NTNU

Kunnskap for en bedre verden

Department of Computer Science

TDT4240 - Project Architectural Description Document

Haakon Svane:

MegaGolf

At the opposite end of minigolf

Group:
Group 29

Author:
Haakon Svane (haakohsv)

COTS:
iOS using Swift with SpriteKit, GamePlay Kit and Game Center

Primary quality attribute:
Modifiability

Secondary quality attribute:
Performance
Usability

April 28, 2021

Preface

While there are plans to release this game to App Store, this is unlikely to happen before the project deadline. The quality of the game is not expected to exceed my standards, and getting the Apple review team to verify the game can also be a lengthy process. This means that this document may include ideas that are planned to be worked on after the project deadline. These ideas however, might still be significant enough to have impact on the software architecture. This text is intended to address the possibility that the final product may or may not contain all the features mentioned in this document.

It should also be mentioned that my project team consists of myself only. This has implications on how the system can be implemented since no parallel work will be performed (my girlfriend has offered to design assets though!). The documentation that will be used in the implementation of the system (mainly the development view) is still written to allow this due to project and course requirements.

Table of Contents

List of Figures

List of Tables

1 Introduction

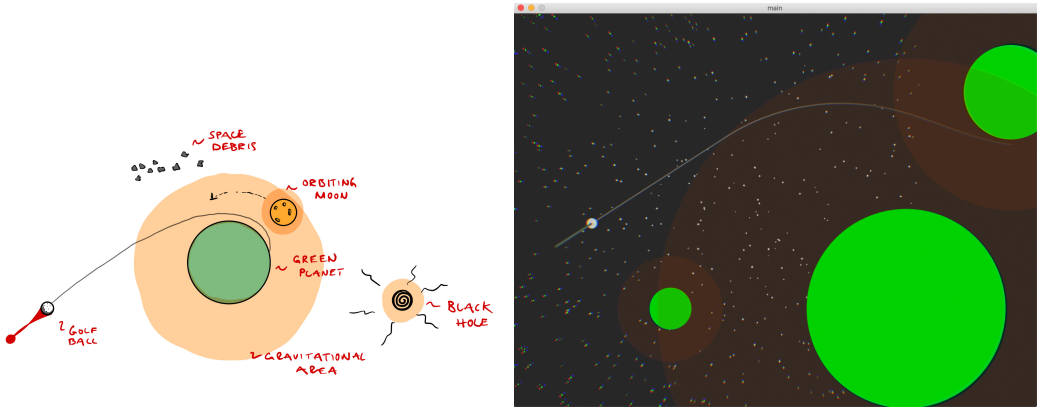
This section provides introductory information about the project, the game concept and the further structuring of this document.

1.1 Description of the project and this phase

This project is a semester project for TDT4240 - Software Architecture. The goal of the project is to design a mobile game (iOS or Android) with an online element to it. A software architecture is to be designed, documented and implemented. The purpose of this document is to act as a basis for the design of an iOS multiplayer game by exploring the architectural decisions that will comprise the final system.

1.2 Description of the game concept

MegaGolf is a golfing game set in space. It draws inspiration from both mini golf and regular golf. The aim of each level in the game is to make use of gravitational assist from planetary bodies to alter the trajectory of a golf ball, guiding it into a black hole which acts as a hole in regular golf. The game play is depicted in Figure 1a. A small proof of concept (POC) demo was made before project startup to evaluate the core mechanics of the game (gravity assisted trajectories). A screenshot of this demo is shown in Figure 1b.



(a) Illustration of the basic tokens of the game. The orange area around the celestial bodies show the gravitational area of effect.

(b) POC demo of the game to test the gravitational assisted trajectory and camera movement.

Figure 1: Game concept illustrated and demonstrated.

1.2.1 Game mechanics

Each level consists of one or several obstacles and one black hole. An obstacle may be a planet, an orbiting moon, a star or space debris. Planets come in the following categories: field planets, rough planets, water planets and bunker planets. The material that comprise the planets defines its category and their physical properties are similar to that one would see on a regular golf course. Space debris does not have any gravity so they act as deflecting obstacles rather than attracting obstacles. The player spawns some distance away from the goal (black hole) and must navigate to it by launching the golf ball using a drag and release command where the relative position of the finger at release time from the golf ball determines the velocity vector for the ball. The scoring of the game follows normal golfing conventions: each level has an associated par in which the difference between the player score and the level par determines the score awarded for the level. Lower is better.

1.2.2 Menus and modes

The main menu of the game presents a single player and a multiplayer mode. Single player gaming consist of unlockable levels that the player can unlock by completing previous ones. Levels are subgroups of solar systems such that a single solar system contains multiple levels. All levels in a solar system share a similar theme in terms of game play. The multiplayer portion of the game will consist of real-time player vs. player matching. Players will be able to invite friends or play vs random opponents around the world. An online game consists of playing through all levels in a solar system and the winner is the player with the best (lowest) score.

1.3 Structure of the document

This document describes the architectural phase of the project. Section 1 provides a description of the game concept and its mechanics. Section 2 lists all the architectural drivers and the significant requirements (ARSs) relevant for the design of the system. Stakeholders of the project and their concerns are listed in Section 3. A presentation of the views that are selected to illustrate the architecture is shown in Section 4. Tactics and patterns chosen for the project are respectively presented in Section 5 and 6. The views with descriptions of them are presented in Section 7. Section 8 and 9 are concerned with discussing the choice of views, the consistency among them and and the consequences of using the chosen architecture. Finally, issues faced during the development of this document and changes made to it are respectively presented in Section 10 and 11.

2 Architectural Drivers / Architectural Significant Requirements (ASRs)

The choice of architectural tactics, architectural patterns and design patterns are motivated by multiple drivers. This section lists important drivers that will guide the development of the system.

2.1 Functional drivers

2.1.1 The medium and its core function

Perhaps the most important driver for the creation of the architectural system is that the resulting system is a video game. In general and broad strokes, video games are filled with states, views and entities. In addition to this, MegaGolf will consist of multiple levels which when looked at from a modifiability perspective must be taken care of properly.

2.1.2 Online multiplayer

The game will offer online multiplayer functionalities, specifically real-time free-for-all game play using peer-to-peer networking with Game Center services. An important aspect to consider is that each of the opposing players will not interact with each other in real time as the golf balls will not interact.

