Chapter 3 Project Application Analysis Phase

3.1 Object Oriented Environment

Object Orientation can be referred as principles of Design and Development, and it originated from the concepts of Object-Oriented Programming (OOP). OOP provides easy modelling in designing and developing real entities and the relationships that exists between them (Yilmaz et al., 2018). Real-world objects and events can be represented in the Object-Oriented environment by *abstracting* the characteristics that best describe the object for specific purposes. To represent objects, in Java Programming language, it uses what is called *classes* (Mullins, 2000). A particular instance of a class is called an *object* (please, see Figure 3.1). For example, the class Book has attributes such as author, title, year of publication etc., and an instance of this class could be the book '*Harry Potter and the chamber of secrets'*, by J.K. Rowling, published in 1998.

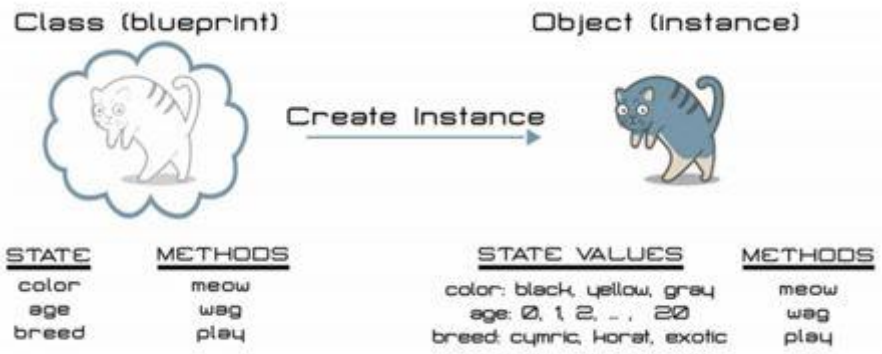


Figure 3.1: Class 'Cat' and the instantiated object *cat* (Yilmaz et al., 2018).

An object has a unique *identity*, and it is composed of *attributes* (properties of the object), has a *state*, and exhibits *behaviour*, with the ability to *interact* with other objects and itself. An object's *state* is defined at specific time by set of values for its attributes. Changing the values of its attributes would change its current state and would bring a change in the object's behaviour. For example, an instance of the object 'Student' may have its current *enrolment* state as 'active', whereas if this student graduates, this *enrolment* state will change to 'inactive'. A change in object's state and its behaviour can be initiated by calling a *method*, which represent actions (operations on the values of its attributes), like *add* a student to a course, *charge* student, *set* name, address, contact details, etc.

The Object Oriented Programming approach provides for data binding, reuse of code, and according to Mullins (2016) there are five key features of the systems built in Object oriented environment: *reusability* (software products can be reused for new applications), *robustness* (software systems continue to function even in abnormal situations), *extendibility/scalability* (easy to adapt to small changes of specification and new user requirements), *correctness* (meet requirements in all situations), and *compatibility* (standardisation that makes possible for software products to interact in well understood ways, e.g. standardized file formats and data structures).

3.1.1 Encapsulation

Encapsulation is a term used to describe hiding of the implementation details of an object class - its entity attributes and methods (a class encapsulates the entity properties). Objects can only interact through their public interfaces. The actual data attributes and the internal logic of the class is hidden from the outside world, therefore the interface encapsulates the object's code and data (Mullins, 2016). Attributes and methods that are declared as 'private' can only be visible in the scope of the class, preventing external direct access to them.

3.1.2 Inheritance

One class can inherit common characteristics from other classes. The class that is inheriting can be denominated as *subclass* (or derived, extended, child class), and the class that the subclass is inheriting from is the *superclass* (or base, parent class). Inheritance allows code reuse since a new class can be created from existing classes.

Inheritance implements the "Is-a" relationship, meaning that one object is type of another one. For example, student is a Person, car is a Vehicle, the dog, cat, panda are all extensions of the superclass Animal. Dog will inherit instance variable declarations and its methods from the base class Animal, as well as implementing its new instance variables and methods that are unique for the object Dog, but not for all other animals. Dog can also override its base class methods, but it can not access private properties of the superclass, due to the concept of encapsulation (Yilmaz et al., 2018).

There are different types of inheritance: *Single* inheritance (derived class inherits from a single base class), *Multi-Level* inheritance (a class that is derived from another derived class), *Hierarchical* inheritance (more than one class is derived from a base class), *Multiple* inheritance (a class derived from more than one other classes) and *Hybrid* inheritance (combination of a *Single* and *Multiple* inheritances).

3.1.3 Polymorphism

Poly is the ancient Greek word for 'many', and *morphism* – for 'forms'. In OOP, *polymorphism* is a property that allows different objects to respond to the same message in different ways. In Java, according to Mullins (2016), there are two types of polymorphism: *compile time* polymorphism (*static binding*) and *runtime* polymorphism (*dynamic binding*). The *method overloading*, which is a *compile time* polymorphism, enables a class to have two or more methods with the same name but with different parameters. Whereas, the *method overriding*, which is a *runtime* polymorphism, can be achieved through *inheritance*. For example, the Animal superclass may have a method, called 'speak', the derived class Dog will override this 'speak' method with a 'woof-woof' method, while the derived class Cat will override the 'speak' method with a 'meow' method.

The *polymorphism* gives flexibility to the program and together with *inheritance*, leads to *code reusability*.

3.1.4 Data Abstraction

Data abstraction is related to the concepts of *encapsulation*, *inheritance*, and *data hiding*. This concept hides any unnecessary implementation code to reduce complexity of the design. Class internals are protected from user-level errors, which breaks state of the objects (Yilmaz et al., 2018).

An *abstract* class is a parent class that allows inheritance and contains abstract properties, which are only declared (not implemented). This class cannot be instantiated, and each derived class must implement the abstract methods on their own way.

3.2 Development Methodologies

A software methodology is a set of guidelines, rules, and practices, used in development projects (Nash, 2003). In this section presented is discussion on popular methodologies used in industry today to select the one that best suits the development purposes of the current Project application.

3.2.1 Waterfall Model

The *Waterfall* model is one of the oldest and simplest, with better budget management. It follows the Systems Development Lifecycle (SDLC) stages in a linear-sequential manner, one stage to the next, not allowing going back to a previous stage (please, see Figure 3.2). This rigidity allows easier management in each stage, allowing to meet deadlines and perform extensive documentation. It is more suitable for small projects.

The typical stages on the SDLC are initiation, feasibility study, analysis of business requirements, systems design, system build and implementation, followed by review and maintenance (Bocij et al., 2008).

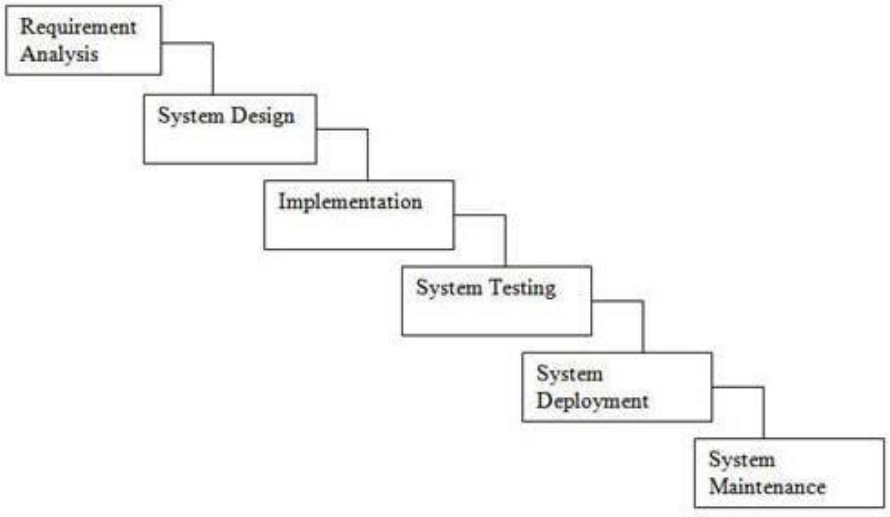


Figure 3.2: Representation of the Waterfall model. Each stage must be completed before starting the next stage (Software Testing Help, 2021a).

