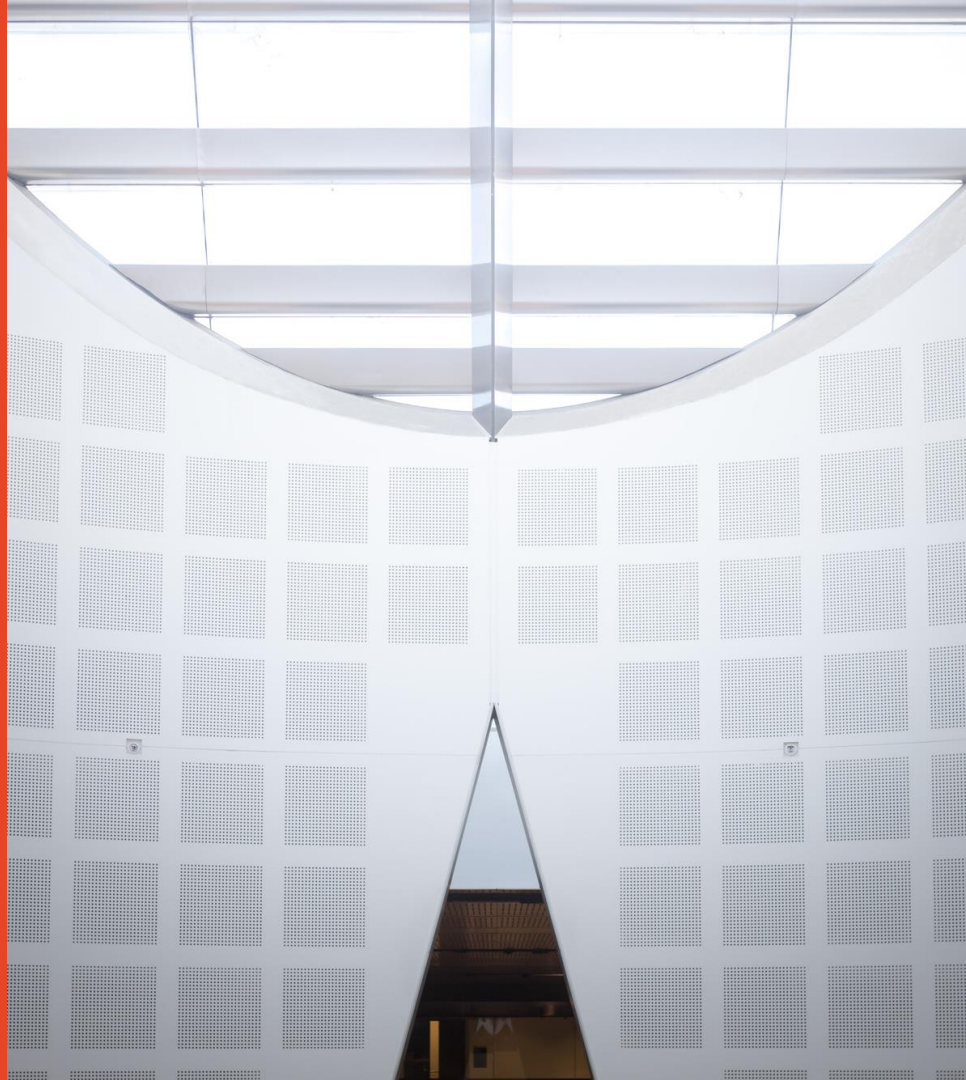
Software Design and Construction 2
SOFT3202 / COMP9202
Enterprise Application Architecture Design Patterns

Dr. Basem Suleiman

School of Information Technologies

THE UNIVERSITY OF SYDNEY



Copyright Warning

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**).

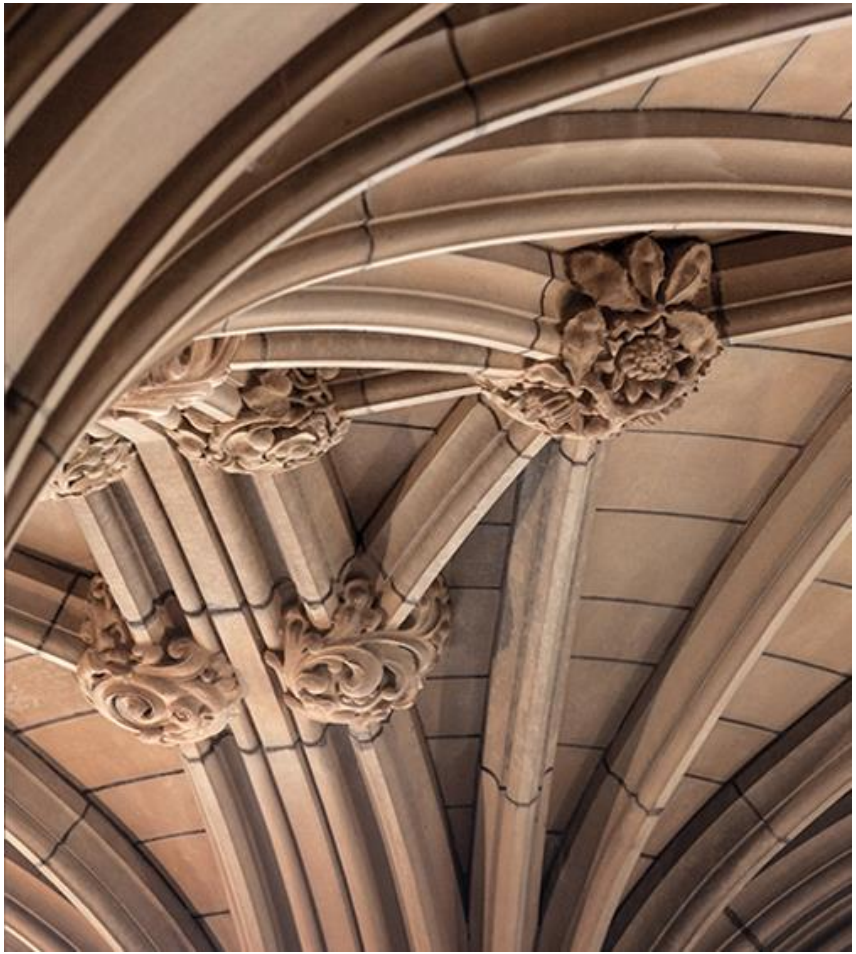
The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

Agenda

- Enterprise Application Architecture
  - Layered Architecture
- Enterprise Application Design Patterns
  - Unit of work
  - Lazy load
  - Value Object