2.2 Quality drivers

2.2.1 Modifiability

Levels The game will have multiple levels structured in a hierarchical order. Functionally similar levels belong to the same planetary system and visually similar levels belong to the same

galaxy. There are multiple planetary systems in a galaxy. Adding and changing galaxies and planetary systems requires a streamlined process that is efficient and effective.

Customization To personalize the game experience, items such as the golf ball, golf clubs and golf ball trails are planned to be customizable. This creates a need for a system pipeline that allows for easily adding new textures into the game from texture design to texture selection by the end-user.

2.2.2 Performance

Physics simulation The game will contain many-body physics simulations. If not implemented correctly, this will negatively affect the performance of the game. Since there will be no additional server component in the online multiplayer, all the calculations must be done on the user clients.

UI responsiveness The responsiveness of the UI is crucial for providing an immersive experience to the end-user. Tactics for ensuring sufficient UI responsiveness must be put in place.

2.2.3 Usability

Ease of use Tutorials/instructions, user interface and multiplayer sessions should be implemented and designed with end-user ease of use as a top priority.

System feedback Interactable elements should provide feedback to best communicate to the end-user the intentions of use.

2.3 Business drivers

Deadline There is a strict time limitation set for the project. This affects the scope of the project.

Acceptable quality Perfection is like infinity: it can be chased, but never reached. The point at which the product quality can be described as *sufficient* is when an end-user can no longer distinguish the developer from the product.

App Store release There are plans to release the game to the App Store once its quality is deemed sufficient. A revenue stream can be created by introducing in-game currency, but the author wishes to express his hatred[1] for advertisements and "pay-to-win" elements in games. Currency will therefore be limited to cosmetic changes only and there will be ways for a player to earn this by playing the game normally as well.

Development team The development team consists of only one person. This can cause issues related to research of frameworks, research of architectures, implementation of the system, testing of the system and producing all the required documentation.

3 Stakeholders and concerns

Though the project can be considered a partially closed system [2], there are multiple stakeholders involved. Table 1 lists the project's stakeholder and their concerns in it. Note that while an ATAM session is to be carried out using this document, the ATAM evaluator will consist of myself only. For this reason, the ATAM evaluation team was omitted from the stakeholders list.

[1] Hate is a strong word. Are there any stronger ones?

[2] The main aim of the project is to teach students about software architecture from experience. This means that in principle, the resulting product (the game) will only be available as a byproduct inside the course environment (closed system). *Partially* is used since the game may or may not be released to the public.

Stakeholder	Concern
Course staff (Lecturer and assistants)	The course staff is concerned about the reviewabillity of the software and the project documentation. The staff is also concerned about how the knowledge of a student is communicated in the project documentation and its practical implementation.
Play-testers	Play-testers are needed for properly testing the multiplayer functionality. These are concerned with the usability of both the game and the distribution platform Test Flight.
Project team	The project team is concerned about the implementation of the architecture as well as its development. Assuring that the implementation follows the specified architecture and all requirements is of importance. The project team is also concerned about the quality of the final product and how well it follows the project requirements.
End-user	Concerned with the usability of the final release of the product. Concerned about the quality of the game, how interesting the gameplay is and what quality of life functionalities the game provides.

Table 1: The stakeholders of the projects and their concerns in it.

4 Selection of Architectural Views (Viewpoint)

The views that will be selected will follow the $4+1$ blueprint model developed by Kruchten [1995]. The views, their purpose, the target audience and their notation are presented in Table 2

5 Architectural Tactics

Ensuring that the architecture meets the quality attribute requirements set for the project requires planning on how to effectively target them. This section is concerned about what specific tactics should be considered for the design and implementation of the system.

5.1 Modifiability

Modifiability is about changes and the costs associated of making them. Developing a system to conform to modifiability requires considering *what* can change, what is the *likelihood* of the change, *when* is the change made, *who* makes it and what is the *cost* of the change [Bass et al., 2013, p. 117-118]. The following subsections presents tactics for modifiability addressing the project requirements and descriptions on how these will be realized in the project architecture.

5.1.1 Coupling and cohesion

All system modules have responsibilities to perform certain tasks. Similarities between inter-responsibilities (responsibilities between modules) define the *coupling* of the system. A system with high coupling is difficult to modify since a change in one module may affect other modules as well. The similarities in a module's intra-responsibilities (responsibilities inside a single module) is what defines a module's cohesion. A module with high cohesion has a unified purpose and modifying such a module reduces the risk of affecting other responsibilities since the modified module's responsibilities are likely to be unique for the system. Increasing the modifiability of a system is a matter of *decreasing* module coupling and *increasing* module cohesion. Specific tactics to ensure this for the project are listed below.

View	Purpose	Target audience	Notation
Logical view	Focuses on the functionality of the system and how it is presented to the end-user.	• Course staff • Play-tester • Project team • End-user	• UML class diagram
Process view	Presents the run-time view of the system processes and how they communicate. Highlights scalability and performance.	• Course staff • Project team	• UML state diagram • UML sequence diagram
Physical view	Describes how the software of the system maps to its corresponding hardware and how different hardware components tie together.	• Course staff • Project team	• UML deployment diagram
Development view	Describes the actual structure of the software in the development environment.	• Course staff • Project team	• UML package diagram

Table 2: Selected architectural views (viewpoints) with their purpose, target audience and notation.

Increasing semantic coherence Ensuring that the responsibilities in two different modules do not serve the same purpose. Moving such responsibilities to common modules increases coherence. Architectural decisions such as utilizing Entities & Components will help enforce this tactic as the developer is forced to isolate functionallity based on behaviour of the individual components.

Encapsulation Providing an explicit interface for interacting with a module instead of providing direct access to its inner functionality reduces coupling. This will utilized and prioritized in the presenter of the system (see: section 6) since it will consist of multiple classes that will need to communicate between them.

Using intermediaries If there are dependencies between responsibilities of different modules, introducing an external intermediate layer to handle the dependency reduces tight coupling between the two modules. This will be solved using the delegate pattern. The delegate provides an interface for modules to communicate.

Refactoring Restructuring (partially) duplicate modules by co-locating common responsibilities reduces coupling between modules. This is to be done throughout development.

