türkiye c++ topluluğu etkinlikleri

modern c++ ile sürdürülebilir yazılım mimarisi

—

hüseyin kutluca

yazılım mimarı

—

23 haziran perşembe
saat 20:30

Content

- Sustainable Code
- Clean Architecture
- Task Design
- Development Test
- Design Your Service Using C4 Diagrams and UML

SUSTAINABILITY : Ability to maintain or support a process continuously over time

Sustainable SW Development-Definition

- ‘software you use today will be available - and continue to be improved and supported - in the future’

- Concerned with concepts of availability, extensibility, and the maintainability of the software
  - Probably in longer term Thus, a long-living software system usually needs to **operate longer than its technical infrastructure**, which for example consists of COTS, middleware, operating systems, and databases

- During Development Improve development tools and methods so that
  - Technical debt and poor code quality do not end up in version control and production.
  - Extensive automated testing of the code,
  - Inspections involving the entire team,
  - improve quality, reduce regression and speed up production

- Cost-efficiently developed, maintained and evolved over its entire life-cycle.

Sustainable Code

How to Sustainable Code

- Follow Clean Code Principals
- Use Modern C++ Features
- Use Coding Standards
- Use Clang Tidy and Clang-Format with IDE
- Use Sonar Cube

- Modern C++ code is **simpler, safer, more elegant**, and still as fast as ever.
- Use Modern C++ Features
  - Avoid Explicit New and Delete
    - Prefer simple object construction on the stack instead of on the heap
    - In a function's argument list, use (const) references instead of pointers
    - To allocate a resource on the heap, use make_shared and make_unique
  - Use Standard Types and Standard Libraries
    - uint16_t, std::vector, std::thread, std::mutex
  - Use auto keyword
  - Use C++ casts Instead of Old C-Style Casts
  - Avoid Macros

- Limit number of parameters

  The ideal number of arguments for a function is zero (**niladic**).

  Next comes one (**monadic**), followed closely by two (**dyadic**).

  Three arguments (**triadic**) should be **avoided** where possible.

  More than three (**polyadic**) requires very special justification — and then **shouldn't be used anyway.**

- Avoid Output Arguments

  For more than one parameter returns use std::tuple AND std::make_tuple or struct

- Avoid Flag Arguments
  - createInvoice(const BookingItems& items, const bool withDetails)
- Don't Pass or Return 0 (NULL, nullptr)

Rule of 30

- **Rule of 30" in :Refactoring in Large Software Projects by Martin Lippert and Stephen Roock**
- If an element consists of **more than 30 subelements**, it is highly probable that there is a serious problem:

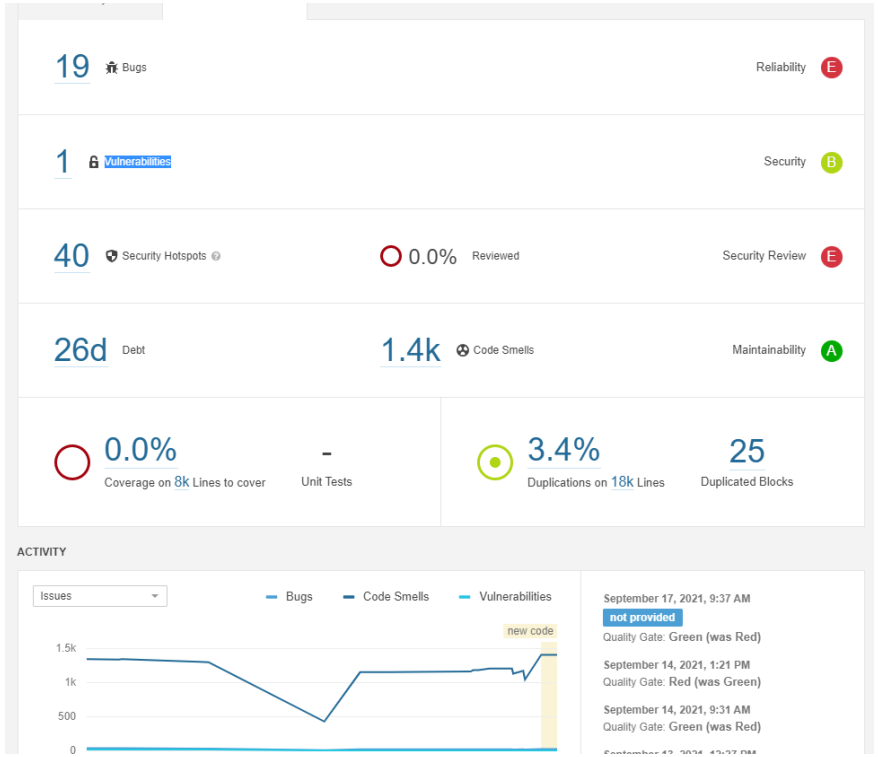
- **Methods** should not have more than an average of 30 code lines (not counting line spaces and comments).
- A **class** should contain an average of less than 30 methods, resulting in up to 900 lines of code.
- A **package** shouldn't contain more than 30 classes, thus comprising up to 27,000 code lines.
- **Subsystems** with more than 30 packages should be avoided. Such a subsystem would count up to 900 classes with up to 810,000 lines of code.
- A **system** with 30 subsystems would thus possess 27,000 classes and 24.3 million code lines

- Use Clang Tidy and Clang-Format
  - Optimize your code
    - Pass by reference
  - Shows and corrects naming rules
  - Autoformats according to indentation rules
  - Shows most of the basic problems. (Unused variables etc.)
  - Checks
    - bugprone-*,
    - google-*,
    - misc-*,
    - performance-*,
    - portability-*,
    - readability-*,

- Modernize your code
  - modernize-avoid-bind
  - modernize-deprecated-headers
  - modernize-loop-convert
  - modernize-make-shared
  - modernize-make-unique
  - modernize-pass-by-value
  - modernize-raw-string-literal
  - modernize-redundant-void-arg
  - modernize-replace-auto-ptr
  - modernize-shrink-to-fit
  - modernize-use-auto
  - modernize-use-bool-literals
  - modernize-use-default
  - modernize-use-emplace
  - modernize-use-nullptr
  - modernize-use-override
  - modernize-use-using

Language

- Use sonar cube from the beginning of project
  - Bugs
  - Vulnerabilities
  - Security Hotspots
  - Code Smells



Roughly 80% of software defects when using the C or C++ language, are attributable to the incorrect usage of 20% of the language constructs

If the usage of the language can be restricted to avoid this subset that is known to be problematic, then the quality of the ensuing software is going to greatly increase

If you don't want defects in your code, then don't put them there!

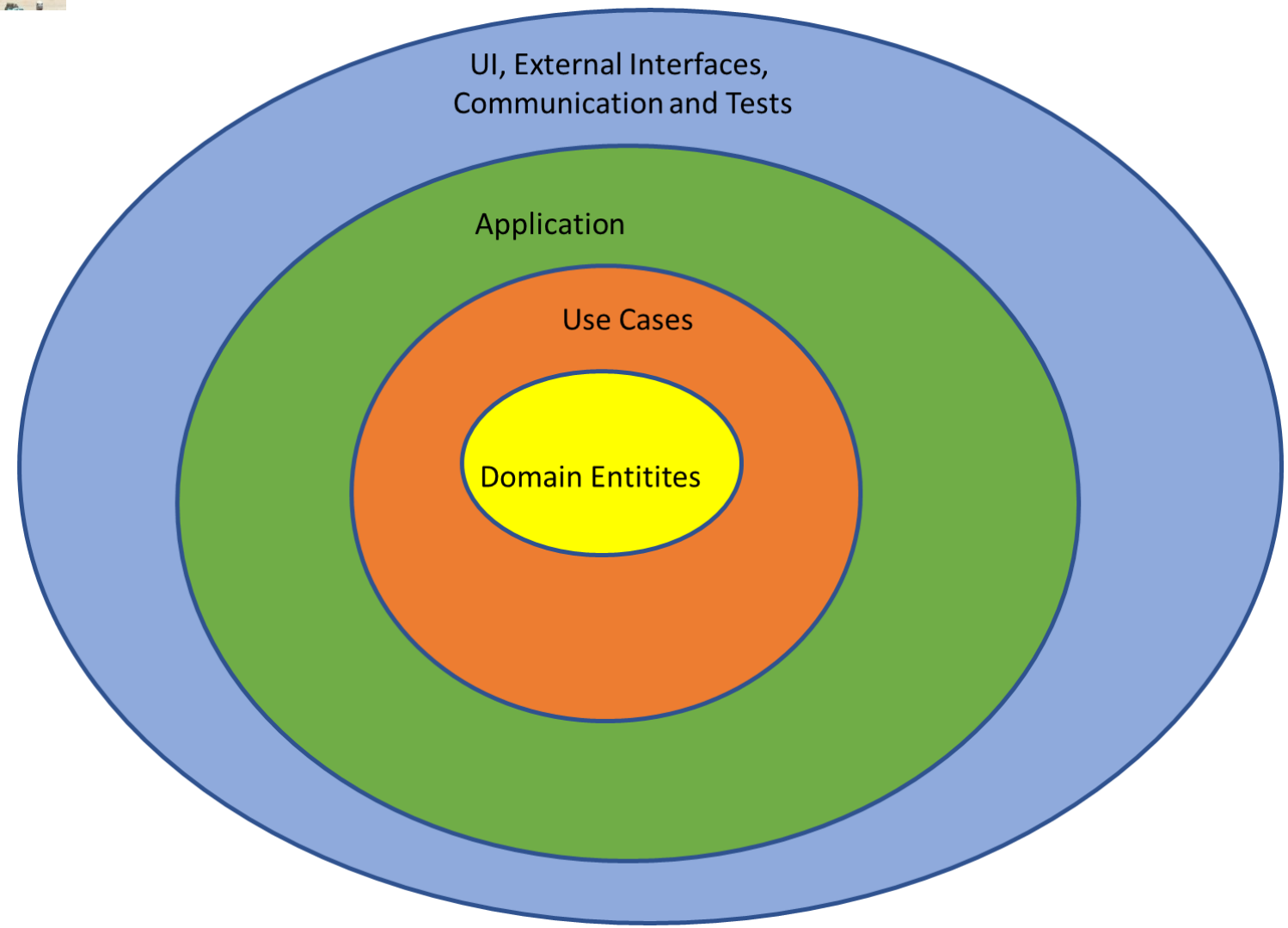
Mandated by many industrial standards Ex: IEC 61508, ISO 26262, DO-178C

