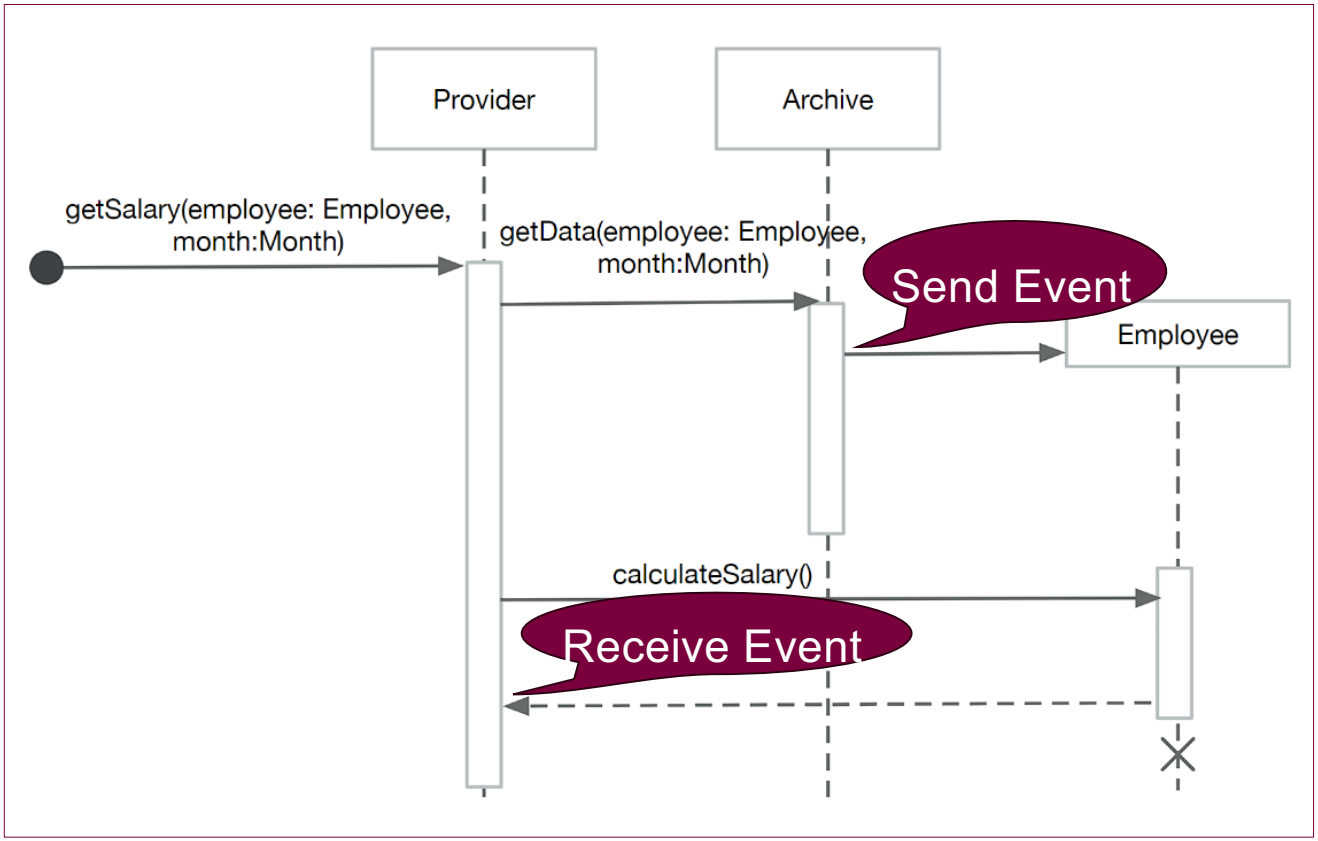


- Specialized for interactions
- Interactions can be everything from component interactions to user interactions
- Models a system by considering:
  - Entities
  - Message exchanges between entities