The ‘initiation phase’, contain the stimulus to develop a new project, it aims to establish whether the project is feasible and prepare to ensure it will be successful (identification of

problem or opportunity). On the ‘feasibility phase’, study will be performed to analyse the need for the proposed application, the impact on the system and its viability (technically, economically, operationally, and organisationally feasible). The capture of the business requirements is done during the ‘analysis phase’, through the end-users input or observing them, or other resources, such as an existing system documentation. The output of this stage is a detailed requirements specification, supported by diagrams showing the flow and processes of the application. How the application will work will be defined in the ‘design phase’, such as interface definition (navigation, menu systems, screen design, etc.), program modules, security, and database transactions. The creation/coding of the application will be done during the ‘build phase’, including testing by programmers and end-users, and writing documentation. The transition from the old system to the new one will be performed at the ‘implementation and changeover phase’, which also includes checking hardware, network infrastructure, testing and training staff to use the new system (preparing for the new system). The ‘maintenance and review phase’ deals with correcting possible issues that may arise after implementation, implementing new requirements, and evaluate the success of the new system and learn for the future.

According to Nash (2003), the traditional *Waterfall* method performs poorly when specifications change frequently and rapidly, therefore it is not largely used today. However, the amount of risk remains essentially the same throughout the project until it reaches the testing phase, where problems may be identified.

The *Waterfall* model is highly inflexible, does not handle unexpected risks well, and is not recommended for complex and long-term projects, since it is difficult to get all the requirements at the initial stage. Mullins (2016) identified that, in software systems in general, regarding of the methodology used, 70% of all modifications had to do with extending systems to deal with new issues, 42% of which accounts to changes in the user requirements.

3.2.2 Incremental/Iterative Methodology

Like the *Waterfall* model, this methodology is also simplistic, as it only describes the overall approach to the development process. The *Iterative* methodology implies a cycle of adding functionality, testing, and documenting (please, see Figure 3.3). Each functionality is

incremented at each stage. Tests are performed in each iteration, which reduces the risks, as each issue identified in the test phase can be corrected before moving to the next iteration.

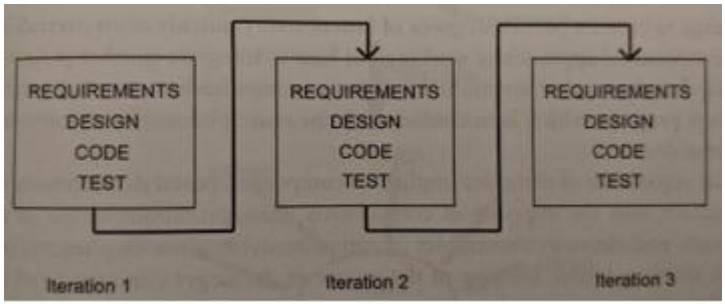


Figure 3.3: Iterative methodology cycle (Nash, 2003).

3.2.3 Boehm's Spiral Model

While handling risks is an issue in the *Waterfall* model, the Spiral model, developed by Barry Boehm in 1986, incorporates *risk assessment* and uses the *iterative prototyping* approach, where analysis, design, code, and review stages are repeating as part of the *prototyping* process. Figure 3.4 shows the *Spiral* model divided into four main activities: planning, risk analysis, engineering (coding and testing), and user evaluation (testing from the end-users point).

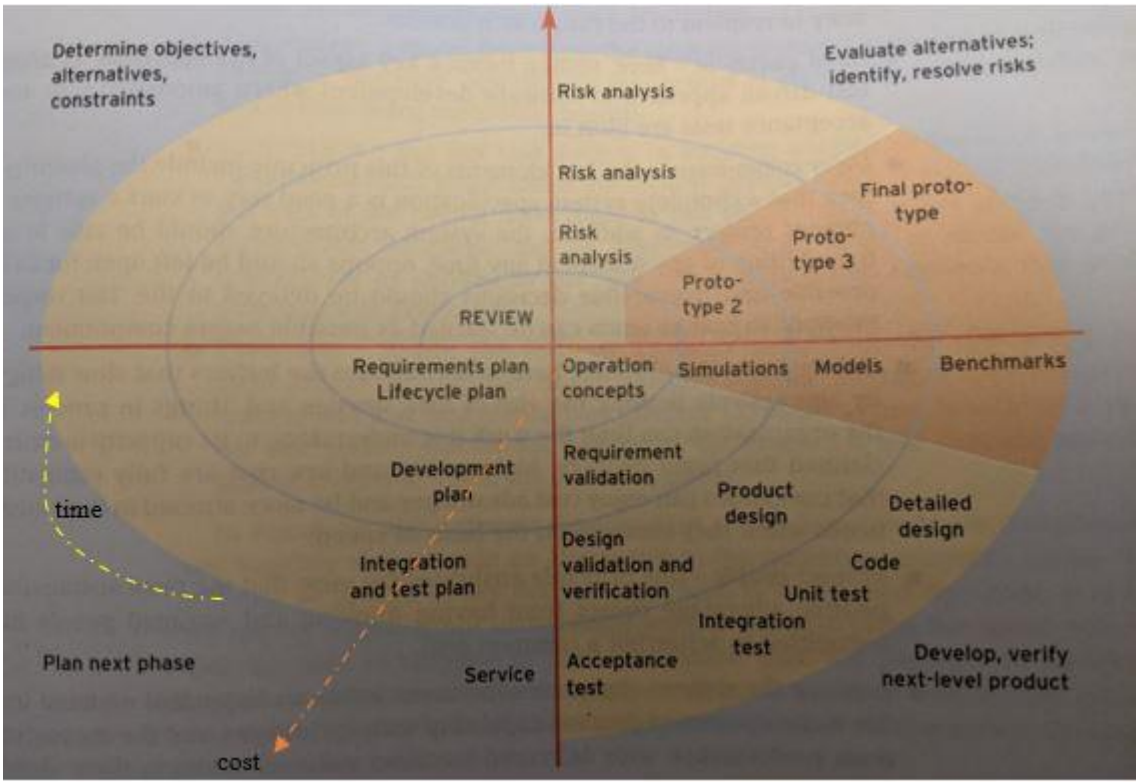


Figure 3.4: The Spiral model approach for systems development (Bocij et al., 2008).

Even though this model combines the benefit of the classic SDLC and the Prototyping approach, adding risk analysis, validation of requirements, and design, the *Spiral* model is not widely used (Bocij et al., 2008). The cost increases in each iteration, meaning it could be an expensive process.

3.2.4 Rapid Application Development (RAD)

According to Nash (2003), *Rapid Application Development* (RAD) is more of a technique than a formalized methodology. It is designed to produce rapid results in short increments, and the *Prototyping* is used in an iterative way to solicit early feedback on design from the end-users, until the application is completed.

RAD solves many of the problems, encountered with the traditional *Waterfall* model, such as the long development times, where requirements may change during the process, and inclusion of the end-users at early stages of the *prototype* development. However, this approach may present some lack of governance, as opposed to the *Dynamic Systems Development Methodology* (DSDM), which is an *Agile* method for software development. Different from the *Spiral* model, *RAD* may not include validation of requirements and design, and does not include *risk analysis*.