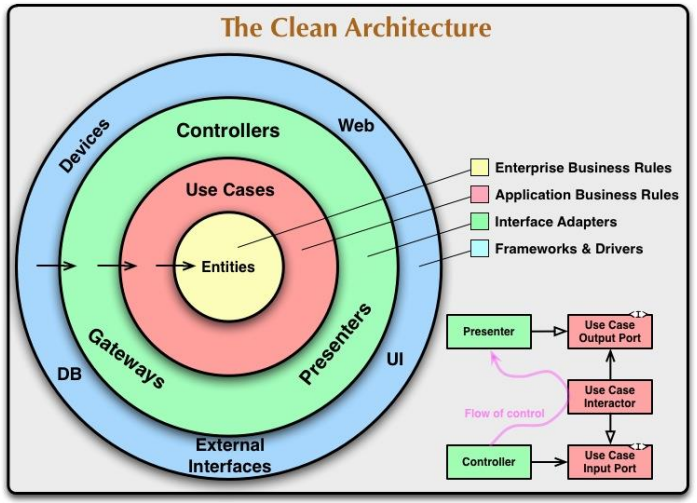
Coding Standards:

- AUTOSAR C++2014
- MISRA C++ 2008

Clean Architecture

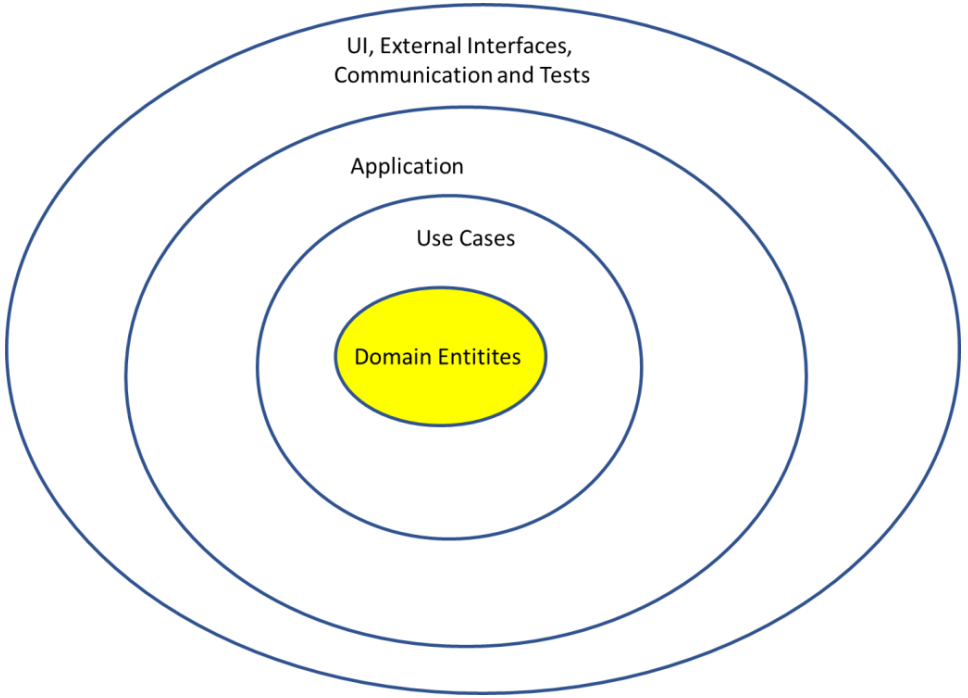
Clean Architecture





Domain Entities

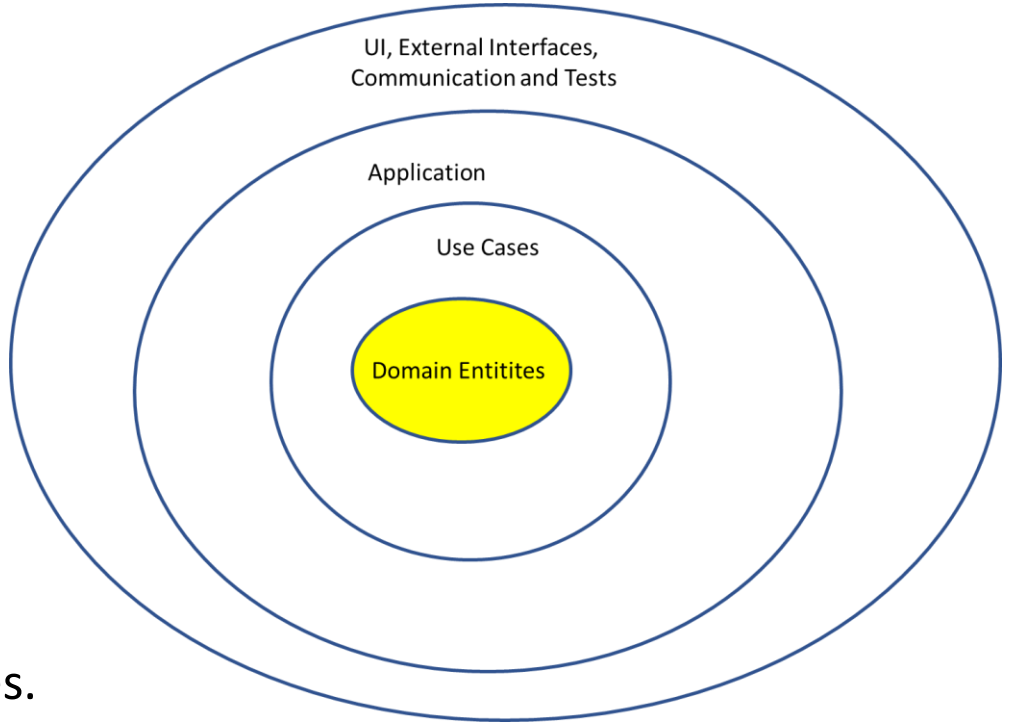
- Business entities identified with the Domain Driven Design (DDD) approach.
- Here we are talking about objects such as Personnel, Order, Product, Call, Delivery, Aircraft etc..
- An object with methods, or it set of data structures and functions.
  - Design these objects as data objects only, independent of the business domain logic
- Since these domain entities will be used by more than one service in large projects, it would be appropriate to design them as a separate module.



How to Manage Core Entities

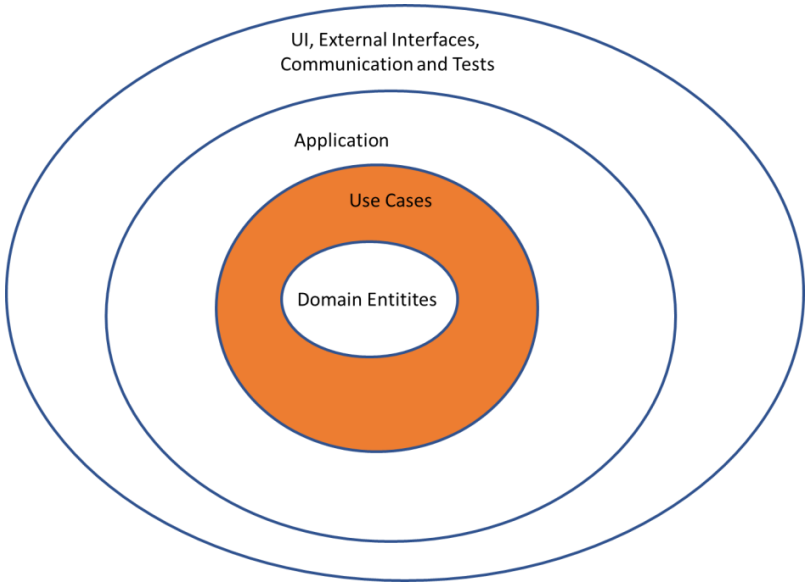
- Manage common data model separately(independent from technology and programming language)
  - Common team to manage changes to these entities.
  - Model and Document
  - Look for extendibility approaches
  - Manage with versioning
  - Be consistent in terms of measurement units, data types, enumarations etc.