UML: Sequence Diagrams – Example



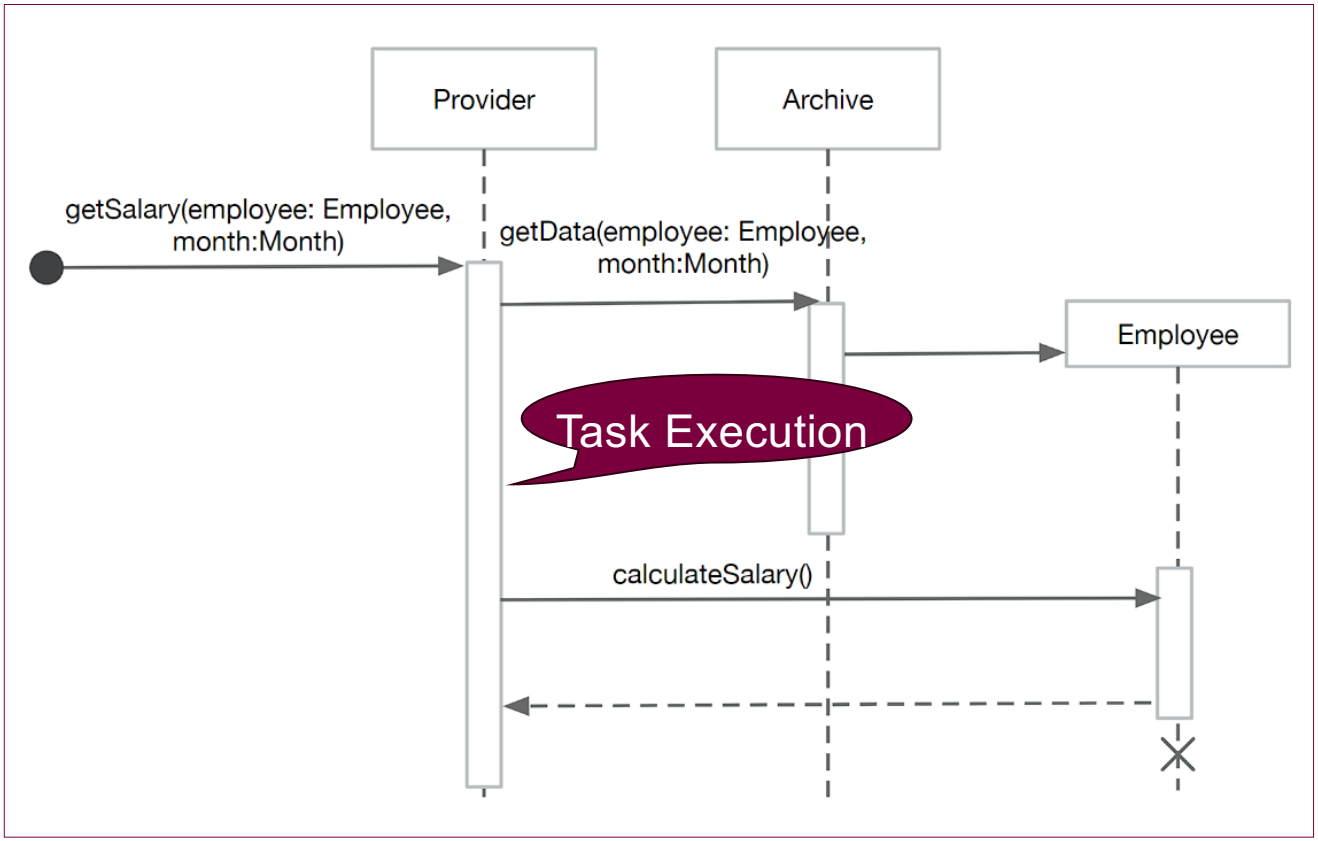
UML: Sequence Diagrams - Example



Timeline

Behaviour Interactions

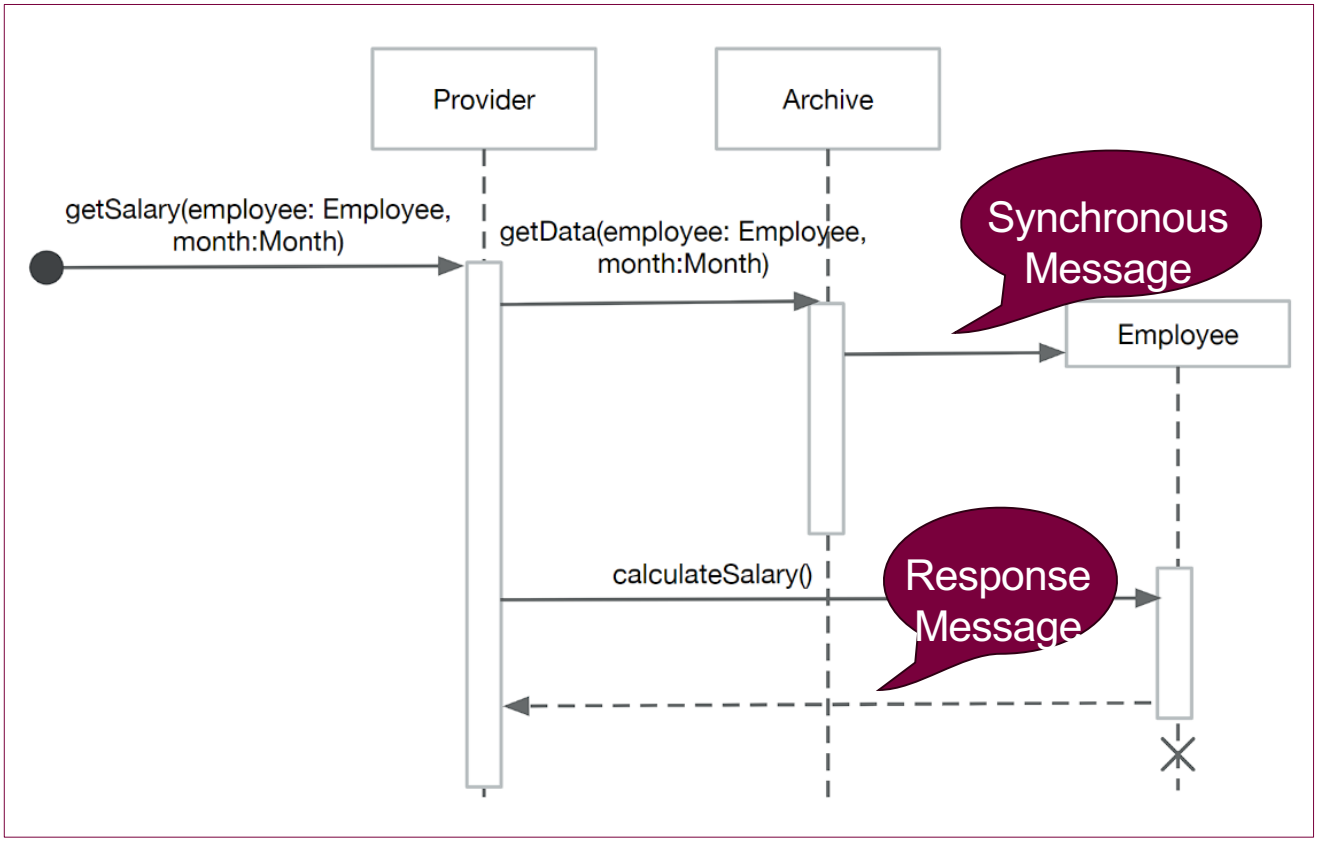
UML: Sequence Diagrams - Example



Timeline

Behaviour Interactions

UML: Sequence Diagrams - Example

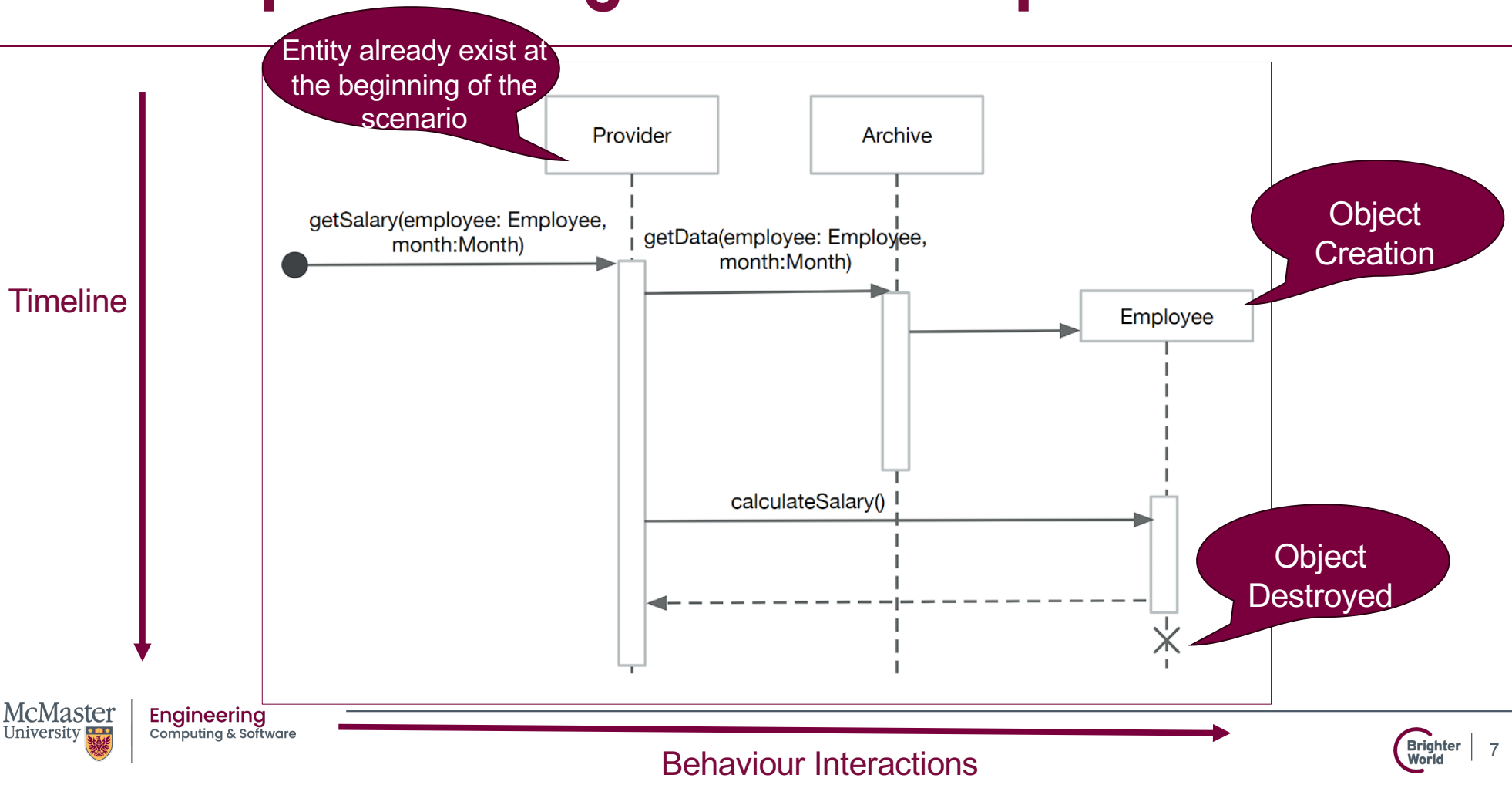


Timeline

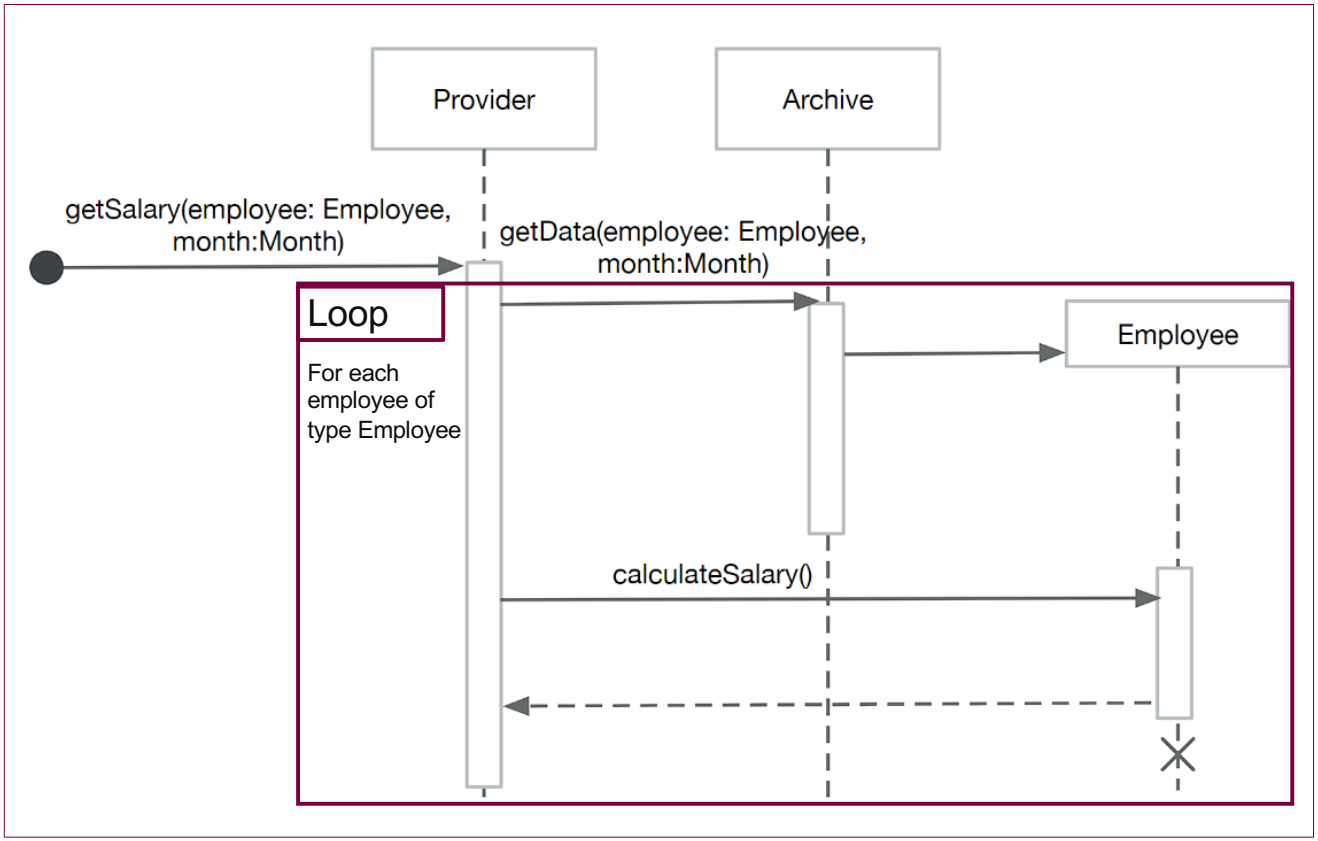
Asynchronous Message

Behaviour Interactions

UML: Sequence Diagrams - Example