3.2.5 Agile and Lean Software Development

The term '*lean*' addresses mostly the philosophy behind software development, whereas the term '*agile*' addresses more the software engineering aspects of the systems development. There are seven principles that define the Lean software development: *eliminate waste* (by focusing on the 20% of features that deliver 80% of system's value); *create knowledge* (follow good practices and standards and improve them); *build high quality software* (follow test-driven approach, with automated unit tests and user acceptance tests); *defer commitment* (learn as much as possible before commitment, as system architecture should be able to support addition of new features at any time); *deliver fast*; *respect people* (committed people working towards reaching a common goal) and *improve the system* (measurement forms to measure process capability with cycle times, team performance, etc.) (Bocij et al., 2008).

The Agile Manifesto (2001) is a set of values and principles that guides how we work as a team and how we develop software. It values **individuals and interactions** more than **processes and tools**, empowering capable people and providing a fun, creative environment; it encourages team to break down complex problems/functionalities into manageable modules, experimenting with small parts of the solution for short period and deliver a **working solution/software** in each cycle, as opposed to focussing on excessive documentation; prioritize what customers value most, test and learn from them, incentivise **customer collaboration** over a rigid contract negotiation; and **respond to change** over following a plan.

Many organisations nowadays have taken advantage of the *Agile* methodology. According to the 14th Annual State of Agile Report (StateOfAgile, 2020), 95% of the respondents reported that their organisation is practicing *Agile* development, and the top five reasons why they adopted *Agile* were *accelerating software delivery* (71%), *enhancing ability to manage changing priorities* (63%), *increasing productivity* (51%), *improving business/IT alignment* (47%), and *enhancing software quality* (42%). Figure 3.5 outlines the major benefits of the *Agile* methodology, as reported by the respondents.

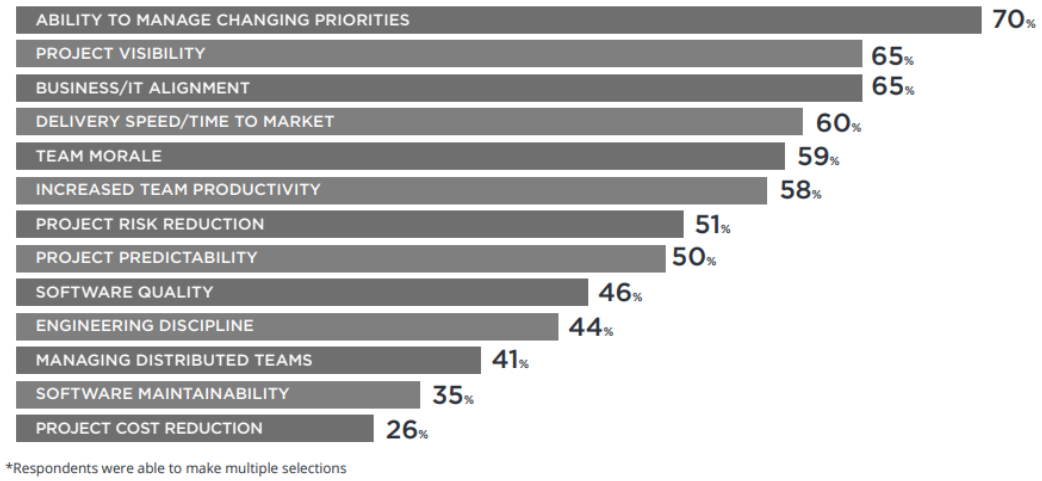


Figure 3.5: Benefits of adopting *Agile* methodology, reported by the respondents (14th Annual State of Agile Report, 2020).

Since 2001, *Agile* methods have been emerging, such as *Adaptative Software Development*, *Agile Unified Process*, *Scrum*, *Extreme Programming*, and *Dynamic Systems Development Methodology*.

3.2.6 Dynamic Systems Development Methodology (DSDM)

Following the principles of *RAD*, *DSDM* uses iterative approach through *Prototyping* to involve users' input while increasing development speed on a tight budget. This approach gathers the user's feedback and knowledge from the previous prototype to improve the next version of the application. In summary, *DSDM* is a methodology that describes how *RAD* can be approached with more governance.

As mentioned in Chapter 1, *Project Research Methodology* section, the DSDM Consortium structured this methodology in nine key principles:

- Active user involvement is imperative.
- Teams must be empowered to make decisions: teams are composed of developers and users, and they must be able to make decisions in changing requirements and improve them.
- Focus on frequent delivery of Products: products will be delivered in an agreed timeframe, which makes possible to evaluate it and improve it until the final working product is released.
- Fitness for business purpose: delivering the business functionality at the required time,
- Iterative and incremental development: system evolves incrementally.
- Changes are reversible: allows flexibility and increases user satisfaction.
- Requirements are baselined at high level: further baselines can be established later in the development.
- Testing is integrated throughout the life cycle: it is not treated as a separate activity. Tested by developers and users to assure the application operates effectively and meets the business needs.
- Collaborative and co-operative approach between all stakeholders.

Due to the rapid development, tight budget and the benefits of Agile Software Development with the *Dynamic Systems Development Methodology*, this feedback-driven approach was chosen to best develop the proposed application, involving end-users to gather feedback on their experience and improve the new version of the application.

3.3 Object Oriented Analysis with UML Diagrams

The system requirements will be structured in a *Conceptual* model through use of Unified Modelling Language (UML) diagrams, allowing to organise, analyse and model the requirements obtained so far.

According to Nash (2003), the UML standard was created by the Object Management Group (OMG) organisation in 1997, and defines twelve standard diagram types:

- Diagrams describing structure → the Object Class, Component, and Deployment diagrams. These diagrams represent the data and static relationships in an Information System.
- Diagrams describing behaviour or operations → Use Case, Sequence (or Interaction), Activity, Collaboration and State-chart diagrams. These diagrams show the dynamic relationships between the instances or objects.
- Diagrams describing the model and its overall organisation → Package, Subsystem, and model diagrams.

For this research, the Use Case modelling together with Sequence and Object Class diagrams will be used to analyse the application's requirements. The diagrams will clearly present the application's functional and business requirements.

3.3.1 Use Case Diagram

Used to identify the application requirements at a high level, without implementation concerns. It provides a functional description of a part of the finished application. It captures the interaction between the *Actors* and the System (Nash, 2003). *Actors,* placed outside of the Systems Boundary, represent hypothetical users of the system, or another client, such as device, or other application, *Actors* are not the specific user, but a role that a user can play, while interacting with the System (Dennis et al., 2009). Figure 3.6 shows the representation of an *Actor*.

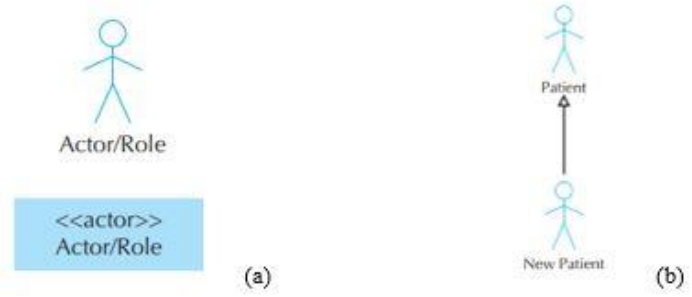


Figure 3.6: (a) The stick figure is the default representation of an Actor. If a non-human actor is involved, it can also be represented by the rectangle with <<actor>> in it. (b) Actors can be associated with other actors using superclass association (Dennis et al., 2009).

Figure 3.7: Systems boundary sets the scope of Use Cases. Use Cases outside the box is considered outside the scope of that system (Lucidchart, 2020).

Figure 3.8: A Use Case is represented by an ellipse and it is placed inside the System Boundary. It is a visual representation of a distinct business functionality in a System (Dennis et al., 2009).