- Also define service only entities (entities in only your bounded context)
  - Put them into separate module or data only header files.



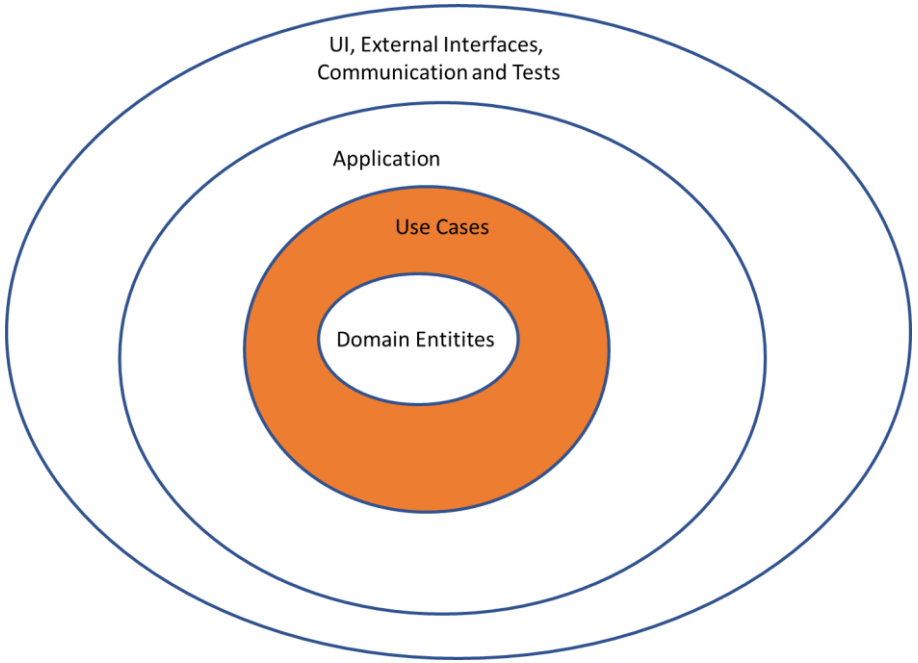
Use Cases/Domain Services

- Application-specific business rules algorithms
- Changes in this layer to affect the entities.
- Changes in technology, UI does not change this layer



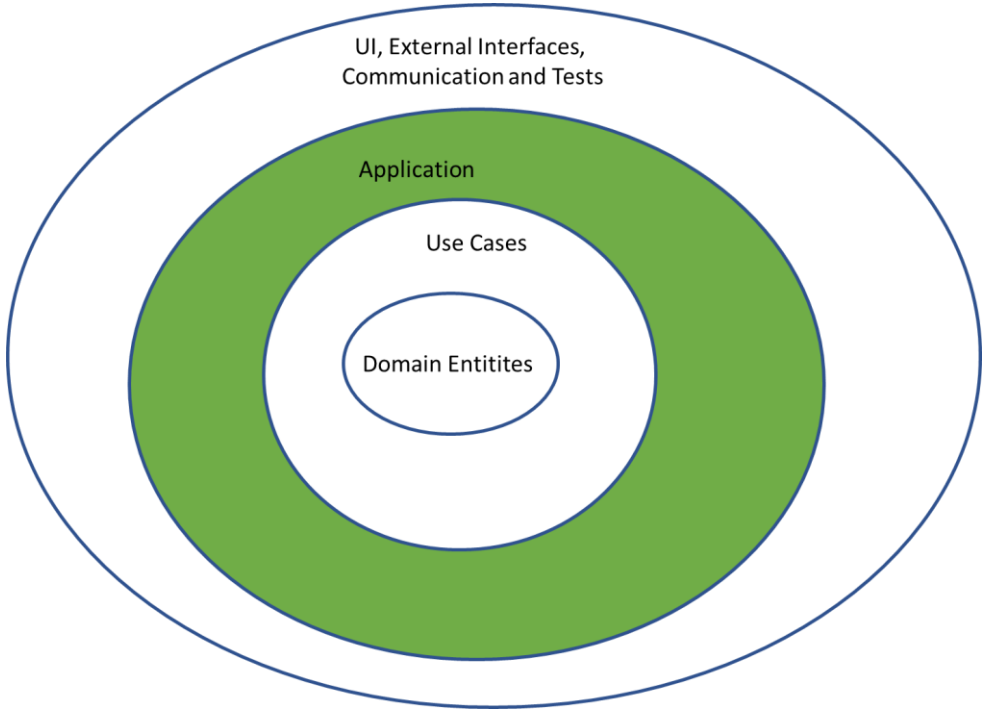
Use cases

- The micro service includes one or more sub-business areas(Use cases).
  - Identify the sub domains within the service
  - Develop them as separate modules.
  - These modules contain mainly passive algorithms and related data structures.
    - Test these algorithms with unit tests (google test etc.).
    - **Most critical gain in terms of writing a clean micro service.**



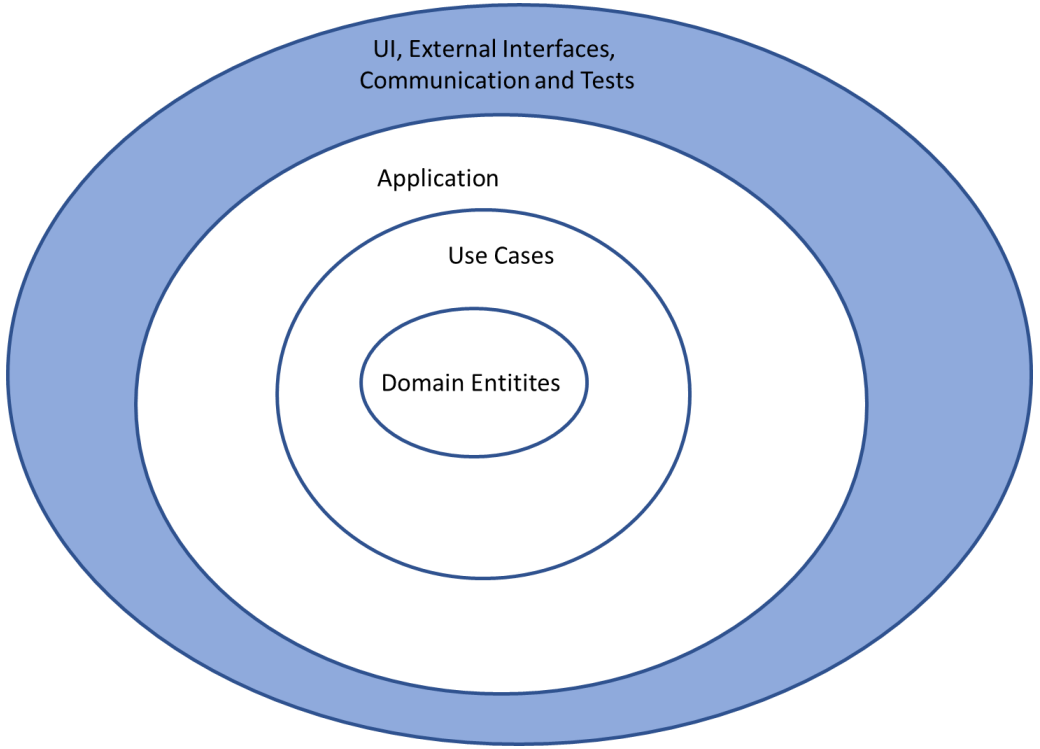
Application Services

- The presenters, views, and controllers all belong to this layer
- Receive data with communication technologies such as REST, MQTT, AMQP, gRPC and provide it to business area services.
- Operations such as managing the relevant technology, transforming the data into an object from a format such as json or Google Protocol Buffer are done.
- Interface/Adapters lays into this layer



