Design Defects and Restructuring

Lecture 13

Sat, Dec 18, 2021

Principles of Package Design

- Granularity – The Principles of Package Cohesion: They help us allocate classes to packages
  - The Reuse Release Equivalence Principle (REP)
  - The Common Reuse Principle (CRP)
  - The Common Closure Principle (CCP)
- Stability – The Principles of Package Coupling: They help us determine how packages should be interrelated
  - The Acyclic Dependency Principle (ADP)
  - The Stable Dependency Principle (SDP)
  - The Stable Abstraction Principle (SAP)

Granularity

- In the UML, packages can be used as containers for a group of classes
- By grouping classes into packages, we can reason about the design at a higher level of abstraction
- The goal is to partition the classes in your application according to some criteria, and then allocate those partitions to packages
- The relationships between those packages expresses the high-level organization of the application

Granularity

- What are the best partitioning criteria?
- What are the relationships that exist between packages, and what design principles govern their use?
- Should packages be designed before classes (Top down)? Or should classes be designed before packages (Bottom up)?
- How are packages physically represented? In C++? In the development environment?
- Once created, to what purpose will we put these packages?

The Reuse Release Equivalence Principle (REP)

- The granule of the reuse is the granule of the release
- The granule of the reuse (package) can be no smaller than the granule of release
- If a package contains software that should be reuse, then it should not also contain the software that is not designed for reuse
- Either all of the classes in a package are reusable or none of them are
- Concept of reusability, concept of re user

The Common Reuse Principle (CRP)

- The classes in a package are reused together
- If you reuse one of the classes in a package, you reuse them all
- This principle helps us to decide which classes should be placed into a package
- It states that classes that tend to be reused together belong in the same package
- Re-distribution problems

The Common Closure Principle (CCP)

- The classes in a package should be closed together against the same kind of changes
- A change that affects a package affects all the classes in that package and no other packages
- SRP re-stated for packages
- It is closely associated with OCP

Stability

- The classic definition of the word stability is “Not easily moved”
- Stability is not a measure of the likelihood that a module will change; rather it is a measure of the difficulty in changing a module
- Modules that are more difficult to change, are going to be less volatile
- The harder the module is to change, the more stable it is, the less volatile it will be
- Classes that are heavily depended upon are called “Responsible”
- Responsible classes tend to be stable because any change has a large impact
- The most stable classes of all are classes that are both Independent and Responsible

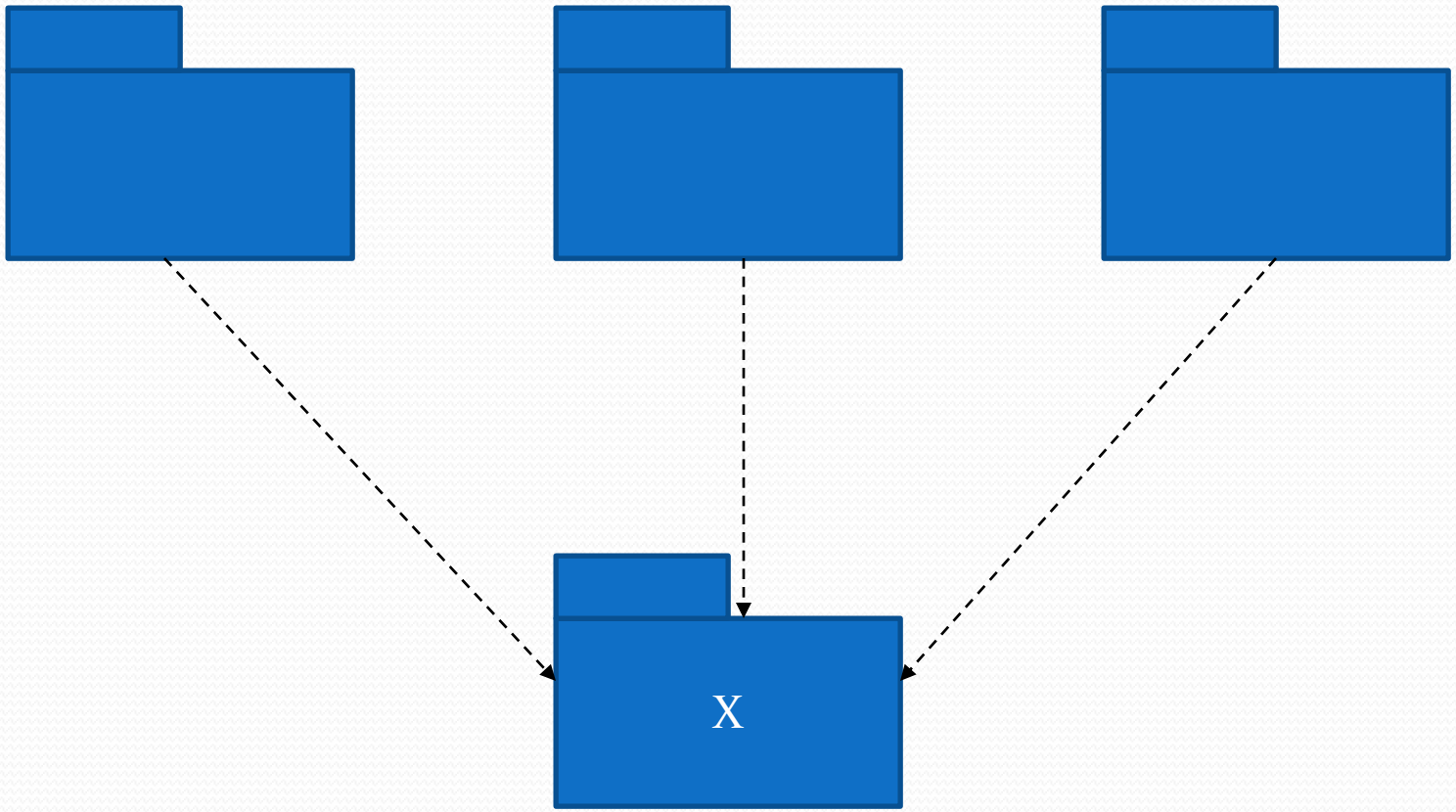
The Acyclic Dependency Principle (ADP)

- Allow no cycle in the package dependency graph
- Morning-after syndrome
- The weekly build – Partitioning the development environment into releasable packages
- Breaking the cycle
  - Apply DIP
  - Create a new package, move the classes that they both depend on into that new package
- Top-Down design

The Stable Dependency Principle (SDP)