Each *Use Case* represents a functionality provided by the application, and the interactions between them and the *Actors* are represented by lines, called associations (please, see Figures 3.8 and 3.9). Each Use Case diagram is followed by a narrative description of the Use Case being described, which includes a step-by-step sequence of the interactions being modelled (Nash, 2003).

Figure 3.9: A solid line links an Actor with the Use Case(s) representing the interaction between them.

A *Use Case* can also depend on another, and it could be represented by two stereotypes: <<include>> and <<extend>> relationships (please, see Figure 3.10). An *include* relationship

represents the inclusion of the functionality of *specific* Use Case, within *another* Use Case (Dennis et al., 2009). In other words, a Use Case (with *more generic* functionality) can be invoked by another Use Case (with *more specific* functionality). Figure 3.11 illustrates an *include* relationship between two Use Cases. An *extend* relationship represents the extension of an Use Case to include optional behaviour or actions

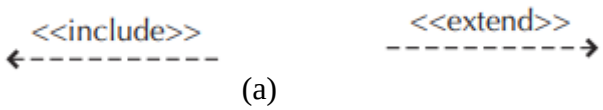


Figure 3.10: (a) Representation of an *include* relationship, (b) representation of an *extend* relationship (Dennis et al., 2009).

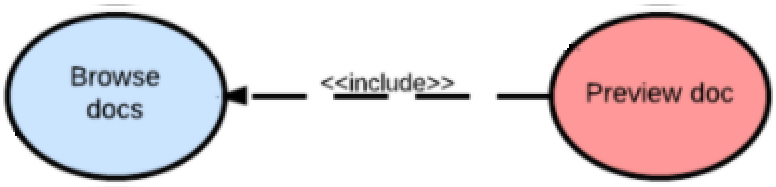


Figure 3.11: Example of an include relationship on a Website Use Case Diagram. '*Preview doc'* Use Case is using '*Browse docs'* Use Case while it is executing itself (Lucidchart, 2020).

Description of the Use Case Diagram for the Proposed Project

Please, refer to *Appendix A4* for the High-end alternative of the Use Case diagram, which involves four actors outside of the System Boundary: User, Admin, Mental Health Organisation Volunteer and a 3rd party Payment System.

First time users will be required to first *Register* online, by providing their personal details along with *strong* password creation. The Mental Health Organisation volunteer will be registered offline to avoid impersonation. Users that have already registered, will only be required to *Login* by providing their *username* and *password*. If the user needs to *Reset the Password*, that can be done with involvement of the Admin.

Once the User has created an account (*Profile*), they can *Set the Goals* they wish to accomplish. These goals can be *edited* anytime through their *Profile* screen. Then, they will be able to access

the *Challenges* (created by the Admin), according to their *Set Goals*. For specific *Challenges* it will be possible to visualise the process of monitoring User's *vital signs*, such as *steps*, *heart rate*, and *sleep quality*, through establishing of a *Smartwatch connection*. The user will be able to *Choose a Challenge* they wish to accomplish in a day, and *points* will be collected at completion of such challenge. The User can *Track Completed Challenges* and visualise a *Progress Report*.

The application will also allow the user to *Add Friends*, with whom they wish to share the accomplished challenges. When *sharing*, it will also be possible to *add a photo* of the completed challenge.

If a User is struggling with some form of mental health issue, the proposed application will allow the user to *Chat with a Mental Health Organisation volunteer* in Ireland. On the Volunteer part, they will access a queue of users, waiting to be replied, select the user at the top of the queue, and reply to them. To support such a Volunteering organisation, the application will avail to the *Donation* functionality, using a Third-Party *Secure Payment* system.

3.3.2 Sequence Diagrams

Differently from the Use Case or Object Class diagrams, the Sequence diagram shows how an application's state changes over time (time increasing toward the bottom of the diagram). It is a key tool to understand the flow and processing of the application and shows the interactions in the form of messages between objects as the application is executed.

Arrows indicate messages being processed from one class to another (please, see Figure 3.12).

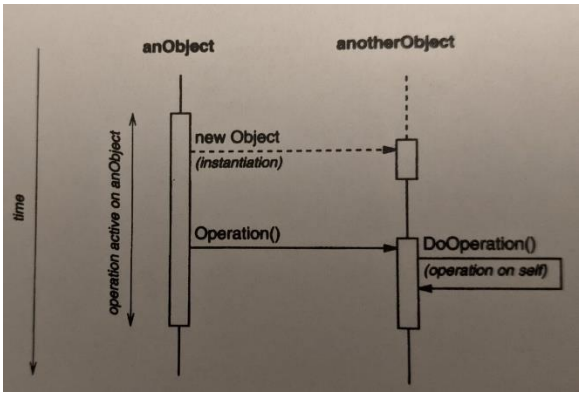


Figure 3.12: Sequence Diagram Notation (Gamma et al., 1995).

Elements of the Sequence Diagram are as follows:

- **Actors**: Like the Use Case Diagram, *Actors* can be a person or another system, that sends messages to the system. It is external to the system and therefore, no interaction between *actors* is reflected on the diagram. Figure 3.13 illustrates the representation of an *Actor*.
- **Messages**: Are presented by horizontal lines with arrowheads and labels passing between objects (please, see Figure 3.14).
- **Lifeline**: Each object has a dotted vertical line, representing the lifeline during a sequence of interactions with other objects (please, see Figure 3.15).
- **Focus of Control or Execution Occurrence**: Represented by thin bars (please, see Figure 3.16) on top of an object's lifeline. It denotes the time when an object is sending or receiving a message (Dennis et al., 2009).
- **Object**: Like the Actors, objects are placed at the top of the diagram, and are responsible for sending/receiving messages. There are three stereotypes (please, see Figure 3.17):
    - Boundary class → encapsulates the connection between *Actors* and Use Cases. Basically, the Boundary class is used by *Actors* to interact with the system, providing an interface, such as menus, dialog boxes, etc.
    - Control class → it is the controller of the system, as it handles the logic of the Use Case. Like the Boundary class, the Control Class does not encapsulate any data and may disappear during the project's time.
    - Entity classes → encapsulate both data and behaviour. Entities are persistent objects, meaning that they continue to exist through the life of the system.

      The usual flow of interaction goes from an *Actor* to the Boundary class, which then, interacts with the Control class, which finally, interacts to an Entity class. The interaction between the objects follows some rules: *Actor* can interact only with a Boundary class. A Boundary class can interact with an Actor and with the Control class. Control class can interact with another Control class, a Boundary class, and an Entity class. An Entity class can only interact with the Control class, but there is no interaction between the Entity classes.

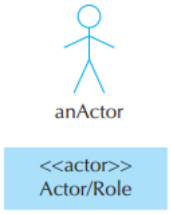


Figure 3.13: The Stick figure is the default representation of an Actor. If a non-human actor is involved, it can also be represented by the rectangle with <<actor>> in it (Dennis et al., 2009).

Figure 3.14: Representation of a message passing from one object, at the base of the arrow, to another object, head of the arrow (Dennis et al., 2009).

Figure 3.15: The dotted vertical line represents an Object's Lifetime (Dennis et al., 2009).

Figure 3.16: Focus of Control or Execution Occurrence representation (Dennis et al., 2009).

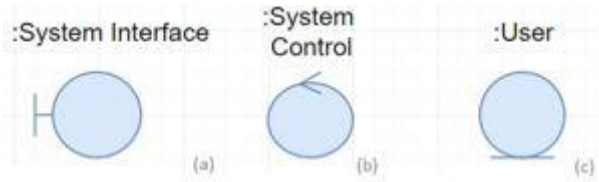


Figure 3.17: Representation of a (a) Boundary class, (b) Control class, (c) Entity class (draw.io, 2020).