Abstracting common services Abstracting common modules to a more general one. The game will consist of multiple levels. By generalizing the notion of a level, creating new are easier as each new level requires less development time.

5.2 Performance

Performance is about ensuring that a software systems meets timing requirements. A performance scenario always begins with an event. The system can either choose to *block* or *process* the event. The time it takes for a system to process an event is defined as *latency*. Since the event may be blocked for a duration before processing, the two components that determine a system's latency is block time and processing time [Bass et al., 2013, p. 131-134]. Processing requires resources and resources are finite. The tactics for fulfilling the performance requirements will therefore target two different perspectives on resources: *Resource demand* and *resource management*.

5.2.1 Control resource demand

Managing sampling rate Reducing the rate at which periodic events sample at. This will be applicable in the networking modules.

Reduce overhead Removing usage of intermediaries can yield performance increases. This needs to be evaluated together with the modifiability requirements as there often occurs trade-offs between performance and modifiability.

Increase resource efficiency Improving algorithms and other crucial processes. A famous example of this is the fast inverse square root algorithm found in Quake III Arena [3].

5.2.2 Manage resources

Increase resources Increasing the amount of resources is an efficient way to cut down latency. Because of the structure of this project, the available resources are those the iOS devices and the end-user's networks has to offer. Since MegaGolf will support online multiplayer, bottlenecks for performance may occur at either the network-end or at the processing-end of the system. If a device suffers from a too high throughput of demand, some of this can be offloaded to other devices, effectively using the idle resources of other devices in the multiplayer session as additional resources.

Concurrency Balancing the load over multiple processors using threads increases performance by reducing blocked time. System processes must be evaluated as thread safe in order for concurrency to be a viable option. This is applicable for loading screens.

5.3 Usability

The usability attribute is meant to aid in ensuring that a user accomplishes a desired task and that the system provides an adequate amount of user support. It is helpful to divide the human-computer interaction into two components:

User initiative Stimuli from the user to the system.

System initiative Stimuli from the system to the user.

Human-computer interaction can be described as a sequence of these components. Scenarios where both components come into play are named *mixed initiative*.

The following subsections presents tactics for usability addressing the project requirements and descriptions on how these will be realized in the project architecture.

5.3.1 User Interface

Interface separation Separating the user interface by making the system flexible to rapidly changing design iterations will enable testing of ideas and quickly receiving feedback from users. This process ties nicely together with the modifiabillity attribute as the user interface will be a coherent and decoupled.

Supporting user initiative Providing feedback to the user about current system processes running and allowing the user to take action betters the user experience. The practical implementations of this will range from cancelling online player search to exiting the game (returning to main menu).

Supporting system initiative More specifically, maintaining the user model will enhance the user experience. This will be used in the game settings where a user sets the preferred settings and the system stores this information for the future.

[3] https://en.wikipedia.org/wiki/Fast_inverse_square_root#Overview_of_the_code

6 Patterns

This section presents all the patterns (architectural and design) that will be utilized in the architecture.

6.1 Architectural patterns

Architectural patterns are strategies to organise the high level end of code. The following subsections describe the architectural patterns that will be used in the system.

6.1.1 Model View Presenter

The Model View Presenter(MVP) pattern aims at separating the system functionality into modules. The View acts as a structure of visual elements. Once a user interacts with these, the Presenter is notified and processes the input using data from the Model. Unlike its sibling pattern, the Model View Controller, The model and the view are completely decoupled in MVP. The pattern will be used as a foundation for the game where the presenter contains logic on the views (GUI), manages scene transitions and handles GameCenter integration.

6.1.2 Peer-to-Peer (P2P)

Apple's Game Center multiplayer services provide multiple options for the networking model between clients, but does not provide a designated server for communications and processing in an online session. This means that all data and processing must be contained within a client mesh network. The peer-to-peer architecture will be used for this. Processing of data will happen on each client device and data will be broadcast to the other connected clients when needed.

6.1.3 Entities & Components (EC)

Entity Component is an architectural pattern inspired by data oriented design. It solves issues that may arise in typical object oriented inheritance design such as *The diamond problem*[4] and also expands the modifiability of a system. Entities are empty containers that are assigned components. Each component has a specific logic and the entity behaviour is a result of the various components that are attached to it. GamePlay Kit provides a template for EC integration that works well with SpriteKit. The specific implementation this EC system will differ from traditional approaches (ECS). The reason for this is that SpriteKit hides much of its rendering and physics pipeline from the developer. The way SpriteKit is structured also posed a problem since SpriteKit scenes need SKNode objects as children if they are to be presented in the game. If all the game token behaviour is to be driven by composition rather than inheritance, the components must be self-contained and entities should not require much work outside of adding components to them. To make the component system as flexible as possible, the components that require nodes must therefore be able to communicate with each other such that they all act on the same SpriteKit parent node instance. For example, if an entity with a SpriteComponent is to be added a LightEmitterComponent, the underlying `SKLightNode` [Apple Inc., 2021b] must be added as a sibling node to the SpriteComponent `SpriteNode` Apple Inc. [2021e] under a common `SKNode` [Apple Inc., 2021c].

6.2 Design patterns

Design patterns aim at solving more localized problems. This subsection describes the design patterns that will be used in the system.

[4] https://en.wikipedia.org/wiki/Multiple_inheritance#The_diamond_problem

6.2.1 Delegation

The delegate pattern helps removing dependencies that occur when using inheritance. Using the pattern has the added benefit of encapsulating modules that use the pattern. The delegate pattern will be used for communication between presenter modules, for communicating with the physics engine (using the `SKPhysicsContactDelegate`) [Apple Inc., 2021a] and for handling touch events where the `TouchableComponent` delegates the responsibility of executing code on touch events elsewhere.

6.2.2 Factory

The factory pattern makes the creation of object instances easier. New entities can be created through a factory class using dynamic initialization. This helps the modifiability of the system as an interface for all possible entities will be provided to the developer. This will be used for creating GUI elements such as buttons, sliders and panes as well as game objects (golf ball, planets, background images).

6.2.3 State

GamePlay Kit provides state machines for use in game development. This will be used in conjunction with the MVP pattern for navigating the user interface, running game play specific logic and for differentiating the different scene states (menu scene, loading scene and game play scene).