Enterprise Applications Architecture



THE UNIVERSITY OF SYDNEY

Catalogue of Design Patterns

Design Pattern	Description
Gang of Four (Gof)	First and most used. Fundamental patterns OO development, but not specifically for enterprise software development.
Enterprise Application Architecture (EAA)	**Layered application architecture with focus on domain logic, web, database interaction and concurrency and distribution. Database patterns (object-relational mapping issues)**
Enterprise Integration	Integrating enterprise applications using effective messaging models
Core J2EE	EAA Focused on J2EE platform. Applicable to other platforms
Microsoft Enterprise Solution	MS enterprise software patterns. Web, deployment and distributed systems

https://martinfowler.com/articles/enterprisePatterns.html

Enterprise Applications

- Payroll, patient records, shipping tracking, insurance, accounting, insurance

- Involve lots of data
  - Information systems or data processing
  - Managing very large data using DBMS
  - Data must be persistent (for many years even with SW/HW changes)
  - Concurrent data access (web-based) require transaction management
  - Data needs to be presented differently to end users

- Enterprise applications need to integrate with other applications

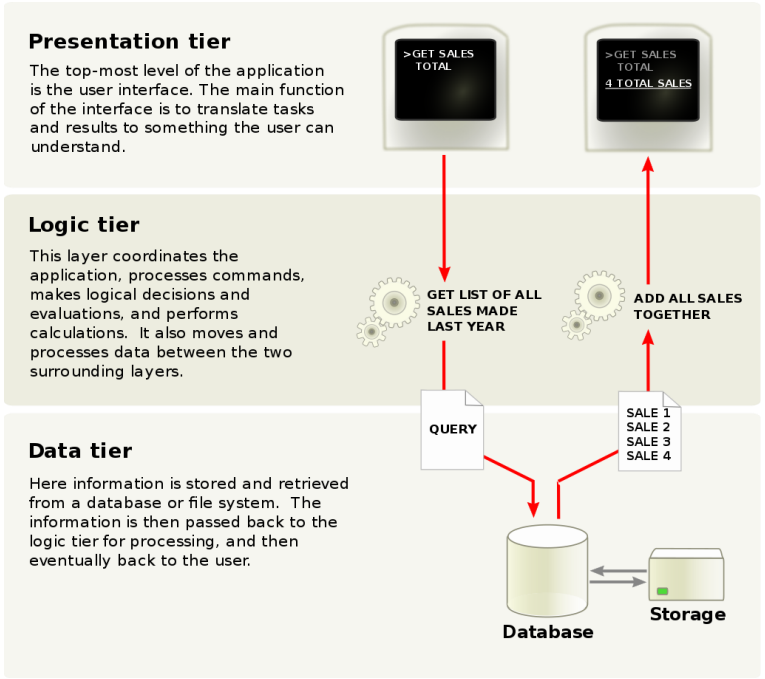
Enterprise Applications

- Enterprise applications need to integrate with other applications
  - Developed using different technologies
  - Data files, DBMS, communication among components

- Differences in business processes
  - Complex business logic/rules
  - Customer (one with current agreement or had a contract)
  - Product sales vs service sales

  - Different enterprise application types impose different design challenges

Enterprise Applications – Layered Architecture

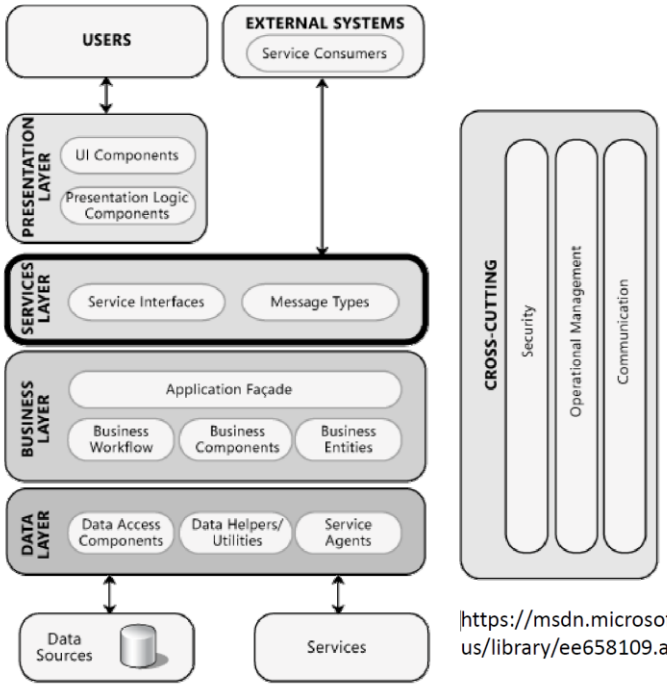


https://commons.wikimedia.org/wiki/File:Overview_of_a_three-tier_application_vectorVersion.svg#/media/File:Overview_of_a_three-tier_application_vectorVersion.svg

Enterprise Applications – Principal Layers

Layer	Description
Presentation	Handle interactions between the user and the application
Domain/Business logic	Application logic including calculations based on inputs, retrieving and storing data from the data source
Data Source	Storing persistence data. Works with other systems such as transaction monitors and messaging system

Enterprise Applications – Layered Architecture



https://msdn.microsoft.com/en-us/library/ee658109.aspx

Layering – Architectural Decision

- Performance is a significant design decision
  - Response time: amount time it takes for the system to process a request
  - Throughput: amount of work an application can handle in a given amount of time
    - Bytes per seconds, requests per second, transaction per second
  - Other measures related to performance include responsiveness, latency, scalability

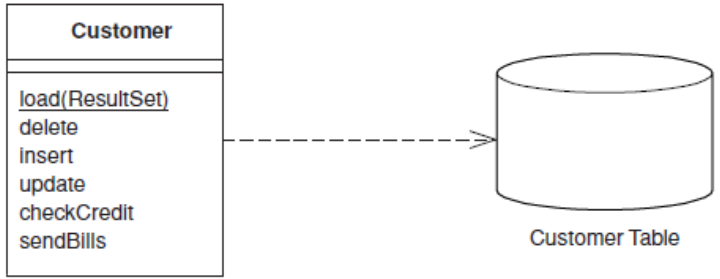
Layering – Architectural Decision

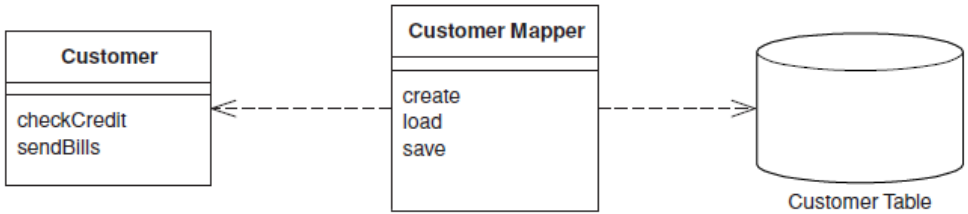
- Many factors can influence performance while designing enterprise applications
  - Communication among different components
    - Trips over the wire (networking)
    - Communication styles (messaging, RPCs)
  - System resources (memory, disk)
  - Expensive system operations (database read/write)
  - Sharing and concurrency

Mapping Objects to Database

- Data is modeled as objects in the domain model but as tables (e.g., in RDB)
  - Objects stored in-memory
  - Tables stored persistently a database

- Issue: how to do mapping between in-memory objects and database tables





What is a Transaction?