Description of the Sequence Diagram for the Proposed Project

Please, refer to *Appendix A5* for the Sequence diagram (Mid-range solution plus *Add a friend*-functionality), which demonstrates the sequential flow of the actions performed by the system users (Actor User).

'Request Login' message: to start using the application, the User will be required to Login with the application (or Register, if a first-time user). The user will need to send message *Request Login* to the :System Interface (system boundary), which passes the message on to the :System Control, which will forward it to the persistent entity :User. The later holds User's information, where the Login details will be verified or, if first-time user - a new record will be created.

'Request Goals Settings' message: this message will be passed from the actor User to the :System Interface, then to the :System Control and to the persistent entity :Goal, where a new goal record will be created (if user is adding a new goal) or updated (if user is removing a goal). Alongside that, the user profile will need to be updated with the new Goal settings on the persistent entity :User (or created, if first-time user).

'Select Challenge to Achieve Selected Goal' message: the User actor will send this message to the :System Interface, which forwards the message to the :System Control, then, the persistent entity that holds challenge's information (:Challenge) will create a new record for selected challenge. Then, all the available challenges for that User will be displayed on the :System Interface. Once the user selects a specific challenge, the message 'get selected challenge details' will be sent to the :Challenge entity.

'Request Manual Update of Tracking Record for User Selected Challenge' message: the message request is sent to the :System Interface and is further passed on to the :System Control, which then sends the request to the :Challenge Progress Report entity to manually update tracking record of the selected challenge for completion. If a smartwatch can be connected to the user selected challenge, the message 'get smartwatch reading' will be sent from the :System Control to the :Smartwatch Interface. The new record of smartwatch reading will be done in the :Smartwatch Recordings Reading entity, through a message passed from the :System Control. In this case, the updating of tracking record will be done automatically as opposed to manually, by passing the message 'system update of tracking record of user selected challenge for completion according to smartwatch reading' from the :System Control to the : Challenge Progress Report.

Furthermore, the :System Control will check the points achieved in the completed challenge. If the user achieved the total points in that challenge, the status of that specific challenge will

be changed to *achieved*. The message 'update challenge status (achieved)' will be passed from the :System Control to the : Challenge Progress Report. If the user's total points meet the total points to achieve a goal, the goal status will be changed to *achieved.* The message 'update goal status (achieved)' will be passed from :System Control to the : Challenge Progress Report. In both messages, the *achieved* value will be either 'Yes' or 'No'. Finally, the :System Control will forward another request to the : Challenge Progress Report to create a new record of challenge completion for the Report. Then, the :System Interface will display the Report.

'Select User Friend' message: At the completion of a challenge, the user can optionally share the results with a friend. In order for this to happen, the User actor will send the message to the :System Interface, which will forward it to the :System Control, which then will send it to the :User Friend entity to get the user's friend details. Finally, the :System Interface will display the Challenge Completion Report to the selected friend.

3.3.3 Object Class Diagram

Object Class diagrams show the relationship between classes, interfaces and instances of the class and their interactions. Differently from the Use Case, Object Class diagram is also concerned with the implementation of the classes and interfaces. As presented on Figure 3.18, each class will contain attributes and methods.

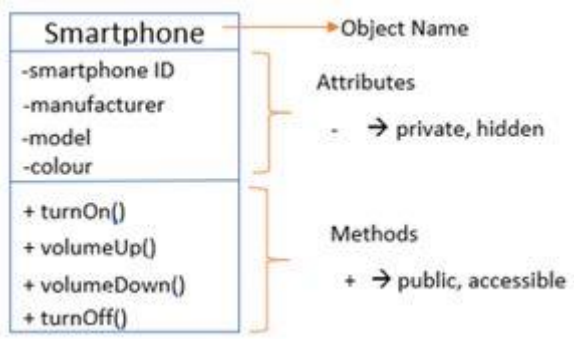


Figure 3.18: A class 'Smartphone' with attributes and methods. The *minus* sign indicates that is a private property, and the *plus* sign indicates that is public.

Attributes describe the characteristics/properties of an object and its current state. It can also be derived from other attributes. The methods represent actions or functions that the object can perform.

Associations between classes can be represented with a solid line (please, see Figure 3.19). It represents a relationship between instances of different classes or within itself, along with multiplicity symbols (see Figure 3.20), representing the minimum and maximum cardinality a class instance can be associated with instance(s) of another class (Dennis et al., 2009).

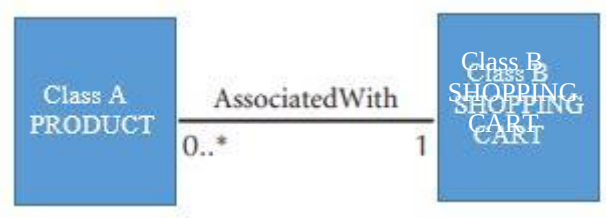


Figure 3.19: Example of a relationship between two classes. Assuming one customer is allowed to have only one shopping cart, the cart may be empty (containing zero products inside) or have multiple products added, hence the notation 0..* (zero to many).

Exactly one	1	Department — 1 — Boss	A department has one and only one boss.
Zero or more	0..*	Employee — 0..* — Child	An employee has zero to many children.
One or more	1..*	Boss — 1..* — Employee	A boss is responsible for one or more employees.
Zero or one	0..1	Employee — 0..1 — Spouse	An employee can be married to zero or one spouse.
Specified range	2..4	Employee — 2..4 — Vacation	An employee can take from two to four vacations each year.
Multiple, disjoint ranges	1..3,5	Employee — 1..3,5 — Committee	An employee is a member of one to three or five committees.

Figure 3.20: Examples of Multiplicity (Dennis et al., 2009).

Object Class diagrams represent the concept of inheritance through Generalization, where classes with common attributes and methods can be grouped together. Figure 3.21 is an example of the Generalization concept and also, the Specialization concept, where an object

may be subdivided into subclasses, which inherit common properties of the Superclass and have their own characteristics. Note the representation with the triangle on the solid line.

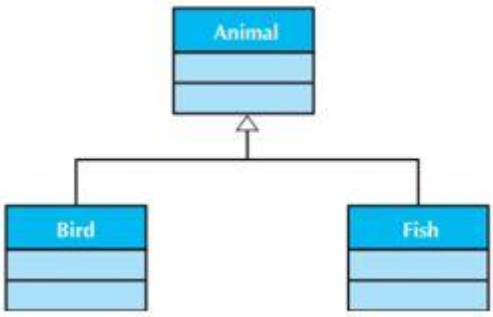


Figure 3.21: Subclasses ‘Bird’ and ‘Fish’ inherits the general properties from the Superclass ‘Animal’, along with its own properties (Dennis et al., 2009).

Aggregation and Composition are special types of associations between classes. An aggregation represents a logical *a-part-of* relationship between different classes, or instances of the class itself. For example, a calculator has multiple buttons, therefore, a button is **part of** the calculator. More simple examples can be observed on Figure 3.22 - note the *diamond* shape on the solid line that represents the Aggregation.

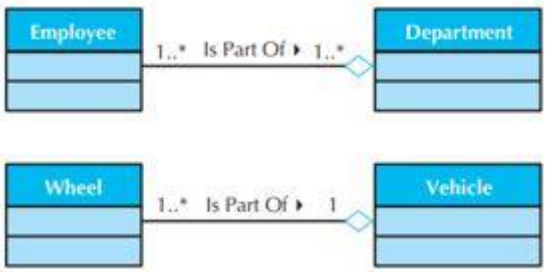


Figure 3.22: Two examples of Aggregation associations. Example one represents a department that contains employees (one department can contain one or more employees, 1..* - one to many cardinality), and each employee can belong to one or more departments. As per the Aggregation association, an employee **is part of** a department. On the example two, one vehicle can have one or more wheels, however, a wheel can only **be part of** one car (Dennis et al., 2009).