UML: Sequence Diagrams - Example

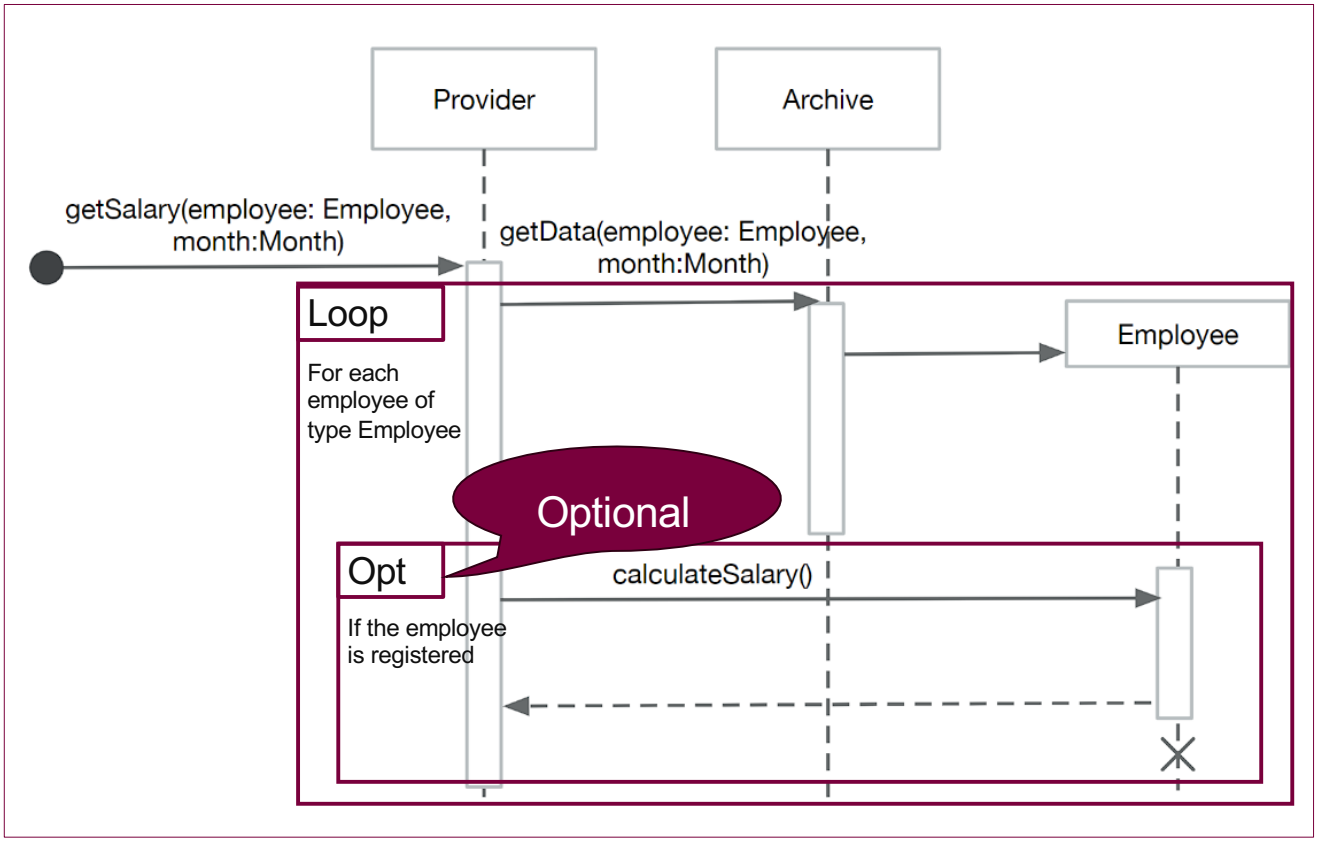


Timeline

Behaviour Interactions

UML: Sequence Diagrams - Example

Timeline



Could also have an "Alt" (Alternative). Have a second option at the bottom for those employees who are registered.

Behaviour Interactions

UML: Summary – Structure vs. Behaviour

Diagram	Purpose	When to Use	Key Elements
Class Diagram	Defines classes, attributes, methods, and relationships.	When designing the core structure of your system: data, logic, and relationships	Classes, attributes, methods, associations, inheritance, aggregation, compositions
Component/Deployment Diagrams	Show how software components are organized and deployed	When focusing on system architecture or physical deployment	Components, nodes, connections

UML: Summary – Structure vs. Behaviour

Diagram	Purpose	When to Use	Key Elements
Use Case Diagram	Captures system functionality from the user's perspective.	Early in design — to define *requirements* and understand *who interacts with what.*	Actors, use cases, include/extend relationships.
Activity Diagram	Models workflows, control flow, or data flow.	When describing *processes, algorithms,* or *business logic* step-by-step.	Start/end nodes, actions, decisions, forks, joins, edges.
State Machine Diagram	Models how an object changes state in response to events.	When designing *reactive systems*, e.g., embedded systems, UI states, control logic.	States, transitions, events, guard conditions.
Interaction Diagram	Category that includes sequence, communication, and timing diagrams.	When focusing on *communication patterns* between objects.	Entities and exchanged messages.
Sequence Diagram	Models interactions between components over time.	When describing *message flow* or *method calls* that implement a use case.	Lifelines, messages, activation bars, returns.

