INTRODUCTION TO SOFTWARE DESIGN

FROM ANALYSIS TO DESIGN, ENTERPRISE APPLICATION ARCHITECTURE (EAA), PERSISTENT LAYER OF EAA
WHAT TO EXPECT FROM SOFTWARE DESIGN

Problem analysis (Komarek)

- Analysis gave us a real-world view/perspective to the problem
- No software constraints were considered
- Objects/concepts of domain

Software design (Cerny)

- Finding an appropriate software structure to capture the problem
- Determining architecture
- Design objects reflect software constraints
  - Data types, structural decomposition, databases, networks
  - Artificial objects that do not have anything to do with real world
WHAT TO EXPECT FROM SOFTWARE DESIGN

Problem analysis

- System is understood as a black box (BB)
- Analysis described input and output expected from the BB
- Black box uses Domain terminology

Software design

- Opens the BB and aims to design what is inside
- Multiple approaches to do that influenced by many factors
WHAT TO EXPECT FROM SOFTWARE DESIGN

Problem analysis
• We know what customer wants
• Vision
• Functional requirements
• Domain model
• Scenarios (use cases)
• Business models

Software design
• Motivation
  • Do not reinvent the wheel
  • Reuse what possible
  • Design something easy to comprehend, easy to read
  • Provide the expected functionality
WHAT TO EXPECT FROM SOFTWARE DESIGN

Problem analysis

- We know what customer wants
- Vision
- Functional requirements
- Domain model
- Scenarios (use cases)
- Business models

Software design

- Object oriented, Component based, Industry standards!
AVOID MISTAKES! FLAWS AND BUGS

Relative costs to fix an issue

Cheap

Hell expensive

Tomas Cerny, Software Engineering, FEE, CTU in Prague, 2016
THINK TWICE BEFORE YOU START

When designing software

• Expect long live
• Expect you work in team
  • what you do, should be understood by others
• When you design something, make it easy for other to understand it
• Expect that other spend most of the time reading how it works!
  • Abstraction!!
  • Interfaces!!
  • Annotations!!
MOTIVATION EXAMPLE

```
Fabia a(ArrayList<Fabia> list) {
    Fabia f = list.get(0);
    f.produce();
    f.test();
    f.ship();
    return f;
}

Car b(List<Car> list) {
    Car c = list.get(0);
    c.produce();
    c.test();
    c.ship();
    return c;
}
```

Be abstract!
- List vs. ArrayList
- Car vs. Fabia
- Virtual methods!
  - Object-oriented
Be **abstract!**
- If you do you get A!

```java
Fabia a(ArrayList<Fabia> list) {
    Fabia f = list.get(0);
    f.produce();
    f.test();
    f.ship();
    return f;
}
```

```java
Car b(List<Car> list) {
    Car c = list.get(0);
    c.produce();
    c.test();
    c.ship();
    return c;
}
```
THINK TWICE BEFORE YOU START

Numbers/Statistics

- Software maintenance
  - 65-75% of life-cycle

- Understanding to the software
  - 40-60% of the entire maintenance!

- User Interface
  - Very expensive!
  - 50% of the entire development of EA

Consider these before designing your home nuclear power plant.
ASSUME NO ONE IS PERFECT, BUT CHARGE THE CUSTOMER
VISUALIZATION OF FIRST STEPS IN DESIGN

• Keywords
  • Use Case (Scenario)
  • Domain Model
  • System Sequence Diagram
  • System Operation
  • Operation Contract
  • Design Model
VISUALIZATION

ANALYSIS OUTCOME

• Domain model
  • Real-world representation of concepts
  • No software constraints

• Scenarios
  • Fully dressed Use Cases
  • Capture requirements
  • Interaction of user and black box
VISUALIZATION
ANALYSIS OUTCOME

- Domain model

- Scenarios
VISUALIZATION
ANALYSIS OUTCOME

• Domain model

• Scenarios
VISUALIZATION DESIGNING

- Domain model

- Scenarios

User – system interaction
VISUALIZATION DESIGNING

• Domain model

• Scenarios

User – system interaction
VISUALIZATION
DESIGNING

- Domain model

- Scenarios
OUTCOME

SYSTEM SEQUENCE DIAGRAM \textsuperscript{SSD}

- Domain model

- Scenarios

System sequence diagram (SSD)
Special usage of UML Sequence diagram to describe communication with black box and user
OUTCOME
SYSTEM OPERATION