A Composition represents a physical *a-part-of* relationship between one or more classes, or a class itself. It is a stronger case of Aggregation, where both parts are meaningless without the other. Figure 3.23 illustrates some examples of Composition - note the *filled diamond* shape on the solid line that represents this type of association.

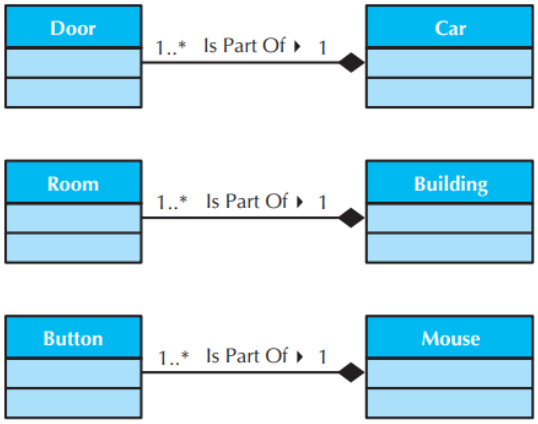


Figure 3.23: Examples of Composition associations. A car needs its doors, and a door is meaningless on its own. Buildings are composed of rooms, there is at least one room in each building and each room **is part of** a building. A mouse and button(s) compose a working object, which each would be useless without the other (Dennis et al., 2009).

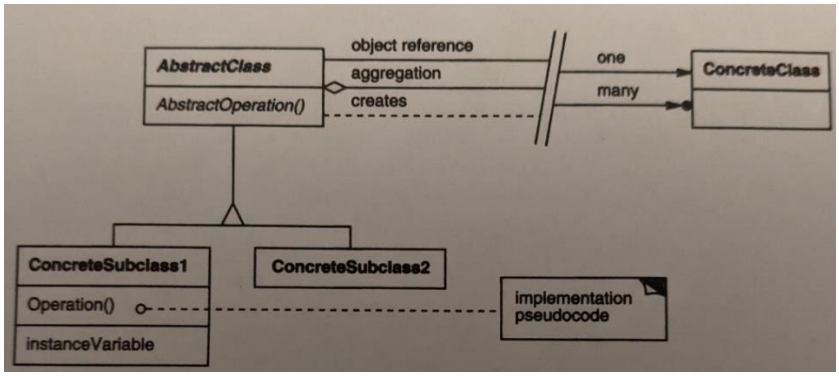


Figure 3.24: Class diagram notation (Gamma et al., 1995).

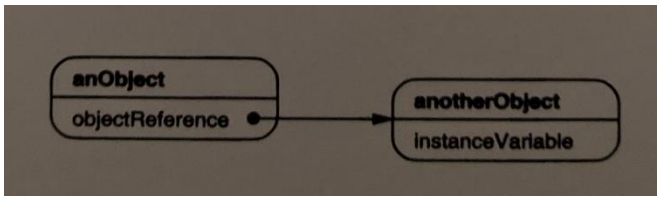


Figure 3.25: Object diagram notation (Gamma et al., 1995).

Description of the Object Class Diagram for the Proposed Project

Please, refer to *Appendix A6* for the Object Class diagram of the Project. The presented Object Classes are as follows:

User

This class stores the following attributes:

- User ID (Primary Key)
- Name
- Last Name
- Username (email address)
- Password
- Goals (Foreign Key)
- Friends (Foreign Key)
- Challenge Progress Report (Foreign Key)
- DOB
- Gender
- Weight
- Height

The methods that operate on the attributes values are:

+ Create new record (if first time user)
+ Verify Login details
+ Create new profile record/ update profile
+ Get user's friend details

Goal

This class stores the following attributes:

- Goal ID (Primary Key)
- Goal total points
- Category (which can be exercises, nutrition, sleep, gratitude, social interactions, or self-awareness)

The methods that operate on the attributes values are:

+ Create new goal record
+ Update record

Challenge

This class stores the following attributes:

- Challenge ID (Primary Key)

- Goal ID (Foreign Key)
- User ID (Foreign Key)
- Title
- Description
- Total Points
- Completed date (derived)
- Points achieved (derived)
- Photo

The methods that operate on the attribute values are:

+ Create new challenge record

+ Get selected challenge details

Smartwatch Recordings Reading

This class stores the following attributes:

- Smartwatch Recording ID (Primary Key)
- User ID (Foreign Key)
- Challenge ID (Foreign Key)
- Goal ID (Foreign Key)
- Steps
- Heart rate
- Sleep

The methods that operate on the attributes values are:

+ Create new record

Challenge Progress Report

This class stores the following attributes:

- Challenge Progress Report ID (Primary Key)
- Challenge ID (Foreign Key)
- Completed challenge (yes, no)
- Points achieved (if yes)
- Challenge status achieved (yes, no)
- Goal status achieved (yes, no)

The methods that operate on the attribute values are:

+ Manually update tracking record of user selected challenge for completion

+ System update of tracking record of user selected challenge for completion according to smartwatch reading

+ Update challenge status (achieved)

+ Update goal status (achieved)

+ Create new record of challenge completion for report

3.4 Alternative Design Solutions

In this section, three alternative design strategies will be developed and compared, but only one alternative will be chosen for the purpose of the application development. Please, see *Appendix 7*.

3.4.1 Low-end Solution

This alternative presents the basic functionalities of the application. It comprises of all elements necessary for the proposed application to function.

Two Actors are involved, the User and the Administrator, who are responsible for handling the password resets and to create new challenges.

The User is required to *Register* to use the proposed application. At the Registration stage, a profile will be created for the user with their personal data and goals. Once the User has registered, they will need to Login to access their account. If the User does not remember the password, they have the option to request the administrator to reset it.

All functionalities proposed for this alternative are as follows:

- Login/ Registration (create profile)/ Reset Password.
- Create Challenges.
- Set Goals at registration process or edit Goals later.
- Access challenges.
- Complete challenges.
- Collect points at a Challenge completion.

- Track completed Challenges.
- Visualise completed Challenges for Progress Report.

As the alternative presents only basic functionalities, this solution on its own will not retain the users.

3.4.2 Mid-range Solution

The Mid-range Alternative includes all basic functionalities from the Low-end solution with some added essential functionalities, which are the ability of the user to *connect with a smartwatch* to visualize and monitor vital signs, such as heart rate, steps, and quality of sleep.

The functionalities proposed for this Alternative are as follows:

- All the functionalities from the Low-end Solution, **plus**:
- Connect with Smartwatch.
- Visualise Monitoring of vital signs.

This Alternative gives more accuracy on the user's progression with the challenges, making it more interesting for the user, as the data is tailored-specific for them. Sleep quality is very difficult to measure without a smartwatch device, as this data will rely on the user's discipline to manually input every day's sleep information into the application. Also, it relies on user's memory on what time they actual fell asleep/ woke up, as they probably will remember what time they went to bed or get up, neglecting the time spent on their phones, or reading a book for example.

3.4.3 High-end Solution

This Alternative includes all the Low-end and Mid-range solutions functionalities, along with some extra, advanced functionalities, which can be good to have.

For this Solution, two more *Actors* are included: Third-party Secure Payment system, and a Mental Health Organisation volunteer. The User will be able to initiate a *chat* with the organisation in case they are struggling with some mental health issues. Also, the User will be able to donate to such an organisation. Along side that, the User will be able to keep a list of friends, who they wish to share a challenge with, and share a photo of the completed challenge.

In summary, all functionalities for the High-end solution are:

- All functionalities of the Low-end and Mid-range solutions, **plus**:
- Share accomplishment of a challenge with a friend.
- Add photo of the accomplished challenge.
- Add a friend.
- Initiate Chat process.
- Access Queue of users in the Chat (for Mental Health Organisation only).
- Make a donation.