Frameworks, External Interfaces, Communication

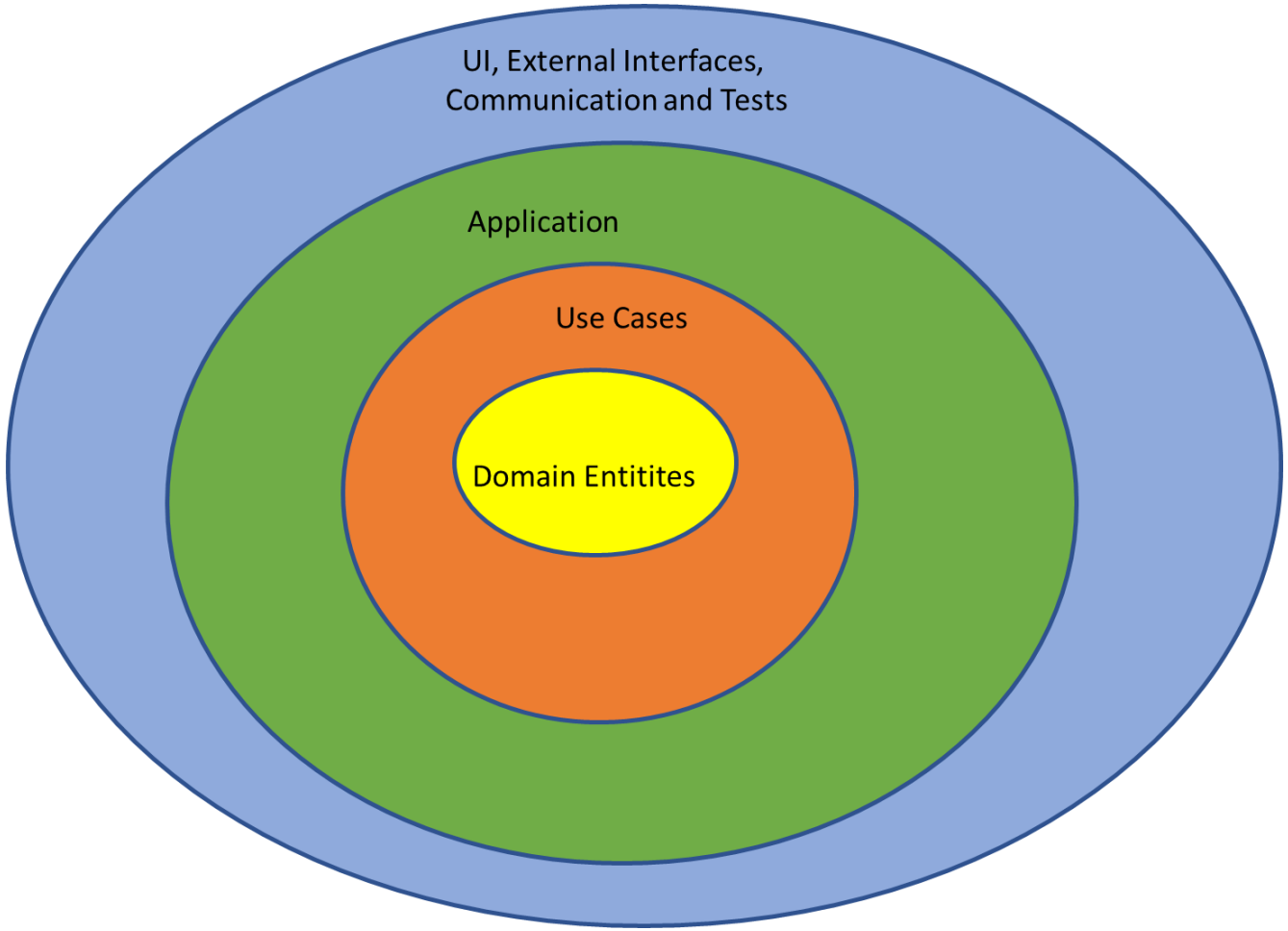
- Frameworks, Communication libraries and databases.
- Generally you don’t write much code in this layer, other than glue code that communicates to the next circle inward.