6.2.4 Template

The template pattern provides a skeleton for modules with common behaviour. The pattern will be used extensively in the creation of entities that share similar behaviour such as the planets. These have similar components added to them, but the parameters of the components might vary slightly. This means that passing these parameters into the template is sufficient for creating different planets.

7 Architectural views

The architectural views aim at providing different perspectives into the architecture of the system. The following subsections present these different views with explanations. Before jumping in to the different views, some terminology must be explained:

SKScene SpriteKit uses `SKScene` [Apple Inc., 2021d] objects as root nodes for presenting SpriteKit content. All visual and auditory objects must be a child node of the scene in order to be presented.

MGScene An MGScene Is a system specific scene inherited from SKScene. All scenes in the system will be an MGScene of some type.

SKNode An SKNode is a general node in the SpriteKit node hierarchy. As a standalone node, it is often used as a container for other types of nodes.

MGView An MGView is a system specific view used for graphical user interface (GUI). The contents of a specific MGView is contained within a SKNode for easier management. The MGView does not contain any logic besides the initialization of its content.

7.1 Logical View

A general logical view of the system is presented in Figure 2 in the form of a class diagram. The `MGPresenter` instance owns a scene manager, a view manager and a Game Center manager that communicate through the presenter using a delegate pattern. The views are controlled by a `MGViewStateMachine` object that switch between different `MGViewState` objects. These states all have an associated MGView that contains all the visual information that a view presents. There are multiple MGScene objects available in the system, namely `MGGameScene` (parent class to all the levels) which contains all the logic for the game and its governing state machine, `MGMenuScene` that contains all the nodes for the main menu scene and a `MGLoadingScreenScene` that contains the nodes that are presented during asset loading.

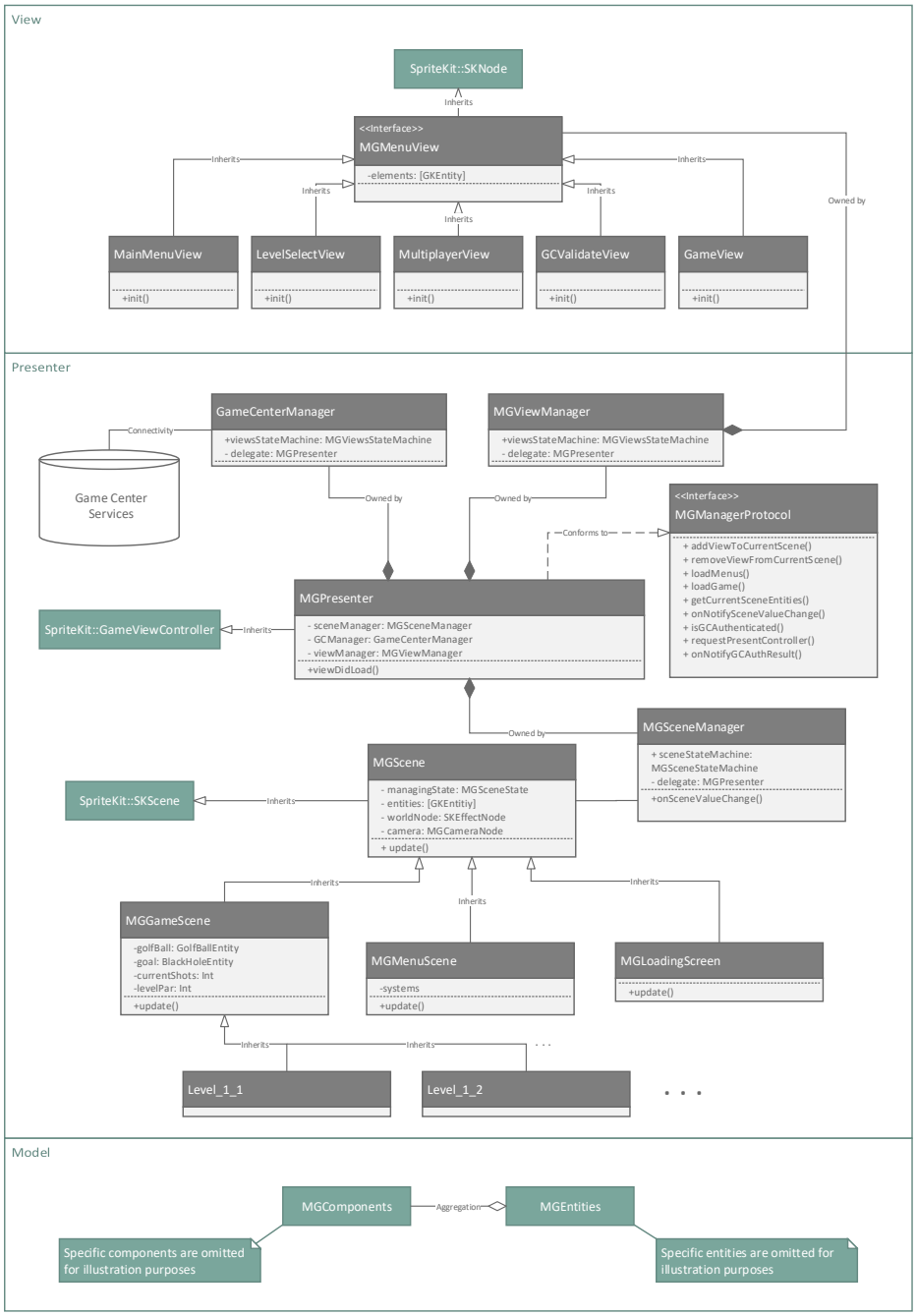


Figure 2: Logical view in UML class notation.

7.1.1 Entities and Components

A more in-depth view of the Entity & Components pattern is shown in Figure 3. This shows how three of the entities in the system will be built up using this pattern. The entity instances are free from any logic (with the exception of its initialization) while the components define its behavior. The components contain all the logic needed for its desired behavior. As mentioned in Section 6.1.3, the specific implementation of the pattern requires that it behaves nicely with SpriteKit's node system. The solution to this problem is to create a node hierarchy that each component needs to follow. This hierarchy is visualized in Figure 4. Components that require any type of SKNode have their nodes added as a child node to a parent `MGNode`. An exception to this is the `PhysicsComponent`. This component is directly added as a property to the parent MGNode since physics simulations should affect the entity as a whole. The caveat to this approach of designing the EC system is that components now become dependant on others. For example, adding a SpriteComponent requires that a NodeComponent is already added.