- Domain model

- Scenarios

SSD helps to deduce System Operations

System Operation
Function called from outside of the system that is processed by the black box to get required results

Tomas Cerny, Software Engineering, FEE, CTU in Prague, 2016
OUTCOME
SYSTEM OPERATION

- Domain model
- Scenarios

System operation
Function called from outside of the system that is processed by the black box to get required results
EXAMPLE:
USE CASE SCENARIO TO SSD

Use case:
1. **Uživatel** odešle dotaz na cenu košíku
2. **Systém** zpracuje košík, dohledá ceny a sečte kvantitu vynásobenou cenou a prezentuje uživateli **cenu košíku**
EXAMPLE:
USE CASE SCENARIO TO SSD

Use case:
1. **Uživatel** odešle dotaz na cenu košíku
2. **Systém** zpracuje košík, dohledá ceny a sečte kvantitu vynásobenou cenou a prezentuje uživateli cenu košíku

SSD:

```
:Uživatel

Dotaz na cenu() → Cena košíku

:System
```
**EXAMPLE:**

**USE CASE SCENARIO** TO SSD

**Use case:**
1. **Uživatel** odešle dotaz na cenu košíku
2. **Systém** zpracuje košík, dohledá ceny a sečte kvantitu vynásobenou cenou a prezentuje uživateli cenu košíku

**SSD:**

- **System**
  - **Uživatel**
  - **Dotaz na cenu()**
  - **Cena košíku**
  - **getTotelCharge()**
    - ..
    - ..
EXAMPLE II: USE CASE SCENARIO TO SSD

Simple cash-only Process Sale Scenario

1. Customer arrives at a POS checkout with goods to purchase.
2. Cashier starts a new sale.
3. Cashier enters item identifier.
4. System records sale line item, and presents item description, price and running total.
   cashier repeats steps 3-4 until indicates done.
5. System presents total with taxes calculated.
6. Get payment
EXAMPLE II: USE CASE SCENARIO TO SSD

Simple cash-only Process Sale Scenario

1. Customer arrives at a POS checkout with goods to purchase.
2. Cashier starts a new sale.
3. Cashier enters item identifier.
4. System records sale line item, and presents item description, price and running total. Cashier repeats steps 3-4 until indicates done.
5. System presents total with taxes calculated.
6. Get payment
EXAMPLE II: SSD TO SYSTEM OPERATION

The set of all required system operations is determined by identifying the system events.

- `makeNewSale()`
- `addLineItem(itemID, quantity)`
- `endSale()`
- `makePayment(amount)`

In the UML the system as a whole can be represented as a class.

<table>
<thead>
<tr>
<th>System</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>makeNewSale()</code></td>
</tr>
<tr>
<td><code>addLineItem(itemID, quantity)</code></td>
</tr>
<tr>
<td><code>endSale()</code></td>
</tr>
<tr>
<td><code>makePayment()</code></td>
</tr>
</tbody>
</table>
TOWARDS DESIGN
OPERATION CONTRACTS

- Domain model

- Scenarios

Operation Contracts
Contracts are documents that describe system behavior.
EXAMPLE II: OPERATION CONTRACTS

Contracts **may be defined for system operations**

- Operations that the system (as a black box) offers in its public interface to handle incoming system events.