As per results received from the Initial Survey performed for the Project, the *donation* functionality did not receive many responses. The additional functionalities of the High-end Alternative will not be included in the first Prototype development process for the application, due to its complexity and time constraints for the project. As most realistic and appropriate Design Alternative for the current Project is considered the *Mid-Range* Solution.

3.5 Chapter Summary

This chapter presented an overview of the Object-Oriented environment and its characteristics, a comparison of different Software Development methodologies, a description of the Unified Modelling Language diagrams (Use Case, Sequence and Object Class diagrams) used to analyse the proposed project and presented three Alternative Design Solutions for the proposed project (ranging from basic functionalities to more complex ones).

As discussed in this chapter, the proposed application is an Object-Oriented system that will be developed, following a feedback-driven approach, using the Dynamic Systems Development Methodology (DSDM) as an Agile method for software development. Object-Oriented Programming allows flexibility, reuse of code, systems built are easy to maintain and can ensure robustness if properly implemented. The DSDM proved to be the most suitable for the time available to develop the application, along with the benefits of developing the prototype system in stages and improving it with each feedback received.

After evaluation of the three proposed alternative design strategies, the *Mid-range* solution proved to be feasible for the time available to produce the first prototype version, presenting not only the basic functionalities, but as well some essential functionalities to gather user's personal data in reliable way regarding progress made in their selected goals, by establishing connection with a smartwatch.

Through storyboards, wireframes and screenshots, the next chapter will describe the Interface and Database development process of the proposed application.

REFERENCES

Agile Manifesto (2001). *Manifesto for Agile Software Development.* [Online]. Available at: https://agilemanifesto.org/ [Accessed: 15 December 2020].

Bocij, P., Greasley, A. (ed.) and Hickie, S. (2008), *Business Information Systems: Technology, Development & Management*. 4th ed. London: Pearson Education.

Dennis, A., Wixom, B. H. and Tegarden, D. (2009). *System Analysis & Design UML Version 2.0: an Object-Oriented Approach.* 3rd ed. USA: John Willey & Sons, Inc. [Online]. Available at: https://saleroo.files.wordpress.com/2015/09/systems-analysis-and-design-with-uml-wiley.pdf [Accessed: 15 December 2020].

Gamma, E., Helm, R., Johnson, R. E. and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. 18th ed. USA: Addison-Wesley.

Lucidchart (2020). *UML Use Case Diagram Tutorial*. [Online]. Available at: https://www.lucidchart.com/pages/uml-use-case-diagram#section_2 [Accessed: 16 December 2020].

Mullins, T. (2000). *A First Course in Programming with Java.* Ireland: ColourBooks Ltd.

Mullins, T. (2016). *Object-Oriented Programming and the Story of Encapsulation in Java.* Faculty of Computing, Griffith College Dublin.

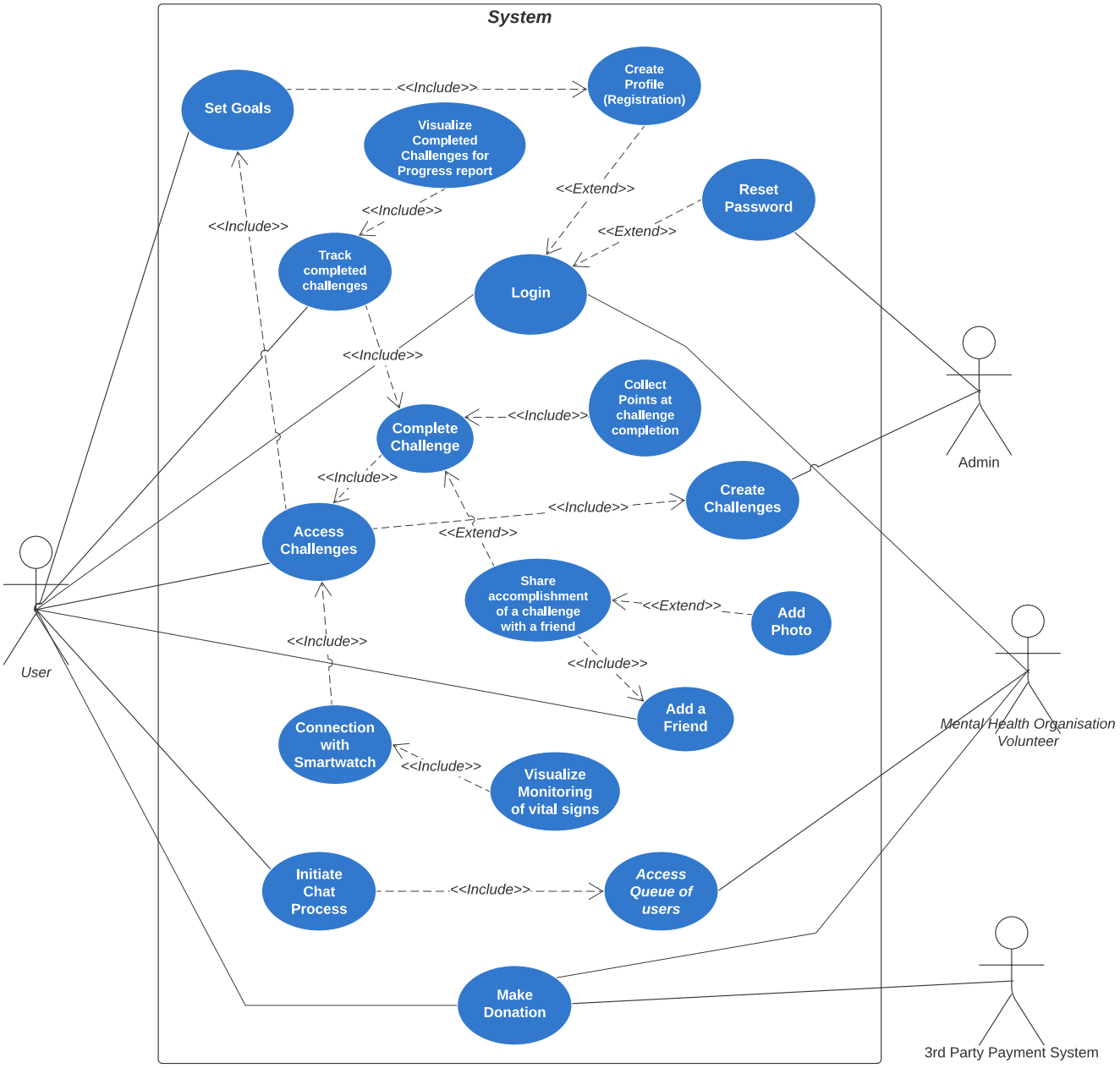
Nash, M. (2003). *Java Frameworks and Components: Accelerate Your Web Application Development.* UK: Cambridge University Press.

StateOfAgile (2020). *14th Annual State of Agile Report*. [Online]. Available at: https://stateofagile.com/#ufh-i-615706098-14th-annual-state-of-agile-report/7027494 [Accessed: 15 December 2020].