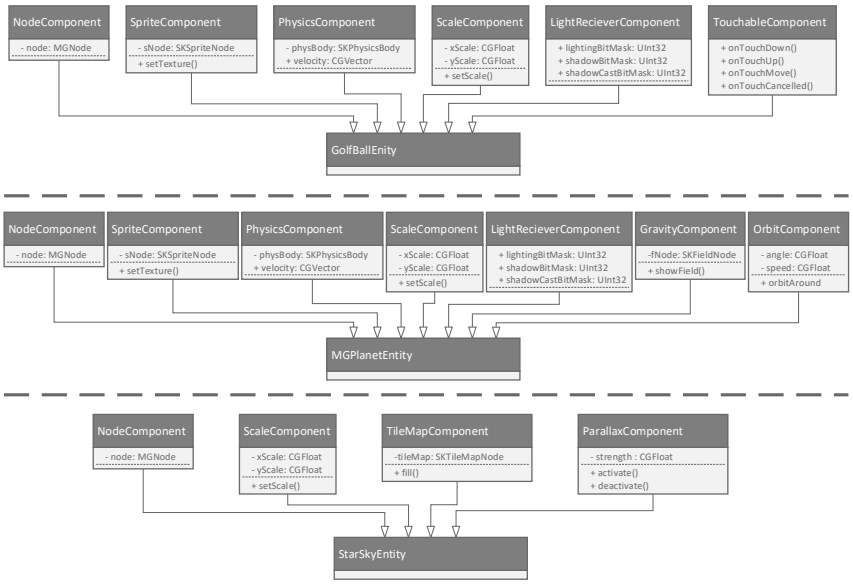


Figure 3: An example of how component composition defines entity behaviour. Note that this is a simplified example where some components and their contents have been omitted for illustration purposes.

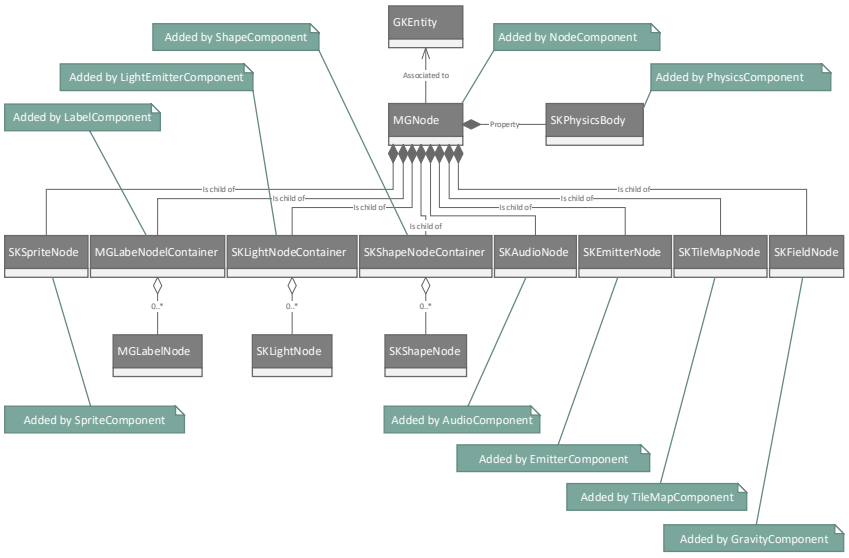


Figure 4: Structure of the node hierarchy in the EC system. Components that contain any type of SKNode are added to a root node of type `MGNode`.

7.2 Process View

A process view of the state machine governing the MGViews using a UML state diagram is presented in Figure 5. As discussed in subsection 7.1, the views state machine (`MGViewStateMachine` object) governs the MGView behaviour. Its states define the logic for each MGView (what to do with button presses, what actions to perform on the scene entities, how to animate the buttons etc.). The system starts in the main menu state where the user can navigate through various menu views (Multiplayer, single player, settings etc.). Since the states of this state machine own their respective view, there is no reason for these to be loaded in memory when playing the actual game. This gives rise to the `MGSceneStateMachine`: A state machine responsible for loading and unloading the scenes in the application (menu scene, loading scene, game scene). A UML State diagram for this state machine is shown in Figure 7. Whenever the MGPresenter makes the transition from one scene to another, the MGViewStateMachine loads/unloads all the MGViews that are needed / not needed for the next state using concurrency. A scenario of this process is illustrated in Figure 6.

The game play logic is contained within the a `MGGamePlayStateMachine` as illustrated in Figure 7. To increase the modularity of the game logic and to reduce clutter in the code, the game play is divided into states. For example, once the game starts, the ball is at rest waiting to be launched. The code for handling the visuals of this state and calculating the impulse vector of the ball at drag and release is contained within the AimingGameState. While the ball is in motion, previous mentioned code is no longer needed, so the game play transitions into the LaunchingGameState. This state checks for collisions between objects handles custom physics calculations. Once the player is finished, the game transitions into the SpectatingGameState. This state allows the player to spectate all other players currently in the level.