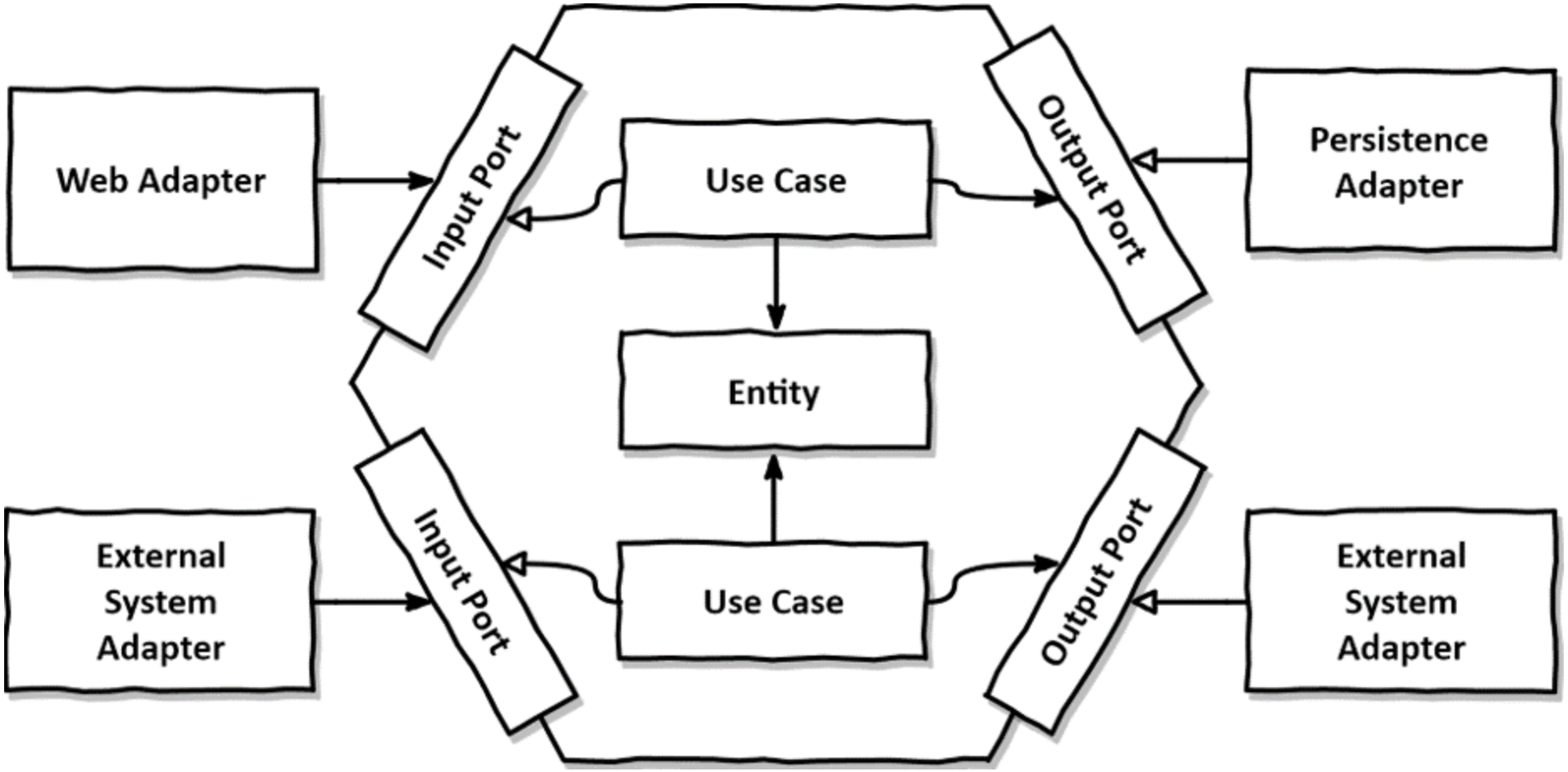


Crossing Boundaries

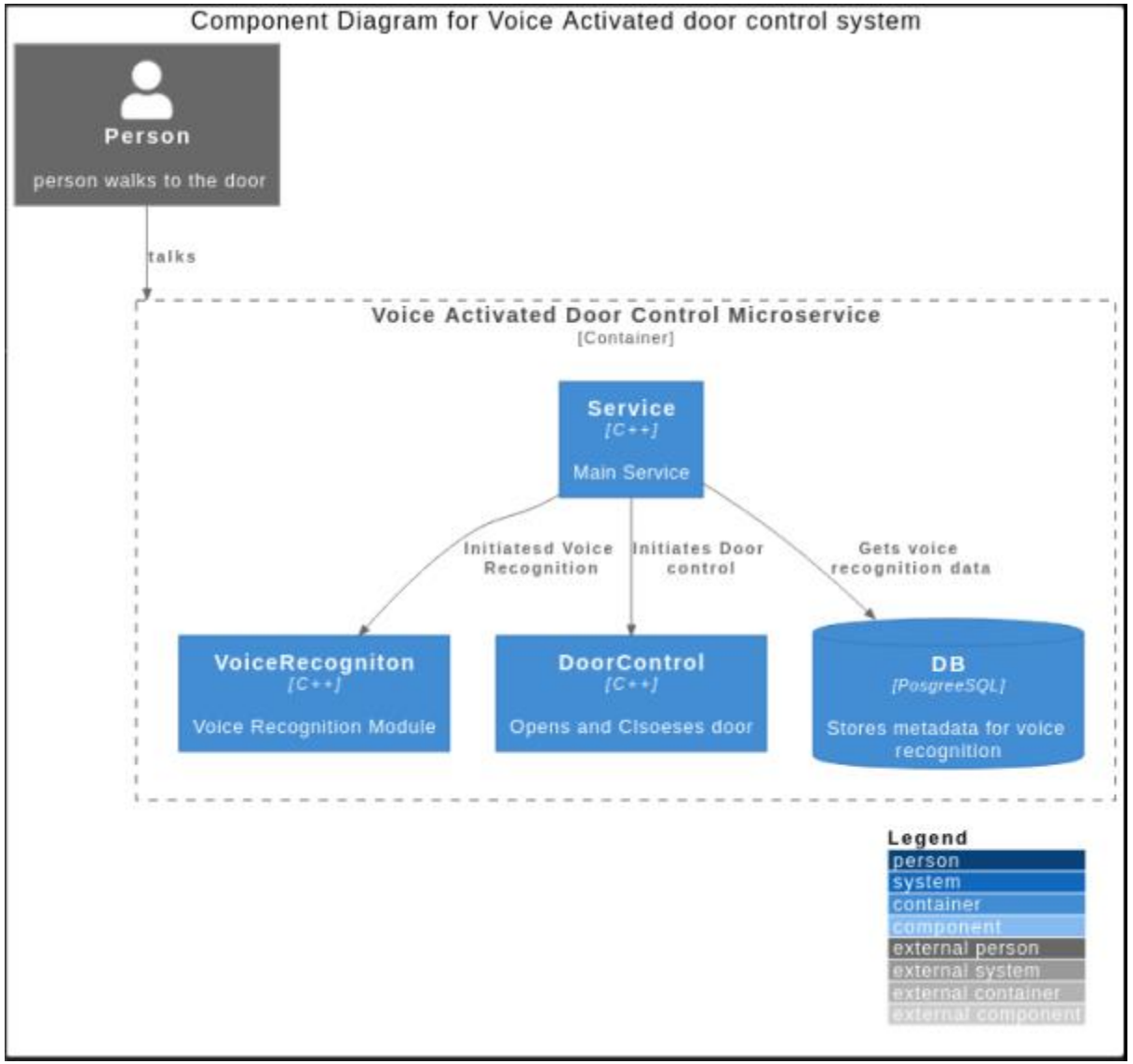
- Dependency Inversion Principle

- Data that crosses the boundaries consists of simple data structures

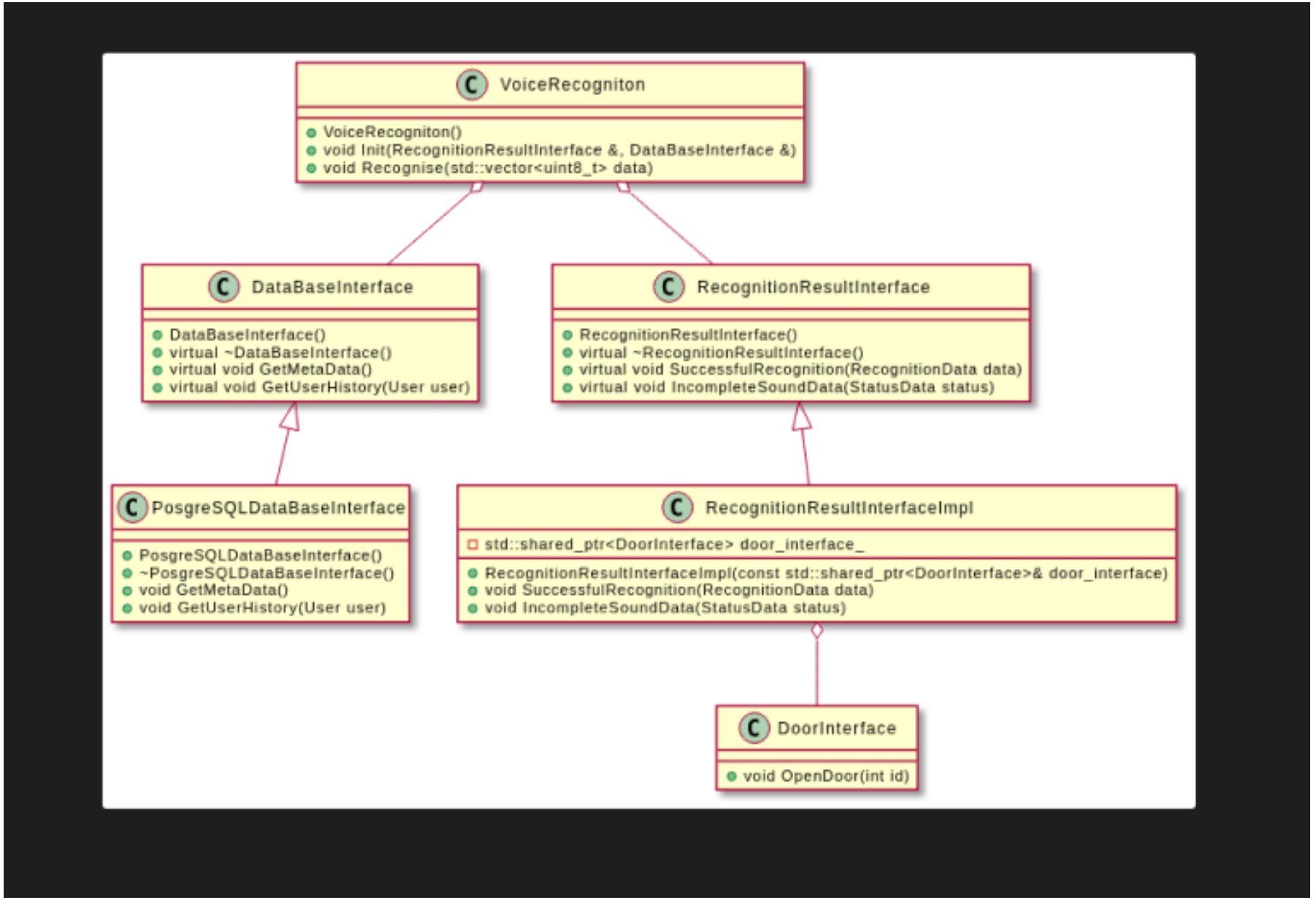



Web Adapter
Input Port
Use Case
Output Port
Persistence Adapter
Entity
External System Adapter
Input Port
Use Case
Output Port
External System Adapter

Simple Example



Simple Example



Use Case : VoiceRecognition


```
// outbound
class RecognitionResultInterface {
public:
  RecognitionResultInterface() = default;
  virtual ~RecognitionResultInterface() = default;
  virtual void SuccessfulRecognition(RecognitionData data) = 0;
  virtual void IncompleteSoundData(StatusData status) = 0;
};

// outbound

class DataBaseInterface {
public:
  DataBaseInterface();
  virtual ~DataBaseInterface();
  virtual void GetMetaData() = 0;
  virtual void GetUserHiustory(User user) = 0;
};

// module interface inbound
class VoiceRecogniton {
public:
  VoiceRecogniton();
  void Init(RecognitionResultInterface &, DataBaseInterface &);
  void Recognise(std::vector<uint8_t> data);
};
```

Managing Outbound Interfaces


```cpp
// Implementation

class RecognitionResultInterfaceImpl : public RecognitionResultInterface {
public:
  explicit RecognitionResultInterfaceImpl(
      std::shared_ptr<DoorInterface> door_interface)
      : door_interface_(door_interface){};
  void SuccessfulRecognition(RecognitionData data) override;
  void IncompleteSoundData(StatusData status) override;

private:
  std::shared_ptr<DoorInterface> door_interface_;
};

void RecognitionResultInterfaceImpl::SuccessfulRecognition(
    RecognitionData data) {
  // Send command to open door
  door_interface_->OpenDoor(data.id);
}

int main() {
  auto door_interface = std::make_shared<DoorInterface>();
  auto recognition_result_interface =
      RecognitionResultInterfaceImpl(door_interface);
  VoiceRecogniton voice_recognition;
  voice_recognition.Init(recognition_result_interface);
}
```

Infrastructure and Code Modules

- It is good practice to wrap Hardware, Socket, Shared memory and Middleware interfaces into one common class.
  - All library related code is in one place. Easy to change vendor, upgrade version etc.
  - Other modules are easier to test without these communication artifacts

Task Design

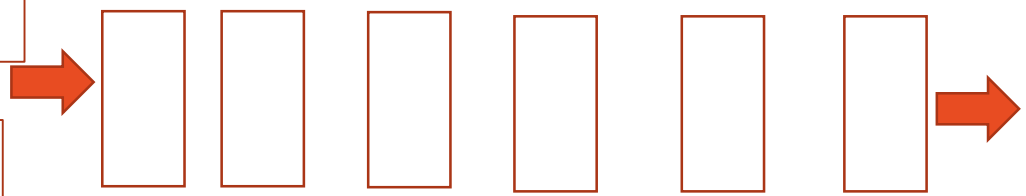
Worker Thread

Produces Thread(s)

```
//receive data  from TCP port and put
to queue

m_WorkerThread->
postMsg(statusData);
```

```
//receive data  from UDP port and put
to queue

m_WorkerThread->
postMsg(statusData);
```

```
//Timer expires in timer thread

 postMsg(timerData);
```