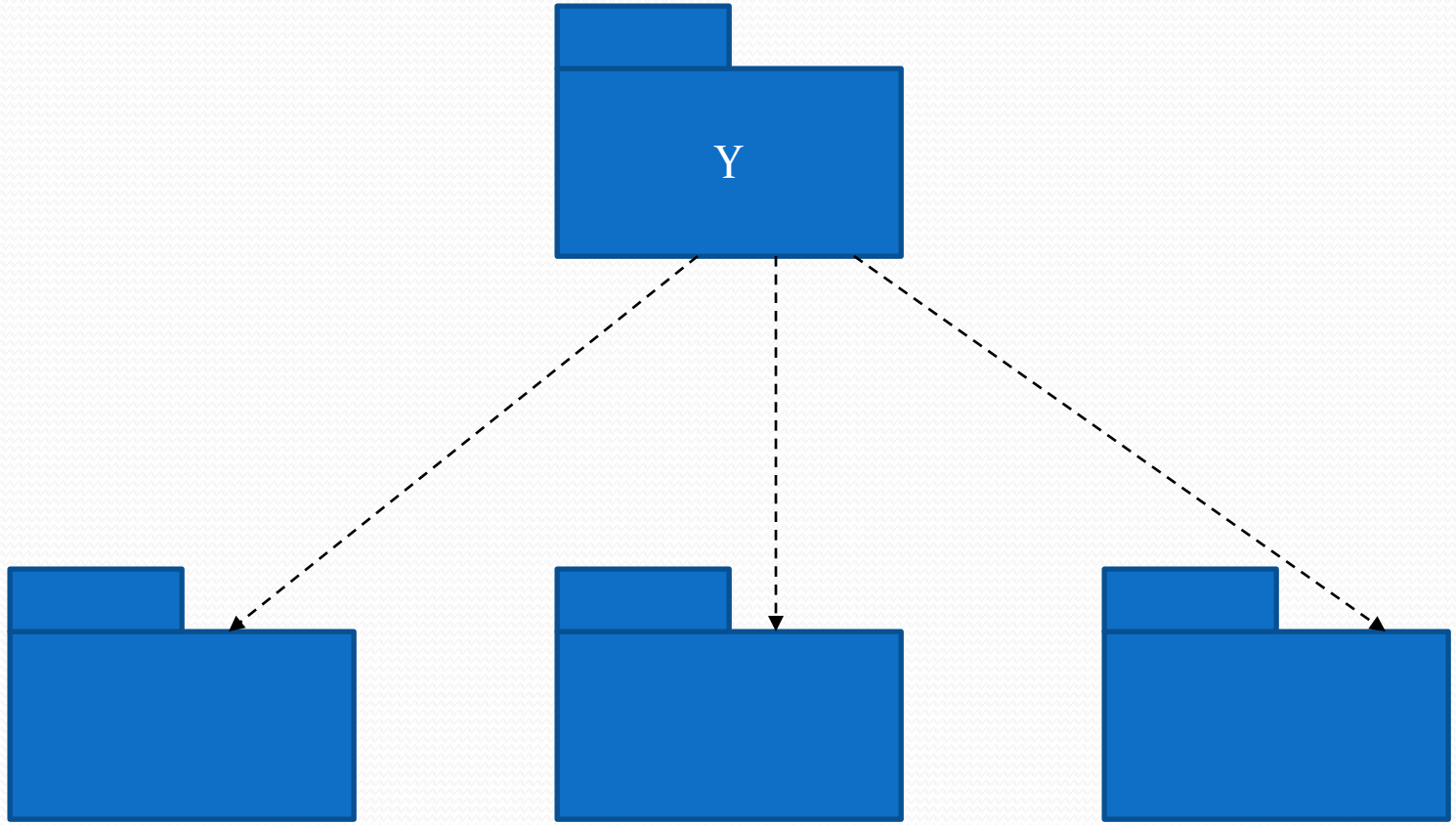
- Depends in the direction of stability
- Any package that we expect to be volatile should not be depended on by a package that is difficult to change, otherwise the volatile package will also be difficult to change
- Stability : amount of work required to make a change

The Stable Dependency Principle (SDP)



X: A Stable Package

The Stable Dependency Principle (SDP)



Y: An Unstable Package

The Stable Dependency Principle (SDP)

- Stability Metrics
  - Afferent Couplings (Ca): The number of classes outside this package that depends on classes within this package
  - Efferent Couplings (Ce): The number of classes inside this package that depend on classes outside this package
  - Instability (I) : $I = \frac{C_e}{C_a + C_e}$
- Range = [0,1]
- I = 0: Maximum Stability
- I = 1: Maximum Instable Package

The Stable Abstraction Principle (SAP)

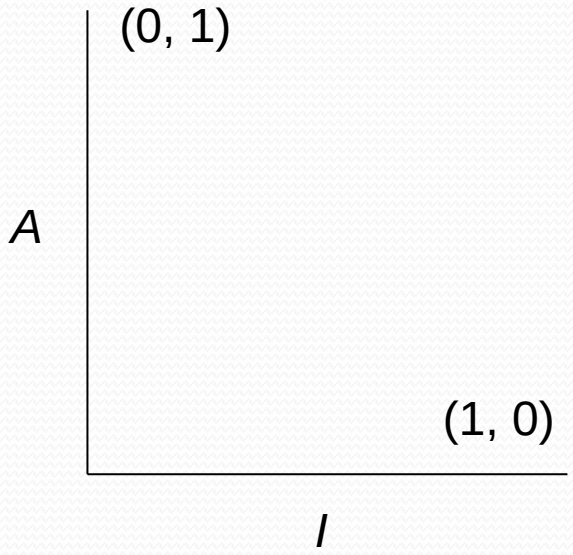
- The package should be as abstract as it is stable
- This principle sets up the relationship between stability and abstractness
- Stable package should also be abstract so that its stability does not prevent it from being extended
- Instable package should also be concrete since its instability allows the concrete code within it to be easily changed

The Stable Abstraction Principle (SAP)

- Abstraction Metrics
  - N_c – The number of classes in the package
  - N_a – The number of abstract classes in the package
  - Abstractness (A) : $$A = \frac{N_a}{N_c}$$
- Range = [0,1]
- A = 0: Package has no Abstract classes
- A = 1: Package contains nothing but the abstract classes

The Main Sequence

- A – I Graph
- Maximally stable and abstract (0, 1)
- Maximally instable and concrete (1, 0)

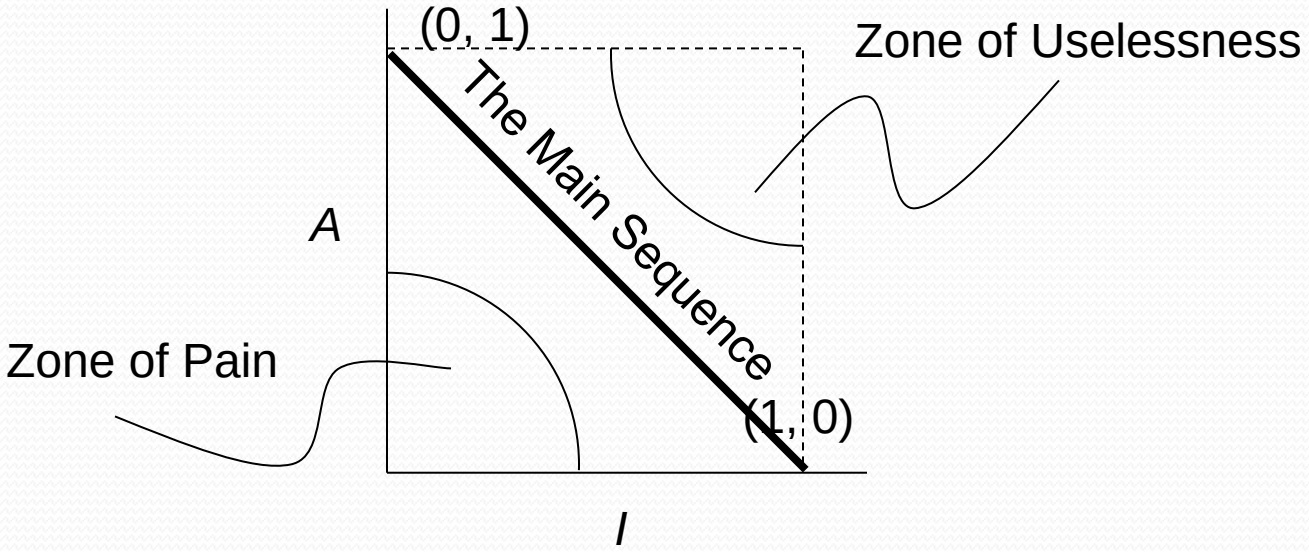


The Main Sequence

- (0, 0) : Highly stable and concrete package
- It cannot be extended because it is not abstract
- It is very difficult to change because of its stability
- Example 1 : Database Schemas
- Example 2 : Concrete Utility Library
- Zone of Pain