<table>
<thead>
<tr>
<th>System</th>
</tr>
</thead>
<tbody>
<tr>
<td>makeNewSale()</td>
</tr>
<tr>
<td>addLineItem(itemID, quantity)</td>
</tr>
<tr>
<td>endSale()</td>
</tr>
<tr>
<td>makePayment()</td>
</tr>
</tbody>
</table>
**EXAMPLE II:**

**OPERATION CONTRACT:** \(\text{addLineItem}\)

**Contract CO2: addLineItem**

**Operation:** \(\text{addLineItem (itemID: ItemID, quantity: integer)}\)

**Cross References:** Use Cases: Process Sale.

**Pre-conditions:** There is a sale underway.

**Post-conditions:**

- A **SalesLineItem** instance \(sli\) was created. (instance creation)
- \(sli\) was associated with the **Sale**. (association formed)
- \(sli.quantity\) was set to quantity. (attribute modification)
- \(sli\) was associated with a **ProductSpecification**, based on \(itemID\) match (association formed)
EXAMPLE II: OVERVIEW

Use Case: Process Sale

- makeNewSale()
- addLineItem (itemID, quantity)
- endSale()
- makePayment()

System Operations

- makeNewSale()
- addLineItem (itemID, quantity)
- endSale()
- makePayment()

Contracts

- Operation: makeNewSale
- Operation: addLineItem
- Operation: endSale
- Operation: makePayment
TOWARDS DESIGN MODEL
ASSIGN SYSTEM OPERATIONS TO OBJECTS

- Domain model

- Scenarios
TOWARDS DESIGN

GRASP

• Domain model

• Scenarios

GRASP
General Responsibility Assignment
Software Patterns.
Assign responsibilities to objects
(Which operation to which object)

System
CallX()
CallY()
CallZ()

Contract
CallX

CallZ()

CallY()
GRASP

• Patterns that help us to choose object into which we assign a particular responsibility

• Name / Problem / Solution

1. Information Expert
2. Creator
3. Low coupling
4. High cohesion
5. Controller
6. Polymorphism
7. Pure fabrication
8. Indirection
9. Do not talk to strangers
GRASP

1. Information Expert
   - Most basic principle to assign responsibility

2. Creator
   - Who should create new instances

3. Low coupling
   - Support low dependency, low change impact, increase reuse

4. High cohesion
   - Keep complexity manageable, keep clarity

5. Controller
   - Handle system events

6. Polymorphism
   - Variation of behavior through polymorphic methods

7. Pure fabrication
   - Make artificial object to help (3) a (4)

8. Indirection
   - Get information through a delegation to associated objects

9. Do not talk to strangers
   - Talk to associated objects only, avoid indirect objects
More about grasps next week
OPENING BLACK BOX

Multiple approaches
- Object-oriented design
- Framework
- Procedural decomposition
- Model-driven development

Component-based development

Design pattern
- Objects
- Packages
- Components
- Artifacts

Architectural styles
- Libraries
- Layers
- Constructs
OBJECT-ORIENTED DESIGN OOD

- Object-oriented constructs
  - Generalization, polymorphism, information hiding
  - Delegation, functional decomposition
- Component design
- Layers
- Subsystems
- Best practice
  - Design patterns (Solution to a repeatable problems)
QUALITY ATTRIBUTES

• Runtime
  • Performance
  • Security
  • Availability
  • Reliability
  • Fault-tolerance
• Functionality
  • Usability
  • Availability

• Static
  • Modifiability
  • Readability
  • Integrability
  • Reuse
  • Testability

*See more at http://en.wikipedia.org/wiki/List_of_system_quality_attributes*
ENTERPRISE APPLICATION (EA)

- Enterprise Application (EA)
  - An *enterprise application* is the term used to describe applications -- or software -- that a business would use to assist the organization in solving enterprise problems.
  - When the word "enterprise" is combined with "application," it usually refers to a *software* platform that is too large and too complex for individual or *small business* use.
  
  - Something big that handles a lot of data and business rules
  - Sometimes use term Enterprise System
ENTERPRISE APPLICATION (EA)

• Enterprise Application (EA) Examples
  • Information system of school, hospital
  • Bank transactions
  • Accounting system
  • Flight booking
  • E-commerce

• Components
  • Database
  • Content management
  • Web-services
  • Lot of User Interface to collect and manipulate data
  • Business rules
**ENTERPRISE APPLICATION ARCHITECTURE (EAA) FRAMEWORK**

- Usually the responsibilities divide to layers
- Example: Java EE Framework to build EA with EAA
• Usually application responsibilities divide to layers
• EAA with 3 layers

presentation layer

business layer

persistence layer
EAA in the context of EAA framework
ENTERPRISE APPLICATION ARCHITECTURE (EAA)
ENTERPRISE APPLICATION ARCHITECTURE (EAA)