- Database technology allows application to define a transaction

- A segment of code that will be executed as a whole
  - System infrastructure should prevent problems from failure and concurrency

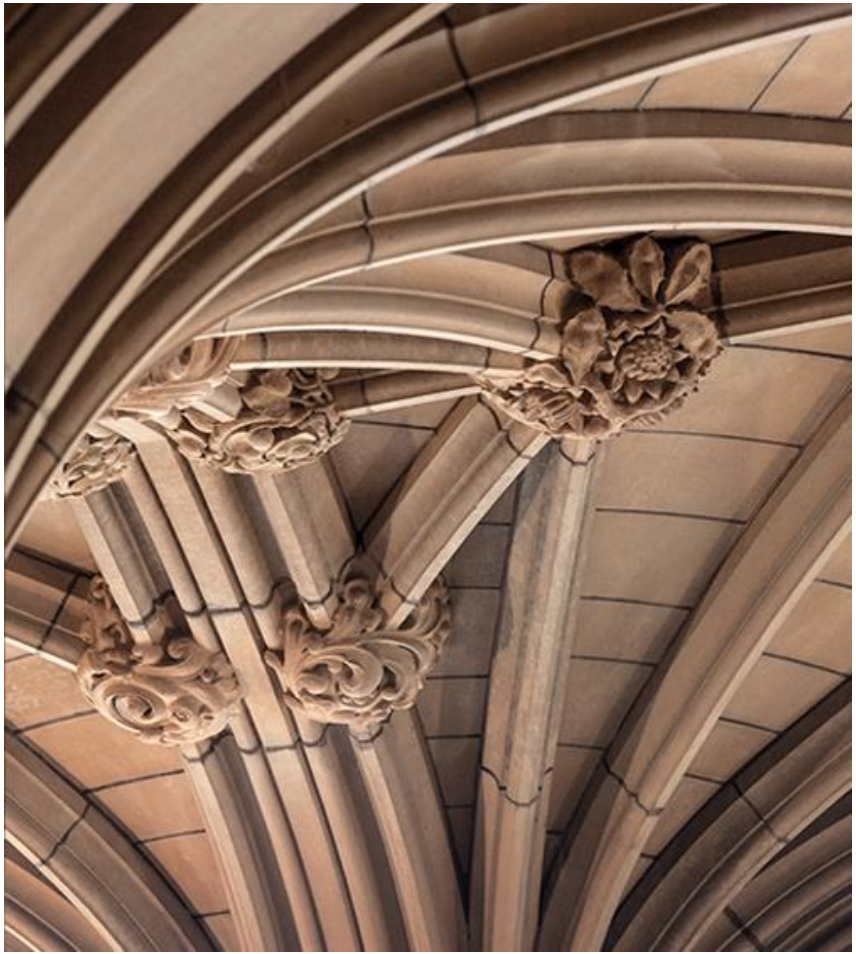
- Transaction should be chosen to cover a complete, indivisible business operation
  - Book a seat
  - Transfer money
  - Withdraw cash
  - Sell something
  - Borrow a book

Transaction – ACID

- The transaction support should pass the ACID test
  - Atomic (all, or nothing)
    - Either transaction commits (all changes are made) or else it aborts (no changes are made)
    - Abort may be application-requested rollback, or it may be system initiated
  - Consistent
  - Isolated (no problems from concurrency; details are complicated)
    - DBMS provides choices of "isolation level"
      - Usually good performance under load is only available with lower isolation
      - But lower isolation level does not prevent all interleaving problems
  - Durable (changes made by committed transactions are not lost in failures)

Unit of Work

Object Relational Behavioural Pattern



THE UNIVERSITY OF SYDNEY

Unit of Work Pattern

"Maintains a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems"

Unit of Work – Applicability

- Want to keep track of everything during a business transaction that can affect the database
  - Pulling data from a database
  - Insert new objects you create and remove any objects you delete
  - Might be useful with session objects

- Why not changing the database with each change to the object model?

Unit of Work – Applicability

- Want to keep track of everything during a business transaction that can affect the database
  - Pulling data from a database
  - Insert new objects you create and remove any objects you delete
- Why not changing the database with each change to the object model?
  - Slow performance (lots of small database calls)
  - Impractical (transactions with multiple request)

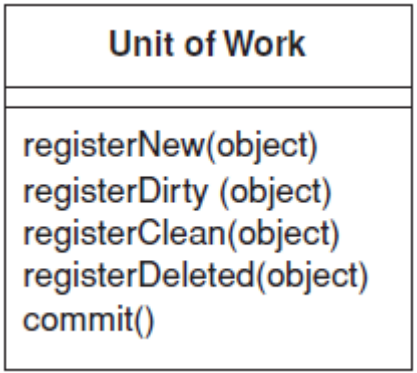
Unit of Work – Applicability

- Batch updates
  - Multiple database updates as a unit
  - One remote call rather than multiple
  - JDBC allow such facility

- Any transactional resource not just databases

Unit of Work – How it Works

- Notify UoW of any change including:
  - New object created
  - Existing objects updated (modified/deleted)
  - Existing objects being read (to ensure consistency)

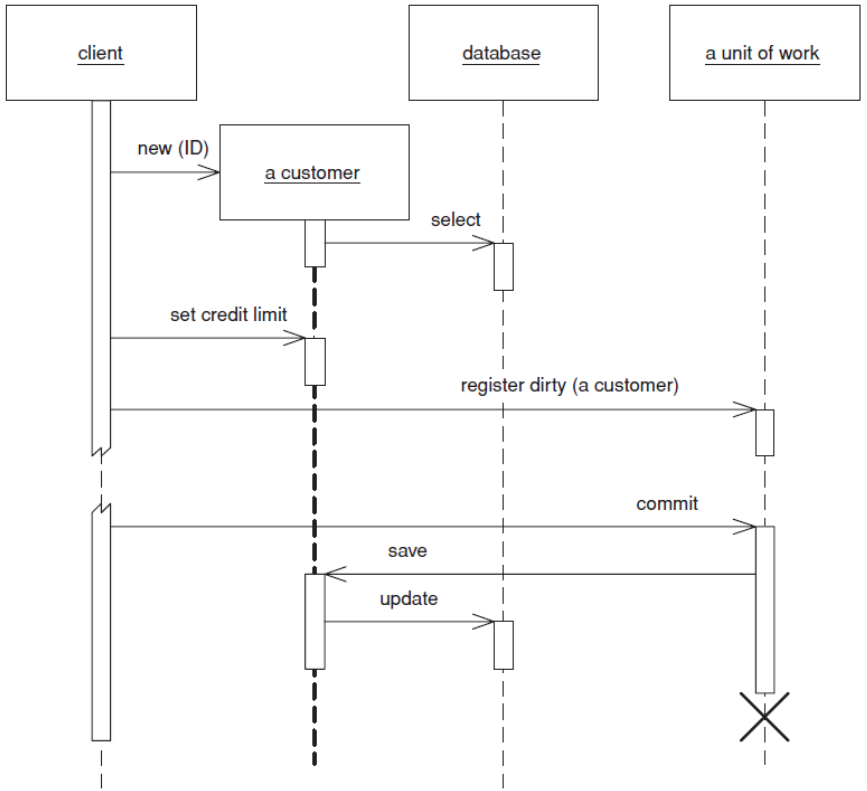


- When a transaction has to be committed, UoW:
  - Open a transaction
  - Check concurrency (pessimistic/optimistic offline lock)
    - Why not programmers make explicit database updates?
  - Write changes out to the database