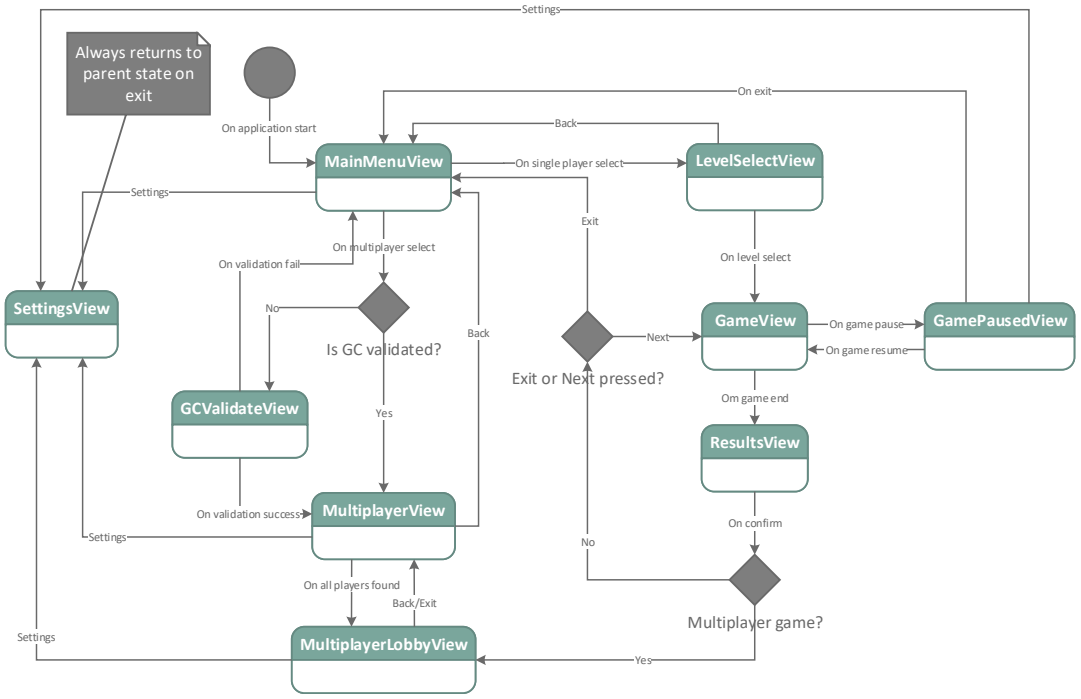


Figure 5: Process view in UML state notation of the states for the views. Note that the illustration shows that SettingsView has no exit state. This has been omitted from the illutration to reduce clutter. SettingsView will always return to its previous state when exited.

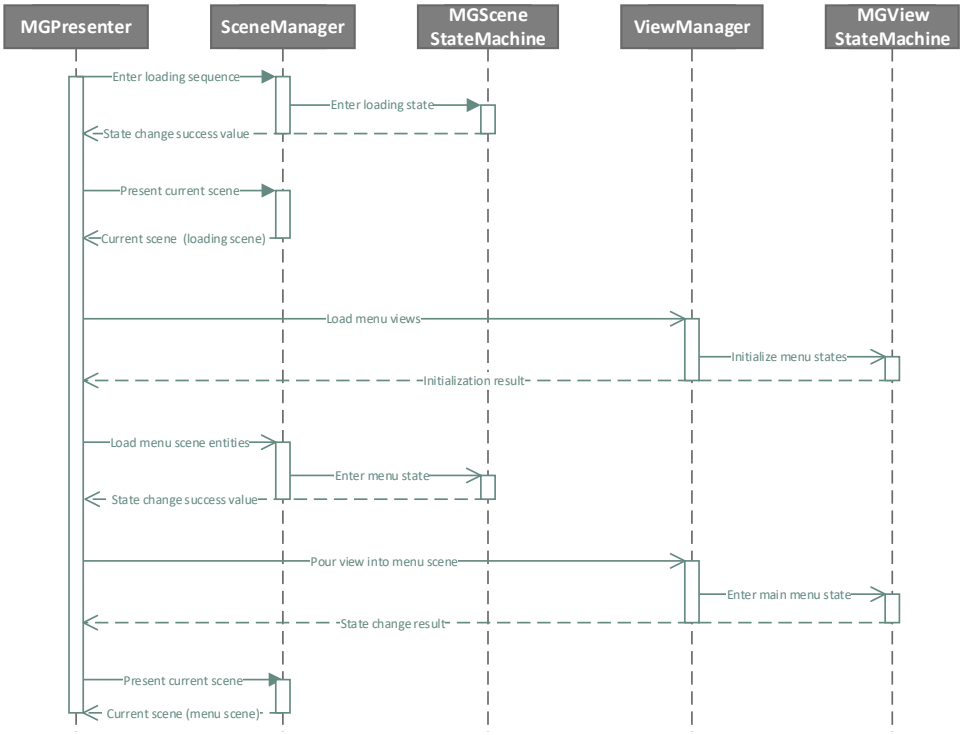


Figure 6: Sequence diagram of a loading scenario as the system moves into the menu scene in UML Sequence notation.

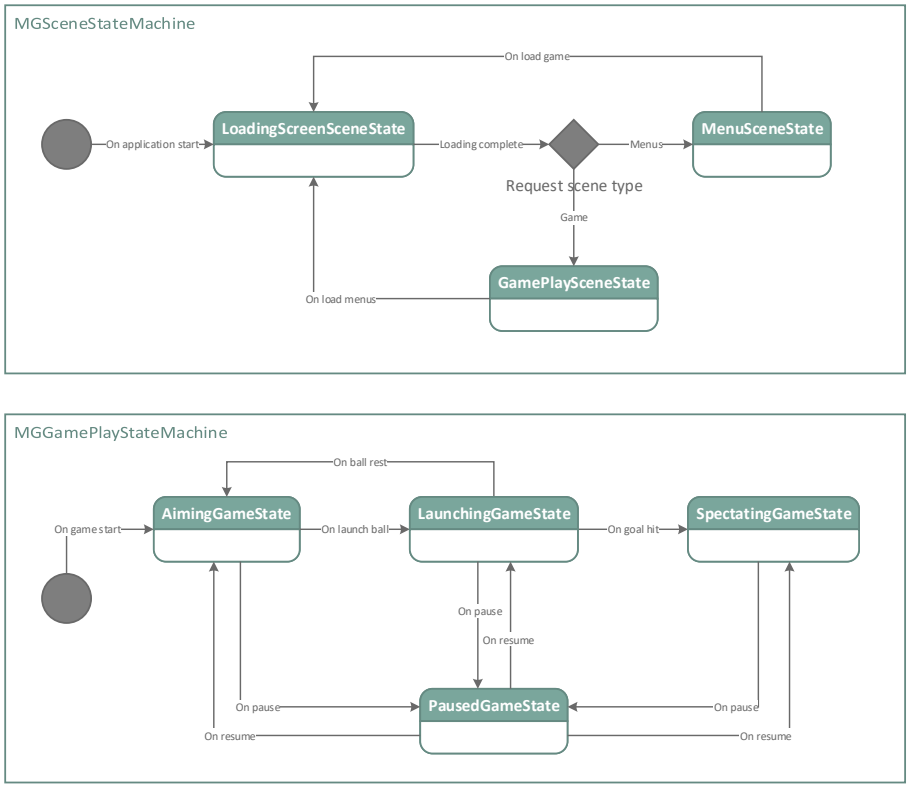


Figure 7: Process view of the scene and game play state machines in UML state notation. The MGSceneStateMachine switches between multiple scenes and the MGGamePlayStateMachine keeps track of the different states of the game play as well as its containing logic.