UML: Does it help modularity?

- Yes!
- Structure Diagrams
  - Recall: it defines the blueprint of the system
  - Architectural modularity: separation of systems into layers/components
- Behaviour Diagrams:
  - Recall: Communication between components
  - Interface modularity: shows how components communicate (communication contracts)

UML: Does it help modularity?

Diagram	Modularity?
Class Diagram	Object-oriented modularity. Encapsulation of data and behaviour.
Use Case Diagram	Functional modularity. Separation of use cases (feature or services)
Activity Diagram	Process modularity Each activity can become an independent function.
State Machine Diagram	Behavioural modularity. Each state machine represents a self-contained behavioural unit.
Sequence Diagram	Interaction boundaries Clarifies which object of module is responsible for which part of the communication flow

Let’s Review!

SFWRENG / MECHTRON 3K04 Software Development

This Week: Verification

November 2nd, 2025

Dr. Angela Zavaleta Bernuy

McMaster University | Engineering Computing & Software

Brighter World

Verification

- The process of checking whether a software system is built correctly according to its specified requirements and design.
- We will discuss:
  - Why we do it?
    - What are the goals of Verification?
  - How we do it?
    - What are the main approaches to Verification?
      - What kind of assurance do we get through testing?
      - How can testing be done systematically?
      - How can we remove defects (debugging)?
  - How successful we can expect it to be?

Need for Verification

- Designers are fallible even if they are skilled and follow sound principles
- Everything must be verified, every required quality, process and products
  - even verification itself…

Properties of Verification

- May not be binary (Pass/Fail)
  - Severity of defect is important
  - Some defects may be tolerated
- May be subjective or objective
  - e.g., usability
- Even implicit qualities should be verified
  - …because requirements are often incomplete
  - e.g., robustness

Approaches to Verification

- Dynamic techniques
  - Experiment with the behaviour of the product
  - Examples: unit testing, integration testing, system testing
  - Strength: can reveal unexpected runtime issues
  - Limitations: can only check a subset if behaviours through selected test cases.
- Static techniques
  - Analyze the product to deduce its adequacy
  - Examples: code reviews, walkthroughs, static analysis tools, formal verification
  - Strength: can catch errors early (before execution).
  - Limitation: may not detect runtime or environment-related issues.

Testing and Lack of Continuity

- Testing is sampling
  - Each test case is an experiment on a specific input or scenario
- The continuity problem:
  - Software behaviour isn’t continuous
- Implication:
  - We could have untested paths or hidden defects
  - Our code can exhibit correct behaviour in infinitely many cases, but it may still be incorrect in some cases.

Goals of Testing

- **Main goal**: *To show the presence of bugs, not their absence* (Dijkstra, 1987).
- Even if no defects are found, it doesn't mean that the system is error free.
- Still, we need to do testing!
- Testing **should** be systematic
  - Planned and based on principles like coverage, risk, and traceability to requirements.
- Testing **should** help to isolate errors
  - Provide diagnostic information
- Testing **should** be repeatable
  - Running the same test under the same conditions should produce the same results
- Testing **should** be accurate
  - Avoid false positives!

Testing Definitions

- Formal notation
  - We can define a program P as a mapping from an input domain D to an output domain R.
  - $P: D \rightarrow R$
- Correctness: $OR \subseteq D \times R$
  - Observed results: OR
  - Cartesian product $D \times R$ represents all possible valid input-output pairs.
  - $P(d)$ is correct if $\forall d,\ P(d) \in OR$
  - That is, P is correct if all $P(d)$ are correct
- Similar to black box testing