- Needs to know objects it should keep track of
  - By caller or through the object itself

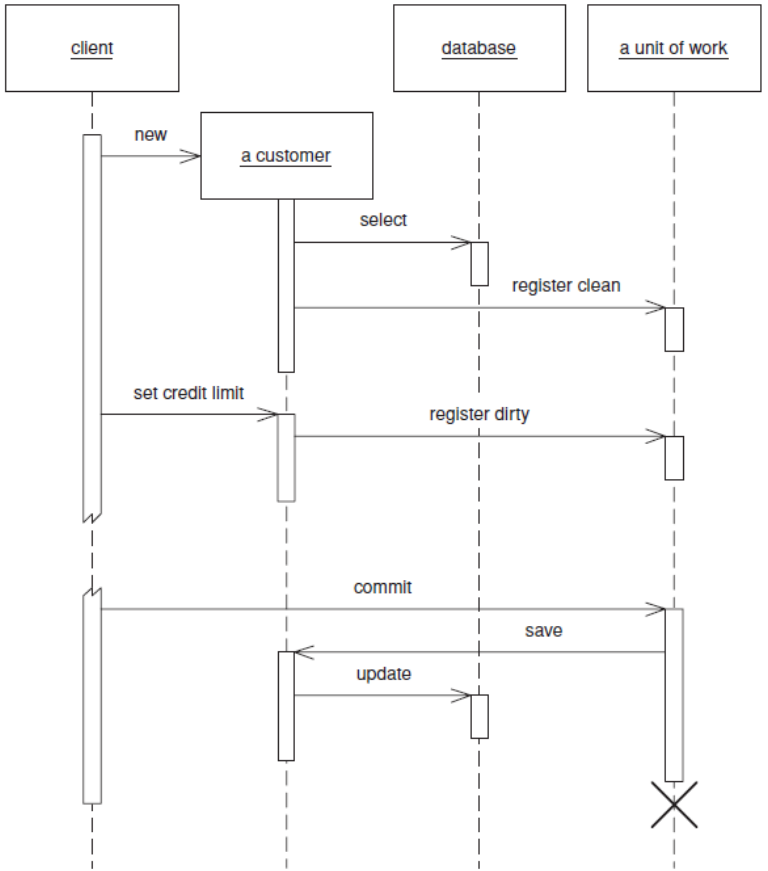
UoW – Caller Registration

- UoW needs to know what objects it should keep track of
- This can be realized by Caller Registration



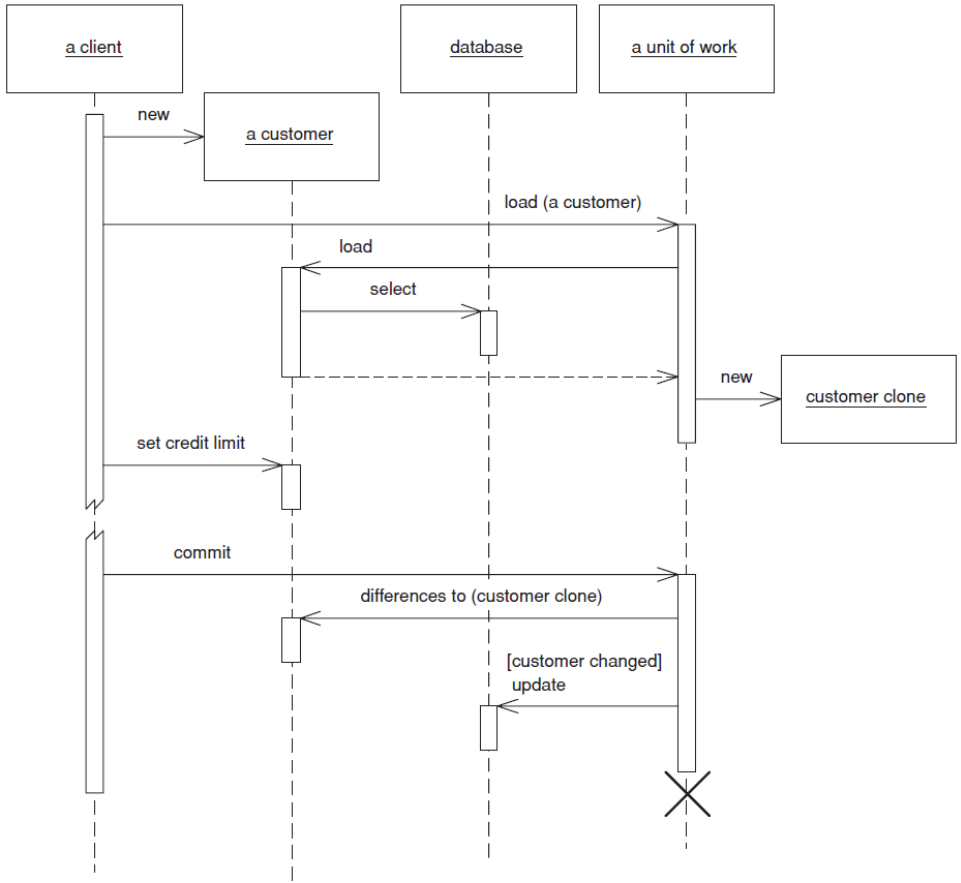
UoW – Object Registration

- UoW needs to know what objects it should keep track of
- This can be realized by Object Registration



UoW – as Controller

– .