- **Layers (Bottom up)**
  - **Persistence**
    - Data model, data persistence, data access
  - **Business logic** *(other names)*
    - Interaction, Business flow, Business constraints/rules
    - Dependency injection, validation, calculation
  - **Presentation**
    - User Interfaces, present the above to user make it easy to process task, automate as much as possible
    - *Do not teach users new things*
ENTERPRISE APPLICATION ARCHITECTURE (EAA)

• Layers - Components
  • Persistence
    • Data model – Entity JPA
    • Data Access - Data Access Object (DAO)
  • Business logic
    • Services – integrate business flow/constraints/rules - EJB
    • Dependency injection – CDI (which DAO/Service to use)
    • Patterns – Façade, Bridge
    • Transitions
  • Presentation
    • Controllers – Java Beans, Session Scopes
    • View code – XHTML/JSP/JSF
ENTERPRISE APPLICATION ARCHITECTURE (EAA)

- Layers - Components
  - Persistence
    - Data model – Entity JPA
    - Data access - Data Access Object (DAO)
  - Business logic
    - Services – integrate business flow/constraints/rules - EJB
    - Dependency injection – CDI (which DAO/Service to use)
    - Patterns – Façade, Bridge
  - Presentation
    - Controllers – Java Beans, Session Scopes
    - View code – XHTML/JSP/JSF

Not everything is that easy!
OBJECT-RELATIONAL MAPPING ORM

• Quotes
  • Ted Neward, late 2004
    • "Object-Relational mapping is the Vietnam of our industry”
  • I've seen developers struggle for years with the huge mismatch between relational database models and traditional object models. And all the solutions they come up with seem to make the problem worse.
WHAT IS ORM?

- **Persistence**
  - Helping to achieve persistence – app data outlive the process
- **Relational database**
  - Persistence to RDBMS in Object-oriented app
- **Object-relational mismatch**
  - Paradigm mismatch – object model and relational model incompatible
- **Granularity**
  - More objects than tables in DB (Person>Address)
- **Subtypes**
  - Inheritance in DB?
- **Identity notion**
  - Primary key vs. a == b and a.equals(b) options
- **Associations**
  - Unidirectional/Bidirectional in OOP vs Foreign Key in RDBMS
- **Data navigation**
  - Fundamental difference in OOP and DRBMS
  - Object network vs. JOIN? Efficiency number of SQL queries (lazy load)
OBJECT-RELATIONAL MAPPING

- Databases are relational
  - Relations and SQL
  - No inheritance
  - Good performance and centralization
  - ACID, fast to save and retrieve data = GOOD for Persistence
- Object works
  - Great for problem decomposition
  - Slow to be used for data persistence
  - Designed to deal with problems not to persist data
  - All nice features and instruments of Object-oriented design
    - Polymorphism, Generics..
PERSISTENCE LAYER

Object-relational mapping (ORM)

- Databases are relational RDBMS
- Objects are not
  - Technical incompatibility
- Solution – mapping to both directions
OBJECT-RELATIONAL MAPPING (ORM)

• Create

```java
public void create() {
    Event e = new Event( "First event!", new Date() );
    Session session = sessionFactory.openSession();
    session.beginTransaction();
    session.save(e);
    session.save(new Event( "Second event", new Date() ));
    session.getTransaction().commit();
    session.close();
}
```
OBJECT-RELATIONAL MAPPING (ORM)

• Read

```java
public void read() {
    Session session = sessionFactory.openSession();
    session.beginTransaction();
    List result = session.createQuery("from Event").list();
    for (Event event : (List<Event>) result) {
        out.print("Event "+event.getDate()+":"+event.getTitle());
    }
    session.getTransaction().commit();
    session.close();
}
```
OBJECT-RELATIONAL MAPPING (ORM)