The Main Sequence

- (1, 1) : Maximally abstract and no dependents
- Zone of Uselessness



Distance from the Main Sequence

- Distance Formula
- Range [0, ~ 0.707]

$$D = \frac{|A + I - 1|}{\sqrt{2}}$$

- Normalized Distance D'
- Range [0, 1]

$$D' = |A + I - 1|$$

Enterprise Application Architecture Prerequisites

- Layering
- Organizing Domain Logic
- Mapping to Relational Databases
- Web Presentation
- Concurrency
- Session State
- Distribution Strategies

Layering

- Benefits
  - Understandability – Coherent, without knowing much about other layers
  - Substitutability – With alternative implementation
  - Minimal Dependency – Adaptation and Abstraction
  - Standardization – Create your own standards for a layer
  - High-level Service Use – Provide barrier to a layer below

Layering

- Downsides
  - Encapsulation
    - Not all elements are encapsulated
    - UI to Database?
  - Too many Layers
    - Data transformation affects performance

Layering

- 3 Principal Layers
  - Presentation
    - Provision of services, display of information (e.g., in Windows or HTML, handling of user request (mouse clicks, keyboard hits), HTTP requests, command-line invocations, batch API)
  - Domain
    - Logic that is the real point of the system
  - Data Source
    - Communication with databases, messaging systems, transaction managers, other packages

Organizing Domain Logic

- Transaction Script
  - Simple Logic
- Domain Model
  - Complex Logic
- Table Module
  - Moderate Logic and good API Tools around
- Service Layer
  - Provides API

Mapping to Relational Databases

- Architecture
  - Mappings and Gateways
- Behavioral Issues
  - Data Reading
  - Data Manipulation
- Using Metadata
  - Code Generation
  - Reflective Programming
- Database Connections

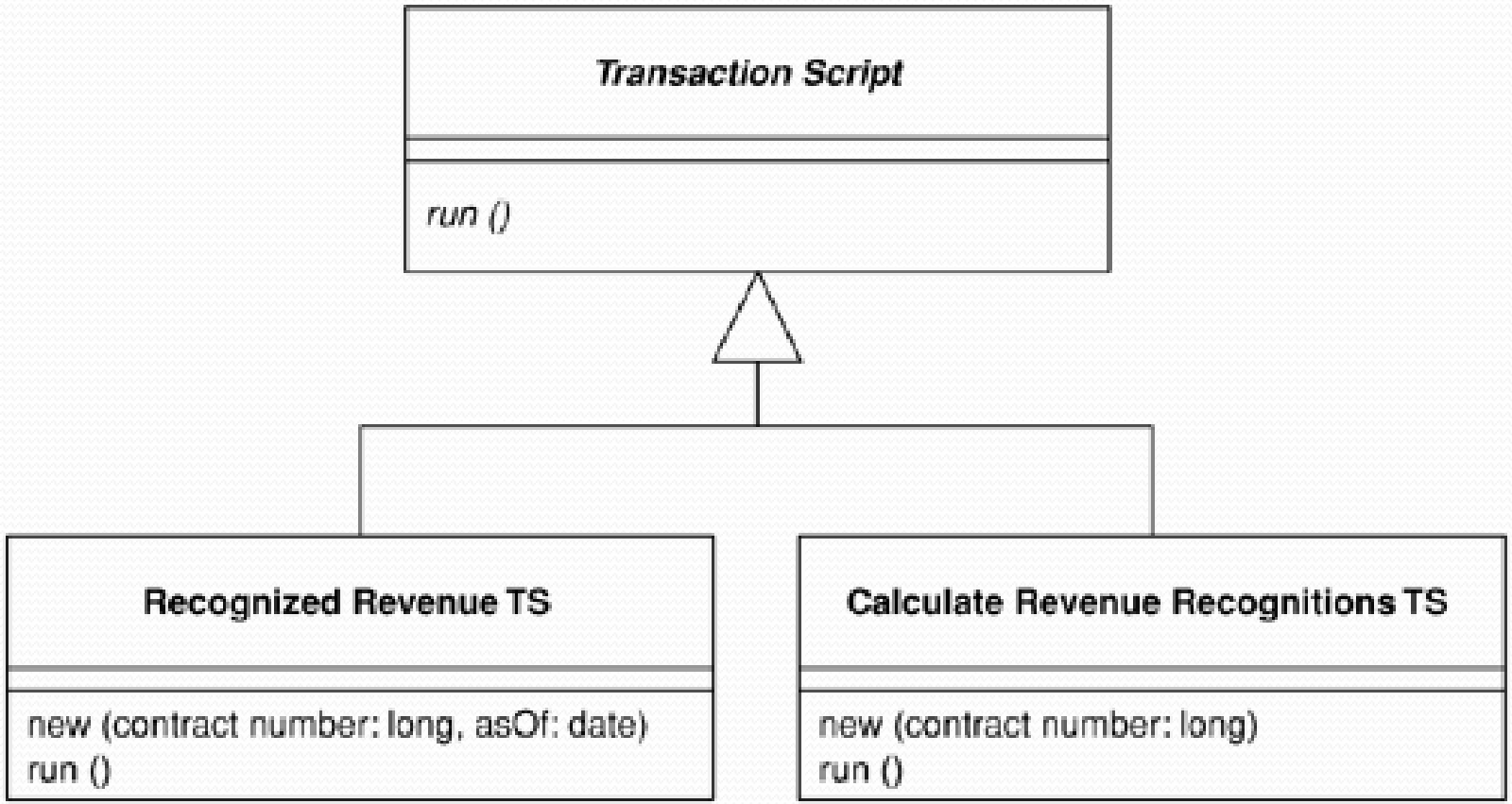
Domain Logic Patterns

- Transaction Script
- Domain Model
- Table Module
- Service Layer

Transaction Script

- Organizes business logic by procedures where each procedure handles a single request from the presentation
- When to use it
  - Simplicity
  - Smaller logic
  - Beware of duplication

Transaction Script



Domain Model

- An object model of the domain that incorporates both behavior and data
- When to use it
  - Complex logic
  - Ever changing business rules
    - Involving validation, calculations, and derivations