7.3 Development View

The development view is a simplified view of all modules/packages that are to be implemented in the system. A UML package diagram of the development view is shown in Figure 8. Dependencies between the different packages are of the greatest importance for this view since identifying these is what makes the system implementable in parallel.

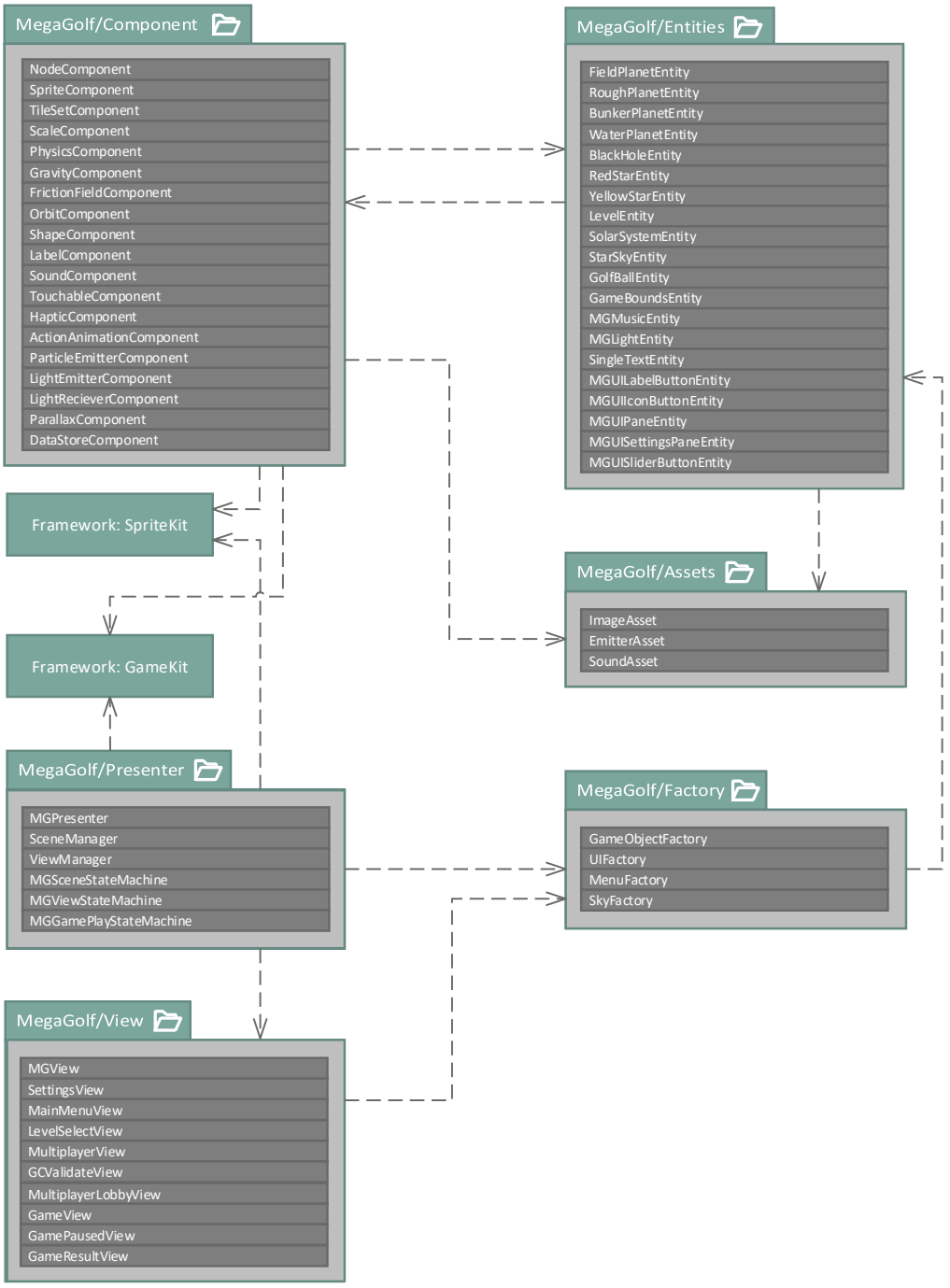


Figure 8: An example of how component composition defines entity behaviour. Note that this is a simplified example where some components have been omitted for illustration purposes.

7.4 Physical View

The physical view, illustrated using a UML Package Diagram in Figure 9, showcases two scenarios for the deployment of the system. One is in the closed developer environment consisting of the compiled bundle and a testing device connected to the developer computer with a data transfer cable, while the other a pre-release version pushed to the App Store Connect servers for beta testing with external devices over the internet using Test Flight.

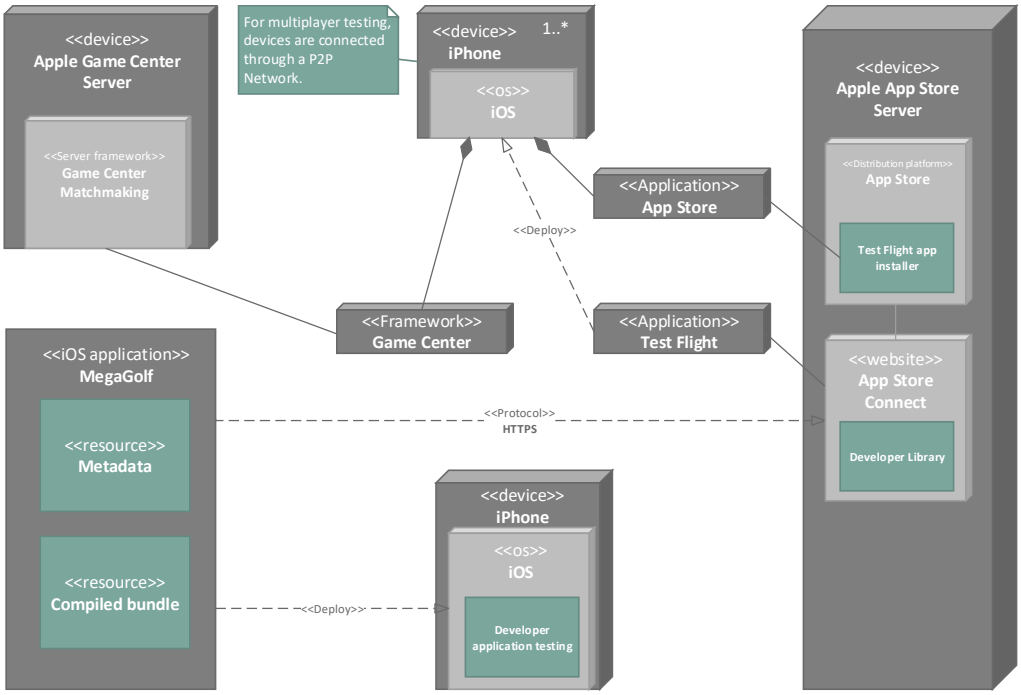


Figure 9: Physical view in UML deployment notation.

