Architecture:   
  
3 –Tier Layered Pattern:

Architectural (structural) patterns are used to constrain the infrastructure between the architectural components. Amongst the most important architectural requirements are

* to provide different access channels to the system with the different access channels potentially changing over time,
* to be flexible around the persistence infrastructures.

The layered architectural pattern provides an infrastructure which limits access of component within one layer to components which are either in the same or the next lower level layer. One of its strengths is that a can be relatively easily replaced. In particular, one can add further access channels without changing any lower layers within the software system and changing the persistence provider would only require changing the persistence API. Using a layered approach can improve the maintainability of your application and make it easier to scale out when necessary to improve performance. However, separating an application into too few or too many layers can add unnecessary complexity; and can decrease the overall performance, maintainability, and flexibility. Thus, we have decided on using 3-tiers in order to get as much performance as possible.

After careful consideration our group decided to use the Layered pattern because it offers us the following:

* **Abstraction**. Layered architecture abstracts the view of the system as whole while providing enough detail to understand the roles and responsibilities of individual layers and the relationship between them.
* **Encapsulation**. No assumptions need to be made about data types, methods and properties, or implementation during design, as these features are not exposed at layer boundaries.
* **Clearly defined functional layers**. The separation between functionality in each layer is clear. Upper layers such as the presentation layer send commands to lower layers, such as the business and data layers, and may react to events in these layers, allowing data to flow both up and down between the layers.
* **High cohesion**. Well-defined responsibility boundaries for each layer, and ensuring that each layer contains functionality directly related to the tasks of that layer, will help to maximize cohesion within the layer.
* **Reusable**. Lower layers have no dependencies on higher layers, potentially allowing them to be reusable in other scenarios.
* **Loose coupling**. Communication between layers is based on abstraction and events to provide loose coupling between layers.

# 1 Application layer

# 2 Business layer

The **application server(??????)** is the architectural component within which request for image processing is captured and handled. It is the component hosting the application processes layer of the application.

## 2.1 Software architecture requirements

The architectural requirements include the refined quality requirements and the architectural responsibilities. The architectural constraints for this lower level component are the same as for the system as a whole.

### 2.1.1 Quality requirements

Many of the quality requirements for the system needs to propagated to this lower level component.

In particular, the scalability, maintainability, auditability and deployability requirements specified for the system as a whole are directly applicable for the **application server(??????)**. However some of the higher level requirements needs to be refined for this lower level component.

#### 2.1.1.1 Flexibility

The system must be able to adapt to changes made to the data it recieves from the input devices (Cameras). Runtime flexibility/configurability will allow for configuration data changes to be made for accepting different image resolutions. If different cameras are used, no core functionality needs to be replaced, only simple configuration of the resolution.

#### 2.1.1.2 Reliability

System must operate full time, have a high level of availability. Have a high detection rate and low false alarm rate.

#### 2.1.1.3 Security

The system does not have potential users accessing methods which should not be accessed, however only admins/owners of the system must be able to configure the configuration data.

#### 2.1.1.4 Testability

All services offered by the system must be testable through automated unit tests, testing components in isolation using mock objects, and automated integration tests where components are integrated within the actual environment.

These functional tests should verify that the service is provided if all pre-conditions are met and that all post-conditions hold true once the service has been provided.

In addition to functional testing, the quality requirements should also be tested.

## 2.2 Architecture design

### 2.2.1 Tactics

#### 2.2.1.1 Flexibility tactics

#### 2.2.1.2 Maintainability tactics

#### 2.2.1.3 Reliability tactics

#### 2.2.1.4 Security

#### 2.2.1.5 Auditability tactics

#### 2.2.1.6 Testability tactics

### 2.2.2 Application Component Concepts and Constraints

The central concept which will be used to specify application logic (the business processes) are the concepts of

* Service contracts which encodes the requirements for a service and
* Service which encodes the concrete implementation of a service.

These services which encapsulate the processes will be constrained to be stateless, i.e. no state are maintained across service requests. A long living state is maintained in domain objects which are typically persisted and which should not have any business logic.