- Entity + annotation

```java
@Entity
@Table( name = "EVENTS" )
public class Event {
    ...
    @Id
    @GeneratedValue("increment")
    @GenericGenerator(
        name="increment",
        strategy = "increment")
    public Long getId() {
        return id;
    }

    public String getTitle() {
        return title;
    }

    @Temporal(TemporalType.TIMESTAMP)
    @Column(name = "EVENT_DATE")
    public Date getDate() {
        return date;
    }
}
```
OBJECT-RELATIONAL MAPPING (ORM)

- Read/Write

```java
Cat fritz = (Cat) sess.load(Cat.class, generatedId);
...
Cat cat = (Cat) sess.get(Cat.class, id);
if (cat==null) {
    cat = new Cat();
    sess.save(cat, id);
}
...
sess.save(cat);
sess.flush(); //force the SQL INSERT
sess.refresh(cat); //re-read the state (after the trigger executes)
...
```
OBJECT-RELATIONAL MAPPING (ORM)

- Inheritance

```java
@Entity
@Inheritance(strategy=
    InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(
    name="planetype",
    discriminatorType=
    DiscriminatorType.STRING
)
@DiscriminatorValue("Plane")
public class Plane { ... }

@Entity
@DiscriminatorValue("A320")
public class A320 extends Plane
{ ... }
```

- Associations

```java
@Entity
public class FlyingObject

    @ManyToOne
    public PropulsionType
    getPropulsion() {
        return altitude;
    }

    @OneToOne(cascade =
        CascadeType.ALL)
    public Navigation getNavi() {
        return navi;
    }
```
ORM FEATURES

- Object property mapping
- Association mapping (One-one, one-many, many-many)
  - Lazy loading
- Inheritance mapping
  - Single/Table per class/Table per concrete class
- Generated Keys
- Cascades
- Locking – Bob and John modify Person with ID = 1
- Hibernate Query Language HQL
- Mapping to many databases
- Cache
- Criteria API
OBJECT-RELATIONAL MAPPING (ORM)

• Issues
  • Needs custom equals and hashCode
  • Too much SQL
    • You can make custom/native query

• More at DOC
DESIGN PATTERNS

• Best practice solution to a particular problem
• Named problem and solution
  • Not relevant to a specific contest

• Good Software Engineer and Developer has good overview of Design Patterns
  • Programming Bible
    • Erich Gamma
    • Martin Fowler
DATA ACCESS OBJECT (DAO)

DESIGN PATTERN

- Problem:
  - How to access persistent objects
  - Persistent in a Data Source
- Object that provides interface to persistence mechanism
- Separates data access
DATA ACCESS OBJECT (DAO)
DESIGN PATTERN

• Design pattern
  • Object that provides interface to persistence mechanism
  • Separates data access
DATA ACCESS OBJECT (DAO)
DESIGN PATTERN

• Interaction
DATA ACCESS OBJECT (DAO) DESIGN PATTERN

• Usual situation
DATA ACCESS OBJECT (DAO) DESIGN PATTERN

• Usual situation
// Interface that all CustomerDAOs must support
public interface CustomerDAO {
    public int insertCustomer(...);
    public boolean deleteCustomer(...);
    public Customer findCustomer(...);
    public boolean updateCustomer(...);
    public RowSet selectCustomersRS(...);
    public Collection selectCustomersTO(...);
    ...
}
public class CloudscapeCustomerDAO implements CustomerDAO {

    public int insertCustomer(...) {
        // Implement insert customer here.
        // Return created customer number
        // or a -1 on error
    }

    public boolean deleteCustomer(...) {
        // Implement delete customer here
        // True - success, false - failure
    }

    public Customer findCustomer(...) {
        // Find a customer using supplied
        // argument values - search criteria
        // Return Transfer Object if found,
        // return null on error / not found
    }

    public boolean updateCustomer(...) {
        // update record here using data
        // from the customerData Object
        // True - success, false - failure
    }

    ...
}
DATA ACCESS OBJECT (DAO)
DESIGN PATTERN

public class Customer implements java.io.Serializable {
    // member variables
    int CustomerNumber;
    String name;
    String streetAddress;
    String city;
    ...

    // getter and setter methods...
    ...
}

Tomas Cerny, Software Engineering, FEE, CTU in Prague, 2016
DATA ACCESS OBJECT (DAO) DESIGN PATTERN

// create the required DAO Factory
DAOFactory cloudscapeFactory = ..