8 Consistency among architectural views

The development view is somewhat lacking in information (networking, utilities and other minor packages are omitted), but since the development team consists of only one individual, this should not cause any major conflicts during the implementation of the system architecture. The development team is inexperienced with the frameworks that are being implemented, and being solely responsible for studying detailed implementation requirements for each framework and designing the system architecture around them is a task too great for the deadlines set. Some improvisation will be needed, and the documents will be modified thereafter. Such is life.

9 Architectural rationale

This section discusses the architectural design choices and motivates them according to the project requirements.

9.1 Entities, components and MVP

At the highest granularity, the system is split up according to the MVP pattern as shown by the encompassing boxes in the logical view (Figure 2). The system also utilizes the entity component architecture. These choices has the following consequences:

Decoupled UI The UI can be modified and iterated upon based on user feedback without affecting the rest of the system. This has effects on both the modifiable and the usability of the system since more UI iterations can be produced before project deadline.

Decoupled logic Much of the game logic is contained within the interactions between the entities. This allows for rapid modifications of modules without affecting other modules. This also makes performance profiling easier since the algorithms to be implemented are easier to target, debug and surveil.

No explicit sampling No additional sampling is required for the communication between the view and the presenter by using the event-handlers provided by Apple's UIKit framework. This increases the performance of the system.

Adding functionality Since the main controller (MGPresenter) keeps track of global system states found inside the managers, adding functionality is a matter of adding more states and defining their behavior. This modular approach to system flow increases modifiability.

9.2 Event driven information flow

Utilizing Apple's event driven framework for touch input without introducing too many intermediaries is key to enhancing the user experience and the performance of the system. The composition driven design from using Entities & Components ties in well with this design philosophy since touch events will only be registered for entities with a `TouchableComponent`. Furthermore, since each component is directly linked to an entity, very little pre-processing of the touch event is needed since by design, it is clear to the developer what entity was touched, and what that specific touch event refers to.

9.3 The delegate as an intermediary

Passing event data around the system, the use of the delegate pattern ensures that the developer clearly understands the flow of information. If for example a pattern like the observer pattern was used instead, it would be unclear exactly what processes are receiving the information and what they are doing to it [5]. This can cause performance issues that may be hard to track down.

9.4 State machines

The generous use of state machines in the system has benefits for the extendability of it. Creating a framework for extending and modifying core functionality yields a system allowing for rapid change. For example, new game play states may be introduced to quickly change the flow of the

[5] I am comparing the observer pattern to the delegate pattern here. Obviously the observer pattern benefits from the fact that it may use multiple observers while the delegate pattern is a more one-to-one communications channel, but for most cases, there is no need for a one-to-many pattern.

game play. Identifying the areas that need work is easy since the game play is wrapped inside other states. The process of making system changes using states and state machines therefore boils down to the following two questions: *When* should the change happen (defining the state transitions) and *what* should happen during this change (adding logic to the state).

10 Issues

Table 3 lists issues faced during the development of this phase.

Date	**Issue**	**Description**
2021-03-01	Views design	Development of game architectural views for systems that have never been used before by project team causes issues.
2021-03-01	View information	Some information is lacking from the logical and development views, mainly general descriptions of the networking systems.

Table 3: Issues during development of this phase.

11 Changes and contributions

Table 4 lists all the changes that was made to this document over time with dates and descriptions. Table 5 lists the individual contributions made to the project at this phase with estimated hours of work.

Date	**Change**	**Description**
2021-03-01	First release	Initial revision of the document.
2021-03-08	Fix typos	Fixed typos and minor formatting errors in the document.
2021-04-15	Introduction	Reworked the introduction according to feedback.
2021-04-15	Patterns	Removed observer pattern and replaced this with the delegate pattern (Sec. 6.2.1).
2021-04-15	Patterns	Added template pattern (Sec. 6.2.4).
2021-04-15	Figures	Added figures and reworked old ones to work better with the overall document structure.
2021-04-16	Views	Rewritten Section 7 to changes in project architecture.
2021-04-16	Minor changes	Fixed minor changes among all sections based on feedback.
2021-04-16	Rationale	Rewritten Section 9 according to feedback.

Table 4: Changes made to this document over time.

Name	**Description**	**∼Number of hours**
Haakon Svane	Written the architectural documentation. Researched architectural tactics, views and design patterns. Designed the architectural views.	85

Table 5: Individual contribution of team member(s) for this phase.

Bibliography

Apple Inc. Protocol: Skphysicscontactdelegate, 2021a. `https://developer.apple.com/documentation/spritekit/skphysicscontactdelegate` [Accessed: 2021-04-15].

Apple Inc. Class: Sklightnode, 2021b. `https://developer.apple.com/documentation/spritekit/sklightnode/` [Accessed: 2021-04-15].

Apple Inc. Class: Sknode, 2021c. `https://developer.apple.com/documentation/spritekit/sknode/` [Accessed: 2021-04-15].

Apple Inc. Class: Skscene, 2021d. `https://developer.apple.com/documentation/spritekit/skscene/` [Accessed: 2021-04-16].

Apple Inc. Class: Skspritenode, 2021e. `https://developer.apple.com/documentation/spritekit/skspritenode` [Accessed: 2021-04-15].

Lee Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesly, 3 edition, 2013.

Philippe Kruchten. Architectural blueprints—the "4+1" view model of software architecture. *IEEE Software*, 12(6):42–50, November 1995.