Worker Threads

```
while (!m_workerExit) {
   ....
   userData = m_queue.front();
   userData->handle(this);
}
```

```
while (!m_workerExit) {
   ....
   userData = m_queue.front();
   userData->handle(this);
}
```

```
while (!m_workerExit) {
   ....
   userData = m_queue.front();
   userData->handle(this);
}
```

- Message is calling handler function in worker thread

```
void handle(WorkerThread* handler)
{
    dynamic_cast<MyWorkerThread*>(handler)->handleMessage(*this);
}
```

- Message is handled by handler method

```
void handleMessage(const ApplicationData& applicationData)
{
…
}
```

Timer Attached to Worker Thread

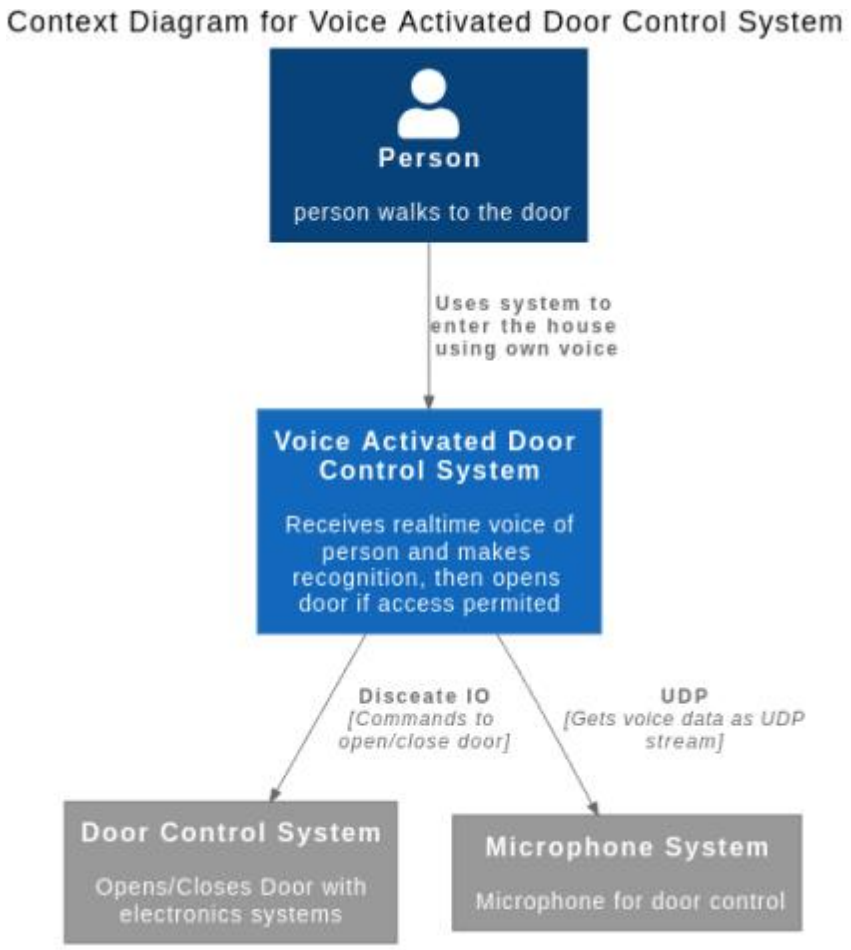
- Additional Timer Thread can be initialized
- Timer thread also puts timer expire event on worker thread queue

Development Test
(Unit Tests and Integration Tests and Simulators)

- Unit test is a method that instantiates a small portion of our application and verifies its behavior **independently from other parts**.
  - Initializes a small piece of an application it wants to test(system under test SUT)
  - Applies some stimulus to the system under test (usually by calling a method on it)
  - Observes the resulting behavior.

- **When it is difficult to write unit test for System Under Test**
  - **It is tightly coupled**
  - Single Responsibility Principle (SRP)

- Simulate Major Components Providing Data to your Service

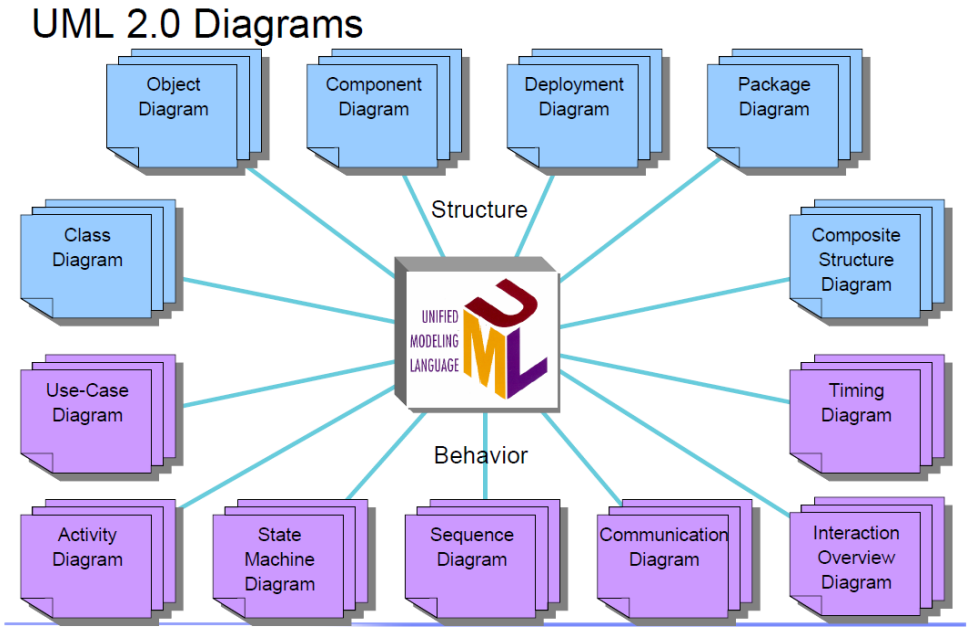


Design Your Service Before Starting to Code
C4 Diagrams and UML

Document Your Architecture in Github/Confluence/Word

- Functional requirements and Non Functional Requirements
- Design Guidelines
- Upper Level System architecture drawings
- Detailed Design
  - Class diagram, Sequence Diagram, State diagram
- Architecture Tradeoff Analysis

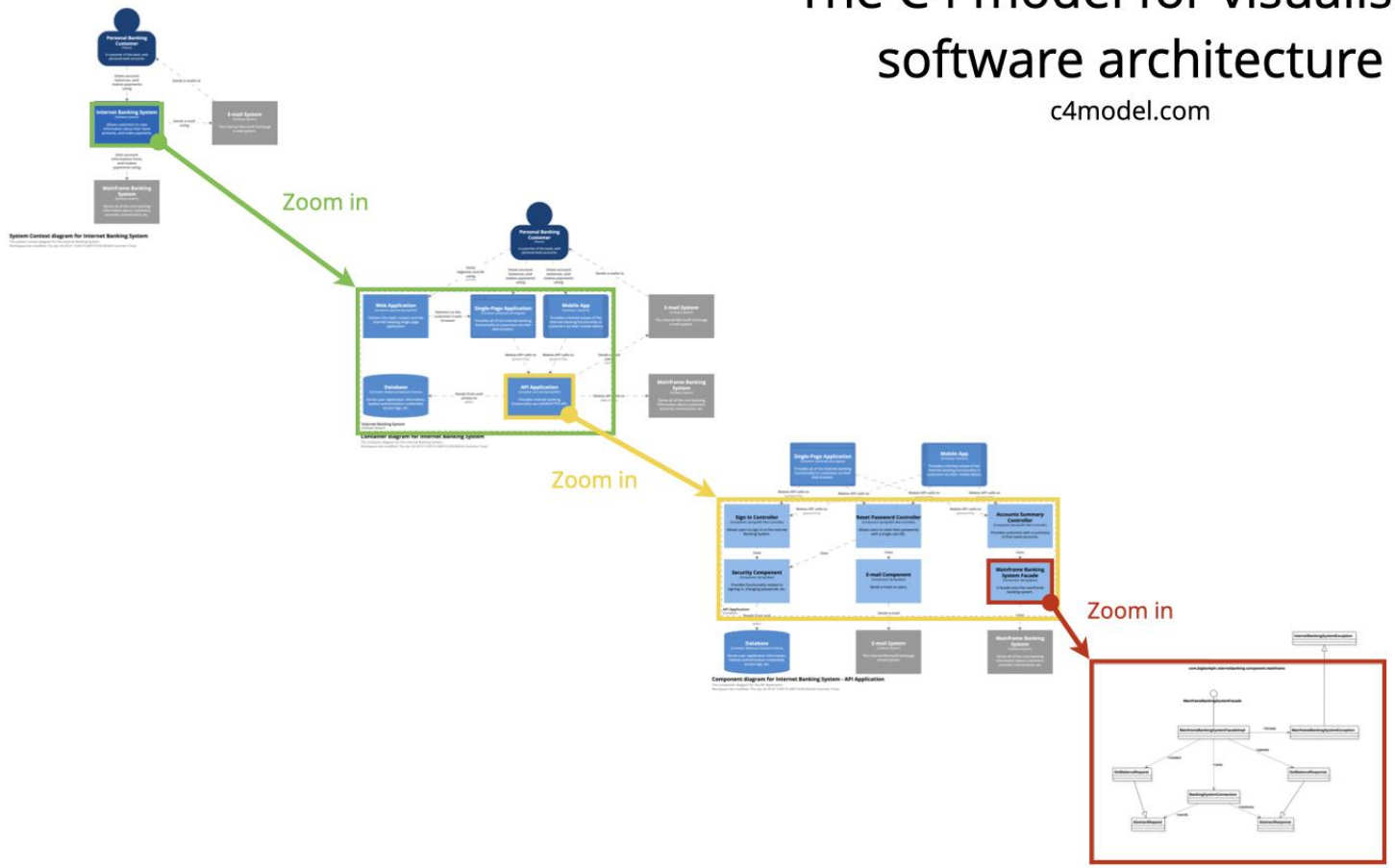
- Different views support different goals and uses.
- We do not advocate a particular view or collection of views.
- The views you should document depend on the uses you expect to make of the documentation.
- Each view has a cost and a benefit; you should ensure that the benefits of maintaining a view outweigh its costs.