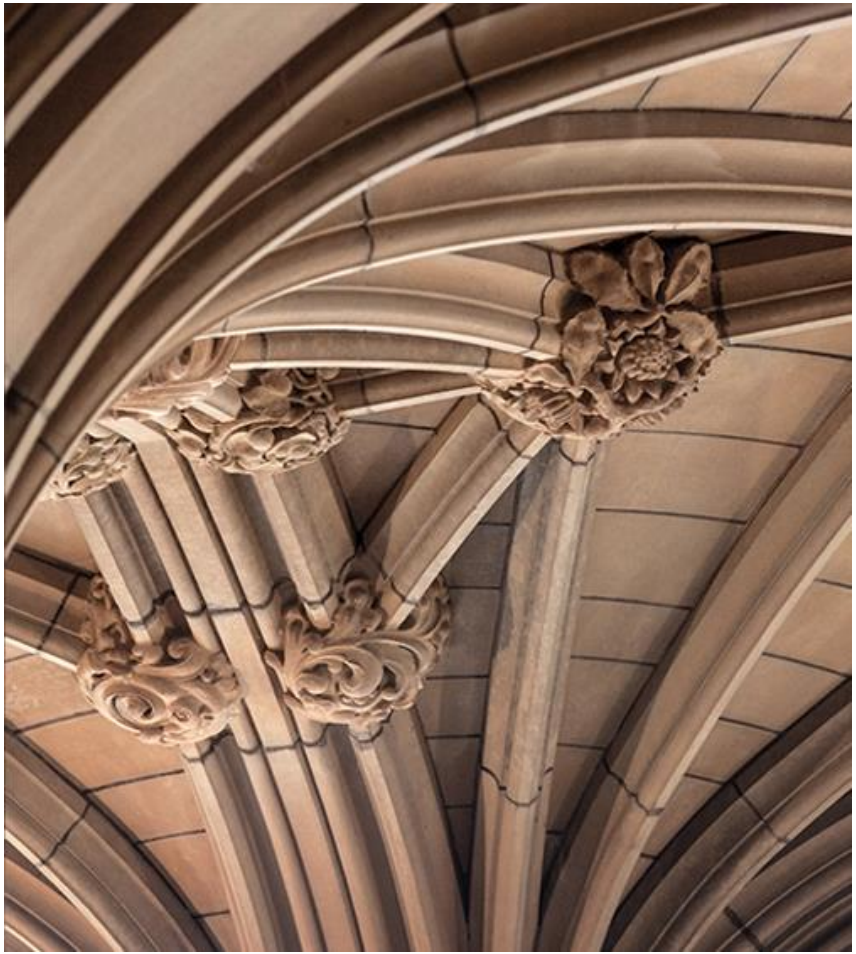


Unit of Work – Consequences

- Keep all information in one place
- Reduce number of database calls
- Handle business operation that span several transactions
- Alternatives can be impractical or difficult to manage
  - Save any object whenever is changed
  - Use variables to track objects that change
  - Setting and finding objects through a special flag (dirty flag)

Lazy Load

Object Relational Behavioural Pattern



THE UNIVERSITY OF SYDNEY

Lazy Load

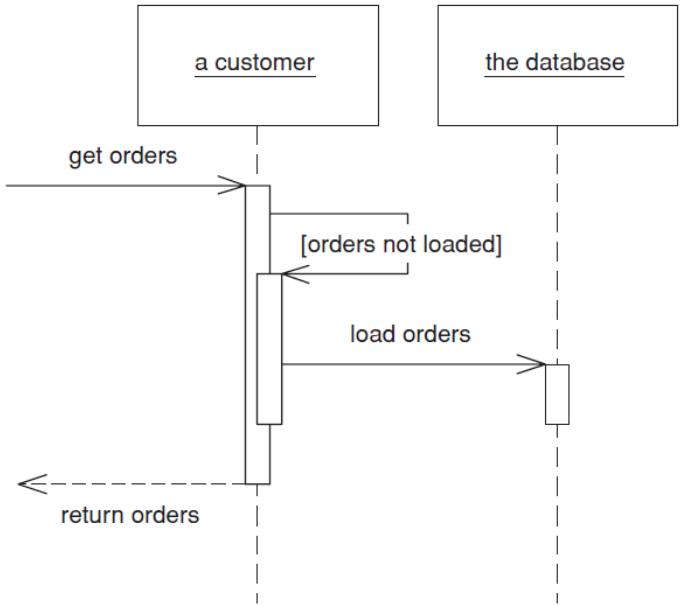
"An object that doesn't contain all of the data you need but knows how to get it."

Lazy Load – Applicability

- Depends on how much data need to retrieve from the database and how many calls required to achieve that
- When the field requires extra database call(s)
- When you need to retrieve everything in one call, especially if it corresponds to a single interaction
- Not useful when a field is stored in the same row as the rest of the object

Lazy Load – How it Works

- Four ways to implement Lazy Load:
  - Lazy initialization
  - Virtual Proxy
  - Value Holder
  - Ghost



Lazy Load – Lazy Initialization

- Use "*null*" to indicate That a field has not been loaded
  - Every access to the field requires a "is-null" check
  - A field with null means the value will need to be calculated before returning the field
- The field should be self-encapsulated
  - Access through getting method
- Use another check in case null is a legal field value

Lazy Initialization – Implementation

- Access of the products field for the first time cases the data to be loaded from the database

```
1 class Supplier...
2 public List getProducts() {
3 if (products == null)
4     products = Product.findForSupplier(getID());
5 return products;
6 }
```

Lazy Load – Virtual Proxy

- Object that looks like the object that should be in the field but it doesn't contain anything
- Virtual Proxy looks like exactly the same object suppose to be there
- Object identities issue
  - Multiple virtual proxies (with different identities) for the same real object
- Dealing with many virtual proxies in statically-typed languages

Virtual Proxy – Implementation (1)

- List of products for a supplier to be held with a regular list field

```
1 class SupplierVL {
2 private List products;
3 //...
4 }
```

- List proxy setup to provide the list that has to be created when it's accessed

```
6 public interface VirtualListLoader {
7 List load();
8 }
```

Virtual Proxy – Implementation (2)

- Virtual list instantiation with a loader that calls the appropriate loader

```
10 class SupplierMapper {
11     //...
12     public static class ProductLoader implements VirtualListLoader {
13         private Long id;
14 
15         public ProductLoader(Long id) {
16         this.id = id;
17         }
18         public List load() {
19             return ProductMapper.create().findForSupplier(id);
20         }
21     }
22 }
```

Virtual Proxy – Implementation (3)

- Product loader assignment to the list field

```
24 class SupplierMapper {
25     //...
26     protected DomainObject doLoad(Long id, ResultSet rs) throws SQLException {
27         String nameArg = rs.getString(2);
28         SupplierVL result = new SupplierVL(id, nameArg);
29         result.setProducts(new VirtualList(new ProductLoader(id)));
30         return result;
31     }
32 }
```

Virtual Proxy – Implementation (4)

- Evaluate the loader on the first reference

```
class VirtualList{
    private List source;
    private VirtualListLoader loader;
    public VirtualList(VirtualListLoader loader) {
        this.loader = loader;
    }
```

Virtual Proxy – Implementation (5)

- Regular list methods

```
41      private List getSource() {
42      if (source == null) source = loader.load();
43          return source;
44      }
45
46      public int size() {
47          return getSource().size();
48      }
49      public boolean isEmpty() {
50          return getSource().isEmpty();
51      }
52      // ... and so on for rest of list methods
53  }
```

Lazy Load – Value Holder

- An object that wraps some other object
- To get the underlying object, ask the value holder for its value on the first access
- Lose explicit strong typing
  - Class needs to know that it is present
- To avoid identity problems, ensure that the value holder is never passed out beyond its owning class

Lazy Load – Ghost

- A real object in a partial state
- An object where every field is lazy-initialized in one go
- When an object is loaded from the database it contains just its ID. It's full state is loaded whenever an access to its filed is attempted

Lazy Load – Consequences (1)

- Virtual proxy/ghost can hold some data
  - Data quick to retrieve and commonly used
- Issues with inheritance (in statically typed languages)
  - Can't decide the type of ghost/virtual proxy to create until the object is loaded properly
- Lazy load may lead to database access more than needed
  - Collection with lazy loads and read them one at a time (*ripple loading*)
  - Make the collection itself as a lazy load and load all the contents
  - That doesn't scale with very large collections

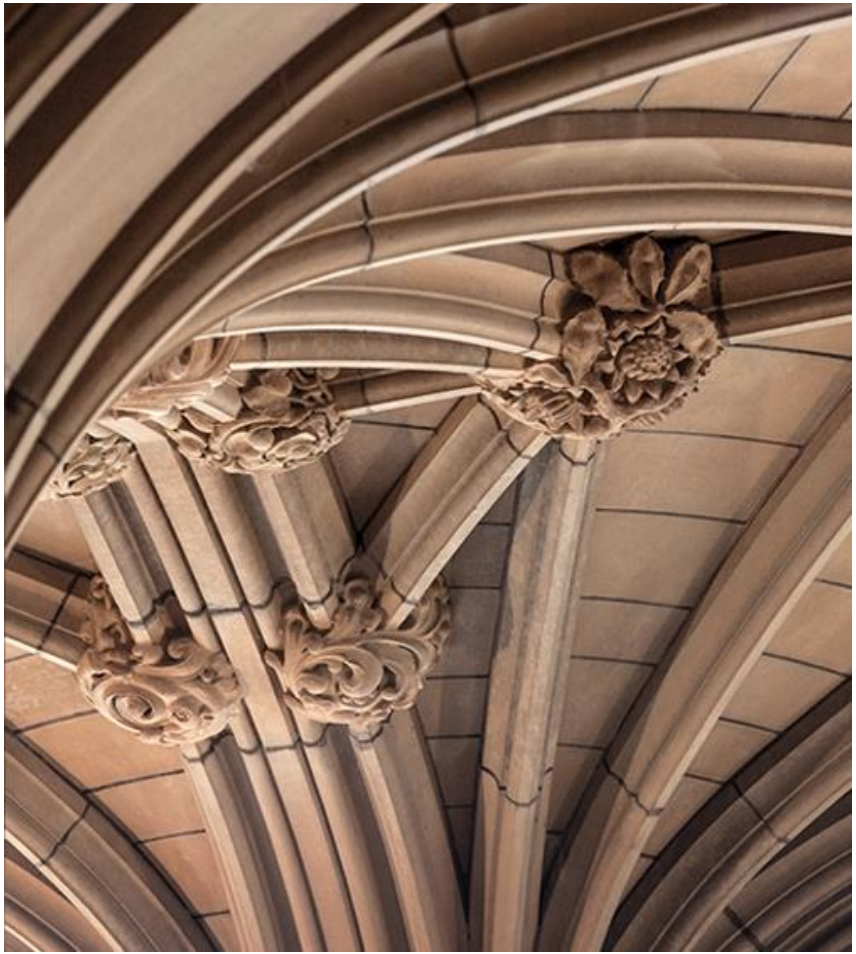
Lazy Load – Consequences (2)

- Dealing with different degrees of laziness (which subset of the object fields is needed)

- Use separate database interaction objects (one for each common scenario)
  - E.g., two order mapper objects; one that loads line items immediately and one loads it lazily
  - More varieties of laziness, complexity overweight benefits
  - Practically two: complete load and just enough load for identification

Value Object

Object Behavioural



THE UNIVERSITY OF SYDNEY

Value Object

- What is the difference between value object and reference objects? Give examples

Value Object Vs. Reference Object

- Based on how they deal with equality
  - Reference object uses identity for equality
    - Have same Id (e.g., built-in Id, primary key)
  - Value object uses field values
    - E.g., two date objects are equal if they have the same values for their day, months and year

- Value objects are passed by value and reference objects are passed by reference

- Most Programing languages do not support value objects. What are the consequences? How to define them?

Value Object Vs. Reference Object

- Aliasing bugs
  - Two objects share the same value object and one of the owners changes the values in it

- Make value objects immutable
  - None of their fields should change once created

Value Object

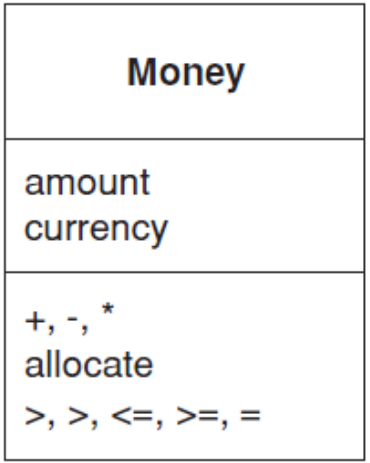
- "*A small simple object whose equality isn't based on identity*"
- Applicability
  - When you want to use equality on something other than identity

Value Object Example – Money

- Money represents a monetary value
- Many issues such as
  - Multiple currencies: e.g., cannot add dollars to yen
  - Rounding errors: losing cents/pennies

Value Object – Example

- Fields should be defined appropriately
  - The *amount* as integral or a fixed decimal type – a float type might introduce rounding problems
  - Monetary values either rounded to the smallest complete unit or with fractional units
- Arithmetic operations should be currency-aware
  - Addition and subtraction
  - Rounding issues in division and multiplication
  - E.g., to add %5 tax, multiply it by 0.05



Value Object Example – Money

- Business rule: allocate the whole amount of a sum of money to 2 accounts:
  - 70% to Acc1 and
  - 30% to Acc2

- How would an application allocate 5 cents?

Value Object Example – Money

- Allocator function
  - Take ratio parameter as a list of numbers
  - Returns list of monies without dropping scents
  - A rule to enforce how to deal with allocation

Value Object Example – Money

- Currency conversion
  - Simple solution Multiply by exchange rate
  - But there might be some rounding issues – conversion rules may have specific rounding
  - Instead use converter object to encapsulate the conversion algorithm

- Money comparison
  - Compare different currencies require currency conversion

Value Object Example – Money

- How to storing money in a database?
  - As embedded value; currency for every money?
  - Store the currency on the account and pull it whenever you load entries

- How to display money on a UI?
  - Use printing behavior to display the correct amount with appropriate currency

Money Example – Implementation

```
1  class Money{
2      private long amount;
3      private Currency currency;
4
5      public Money(double amount, Currency currency) {
6          this.currency = currency;
7          this.amount = Math.round(amount * centFactor());
8      }
9
10     public Money(long amount, Currency currency) {
11         this.currency = currency;
12         this.amount = amount * centFactor();
13     }
14
15     private static final int[] cents = new int[] { 1, 10, 100, 1000 };
16         private int centFactor() {
17         return cents[currency.getDefaultFractionDigits()];
18     }
19 }
```

Money Example – Implementation

- Access the underlying data (amount)

```
19      public BigDecimal amount() {
20          return BigDecimal.valueOf(amount, currency.getDefaultFractionDigits());
21      }
22      public Currency currency() {
23          return currency;
24      }
```

- Accessors

```
25      public static Money dollars(double amount) {
26          return new Money(amount, Currency.USD);
27      }
```

Money Example – Implementation

- Money equality

```
29      public boolean equals(Object other) {
30          return (other instanceof Money) && equals((Money)other);
31      }
32      public boolean equals(Money other) {
33          return currency.equals(other.currency) && (amount == other.amount);
34      }
35      public int hashCode() {
36          return (int) (amount ^ (amount >>> 32));
37      }
```

Money Example – Implementation

- Money equality

```
29      public boolean equals(Object other) {
30          return (other instanceof Money) && equals((Money)other);
31      }
32      public boolean equals(Money other) {
33          return currency.equals(other.currency) && (amount == other.amount);
34      }
35      public int hashCode() {
36          return (int) (amount ^ (amount >>> 32));
37      }
```

Money Example – Implementation

- Money Addition

```
38      public Money add(Money other) {
39          assertSameCurrencyAs(other);
40          return newMoney(amount + other.amount);
41      }
42      private void assertSameCurrencyAs(Money arg) {
43          Assert.equals("money math mismatch", currency, arg.currency);
44      }
45      private Money newMoney(long amount) {
46          Money money = new Money();
47          money.currency = this.currency;
48          money.amount = amount;
49          return money;
50      }
```

Money Example – Implementation

- Money comparison

```
51      public int compareTo(Object other) {
52          return compareTo((Money)other);
53      }
54      public int compareTo(Money other) {
55          assertSameCurrencyAs(other);
56      if (amount < other.amount)
57          return -1;
58      else if (amount == other.amount)
59          return 0;
60      else
61          return 1;
62      }
```

Money Example – Implementation

- Money Multiplication

```
64      public Money multiply(double amount) {
65          return multiply(new BigDecimal(amount));
66      }
67      public Money multiply(BigDecimal amount) {
68          return multiply(amount, BigDecimal.ROUND_HALF_EVEN);
69      }
70      public Money multiply(BigDecimal amount, int roundingMode) {
71          return new Money(amount().multiply(amount), currency, roundingMode);
72      }
```

Money Example – Implementation

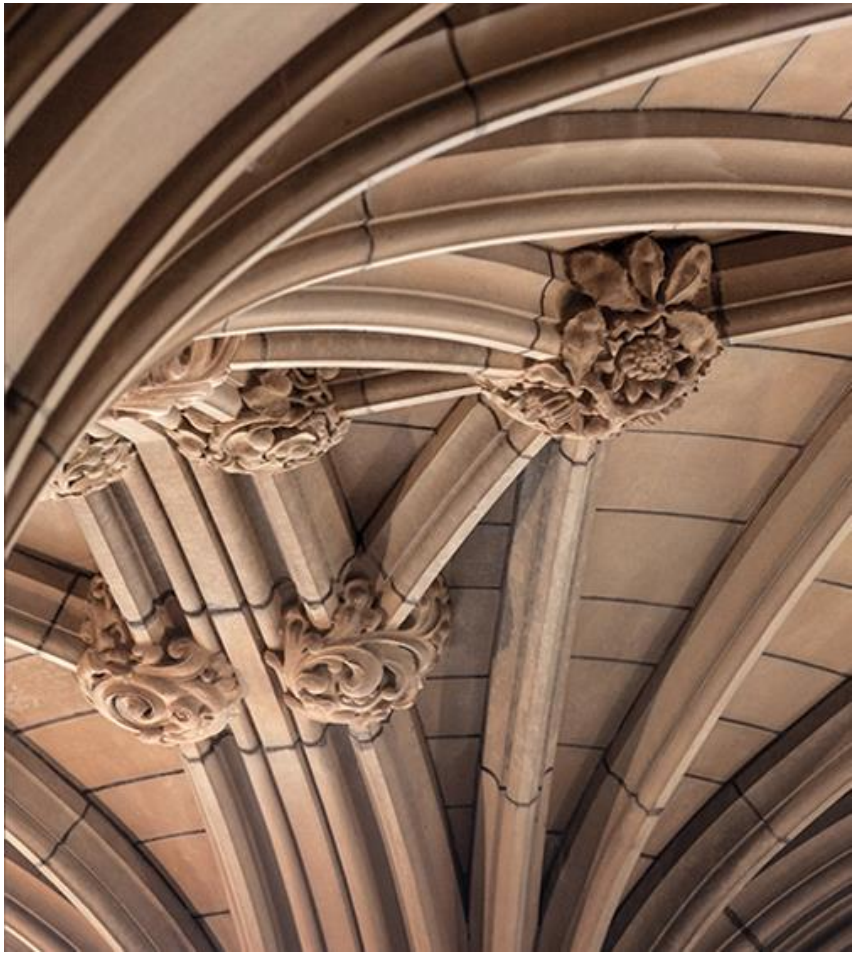
- Allocate sum of money among many targets without losing cents

```
73      public Money[] allocate(int n) {
74          Money lowResult = newMoney(amount / n);
75          Money highResult = newMoney(lowResult.amount + 1);
76          Money[] results = new Money[n];
77          int remainder = (int) amount % n;
78          for (int i = 0; i < remainder; i++)
79              results[i] = highResult;
80          for (int i = remainder; i < n; i++)
81              results[i] = lowResult;
82          return results;
83      }
```

Value Object – Consequences

- Performance improvements
  - Specifically with binary serializing

Lazy Loader - Value Holder Implementation



Value Holder – Implementation (1)

- Product field typed as a value and use *getProducts()* to hide this from the clients

```
1 class SupplierVH {
2     private ValueHolder products;
3     public List getProducts() {
4         return (List) products.getValue();
5     }
6 }
```

Value Holder – Implementation (2)

- Value holder performs the lazy initialization

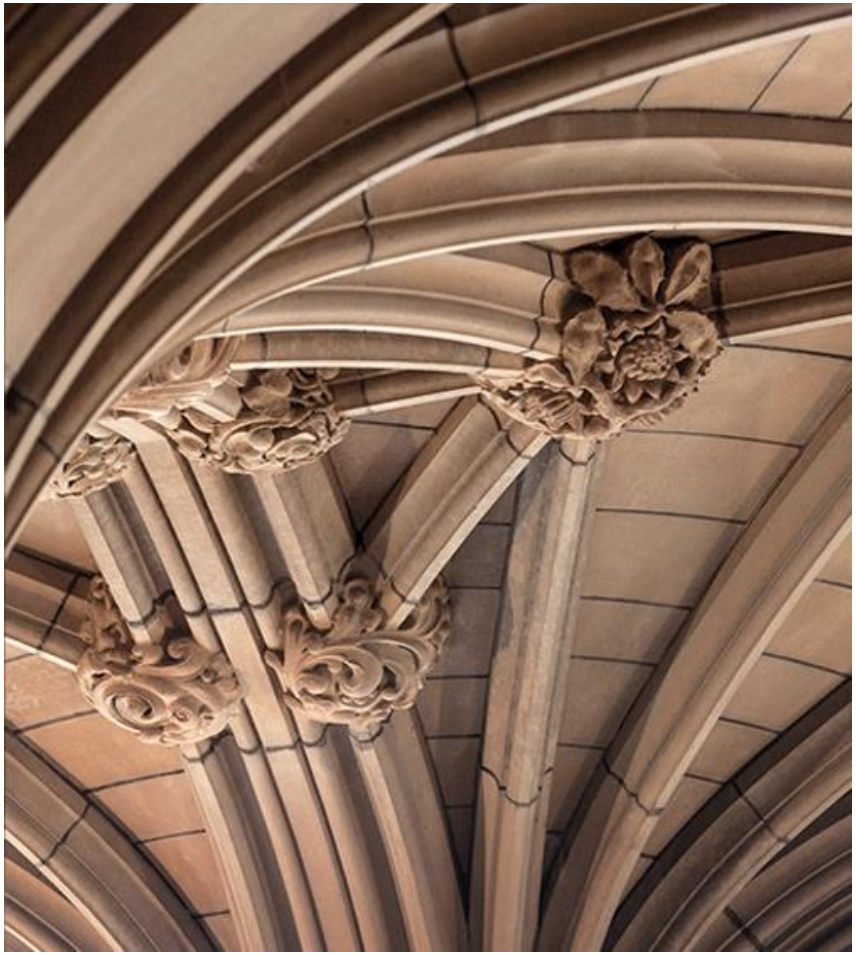
```
 8 class ValueHolder {
 9     //....
10
11     private Object value;
12     private ValueLoader loader;
13     public ValueHolder(ValueLoader loader) {
14         this.loader = loader;
15     }
16     public Object getValue() {
17         if (value == null)
18             value = loader.load();
19             return value;
20     }
21     public interface ValueLoader {
22         Object load();
23     }
24 }
```

Value Holder – Implementation (3)

- Value holder performs the lazy initialization

```
26 class SupplierMapper {
27     protected DomainObject doLoad(Long id, ResultSet rs) throws SQLException {
28         String nameArg = rs.getString(2);
29         SupplierVH result = new SupplierVH(id, nameArg);
30         result.setProducts(new ValueHolder(new ProductLoader(id)));
31         return result;
32     }
33     public static class ProductLoader implements ValueLoader {
34         private Long id;
35         public ProductLoader(Long id) {
36         this.id = id;
37     }
38     public Object load() {
39         return ProductMapper.create().findForSupplier(id);
40     }
41     }
42 }
```

Unit of Work – Implementation



THE UNIVERSITY OF SYDNEY

Unit of Work – Implementation (1)

- Unit of work that can track all changes for a given business transaction and then commit them to the database when required

- Registration methods

```
1  class UnitOfWork {
2      private List newObjects = new ArrayList();
3      private List dirtyObjects = new ArrayList();
4      private List removedObjects = new ArrayList();
5
6      public void registerNew(DomainObject obj) {
7          Assert.notNull("id not null", obj.getId());
8          Assert.isTrue("object not dirty", !dirtyObjects.contains(obj));
9          Assert.isTrue("object not removed", !removedObjects.contains(obj));
10         Assert.isTrue("object not already registered new", !newObjects.contains(obj));
11         newObjects.add(obj);
12     }
```

Unit of Work – Implementation (2)

- Registration methods

```
14      public void registerDirty(DomainObject obj) {
15          Assert.notNull("id not null", obj.getId());
16          Assert.isTrue("object not removed", !removedObjects.contains(obj));
17          if (!dirtyObjects.contains(obj) && !newObjects.contains(obj)) {
18              dirtyObjects.add(obj);
19          }
20      }
21      public void registerRemoved(DomainObject obj) {
22          Assert.notNull("id not null", obj.getId());
23          if (newObjects.remove(obj))
24              return;
25          dirtyObjects.remove(obj);
26          if (!removedObjects.contains(obj)) {
27              removedObjects.add(obj);
28          }
29      }
30      public void registerClean(DomainObject obj) {
31          Assert.notNull("id not null", obj.getId());
32      }
```

Unit of Work – Implementation (3)

- Commit method to locate data mapper for each object and invoke appropriate mapping method

```
33      public void commit() {
34          insertNew();
35          updateDirty();
36          deleteRemoved();
37      }
38      private void insertNew() {
39          for (Iterator objects = newObjects.iterator(); objects.hasNext();) {
40              DomainObject obj = (DomainObject) objects.next();
41              MapperRegistry.getMapper(obj.getClass()).insert(obj);
42      }
43  }
```

Unit of Work – Implementation (4)

- Facilitate object registration

```
44      // ...
45      private static ThreadLocal current = new ThreadLocal();
46      public static void newCurrent() {
47          setCurrent(new UnitOfWork());
48      }
49      public static void setCurrent(UnitOfWork uow) {
50          current.set(uow);
51      }
52      public static UnitOfWork getCurrent() {
53          return (UnitOfWork) current.get();
54      }
```

Unit of Work – Implementation (5)

- Methods to register objects as unit of work

```
56 class DomainObject {
57     protected void markNew() {
58         UnitOfWork.getCurrent().registerNew(this);
59     }
60     protected void markClean() {
61         UnitOfWork.getCurrent().registerClean(this);
62     }
63     protected void markDirty() {
64         UnitOfWork.getCurrent().registerDirty(this);
65     }
66     protected void markRemoved() {
67         UnitOfWork.getCurrent().registerRemoved(this);
68     }
69 }
```

Unit of Work – Implementation (5)

– Domain objects need to mark themselves new and dirty where appropriate

```
class Album {
    public static Album create(String name) {
        Album obj = new Album(IdGenerator.nextId(), name);
        obj.markNew();
        return obj;
    }
public void setTitle(String title) {
    this.title = title;
    markDirty();
}
```

Unit of Work – Implementation (6)

- Register and commit unit of work

```
81 class EditAlbumScript {
82     public static void updateTitle(Long albumId, String title) {
83         UnitOfWork.newCurrent();
84         Mapper mapper = MapperRegistry.getMapper(Album.class);
85         Album album = (Album) mapper.find(albumId);
86         album.setTitle(title);
87         UnitOfWork.getCurrent().commit();
88     }
89 }
```

References

- Martin Fowler (With contributions from David Rice, Matthew Foemmel, Edward Hieatt, Robert Mee, and Randy Stafford). 2003. *Patterns of Enterprise Applications Architecture*. Pearson.

- Enterprise-Scale Software Architecture (COMP5348). Slides 2018.

W8 Tutorial: Practical Exercises/coding + quiz
W8 Lecture: Enterprise Design Patterns
Design Pattern Assignment

THE UNIVERSITY OF SYDNEY