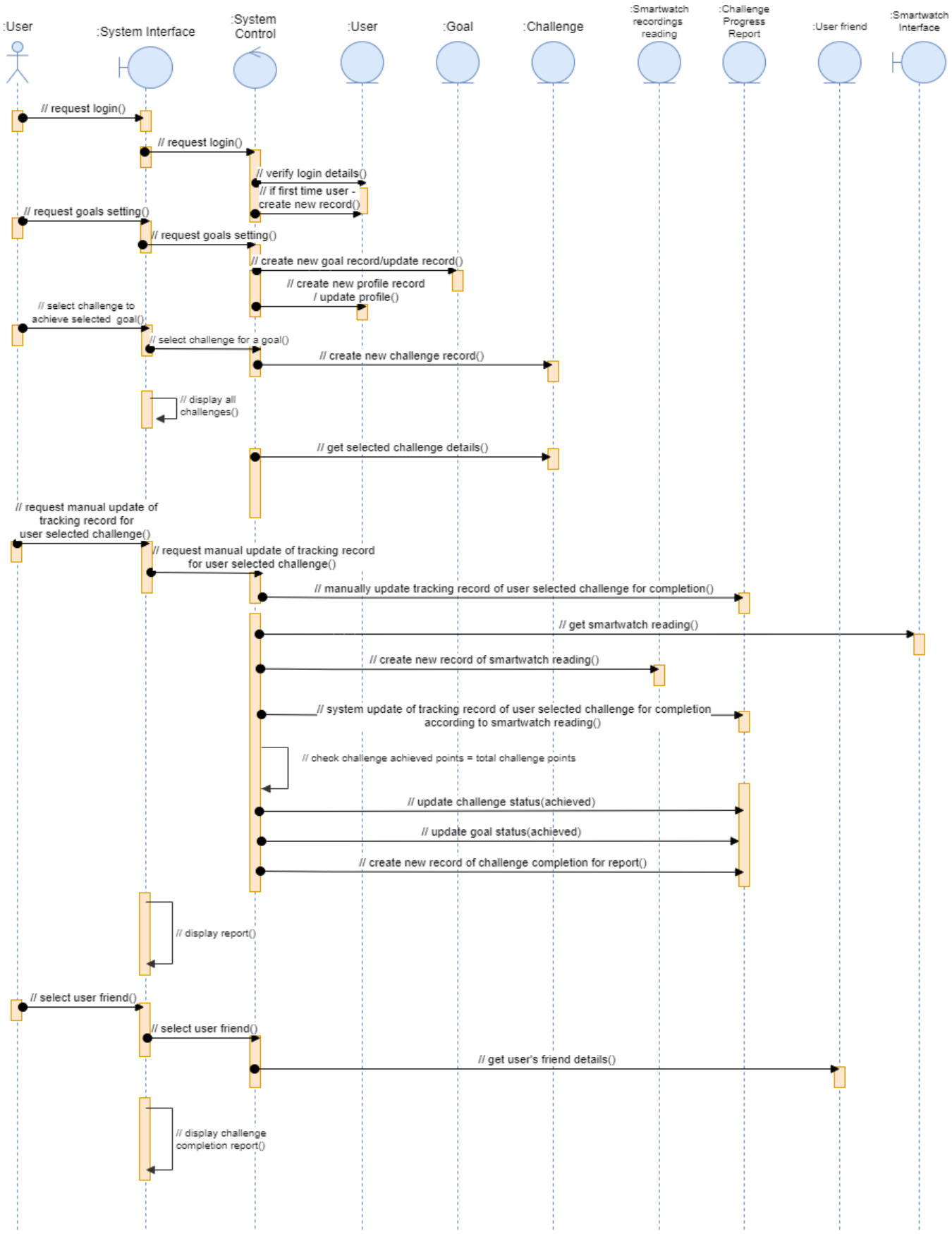
Software Testing Help (2021a), *What is SDLC Waterfall Model?* [Online]. Available at: https://www.softwaretestinghelp.com/what-is-sdlc-waterfall-model/ [Accessed: 15 December 2020].

Yilmaz, R., Sezgin, A., Kurnaz, S. and Arslan, Y. (2018). *Encyclopaedia of Information Science and Technology*. 4th ed. USA: IGI Global. [Online]. Available at: https://www.researchgate.net /publication/317957956_Object-Oriented_Programming_in_Computer_Science [Accessed: 15 December 2020].

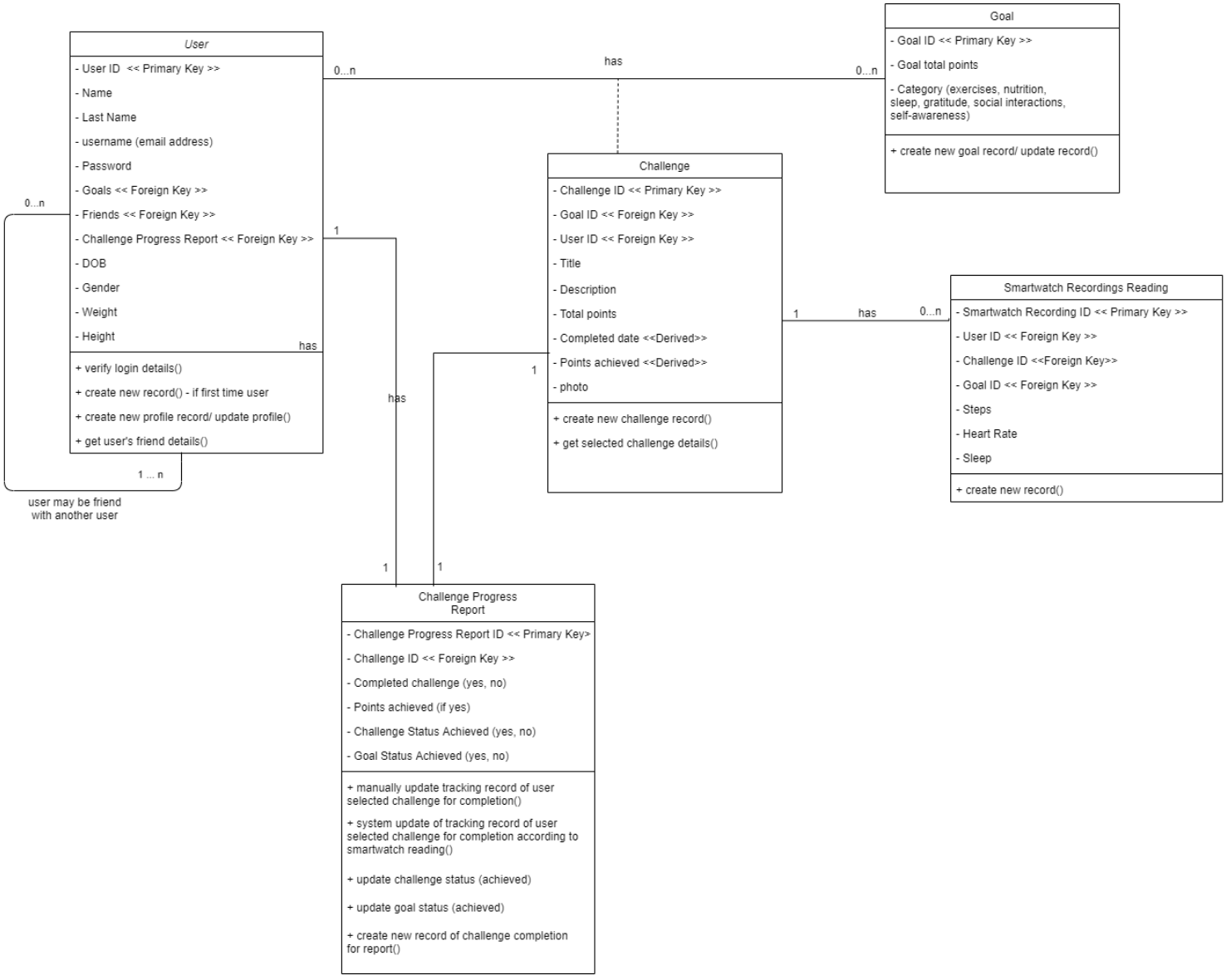
APPENDIX A4: Use Case Diagram for the Proposed Project



APPENDIX A5: Sequence Diagram for the Proposed Project



APPENDIX A6: Object Class Diagram for the Proposed Project



APPENDIX A7: Alternative Design Solutions for the Proposed Project