### 2.2.3 Frameworks and technologies

Framework

* JAVA EE satisfies the following quality requirements:
  + Scalability
    - The research support system does not have particularly stringent scalability requirements. Nevertheless, at times the system can be used by at least many tens of concurrent users. Java-EE uses thread-pooling, object-pooling, connection-pooling and clustering to improve scalability.
  + Reliability
    - Typically good through transaction support, clustering, session replication and support for messaging.
    - Java-EE supports container-managed transactions across multiple transaction-supporting resources with two phase-commit.
    - Similarly, if this service is called from a higher level service which is already under transactional scope, it will be executed within the transaction of the higher level service.
    - In addition to transactions support, Java-EE application servers also commonly support deployment within a clustered environment to further improve reliability (and scalability).
  + Flexibility
    - Java-EE is a component-based framework requiring decoupling through interfaces/contracts.
    - CDI (Context and Dependency Injection) is fully supported within Java-EE with dependency injection requested via @Inject annotations.
    - Hot-deployment is supported by most application servers including the most widely used open-source application servers like JBoss and Glassfish.
  + Performance
    - Average because of layers, and communication overheads.
    - RMI/IIOP is reasonably efficient protocol.
    - Improved through JPA-based object cache.
  + Security
    - Good support for authentication, authorization and confidentiality.
  + Integratibility
    - Quite good with support for CORBA, SOAP-based & Restless web services, DB integration
    - Integration to systems using proprietary protocol via JCA.
  + Testability
    - Java-EE has good support for dependency injection. Furthermore, the architecture (e.g. the fact that the application code is executing within a Java-EE application server) is not hard-coded within the application logic, but any Java-EE specific information is specified via annotations. All application code can thus be executed outside a JavaEE container (e.g. for unit, integration and regression testing). Dependency injection can be used to inject wither mock or actual dependencies allowing for a unit test to be reused for integration and regression testing.
    - Finally, Java-EE-6 onwards includes the requirement for embedded containers.
    - This does allow for bringing up a Java-EE container (as well as an embedded (in-memory) database) within the run-time environment of the unit test.

## Technology:

* Open CV

Open CV uses an algorithm called HOG (Histogram of Oriented Gradients) which can be used to detect any kind of objects, as to a computer, an image is an assortment of pixels and you may extract features regardless of their contents.

HOG and other such feature extractors are methods used to extract relevant information from an image to describe it in a more meaningful way. When you want to detect an object or person in an image with thousands (and maybe millions) of pixels, it is inefficient to simply feed a vector with millions of numbers to a machine learning algorithm as

1. It will take a large amount of time to complete
2. There will be a lot of noisy information (background, blur, lightning and rotation changes) which we do not wish to regard as important

The HOG algorithm, specifically, creates histograms of edge orientations from certain patches in images. A patch may come from an object, a person, meaningless background, or anything else, and is merely a way to describe an area using edge information. This information can then be used to feed a machine learning algorithm such as the classical support vector machines to train a classifier able to distinguish one type of object from another.

The reason HOG has had so much success with people-detection is because a person can greatly vary in color, clothing, and other factors, but the general edges of a person remain relatively constant, especially around the leg area. This does not mean that it cannot be used to detect other types of objects.

**Open CV also provides the following benefits:**

* + Open source, free for both academic and commercial use.
  + It has C, C++, Python and Java interfaces.
  + Supports Windows, Linux, Mac OS, iOS and Android.
  + Provides a strong focus on real-time applications.
  + Provides a range of image processing functionality.

# 3 Persistence Layer

The persistence API provides abstracted access to persistence to a persistence provider (database) whilst remaining decoupled from the database technology as well as the concrete database selected for the system. It also implements a range of tactics in order to concretely address quality requirements required from the persistence domain.

## 3.1 Software architecture requirements

The architectural requirements for the persistence API include the refined quality requirements and the architectural responsibilities. The architectural constraints for this lower level component are the same as for the system as a whole.

### 3.1.1 Quality requirements

Particularly, scalability, reliability, flexibility and maintainability are important for the persistence API.

* Hot deployment
* Scalability
* Security (authorization)
* Scheduling
* CPU access (Thread provision)
* Auditability

## 3.2 Software architecture design for the persistence API

### 3.2.1 Architectural tactics

The persistence API should use:

* Object-relational mapping to reduce code bulk, improve maintainability and allow for decoupling from the persistence provider
* Query mapping from queries across a graph of Java objects onto the database queries used in the selected database technology and provider,
* Object caching to improve scalability and performance,
* Transactions with 2-phase commit to improve reliability of processes, and connection pooling to improve performance and scalability.

### 3.2.2 Application component concepts and constraints

Within the persistence domain the application concepts (i.e. concepts within which the application functionality is specified) are:

* Domain objects (entities) which host long-living state, and
* Queries across object graph of domain objects through which the state information is retrieved and through which the state of the domain objects is modified.

These domain objects are constrained to be devoid of any business processes - they purely hold data.

### 3.2.3 Frameworks and technologies

A JPA (Java Persistence API) provider will be used as a persistence API. The default concrete implementation packaged with the chosen application server (EclipseLink in the case of Glassfish) will be used. The persistence context (EntityManager) will be dependency injected into services requiring access to persistent data. JPA provider do implement

* Object-relational mapping (ORM) including mapping of relationships between objects via a provided ORM,
* Query mapping from object-oriented queries across the domain objects graph to queries for a specific database provider (e.g. onto SQL),
* Object caching (within the persistence context),
* Transaction support though the Java Transaction API, and
* Connection pooling through a JCA connector based implementation of a JDBC driver.

Queries will be specified as JPQL (Java Persistence Query Language) queries which are queries across Java entities (domain objects).