Domain Model

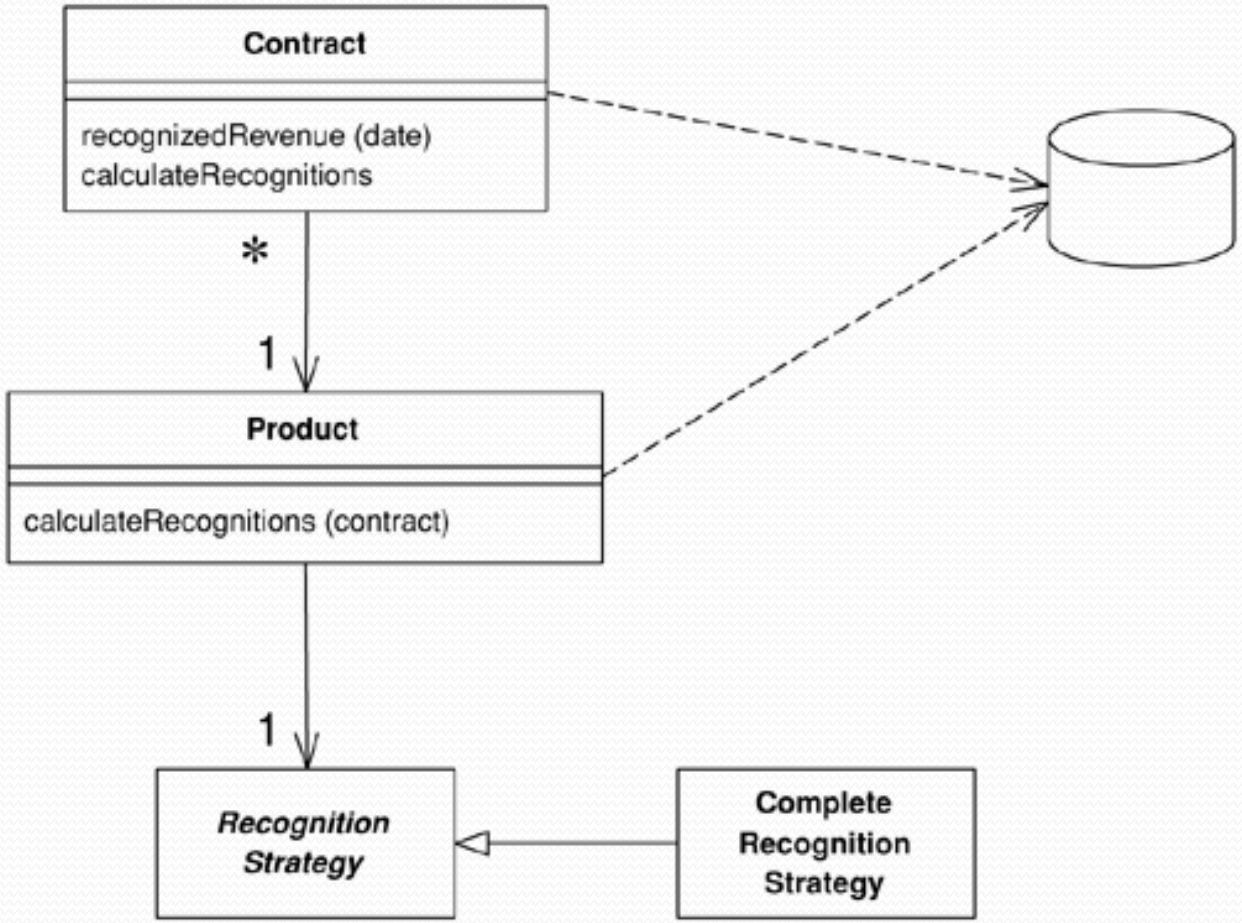
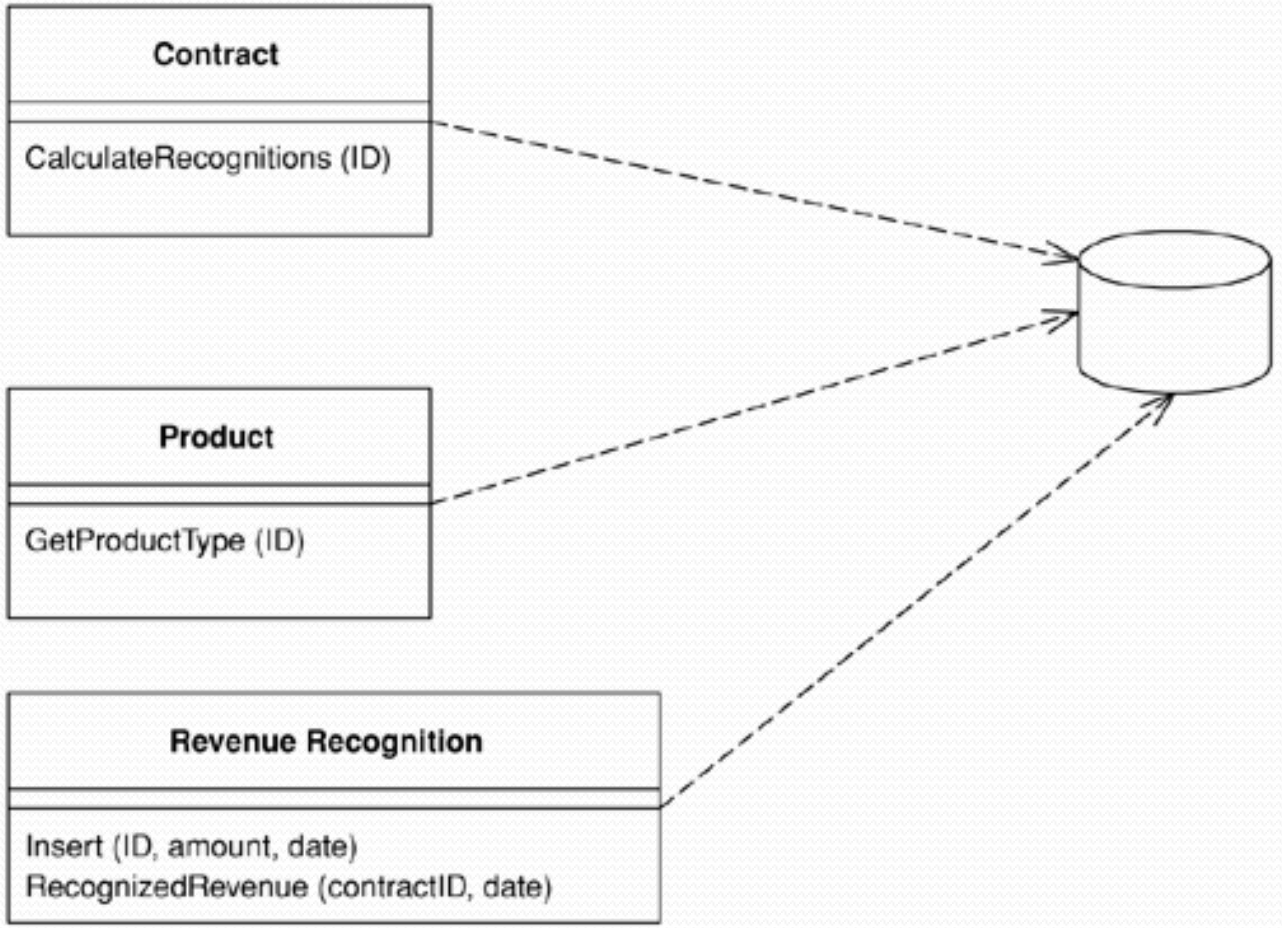


Table Module

- A single instance that handles the business logic for all rows in a database table or view
- When to use it
  - Based on table oriented data
    - Access using Record Set
  - Data structure are fairly straightforward
  - No direct instance-to-instance relationships

Table Module



Service Layer

- Defines an application's boundary with a layer of services that establishes a set of available operations and coordinates the application's response in each operation
- When to use it
  - In an application with more than one kind of client of its business logic, and complex responses in its use cases involving multiple transactional resources, it makes a lot of sense to include a Service Layer with container-managed transactions

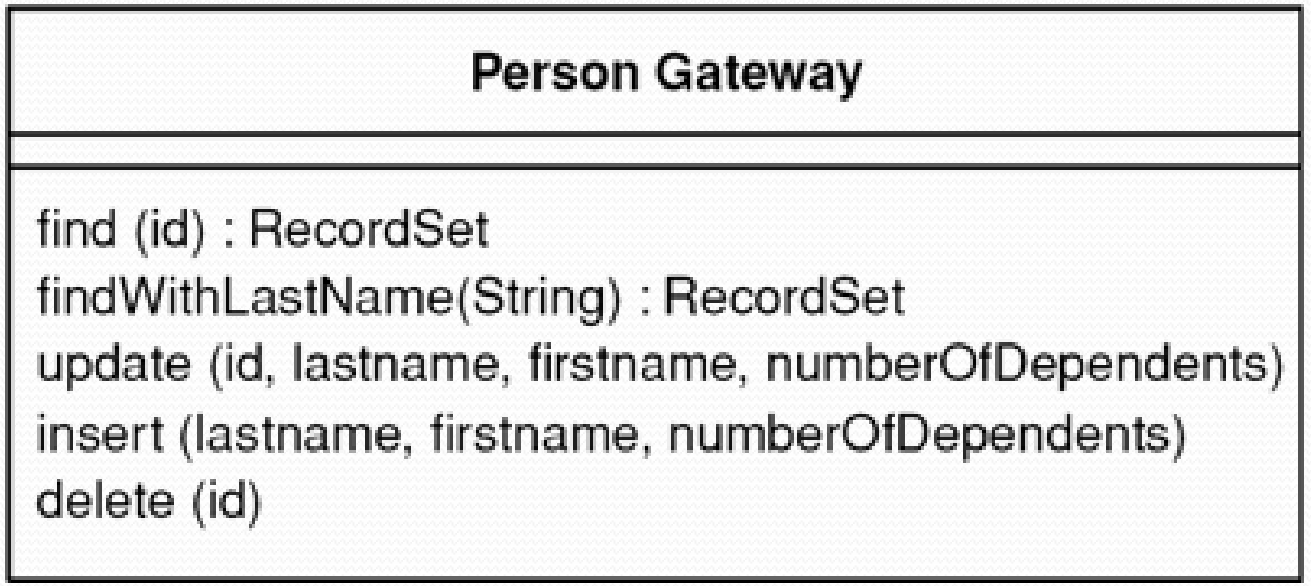
Data Source Architectural Patterns

- Table Data Gateway
- Row Data Gateway
- Active Record
- Data Mapper

Table Data Gateway

- An object that acts as a Gateway to a database table
  - One instance handles all the rows in the table
- When to use it
  - Work well with Table Module and Transaction Script
  - Encapsulation of database access

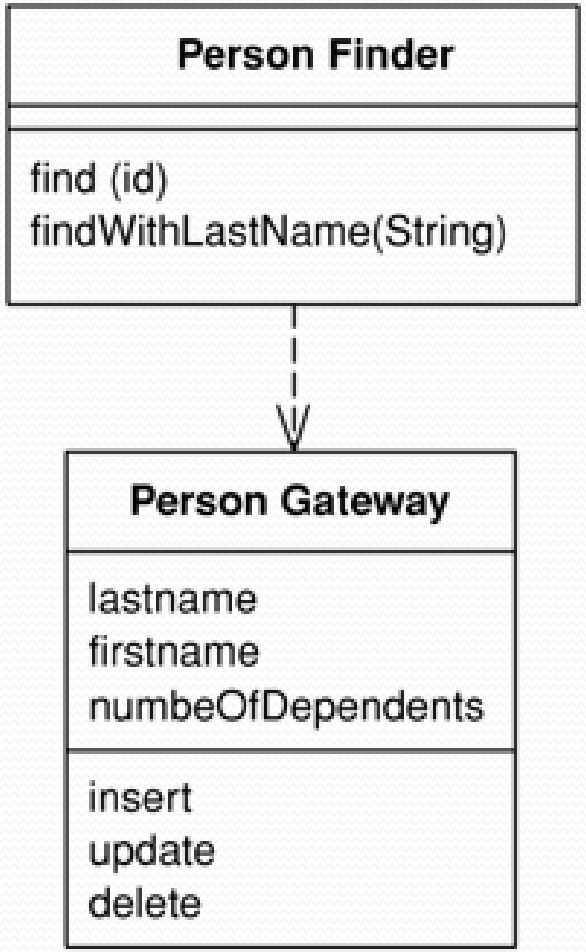
Table Data Gateway



Row Data Gateway

- An object that acts as a Gateway to a single record in a data source
  - There is one instance per row
- When to use it
  - Use it with Transaction Script

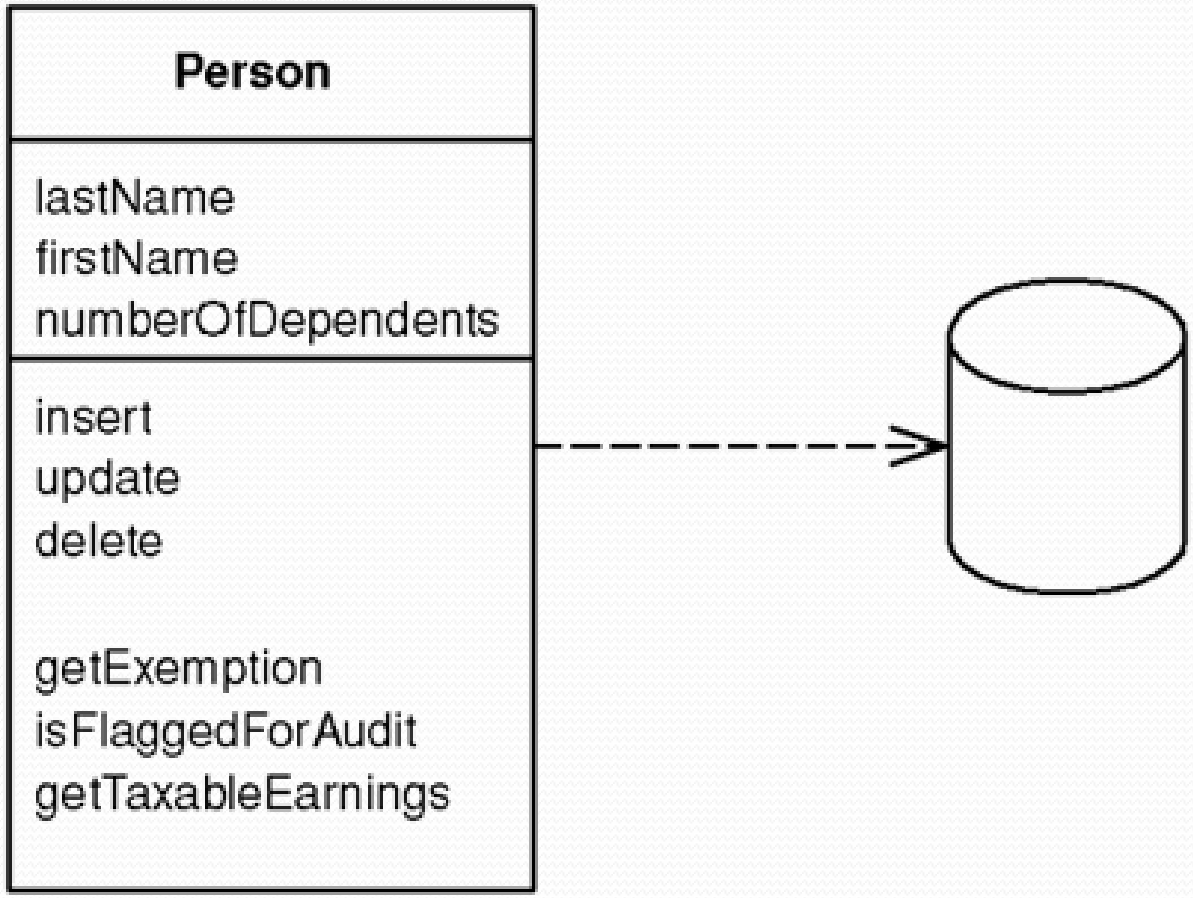
Row Data Gateway



Active Record

- An object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on that data
- When to use it
  - Logic is not complex

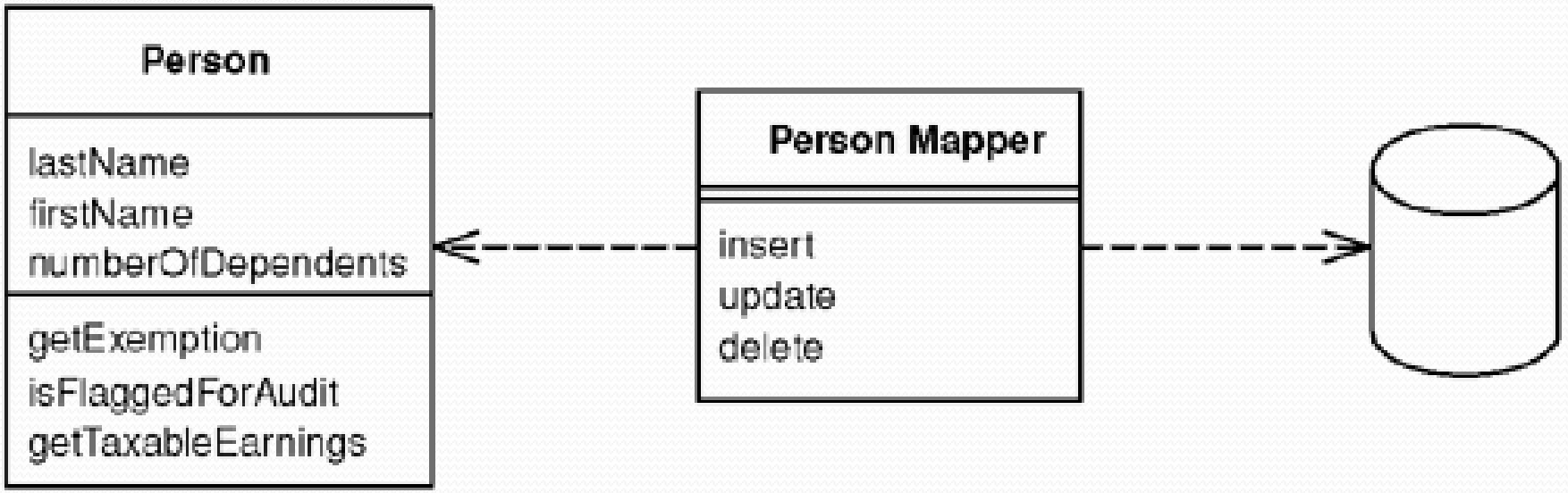
Active Record



Data Mapper

- A layer of Mappers that moves data between objects and a database while keeping them independent of each other and the mapper itself
- When to use it
  - When you want the database schema and the object model to evolve independently
  - When you are using Domain Model

Data Mapper


Person
lastName
firstName
numberOfDependents
getExemption
isFlaggedForAudit
getTaxableEarnings
Person Mapper
insert
update
delete

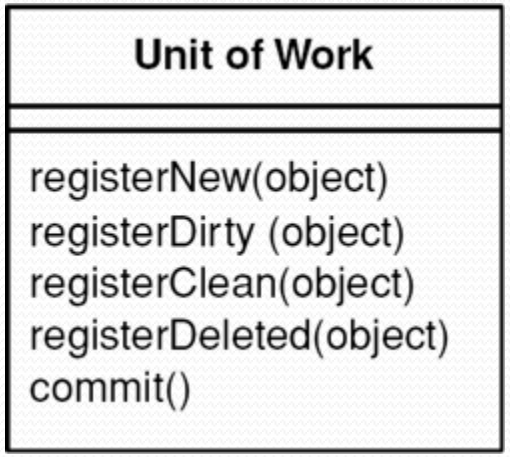
Object – Relational Behavioral Patterns

- Unit of Work
- Identity Map
- Lazy Load

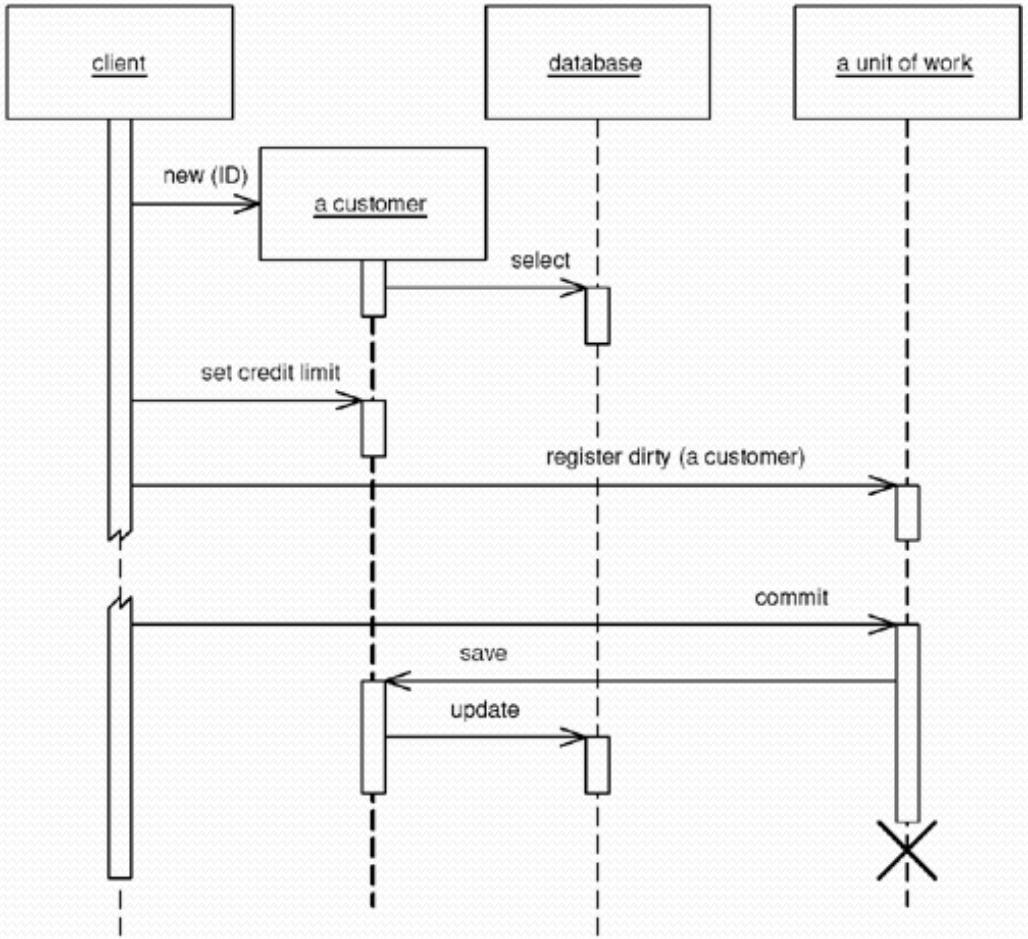
Unit of Work

- Maintains a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems
  - Caller Registration
  - Object Registration
  - Unit of Work Controller
- When to use it
  - Keeping track of various objects
  - Reducing database access

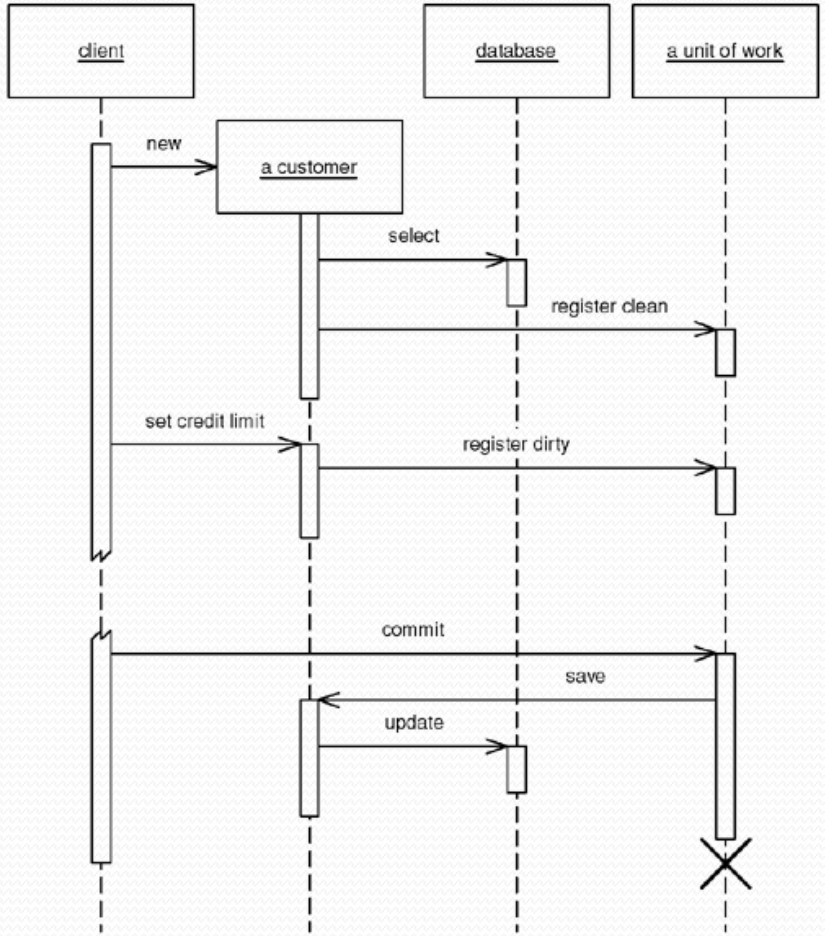
Unit of Work


Unit of Work
registerNew(object)
registerDirty (object)
registerClean(object)
registerDeleted(object)
commit()

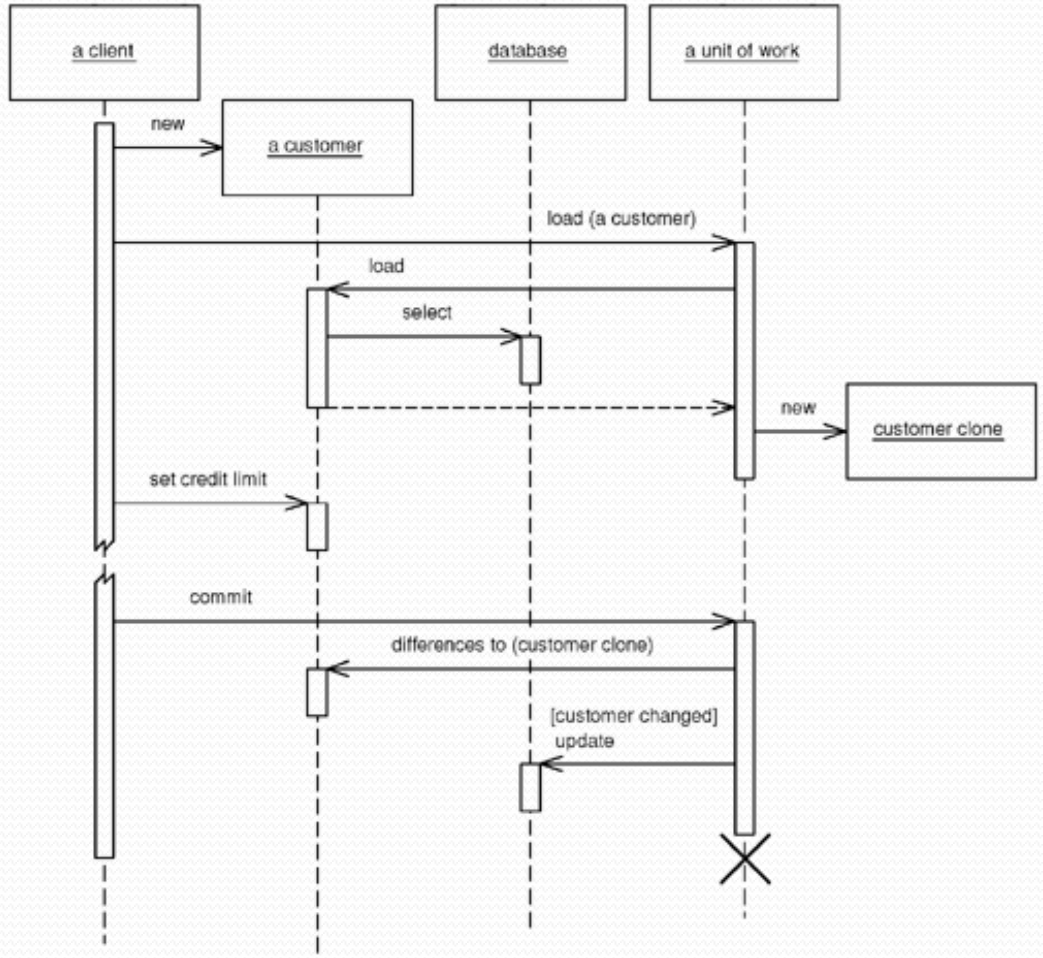
Unit of Work – Caller Registration



Unit of Work – Object Registration



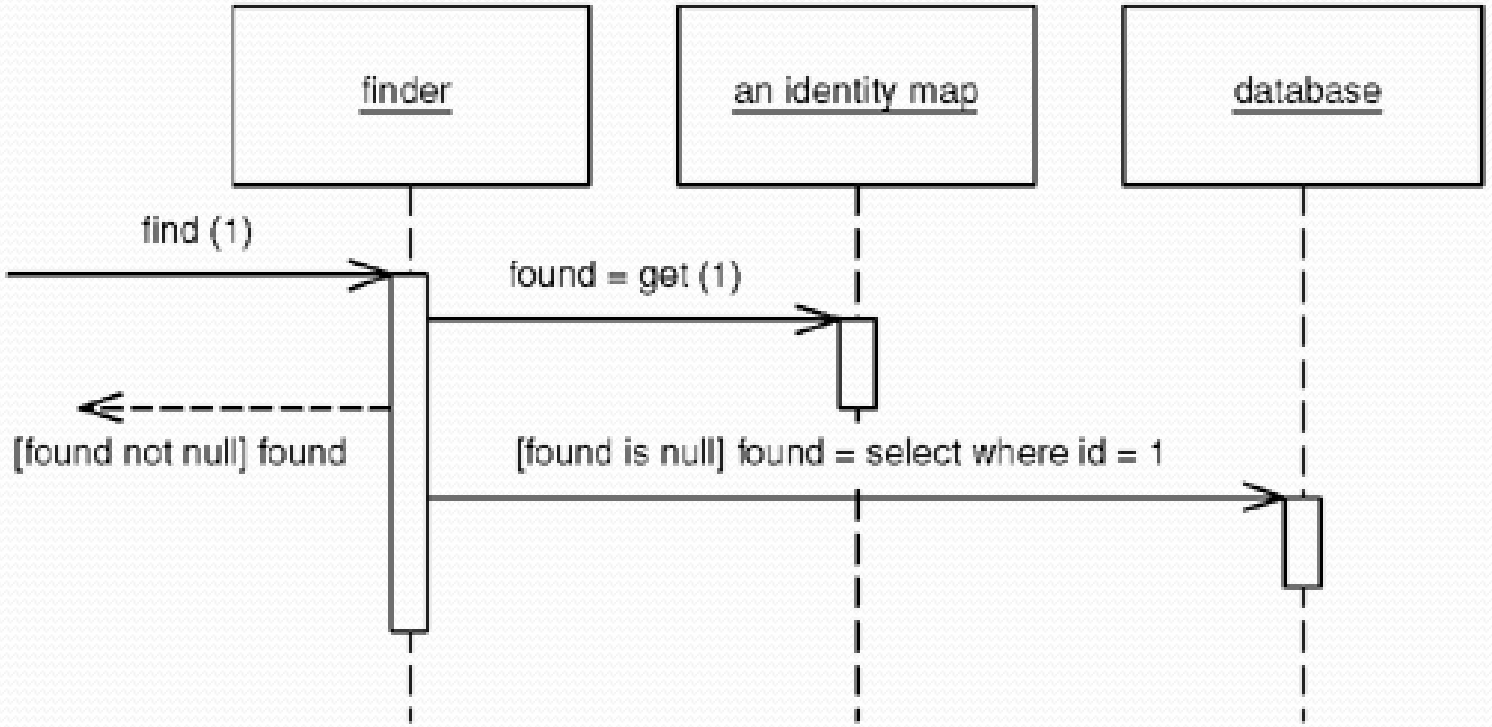
Unit of Work – Controller



Identity Map

- Ensures that each object gets loaded only once by keeping every loaded object in a map. Looks up objects using the map when referring to them
- When to use it
  - Treat it as an in memory cache
  - To remove in consistency
  - Key field – Surrogate Key
  - Explicit or Generic
  - Mapper per Class or Mapper per Session

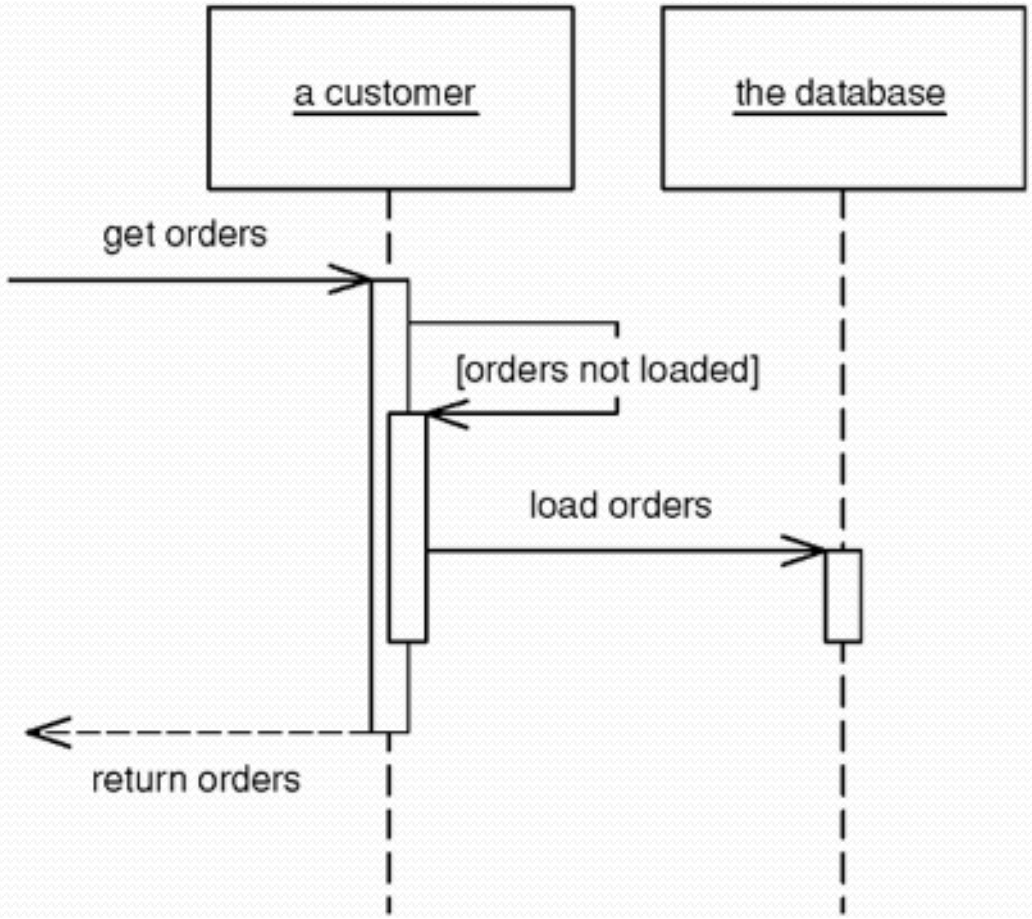
Identity Map



Lazy Load

- An object that does not contain all of the data you need but knows how to get it
  - Lazy initialization
    - Gateways
    - Getter Method
  - Virtual proxy
    - Mappers
    - Proxy pattern
  - Value holder
    - Wraparound object
    - Single loading
  - Ghost

Lazy Load


a customer
the database
get orders
[orders not loaded]
load orders
return orders