C4 Model

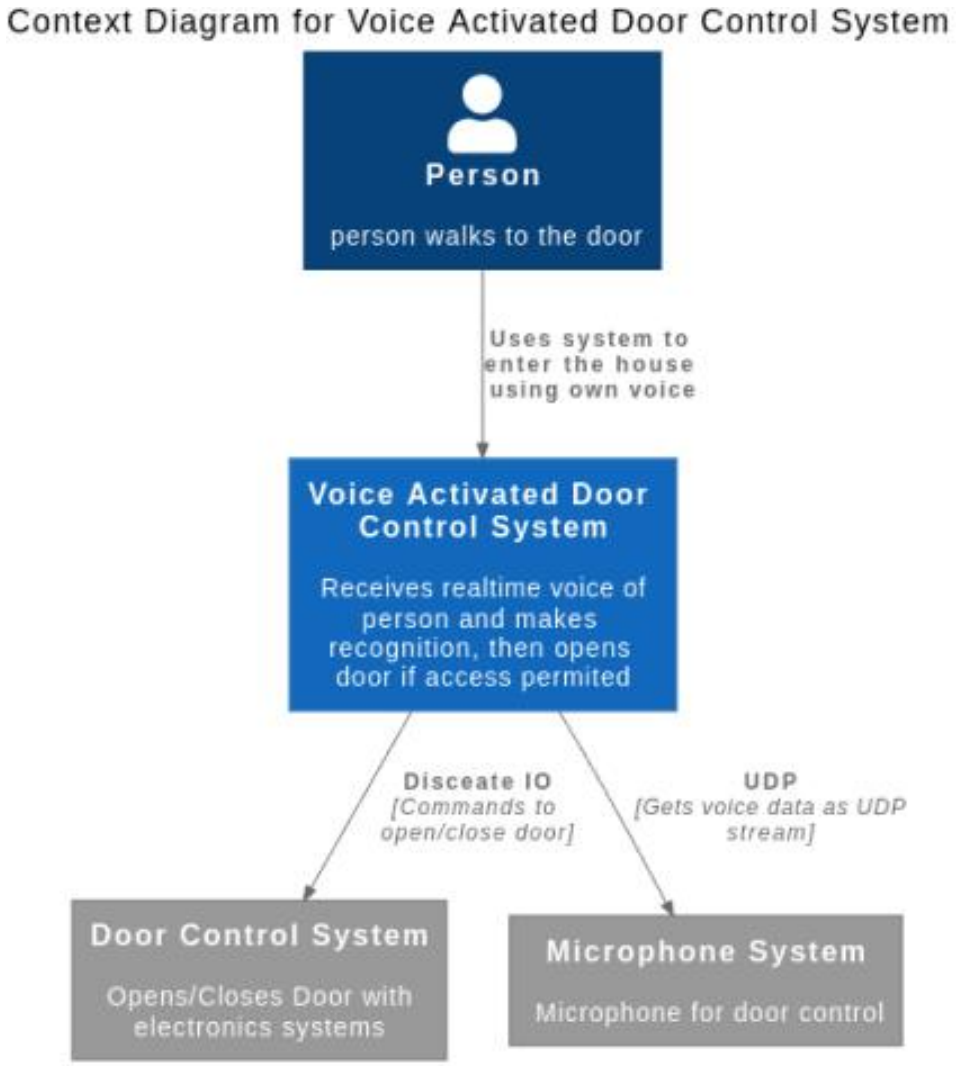
The C4 model for visualising software architecture

c4model.com

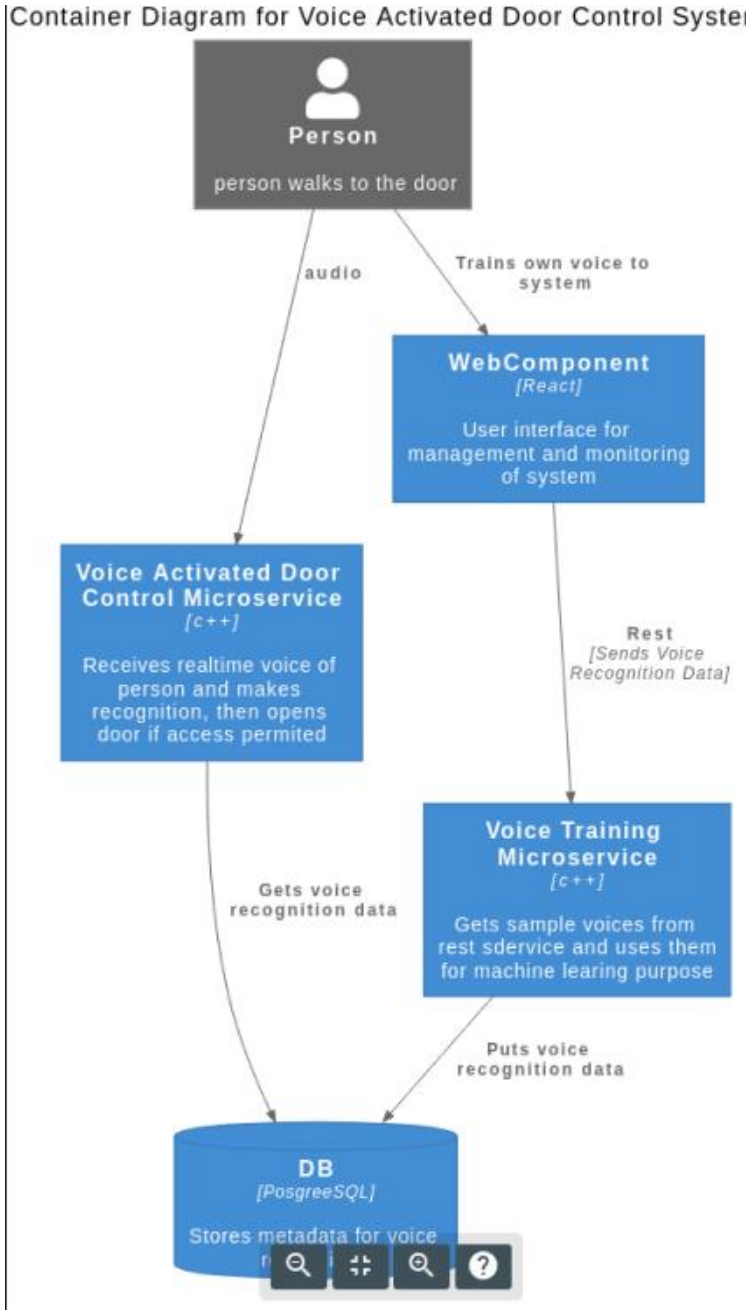


Level 1	Level 2	Level 3	Level 4
Context	Containers	Components	Code

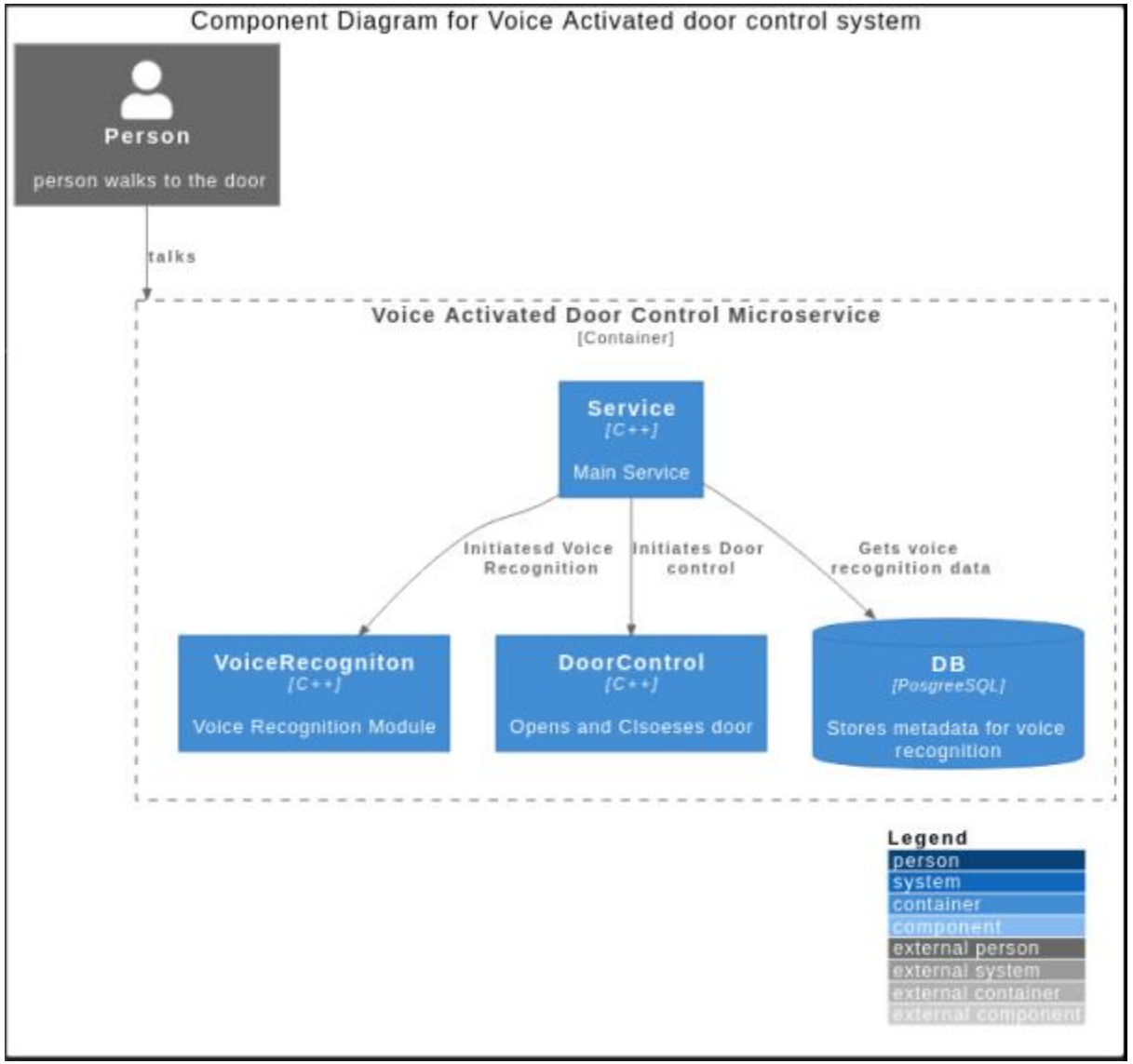
Context


Context Diagram for Voice Activated Door Control System
Person
person walks to the door
Uses system to enter the house using own voice
Voice Activated Door Control System
Receives realtime voice of person and makes recognition, then opens door if access permited
Disceate IO
[Commands to open/close door]
UDP
[Gets voice data as UDP stream]
Door Control System
Opens/Closes Door with electronics systems
Microphone System
Microphone for door control

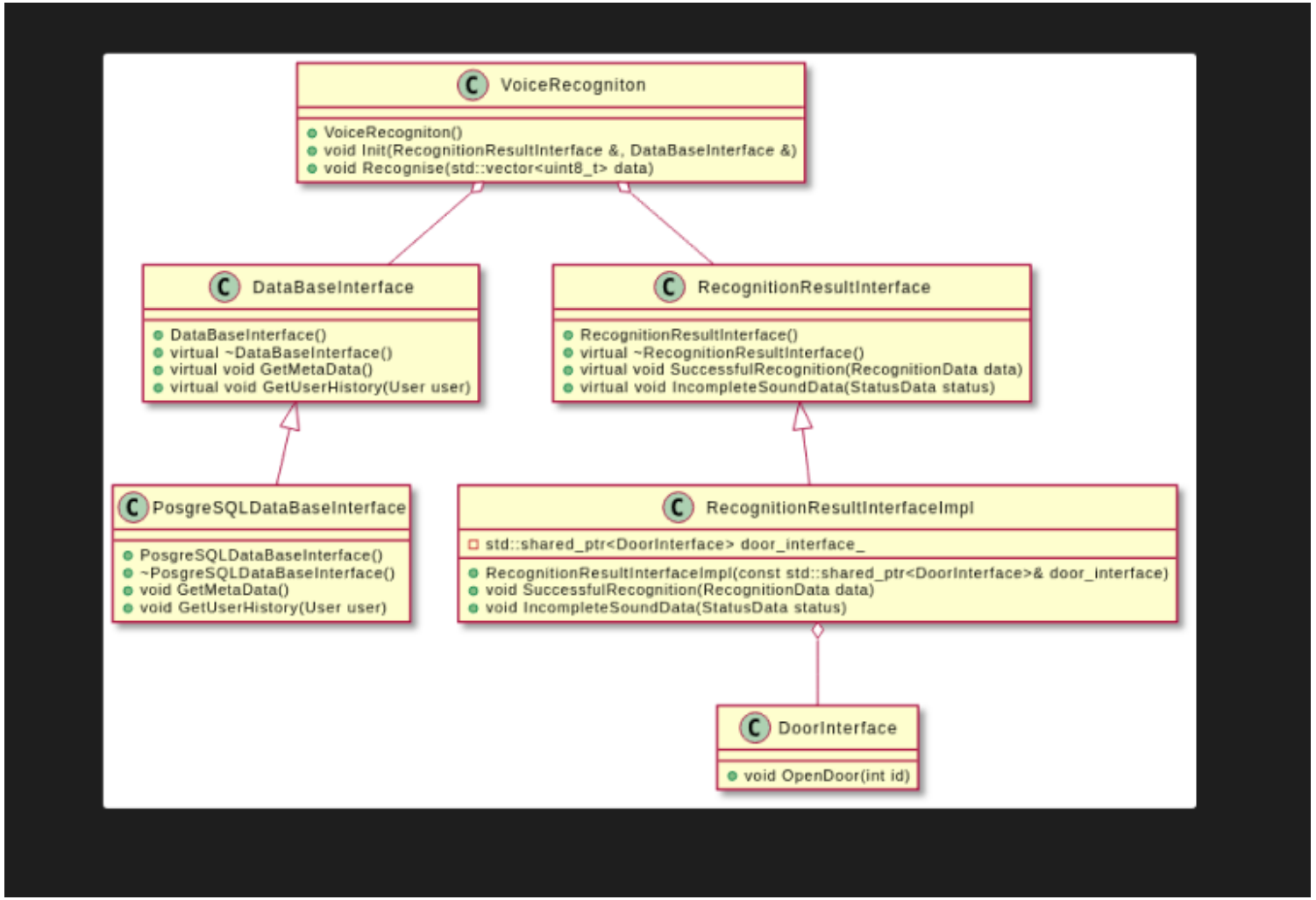
Container


Container Diagram for Voice Activated Door Control System
Person
person walks to the door
audio
Trains own voice to system
WebComponent
[React]
User interface for management and monitoring of system
Voice Activated Door Control Microservice
[c++]
Receives realtime voice of person and makes recognition, then opens door if access permited
Rest
[Sends Voice Recognition Data]
Gets voice recognition data
Voice Training Microservice
[c++]
Gets sample voices from rest sdervice and uses them for machine learing purpose
Puts voice recognition data
DB
[PosgreeSQL]
Stores metadata for voice

Component


Component Diagram for Voice Activated door control system
Person
person walks to the door
talks
Voice Activated Door Control Microservice
[Container]
Service
[C++]
Main Service
Initiatesd Voice Recognition
Initiates Door control
Gets voice recognition data
VoiceRecogniton
[C++]
Voice Recognition Module
DoorControl
[C++]
Opens and Clsoeses door
DB
[PosgreeSQL]
Stores metadata for voice recognition
Legend
person
system
container
component
external person
external system
external container
external component

Code


VoiceRecogniton
VoiceRecogniton()
void Init(RecognitionResultInterface &, DataBaseInterface &)
void Recognise(std::vector<uint8_t> data)
DataBaseInterface
DataBaseInterface()
virtual ~DataBaseInterface()
virtual void GetMetaData()
virtual void GetUserHistory(User user)
RecognitionResultInterface
RecognitionResultInterface()
virtual ~RecognitionResultInterface()
virtual void SuccessfulRecognition(RecognitionData data)
virtual void IncompleteSoundData(StatusData status)
PosgreSQLDataBaseInterface
PosgreSQLDataBaseInterface()
~PosgreSQLDataBaseInterface()
void GetMetaData()
void GetUserHistory(User user)
RecognitionResultInterfaceImpl
std::shared_ptr<DoorInterface> door_interface_
RecognitionResultInterfaceImpl(const std::shared_ptr<DoorInterface>& door_interface)
void SuccessfulRecognition(RecognitionData data)
void IncompleteSoundData(StatusData status)
DoorInterface
void OpenDoor(int id)