// Create a DAO
CustomerDAO custDAO =
    daoFactory.getCustomerDAO();

// create a new customer
int custNo = custDAO.insertCustomer(...);

// Find a customer object.
Customer cust = custDAO.findCustomer(...);

// modify the values in the Transfer Object. ..
cust.setAddress(...);
cust.setEmail(...);

// update the customer object using the DAO
custDAO.updateCustomer(cust);

// delete a customer object
custDAO.deleteCustomer(...);

// select all customers in the same city
Customer criteria=new Customer();
criteria.setCity("New York");
Collection customersList =
    custDAO.selectCustomersTO(criteria);

// returns customersList
DATA ACCESS OBJECT (DAO)
DESIGN PATTERN

• Defines CRUD
  • Create
  • Read
  • Update
  • Delete

• See more at
  • http://www.oracle.com/technetwork/java/dataaccessobject-138824.html
DATA REPOSITORY
DESIGN PATTERN

• Problem
  • CRUD
  • Look up a particular object.
  • We know the ID, name and need the object.
  • Find all orders from a customer.

  • Concentration of query construction code.
  • Large number of domain classes or heavy querying.

• The goal of Data Repository is to significantly reduce the amount of boilerplate code required to implement data access layers for various persistence stores.
DATA REPOSITORY
DESIGN PATTERN

Concept

• Central Interface <Generics> <Entity,Key>
• CrudRepository
  • CRUD functionality for the entity class that is being managed.

```java
public interface CrudRepository<T, ID extends Serializable>
    extends Repository<T, ID> {

    <S extends T> S save(S entity);
    T findOne(ID primaryKey);
    Iterable<T> findAll();
    Long count();
    void delete(T entity);
    boolean exists(ID primaryKey);
    // ... more functionality omitted.
}
```
DATA REPOSITORY
DESIGN PATTERN

Query methods over declared interface

```java
public interface PersonRepository extends Repository<User, Long> { ... }
```

Declare query methods

```java
List<Person> findByLastname(String lastname);
```

Usage

```java
public class SomeClient {
    @Autowired
    private PersonRepository repository;

    public void doSomething() {
        List<Person> persons = repository.findByLastname("Matthews");
    }
}
```
public interface PersonRepository extends Repository<User, Long> {

    List<Person> findByEmailAddressAndLastname(EmailAddress emailAddress, String lastname);

    // Enables the distinct flag for the query
    List<Person> findDistinctPeopleByLastnameOrFirstname(String lastname, String firstname);
    List<Person> findPeopleDistinctByLastnameOrFirstname(String lastname, String firstname);

    // Enabling ignoring case for an individual property
    List<Person> findByLastNameIgnoreCase(String lastname);

    // Enabling ignoring case for all suitable properties
    List<Person> findByLastNameAndFirstnameAllIgnoreCase(String lastname, String firstname);

    // Enabling static ORDER BY for a query
    List<Person> findByLastNameOrderByFirstnameAsc(String lastname);
    List<Person> findByLastNameOrderByFirstnameDesc(String lastname);
}
Using Pageable and Sort in query methods

Page<User> findByLastname(String lastname, Pageable pageable);

List<User> findByLastname(String lastname, Sort sort);

List<User> findByLastname(String lastname, Pageable pageable);
Sample usage

```java
@Controller
@RequestMapping("/users")
public class UserController {

    @Autowired
    UserRepository repository;

    @RequestMapping
    public String showUsers(Model model, Pageable pageable) {

        model.addAttribute("users", repository.findAll(pageable));
        return "users";
    }
}
```
Data Repository
Design Pattern

Sample usage

```java
@Controller
@RequestMapping("/users")
public class UserController {

    private final UserRepository userRepository;

    @Autowired
    public UserController(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    @RequestMapping("/{id}")
    public String showUserForm(@PathVariable("id") Long id, Model model) {
        User user = userRepository.findOne(id);
        model.addAttribute("user", user);
        return "user";
    }
}
```
DATA REPOSITORY

DESIGN PATTERN

Details: