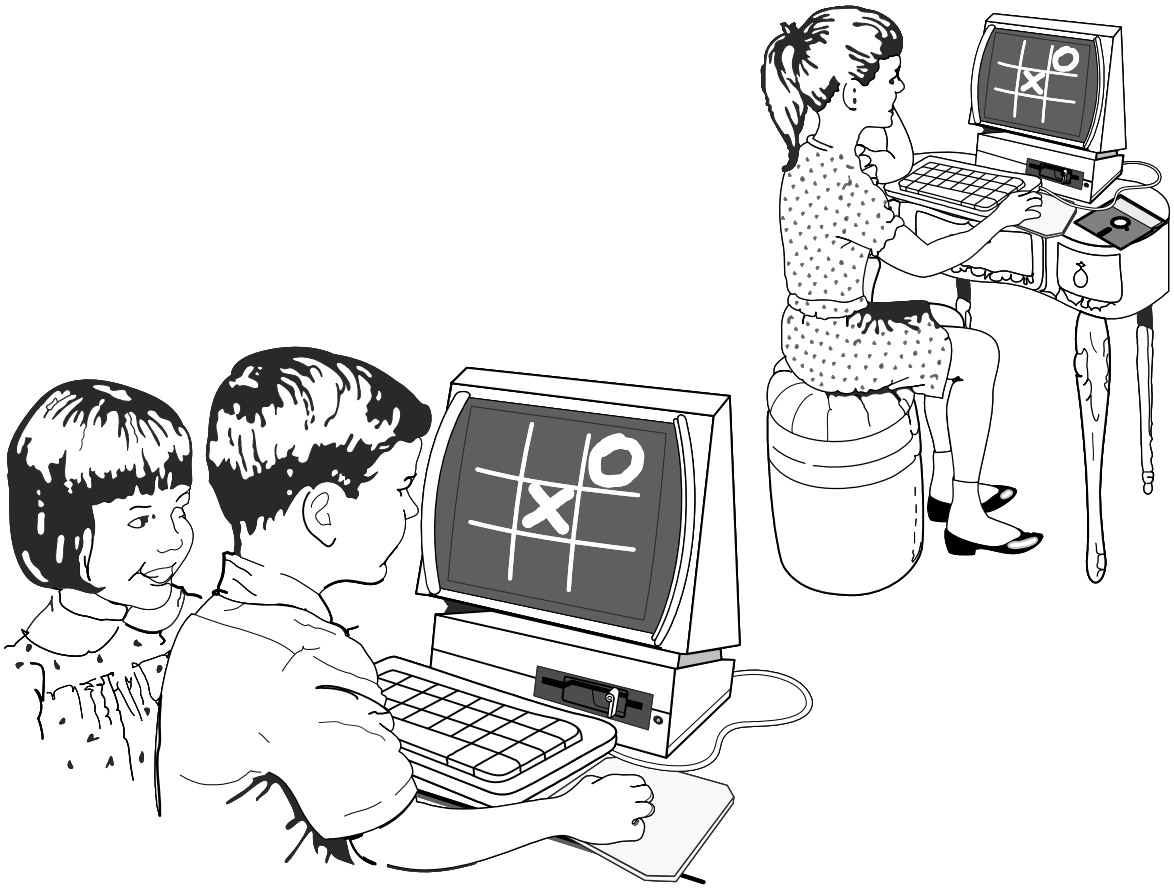
Tic-Tac-Toe Game

Example Project

This appendix illustrates a worked example of a full software engineering project. The problem is developing a distributed game of tic-tac-toe. Our development team is three strong and counts as members Me, Myself & Irene. We go under the nom de guerre Gang of Three - in team, or simply GoT-it. Me Go hails from Hunan and Irene is from Ukraine. Our strengths include: Me likes coding and prefers doing it all alone; he thinks that everything other than code is fluff. Me believes that the most successful way to solve a problem is to tackle it as a whole because you get the work done and finish it faster, instead of getting bogged down in minor details. Irene would like to manage the project. As for Myself, programming is not my forte; I lean towards my creative and critical thinking skills—I like sketching user interfaces and other impressions.



Not everyone had the same experience at the beginning of the project and not every team member had the same aptitude for learning. We hoped that the size of scope of the assignment, however,

would allow everyone to find a niche and work on different parts of the project. We figured out, with our complementary skills we are well equipped to tackle any challenges of teamwork on a large-scale project.


ME


MYSELF

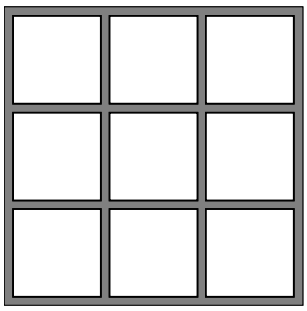

IRENE

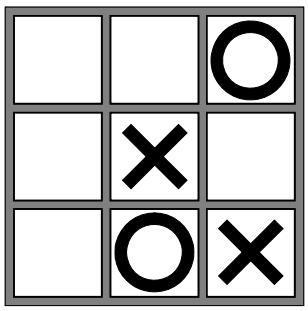
G.1 Customer Statement of Work

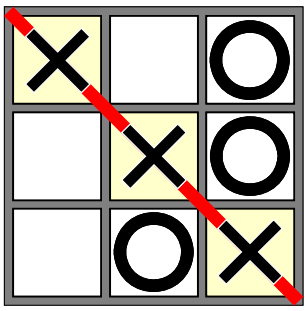
This section describes the initial "vision statement" that we received from our customer. It only roughly describes the system-to-be and the details will need to be discovered during the requirements engineering phase of our project (Sections G.2 and G.3).

G.1.1 Problem Statement

The GoT-it team is charged with building software that will allow players to play the game of tic-tac-toe from different computers. Tic-tac-toe is a game in which players alternate placing pieces (typically **X**s for the player who goes first and **O**s for the second) on a 3×3 board. The first player to get three pieces in a line (vertically, horizontally, or diagonally) is the winner. The game may end in a "draw" or "tie", so that neither of two players wins.







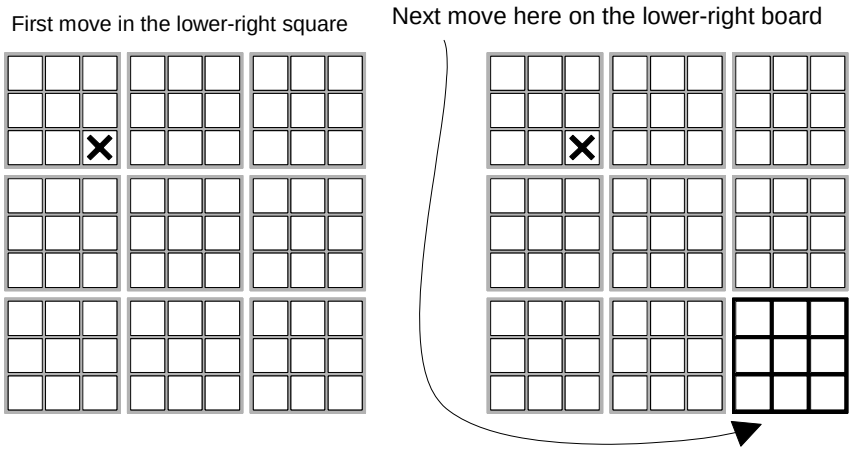


Figure G-1: Nine-board tic-tac-toe. If a move is made to the lower-right-corner cell of the first board, then the next move must be on any empty cell of the lower-right-corner board.

Motivation: Tic-tac-toe is a simple game that is fun to play. A quick search of the Web reveals that all free implementations allow the user to play against the computer. We will make it possible to play against other users and will support different versions of the game.

Vision: In addition to the default standard version, the players will be able to play two variants of the game:

- "revenge" tic-tac-toe
- nine-board tic-tac-toe

Our business plan is to offer the game free and support the operations from commercial advertisement proceeds. Other versions are planned for the future, if the game proves popular and the budget allows it.

Each player will be able to invite an opponent for a match, or may just wait to be invited by another player.

The game will also show the leaderboard—a scoreboard displaying the names and current scores of the leading competitors.

G.1.2 Glossary of Terms

● **Leaderboard** — a scoreboard displaying the names and current scores of the leading competitors.

● **Nine-board tic-tac-toe** — nine tic-tac-toe boards are arranged in a 3×3 grid to form a 9×9 grid (Figure G-1). The first player's move may go on any board; all subsequent moves are placed in the empty cells on the board corresponding to the square of the previous move (that is, if a move were in the lower-right square of a board, the next move would take place on the lower-right board). If a player cannot move because the indicated board is full, the next move may go on any board. Again, the first player to get three pieces in a line is the winner.

● **Game board** — the board with a 3×3 grid in the standard tic-tac-toe, on which the players move their pieces.

● **Game lobby** — the initial screen shown to the user when he or she logs into the system or after a match is finished and the game board is removed.

● **Gameroom** — the private session established between two players to play a match of tic-tac-toe. After the match is finished, the gameroom is destroyed and the players are brought to the initial screen. The rationale for this design is explained later in Section G.3.4.

● **Player list** — the list of all players that are currently logged in the system and available to play the game of tic-tac-toe.

● **"Revenge" tic-tac-toe** — the first player with 3-in-a-line wins, but loses if the opponent can make 3-in-a-line on the next move.

● **Response time limit** — the interval within which the remote player is expected to respond to local player's actions. If no response is received, it is assume that the remote player lost network connection or became disinterested in the game. Additional description provided in Section G.2.1.

Additional information from Wikipedia: http://en.wikipedia.org/wiki/Tic-tac-toe

G.2 System Requirements Engineering

The key lesson of this section that the reader should learn is that we are not just writing down the system specification based on the customer statement of work. Instead, we are *discovering* the requirements. The customer statement of work contains only a small part of knowledge that we will need to develop the system-to-be. Most of the knowledge remains to be discovered through careful analysis and discussion with the customer. We will discover issues that need to be resolved and make decisions about business policies that will be implemented by the system-to-be.

G.2.1 Enumerated Functional Requirements

We start by deriving the functional requirements from the statement of work (Section G.1.1). In our case, most functional requirements are distilled directly from the statement of work. However, some requirements will emerge from the requirements analysis. In addition, all requirements should be analyzed for their clarity, precision, and how realistic they are given the project resources and schedule.

We also need to consider if we need any non-functional requirements, which are usually less conspicuous in the statement of work. We realize that the players are remote, which raises the issue of latency and generally of poor awareness about each other's activities. We do not specify any latency requirements, because it is not critical that the other player immediately sees each move. The players are allowed time to contemplate their next move, so any network latency cannot be distinguished from a thinking pause. However, to avoid awkward situations where

players have to wait for the other's response for annoyingly long intervals (e.g., a player quits in the middle of a game), we introduce a RESPONSE TIME POLICY:

TTT-BP01: each player is required to respond within a limited interval or lose the match

This policy is not really a non-functional requirement and will not be stated as such. It may be refined in the future, so that players can request to suspend a match for a specified interval, or instead of automatically penalizing an unresponsive player, the system may first send alerts to this player.

Enumerated requirements for the system-to-be are as follows:

Table G-1: Enumerated functional requirements for the distributed game of tic-tac-toe.

Identifier	Requirement	PW
REQ1	The system shall allow any pair of players to play from different computers	5
REQ2	The system shall allow users to challenge opponents to play the game	4
REQ3	The system shall allow users to negotiate the game version to play	3
REQ4	The system shall allow users to play the standard tic-tac-toe or any of the two variants, as selected by the users	4
REQ5	The system should show a leaderboard of leading competitors and their ranking	2
REQ6	The system shall allow users to register with unique identifiers	1
REQ7	The system shall allow users to set their status, such as "available," "engaged," or "invisible"	2

Our customer explained the priority weights (PW) as follows. Obviously, the key objective for this system is to allow playing a distributed game of tic-tac-toe. So, REQ1 has the highest priority. It is desirable that the players can challenge other players (REQ2), but this is not a top priority in case it proves difficult to implement and a simple solution can be found, such as communicating by other means (e.g., telephone) and connecting only two players at a time. Thus, REQ2 has a lower priority. REQ3 has an even lower priority. REQ4 says that the system should support different variants, but they may not be negotiable using our system (REQ3). Instead, the players may use different means to negotiate the version (e.g., telephone) and the start our system. The last three requirements (REQ5–REQ7) are desirable, but may be dropped if time and resources become exhausted.

The requirement REQ2 does not specify if any player can invite any other player or only the players who are not already playing. We may introduce an option that the system informs the inviter that they must wait until the invitee completes the ongoing game first. This issue is related to REQ7 and will be discussed below and in Section G.3.3.

The requirement REQ4 is compounded because it demands the ability to play a game variant, as well as the ability to select among different variants. To facilitate acceptance testing of this requirement, it is helpful to split it into two simpler requirements:

REQ4a: The system shall allow users to play a variant of tic-tac-toe (standard, revenge, and nine-board)

REQ4b: The system shall allow users to select which variant of tic-tac-toe to play

The last requirement (REQ7) originally was not requested by the customer, but the developer may suggest it as useful and introduce it with customer's approval. However, adding this requirement is not as simple as it may appear at first. What is meant by "available" or "engaged"? Does "available" mean that this player is generally open to invitations to play the game, or it has a more narrow meaning representing a currently idle player? If former, what the system should do if an "available" player receives an invitation while he or she is playing the game? Should the system pop up (possibly annoying) dialog box and ask whether he or she accepts the invitation? These issues are also related to REQ2. In the latter case where the status represents the status in the current instant, will the player explicitly manage his or her status, or will the system do it automatically. That is, when a player finishes a match, his or her status will automatically change to "available." If the system will support different statuses, instead only idle versus playing, then players will need explicitly to indicate their availability to other players. Allowing players to play multiple matches at the same time would further complicate the player status issue. The reader must be aware that such issues must be resolved at some point. Because of this difficulty, we will leave requirement REQ7 out of further consideration.

The NOT List—What This Project is *Not* About: Additional requirements may be conceived, such as allowing users to search for opponents by name or some other query parameter, invite their social network friends to play the game, organize tournaments, etc. In such cases, invitations can be sent not only to currently logged-in idle players, but also to any person possibly outside of our system. The system could save the state of the matches at each turn so that players may play at their own pace. The system could maintain the history of all matches for all users, and each user would be able to view his or her statistics: the history of matches, scores, and the past opponents. Playing against a computer opponent may also be supported. The system could also allow players to chat with each other. These extensions were not asked for in the problem statement, so they will not be considered.

Note that we may need to provide more details for some requirements. Some things in the listed requirements may be tacitly assumed by one person, but may not be self-evident, so it is safer to be specific about them and avoid issues down the road. Here are some examples of additional details for some of the requirements:

Table G-2: Extending the list of functional requirements from Table G-1.

Identifier	Requirement
REQ2a	(as REQ2 in Table G-1)
REQ2b	The system shall allow the invited user to accept or decline the challenge
REQ2c	The system shall allow the user to challenge only one opponent at a time—no simultaneous pending invitations are allowed
REQ3a	(as REQ3 in Table G-1)
REQ3b	The system shall not allow the players to change the game version during an ongoing match
(REQ4a, REQ4b listed above)	
REQ4c	The system shall allow each player to play no more than one match at a time
REQ4d	The system shall allow a player to forfeit an ongoing match
REQ4e	The system will end every match in either a win or a draw and adjust players'

	rankings accordingly
(REQ5, REQ6 remain same as in Table G-1; REQ7 is rejected)	

REQ2c in Table G-2 is intended to keep the system simple, so that the program does not need to keep track of pending invitations and resolve multiple acceptances. The reader should recognize that this is a BUSINESS POLICY, which may have different solutions:

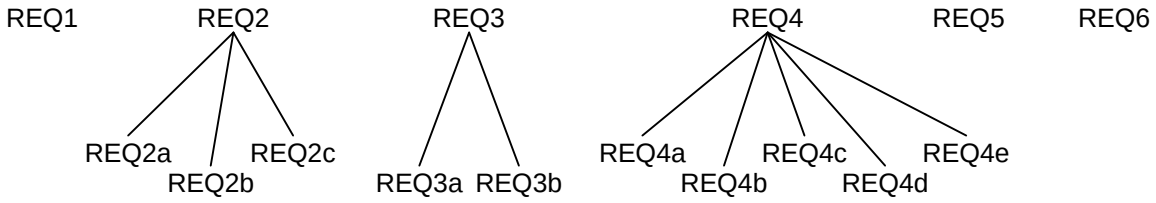
- Option 1: Allow no more than one pending invitation per player
- Option 2: Allow no additional invitations for players already engaged in a match
- Option 3: Allow unlimited pending invitations at any time

We select the first option for simplicity, but a real-world implementation may provide the advanced options.

TTT-BP02: no more than one pending invitation per player are allowed

This policy does not necessarily imply that the user can play only one match at a time. However, such a more restrictive version will be introduced in Section G.3.4, when we will introduce an operational model for our system-to-be to keep the project manageable.

Further analysis would reveal more details, in addition to those shown in Table G-2. For example, perhaps as part of REQ4 the system should also allow the players to agree on a "draw" before the winner becomes obvious? Because tic-tac-toe is a simple, short and inconsequential game, we will stop here:



The above details should not be listed as requirements on their own in the table of requirements, because such fragmentation of functional requirements would complicate the understanding of the system's purpose. Their appropriate role is as details of the main requirements.

REQ3 appears to complicate the setup process and one may look for alternate solutions. Instead of players having to negotiate the game version, it may be more convenient to have each user specify in the challenge which game version they want to play. Then, the player who accepts the challenge also agrees to play the proposed game version and the match is started immediately. Similarly, players could indicate their preferred game version as part of their availability status. Then, each player could search the player list for players interested in playing a certain version and challenge one of them.

This solution has its issues as well, because the challenged player may never receive an invitation for the game version that he wishes to play. Of course, he may send his own invitations for the desired game version, but then needs to keep sending to different players on the "available" list until one accepts. However, there is a more important reason that we should *not* choose this solution—because it makes the developer's task easier, while at the same time making the user's task harder. Each user would always be required to choose the version they wish to play, when in

reality most users might be happy to play the standard version. Solutions that make the developer's work easier while making the user's work harder should be avoided, unless there is a good reason, such as constraints on the product development time or resources. The customer must always be involved in making such compromises. At this point, we do not know how much more complex the user-friendly solution is than the developer-friendly solution, so for now we proceed with the user-friendly solution as originally formulated in REQ3. However, see Sidebar G.1 for additional issues.

SIDEBAR G.1: Playing Multiple Matches at a Time

♦ Requirement REQ2c in Table G-2 and business policy TTT-BP02 allow the user to have at most one pending invitation. The user must wait for the opponent to accept a challenge. The challenger cannot do anything until the opponent responds or response timeout expires. Later on, in Section G.3.4 we will make an even more restrictive choice to allow the user to participate in no more than one match at a time.

The reader who is also an avid game player will know that many existing games, such as Scrabble, Words With Friends, Draw Something, all allow users to engage in multiple matches at the same time. Why not make our Tic-tac-toe to do the same? A user would challenge an opponent, automatically enter a gameroom for this match, and then return to the game lobby to play in other matches while he or she waits for the opponent. This method would also allow users to challenge players who are currently not logged into the system. A logged-off user would simply receive a challenge notification when he or she logs in.

By adopting the multiple-simultaneous-opponents version, we would drop the policy TTT-BP02. However, should we drop the response time policy TTT-BP01? On one hand, users may find it annoying to have many unresolved matches at a time. It may be helpful if the system provided some indication of the underlying cause, such as network outage, or logged-out opponent. On the other hand, one may argue that such scenarios will be rare and most of the matches will be played within a short interval, without interruptions. We may introduce a policy that the gamerooms which have seen no activity for several days will be automatically terminated.

Although the multiplayer version of distributed tic-tac-toe appears to remove some complexity related to game setup, we still need to decide how the players would select the game version to play. At this stage, we decide not to adopt this version as the target version of our system-to-be out of concern that it may be too complex to implement. User convenience must always take priority over developer's convenience, unless it is impossible to achieve with the given resources and time constraints. However, we will consider the merits of the multiplayer option as we go and may even adopt it as the target version for the system-to-be if its merits are deemed high and the costs acceptable.

The reader should particularly observe that instead of simply cataloguing the system requirements by reading the customer statement of work, we started the *discovery process* of learning the details of what exactly we are expected to develop. In other words, we started ***requirements analysis***. Based on the analysis we detected issues that could be solved in different ways and made choices of *business policies*, such as the response time limit, and rejected some requirements (REQ7). We also uncovered additional details that need to be stated explicitly in the

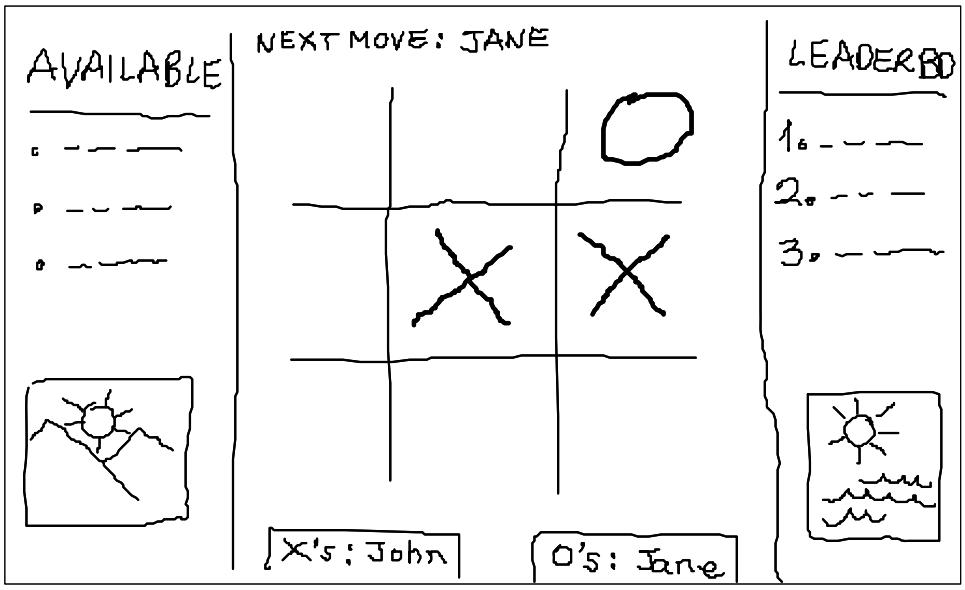


Figure G-2: On-Screen Appearance Requirements: Customer's sketch of the user interface.

requirements. Requirements analysis for this system will be continued in Section G.3.4, during the detailed use case analysis.

G.2.2 Enumerated Nonfunctional Requirements

As stated in Section G.2.1, we do not specify any latency requirements, because it is not critical that the other player immediately sees each move.

G.2.3 On-Screen Appearance Requirements

REQ8 Figure G-2 shows a customer-provided initial sketch of the user interface appearance. The screen real estate is divided into three main areas. The area on the left will show the list of currently available players. The central area will be empty when the user is in the game lobby, while waiting to be invited or inviting an opponent. The central area will show the game board once the players agree the play the game. The area one the right will show the current leaderboard. Notice also that two parts on the bottom of left and right areas are provisioned to show sponsor advertisements. The customer requested that the advertisements should be subtle rather than distractive.

G.2.4 Acceptance Tests

Acceptance tests that the customer will run to check that the system meets the requirements are as follows. Note, however, that these test cases provide only a coarse description of how a requirement will be tested. It is insufficient to specify only input data and expected outcomes for testing functions that involve multi-step interaction. Use case acceptance tests in Section G.3.5 will provide step-by-step description of acceptance tests.

Acceptance test cases for REQ1:

ATC1.01 Ensure a working network connection between two computers, run the game program on both computers (pass: each user is shown as "available" on the other user's screen)

ATC1.02 Ensure a working network connection between more than two computers, run the game program on computers at random times (pass: the displayed list of "available" users is updated correctly on all computers)

Note that REQ1 states that any pair of players will be able to play from different computers; however, the acceptance tests do not specifically test this capability. The acceptance tests check whether the users can see each other as available, assuming that they successfully joined the game. I leave it to the reader to formulate more comprehensive test cases for REQ1.

Acceptance test cases for REQ2:

ATC2.01 Challenge a user who is logged in and accepting invitations (pass)

ATC2.02 Challenge a user who is currently not logged in (fail)

ATC2.03 Challenge a user who is logged in but not accepting invitations (fail)

ATC2.04 Challenge another user accepting invitations after a declined invitation (pass)

ATC2.05 Challenge another user immediately after one user accepted the challenge (fail)

ATC2.06 Challenge another user during an ongoing match (fail)

ATC2.07 Challenge another user after a finished match (pass)

Note that ATC2.02 may not be possible to run directly from the user interface if the user interface is designed to force the user to select from the set of available players. In addition, ATC2.05 and ATC2.06 appear to test the same scenario and one of them may be redundant. Finally, we may wish to add one more test case (ATC2.08), to challenge the local user after a finished match. This case is reciprocal to ATC2.07, which allows the local user to challenge another (remote) user.

Acceptance test cases for REQ3:

ATC3.01 During a match in progress, select a different game variant from the one currently played (fail)

How exactly this test case will be executed depends on how the user interface is implemented and whether it allows the user to perform such actions in different contexts.

Acceptance test cases for REQ4a:

Test cases for this requirement are difficult to formulate in a simple, one-sentence version as for other use cases. We may test that a player can move a piece to any empty cell, or in case of nine-board tic-tac-toe the player moves to an empty cells on the board corresponding to the square of the previous move, but this does not cover the whole REQ4a. We also need to test that the match is correctly refereed and that players' high scores are correctly updated.

This is why the acceptance test formulation for this requirement is deferred to Section G.3.5.

Acceptance test cases for REQ4b:

ATC4.01 In a resting state, select the revenge tic-tac-toe game (pass: the opponent is asked to accept the selection or counteroffer a different selection; if accepted, the revenge

version is shown on both players' screens; else, the first player is asked to accepts the counteroffer selection or make another counteroffer)

ATC4.02 In a resting state, select the nine-board tic-tac-toe game (pass: similar as for the revenge case)

ATC4.03 In an ongoing match, select to change the game variant (fail)

Note that this test case formulation is inelegant and will be better represented with a test case for the corresponding use case (see Section G.3.5).

Acceptance test cases for REQ5:

ATC5.01 A visitor user not logged in requests to see the leaderboard (pass)

Acceptance test cases for REQ6:

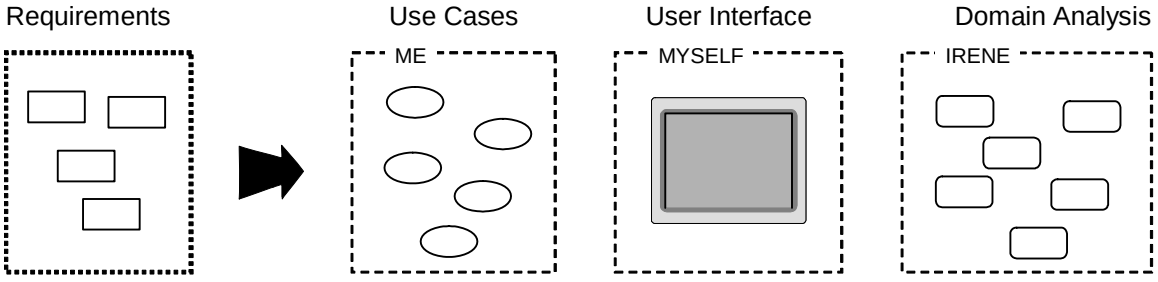
ATC6.01 A visitor not logged in fills out the registration form using an unused identifier (pass)

ATC6.02 A visitor not logged in fills out the registration form using a taken identifier (fail)

How We Did It & Plan of Work

After reading the customer statement of work, Me said he will start coding right away, while Irene and Myself find out what needs to be done. It turned out real bad. Irene and Myself met several times, discussed the statement of work and came up with a paper-based prototype of the game. We realized we were missing Me's technical skills but he just hunkered down in his lair and made himself unreachable. Then finally, we all met the night before the deadline. It turns out that Me wrote his codes in a foreign language—the program crashed even before it rendered the welcome screen. But Me kept saying that the software is done and we just needed to get it to work. We realized we needed to work as the Gang of Three - in team, instead of three Gangs of One. We ordered pizza and coke, and went on burning the midnight oil all night long to derive the system requirements, as described above. Requirement analysis helped us greatly as it allowed us to enumerate the requirements and pool all of the ideas of all the team members together in an orderly manner. GoT-it!

Then we faced the problem of how to split our future work. Me volunteered to do the use cases and Myself jumped on the opportunity to do some interface design; we suggested that Irene does the domain analysis. Me quickly drew a sketch that shows how we will organize our teamwork:



However, Irene, being the project manager, pointed out that the problem with working with a team on a project of this scale is that a member may or may not get their job done, which affects the team as a whole. She suggested that we might face issues where one person's part was

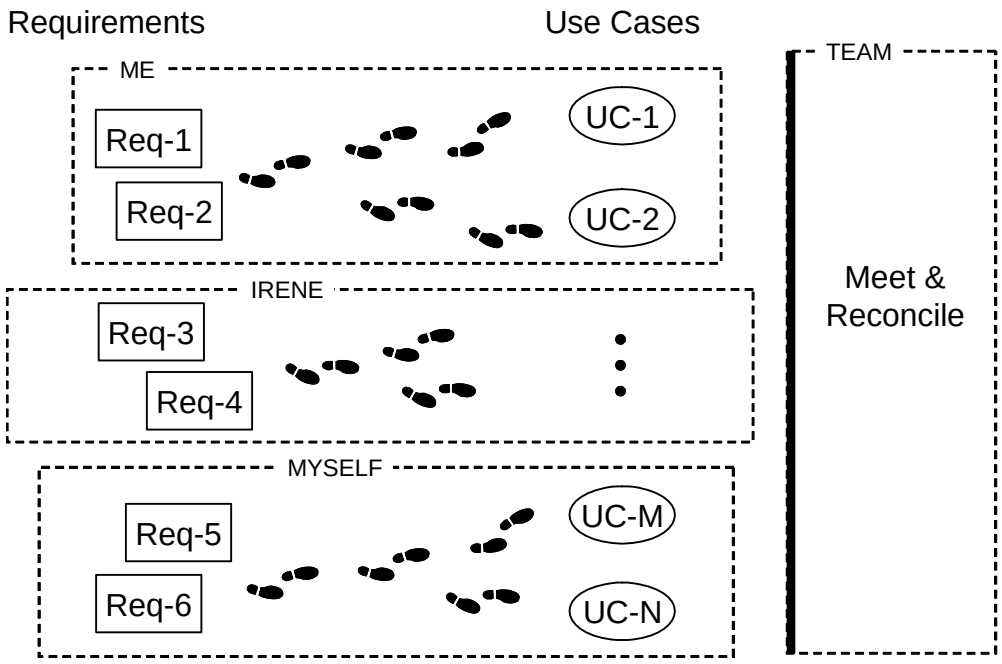


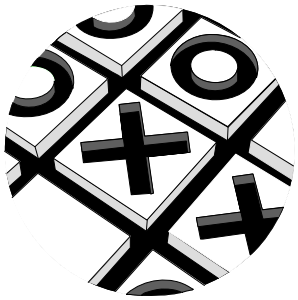
Figure G-3: Ownership diagram for splitting up the teamwork in *parallel* instead of *series*, to avoid roadblocks to successful teamwork. (Continued in Figure G-7.)

required in order to continue the flow of work; which, resulted in roadblocks during the process. Some parts of the program cannot be worked on without first finishing other functions. Irene pointed out that, for example, she would not be able to do anything before receiving the elaborated use cases and the interface design from Me and Myself. Instead, she proposed that we do work in parallel (Figure G-3), so that each team member takes ownership of several system requirements and derives the corresponding use cases. Although Figure G-3 shows that our team will meet only once at the end of this development stage, clearly we will need to meet often, reconcile any issues with our use cases, and jointly decide on next steps.

Me and Myself agreed that this is a great idea because it minimized mutual dependency of team members on each other's progress. From this, we learned better time management and cooperation with others.

We also agreed that everyone will be responsible for writing the part of the project report describing his or her component. At the end, Irene will collect all report contributions and integrate them into a uniform whole. Everyone felt Irene was the one who paid attention to detail most for things such as naming conventions and report format so she wore the additional hat of editor-in-chief, making sure everyone's work made sense before we submitted our reports. For these reasons, when it came to deciding who would work on what, Irene was on the business/report side.

G.3 Functional Requirements Specification



Although this section is entitled "Requirements Specification," we will see that we are still *discovering* the functional system requirements for the system-to-be, as well as *specifying* the discovered requirements.

We start by selecting an architectural style for our system, because the user experience will depend on the choice of architectural style and because some use case scenarios would not be possible to specify in detail without knowing the architectural style.

ARCHITECTURAL STYLE: CENTRAL REPOSITORY – The problem statement (Section G.1.1) does not mention that the system should be Web-based, so we will assume that programs will run on different computers without a dedicated server application, but all users will connect to a common database server, such as MySQL. This means that all "clients" will store their game-related data to the database. To enable communication between "clients", each client will periodically check the database when it expects a message. After the sender stores its message in the database, the receiver will pick it up in the next round of checking. More sophisticated architectural styles may be considered in a future version of this system.

G.3.1 Stakeholders

Identify anyone and everyone who has interest in this system (users, managers, sponsors, etc.). Stakeholders should be humans or human organizations.

G.3.2 Actors and Goals

We identify four types of actors:

1. Player – a registered user
2. Opponent – a special case of Player actor, defined relative to the Player who initiated the given use case; this actor can do everything as Player, but we need to distinguish them to be able to describe the sequence of interactions in use case scenarios
3. Visitor – any unregistered user
4. Database – records the Players' performance

To implement the RESPONSE TIME POLICY defined in Section G.2.1, we will need a timeout timer to measure the reaction time of each player. Because this timer will be part of a use case execution, but will *not* initiate full use cases, there is no need to consider it an actor.

G.3.3 Use Cases Casual Description

The summary use cases are as follows:

UC-1: PlayGame — Allows the Player to play the standard tic-tac-toe game (default option). Extension point: the Player has an option to challenge an Opponent, or just wait to be challenged

by an Opponent. Extension point: the Player has an option to suggest another game variant. Derived from requirement REQ1 – REQ4.

Note that UC-1 mentions only the Player actor so, by implication, it is available only to registered players who are logged in into the system.

UC-2: Challenge — Allows the Player to challenge an Opponent to play a match (optional sub use case, «extend» UC-1: PlayGame).
Derived from requirement REQ2.

UC-3: SelectGameVariant — Allows the Player and Opponent to negotiate a variant that they will play, different from the default standard tic-tac-toe (optional sub use case, «extend» UC-1: PlayGame).
Derived from requirements REQ3 and REQ4.

UC-4: Register — Allows a Visitor to fill out the registration form and become a member of the game. (For simplicity, we omit the use case that allows the user to modify his or her profile.) Derived from requirement REQ6.

UC-5: Login — Allows the Player to join the game and have the system track his or her performance on the leaderboard (mandatory sub use case, «include» from UC-1: PlayGame). Derived from requirement REQ6.

UC-6: ViewLeaderboard — Allows a Visitor to view the leaderboard of player rankings without being logged in the system. Players will always be able to run this use case. Another option that may be considered is to have the leaderboard displayed in any screen that the player visits. At this point, we decide that the player will explicitly run UC-6 to avoid screen clutter. A more detailed analysis in the future may sway the developer to switch to the always-shown option. Derived from requirement REQ5.

Some alternative use cases may be considered. For example, instead of the use cases to challenge an opponent (UC-2) and negotiate the game version (UC-3), one may propose a single use case where the player will challenge an opponent to play a specific version of tic-tac-toe. I have not carefully considered the merits of this alternative solution, so for now we go with two separate use cases.

The last use case (UC-6) allows any visitor to view the leaderboard, which is not strictly implied by REQ5. We provide it anyway to allow visitors to view the current state of the game, perhaps to attract them to become active members.

If the player passively waits to be invited by another player, this is not a use case, because this player does *not* initiate any interaction with the system to achieve his or her goal. This player will play a participating actor role when another player initiates UC-2.

It is important to choose the right level of granularity for use cases. Introducing a Make-Move use case to place a piece on the board is too fine granularity and does not confer any benefit. Therefore, making a move should be considered a step in use case UC-1: PlayGame. Similar argument applies against having Start-New-Match as a use case. Another example is Record-Result, which is a step in successful completion of UC-1, and *not* a standalone use case initiated by Database. Yet another example is View-Pending-Invitations to check for invitations and accept or decline, which should also be considered a step in UC-1. And so forth.

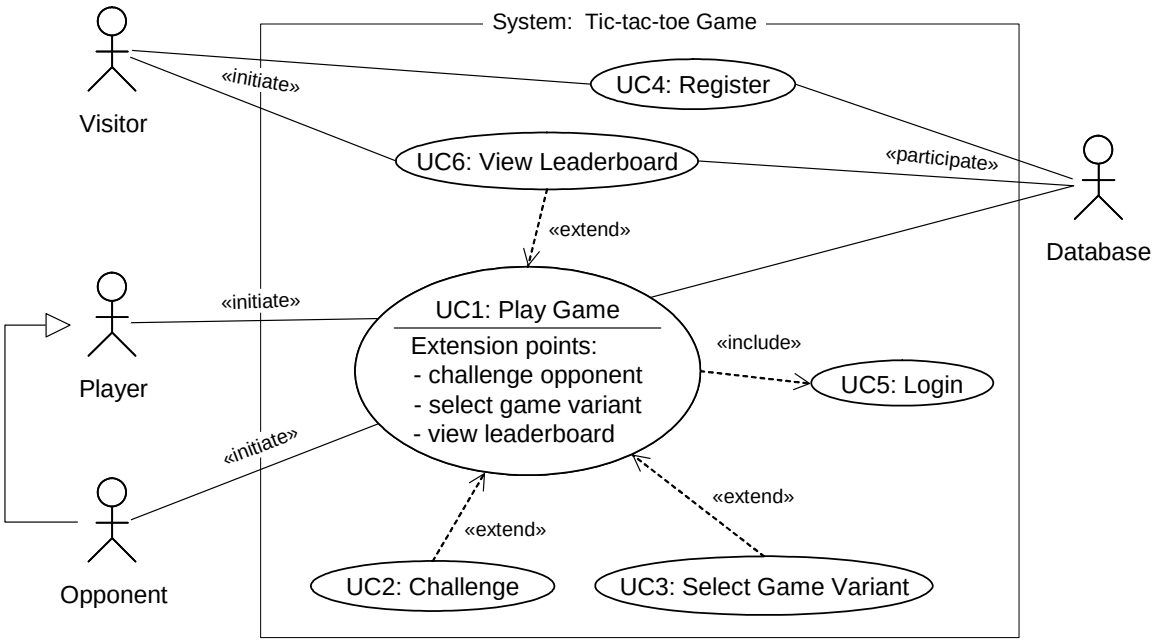


Figure G-4: Use case diagram for the distributed game of tic-tac-toe.

Note that options to play different game variants (standard, revenge, or nine-board) are *not* shown as extension use cases. The reason for this choice is that it is difficult to specify different game rules in use cases notation. On the other hand, there is no value in indicating three more use cases for the three variants if those use cases will have identical specification. Tic-tac-toe game involves a single type of interaction for all game variants: placing a piece on the board. In case of games with many or more sophisticated interaction types, it may be appropriate to consider sub-use cases for different game variants. Therefore, we leave the game variant specification for the next stage of development lifecycle: domain analysis (Section G.5).

Use Case Diagram

The use case diagram is shown in Figure H-1. The diagram indicates the «include» and «extend» sub-use-case relationships. Also indicated is that Opponent is a specialization of Player. Both players must «initiate» the game before they can play. Each player has an option of waiting to be invited, or inviting an opponent. When the players connect, they are shown the default version of the game, and they may select a different variant.

Traceability Matrix

The traceability matrix in Figure G-5 shows how our system requirements map to our use cases. We calculated the priority weights of the use cases, and we can order our use cases by priority:

UC1 > UC3 > UC2 > UC6 > UC4, UC5

We select the three use cases with the highest priority to be elaborated and implemented for the first demonstration of our system.

Req't	PW	UC1	UC2	UC3	UC4	UC5	UC6
REQ1	5	X					
REQ2	4	X	X				
REQ3	3	X		X			
REQ4	4	X		X			
REQ5	2						X
REQ6	1				X	X	
Max PW		5	4	4	1	1	2
Total PW		16	4	7	1	1	2

Figure G-5: Traceability matrix mapping the system requirements to use cases. Priority weight (PW) given in Table G-1. (Traceability continued in Figure G-13.)

G.3.4 Use Cases Fully-Dressed Description

We start deriving the detailed ("fully-dressed") specification of use cases by sketching usage scenarios.

Start with UC-2: Challenge, which allows the Player to challenge an Opponent to play a match, because this is the first logical step in the game. A possible scenario may look something like this:

1. Player sends an invitation to an Opponent to play a match
2. Opponent accepts or declines
3. Players are brought to the main use case (UC-1) to play a match

However, we realize that we must be more specific in Step 1 about how the Player *selects* an Opponent, and what if this Opponent is already playing with another player. Would an "engaged" player be interested in accepting new invites? We already discussed this issue in Section G.2.1, when analyzing the feasibility of the requirement REQ7. We choose the following simple solution. Every player will be shown a list of currently available players. To avoid annoying invitations while the player is already engaged in a game, the system will automatically remove this player from the list of available players. The players follow a simple invitation *protocol*, which is a BUSINESS POLICY, specified as a sequence of interactions between the players (i.e., "protocol") shown in Figure G-6.

TTT-BP03: The invitation protocol allows a player to challenge only the "available" opponents. (Note that the first two business rules were identified in Section G.2.1.) The opponent accepts or declines and these two players are brought into a "game room" to play only a single match. After the match is finished, the players are brought back to the main screen and they must again send match invitations. (The players will remain logged in.)

There may be other ways to operationalize this game. For example, a user may start a new match by selecting a game variant and then challenge an opponent to play this match. In other words, the game variant would not be negotiable. In addition, given the concept of a "gameroom" we may now simplify the problem of determining player availability. We may operationalize the

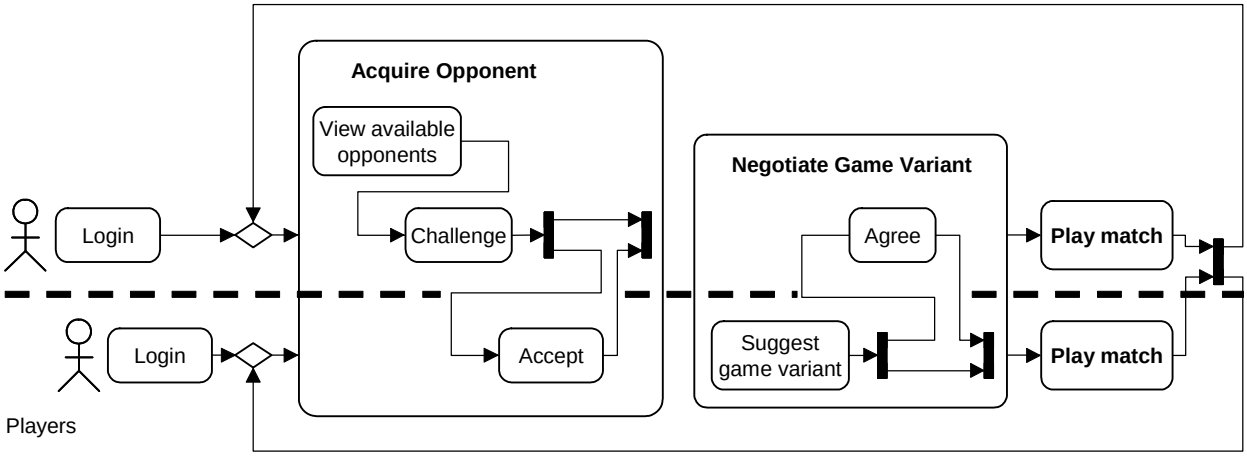


Figure G-6: Operational model for the distributed game of tic-tac-toe.

game so that once two players enter a gameroom, they can play as many matches as desired. When one player leaves the gameroom, both players will become available for invitations. This operational model is left to the reader as an exercise and will not be considered here.

The requirements state that players takes turns, so that the **X**s-player goes first and the **O**s-player goes second, but it is not specified how **X**s and **O**s are assigned, so we decide that before each match the system randomly designates and informs the players. See Section G.5.3 for the reasoning behind this choice.

Use Case UC-1:	**Play Game**
Related Requirements:	REQ1 – REQ4
Initiating Actor:	Player
Actor's Goal:	To play the game of tic-tac-toe
Participating Actors:	Opponent, Database
Preconditions:	• Player is a registered user
Success End Condition:	• If completed ≥1 matches, Player's score is updated in Database
Failed End Condition:	• Forfeited matches counted as losses in Database
Extension Points:	*Challenge* (UC-2) in step 1; *Select Game Variant* (UC-3) in step 3; *View Leaderboard* (UC-6) in any step
Flow of Events for Main Success Scenario:	
	include::*Login* (UC-5)
←	1. **System** displays the list of remote players that are available to play the game; the display is continuously updated and any incoming invitations are shown
→	2. **Player** accepts an incoming invitation
←	3. **System** (a) brings both **Player** and **Opponent** into a gameroom, (b) displays the default standard tic-tac-toe, (c) randomly assigns **X**s or **O**s to the **Player** and **Opponent** and displays their designations
→	4. The **X**s-player is prompted to make the first move anywhere on the board, then **O**s-

player is prompted to make a move [for each move, a timer is started to limit the response time]

Players repeat Step 4 until System declares a winner or detects that the board is filled to end in a draw

← 5. **System** (a) signals the match end, (b) erases the screen, (c) stores the updated scores for both players in the **Database**, and (d) closes the gameroom and brings players back to the main screen (Step 1)

Flow of Events for Extensions (Alternate Scenarios):

any step: **Player** requests to forfeit an ongoing match

← 1. **System** (a) shows a message to the opponent and asks for an acknowledgement, (b) starts a timer for a fixed interval, say 10 seconds, (c) when the opponent acknowledges or timer expires, System goes to Step 5 of the main success scenario

4a. **Player** tries to make an out-of-order move (e.g., the **O**s-player tries to go first, or any player tries to move during their opponent's turn)

← 1. **System** signals an error to **Player** and ignores the move

4b. **Player** tries to place a piece over an already placed piece

1. **System** signals an error to **Player** and ignores the move

4c. **Player** fails to make the next move within the timeout interval

← 1. **System** applies TTT-BP01: RESPONSE TIME POLICY, declares a "win" for the other Player and goes to Step 5

any step: network connection fails (**System** should continuously monitor the health of network connections)

← 1. **System** detects network failure and (a) cancels the ongoing match and closes the gameroom, (b) signals the network failure to **Player** and informs about a possibly forfeited match, (c) blocks use cases that require network connectivity and goes to Step 1

Note that alternate scenarios 4a and 4b could be combined into a single scenario: Player tries to make an *invalid* move (out-of-order or to a played cell).

Also, the alternate scenario 4c should be more precisely stated: Player fails to make a *valid move* within the timeout interval. We decide that in alternate scenarios when the player makes an invalid move (4a and 4b) no remote notifications are sent—the response timer should time out and the system should declare our player loser regardless of whether the player is unresponsive or just being silly.

The reader should note that handling the last alternative scenario of UC-1 involves another BUSINESS POLICY:

TTT-BP04: When the system detects network failure, it cancels the ongoing match and closes the gameroom, signals the network failure to Player and informs about a possibly forfeited match, and blocks use cases that require network connectivity.

This policy may be formulated differently. One may object to forfeiting the match because of a lost network connection and propose instead saving the current board state and resuming the match when the connection is reestablished. Note that the other player may still be connected and

waiting for this player's response. Distinguishing a lost connection from an unresponsive user would introduce additional complexity into our system. Given that losing a tic-tac-toe match is inconsequential, we decide that the effort needed to support policies that are more sophisticated is not justified.

This use case is general for all three variants of the game. As noted, specifics for the revenge and nine-board variants will be considered in domain analysis (Section G.5).

In UC-2, we assume that the player is shown only the list of players that are currently available and no other players can be invited. As mentioned in Section G.2.1, possible future extensions are to allow users to search for opponents by name or some other keyword, invite their social network friends, etc.

The player can send only one invitation at a time. Acceptance tests listed in Section G.2.4 provide ideas about preconditions and alternative scenarios.

Use Case UC-2:	**Challenge**
Related Requirements:	REQ2
Initiating Actor:	Player
Actor's Goal:	To challenge an opponent to play the game
Participating Actors:	Opponent
Preconditions:	• The initiating Player is logged in and "available" (*not* in gameroom) • Only "available" remote players are listed
Success End Condition:	• Opponent accepted; both players marked as "engaged" and removed from the "available" list
Failed End Condition:	• Opponent declined or failed to respond before timeout interval

Flow of Events for Main Success Scenario:

→ 1. **Player** selects an opponent from the list of available remote players and sends invitation for a match

← 2. **System** (a) asks the **Opponent** to accept the invitation and (b) starts a timer for a fixed interval, say 1 minute

→ 3. **Opponent** indicates acceptance

← 4. **System** goes to Step 3 of UC-1 (Play Game)

Flow of Events for Extensions (Alternate Scenarios):

2a. **System** receives several simultaneous invitations for the same Opponent

← 1. **System** (a) picks one challenger randomly, (b) notifies the remaining challengers that the Opponent became engaged, and (c) goes to Step 2 of the main success scenario

3a. **Opponent** declines the invitation

← 1. **System** notifies **Player** about a refusal and goes to Step 1 of UC-1 (Play Game)

3b. **Opponent** fails to respond within a timeout time

← 1. **System** (a) removes the pending invitation from **Opponent**'s screen, (b) notifies

Player about a refusal and goes to Step 1 of UC-1 (Play Game)

Player receives invitation(s) while waiting for an **Opponent**'s answer to own invitation

1. **System** intercepts such invitations and notifies their senders about the failure

After step 1, **Player** quits (logs out) without waiting for **Opponent** to answer

← 1. On Opponent's terminal, **System** notifies **Opponent** about the desertion and goes to Step 1 of UC-1 (Play Game) for the Opponent

A precondition for UC-2 is that the initiating actor is "available." In the future, we may need to consider an option of allowing the initiating actor to cancel the current engagement *without* playing a match. The engagement is automatically cancelled after a match is finished and the player must challenge another opponent before the next match.

Note that the first extension deals with potential simultaneous received invitations. In Section G.2.1, we decided that a player could not send a new invitation before a pending invitation is answered. Here, the reader should note another BUSINESS POLICY:

TTT-BP05: in case of simultaneously received invitations, one is selected randomly

This policy may be decided differently. For example, the system may select one invitation based on the inviter's leaderboard ranking or friendship connections. It is important to make such choices explicit, so that the customer can participate in the decision-making and change the policy in the future.

The reader familiar with network security issues may detect a more serious problem with the policy TTT-BP05. This policy makes our system susceptible to the so-called denial-of-service attacks (http://en.wikipedia.org/wiki/Denial-of-service_attack). An adversary who is familiar with our system may saturate our central database server with match invitations, such that it cannot respond to legitimate traffic and rendering it effectively unavailable. The legitimate challengers would wonder why their invitations always go unanswered and the challenged players would find that their opponents are fake. A potential solution to this problem is to show the user all invitations and let the user select. This solution is more complex and it is not clear that the user will always have sufficient information to discern fake from legitimate invitations.

The decision to intercept and discard the invitations received while the player waits for an answer to a pending invitation is another BUSINESS POLICY that may be decided differently.

TTT-BP06: intercept and discard the invitations received during an outstanding invitation

We start the analysis of UC-3 by sketching a possible scenario, like so:

1. Player suggests a version of tic-tac-toe to play
2. Opponent disagrees and counteroffers a different version
3. Player disagrees and counteroffers a different version
4. and so forth…

We realize that this cycle can go on forever, so we need to specify the negotiation *protocol*. This is another BUSINESS POLICY:

TTT-BP07: the negotiation protocol is specified as a sequence of interactions between the players (i.e., "protocol"). We adopt a simple protocol where one player suggests the variant to

play. The opponent either agrees or responds with a counteroffer. If the first player does not agree to the counteroffer, the match is cancelled before it started and the gameroom is closed. Both players are brought back to the main screen and they enter the pool of "available" players. Similarly, if a response is not received within a specified interval, match is cancelled before it started, and both players go to the pool of "available" players. One may conceive a more complex protocol, for example, the system automatically initiates a chat where the two users can discuss how to continue, but given that the game of tic-tac-toe is quick and of no consequence, such complex features are unnecessary.

If the players agree on a game version, the newly agreed game version is loaded. The players' previous designations remain unchanged when the different game version is loaded: **X**s-player remains **X**s and **O**s-player remains **O**s.

The detailed use case UC-3 can then be specified as follows:

Use Case UC-3:	**Select Game Variant**
Related Requirements:	REQ3
Initiating Actor:	Player
Actor's Goal:	To negotiate the version of tic-tac-toe to play
Participating Actors:	Opponent
Preconditions:	• Player and Opponent are already in a gameroom • The game is in the resting state (no match is in progress)
Success End Condition:	• Player and Opponent agreed on the game version
Failed End Condition:	• agreement not reached or Opponent failed to respond; match cancelled and both players join the pool of "available" players

Flow of Events for Main Success Scenario:

→ 1. **Player** selects a choice from the list of available versions of tic-tac-toe

← 2. **System** (a) asks the **Opponent** to accept the choice or provide a counteroffer, and (b) starts a timer for a fixed interval, say 1 minute

→ 3. **Opponent** indicates agreement

← 4. **System** goes to Step 3 the main success scenario of UC-1 (Play Game)

Flow of Events for Extensions (Alternate Scenarios):

3a. **Opponent** provides a counteroffer

← 1. **System** notifies **Player** about counteroffer and goes to Step 1 of UC-1 (Play Game)

3b. **Opponent** fails to respond within a timeout time

← 1. **System** (a) cancels the match, (b) notifies both players about a preempted match and goes to Step 1 of UC-1 (Play Game)

Player receives version request that the **Opponent** sent nearly simultaneously, before receiving this player's version request

1. **System** intercepts such requests and discard the most recent request silently

After step 1, **Player** quits (logs out) without waiting for **Opponent** to answer

←	1. On Opponent's terminal, **System** (a) cancels the match, (b) notifies **Opponent** about the desertion and goes to Step 1 of UC-1 (Play Game) for the Opponent

The remaining use cases are relatively simple and are left to the reader as exercise. However, I want to use this occasion to emphasize an important principle of agile development. My main reason for omitting the remaining use cases is that I did not have enough time. When faced with too much work to do and not enough time, the agile developer will cut the project scope. I decided which use cases should be ignored based on the priority weights from Table G-1 and the traceability matrix in Figure G-5.

Notes on the remaining use cases:

• UC-6: ViewLeaderboard — one may wish to state as a precondition that at least one match has finished; however, it is not clear why UC-6 must not be executed if no match was played, so we do not consider it a precondition.

The decision to intercept and discard a version request received while the player awaits an answer from the opponent is another BUSINESS POLICY that may be decided differently.

TTT-BP08: intercept and discard version requests received while awaiting an answer to a version offer.

SIDEBAR G.2: Playing Multiple Matches at a Time

♦ Sidebar G.1 discussed the option of allowing the user to play multiple matches at a time.

The reader should particularly observe the continuing *knowledge discovery* about the system-to-be (i.e., *requirements analysis*). We have not just written down the detailed use cases (i.e., functional requirements specification). Instead, we needed to invent strategies for tackling the identified ambiguities and constraints and analyze their feasibility. The outcomes of this knowledge discovery process include the operational model that specifies the system-to-be and two business policies for invitation and negotiation protocols. It is critical properly to document this discovery process and the choices that we made.

G.3.5 Acceptance Tests for Use Cases

The acceptance test cases for the use cases are similar to acceptance test cases in Section G.2.4. As mentioned, testing functions that involve multi-step interaction requires more than just specifying the input data and expected outcomes. We also need to specify the step-by-step how the user interacts with the system and what is the system expected to do. Simple test cases listed in Section G.2.4 need not be repeated. Here we show only the test cases that were not already shown or needed a more structured presentation.

Acceptance test cases for UC-1: Play Game include but are not limited to:

Test-case Identifier:	TC-1.01
Use Case Tested:	UC-1: Play Game – main success scenario for ***standard*** tic-tac-toe

Pass/Fail Criteria:	If either user aligns three pieces in a line, he is declared the winner If the board fills up with no winner, a draw is declared
Input Data:	Players' moves on the game board

Test Procedure:	**Expected Result:**
Set Up: Two or more users log into the program and verify that each is given the option to challenge an opponent	System displays the list of currently available players
Step 1. Challenge an opponent as in test case TC-2.01 for UC-2	System displays a gameroom and informs each player that they are randomly assigned **X**s or **O**s
Step 2. Players alternate placing their pieces on the game board	● Valid moves accepted & consistently displayed for both players (a small delay for remote player)
	● Invalid moves rejected with an error message
Step 3. Loop back to Step 2 until the match is finished	● If either player aligned three pieces in a line, he or she is declared the winner; If the board filled up with no winner, a draw is declared
	● Both players are shown the outcome and taken out of the gameroom back to the main screen
	● The leaderboard is updated accordingly

In addition to the above test procedure, the user must verify that the system correctly maintains the scoreboard. The user would play a few matches and keep a hand-drawn tally of scores. The user would then compare the leaderboard to verify whether his number of matches played and their outcomes have been counted.

Test case for UC-1: Play Game – main success scenario for ***revenge*** tic-tac-toe is the same as TC-1.01, except that:

● If either user aligns three pieces in a line, the system gives the opponent one more move. If the opponent aligns three in a line with the next move, he or she wins; otherwise, the first player is declared the winner.

Similarly, test case for UC-1: Play Game – main success scenario for ***nine-board*** tic-tac-toe is the same as TC-1.01, except that:

● After the first move, every subsequent move must be on the board corresponding to the cell of the previous move.

A player should have the opponent quit an ongoing match and verify that the remaining player is declared as the winner.

Test-case Identifier:	TC-1.02
Use Case Tested:	UC-1: Play Game – alternate scenario 4.c
Pass/Fail Criteria:	The test passes if either player delays his response longer than the response time limit
Input Data:	Players' moves on the board & the time to wait before responding

Test Procedure:	**Expected Result:**
Set Up: Two or more users log into the program and verify that each is given the option to challenge an opponent	System displays the list of currently available players
Step 1. Challenge an opponent as in test case TC-2.01 for UC-2	System displays a gameroom and informs each player that they are randomly assigned **X**s or **O**s
Step 2. Players alternate placing their pieces on the game board	● Valid moves accepted & consistently displayed for both players (a small delay for remote player) ● Invalid moves rejected with an error message
Step 3. Before the match end, one player delays his response longer than the response time limit	● System applies the RESPONSE TIME POLICY and declares the other player the winner ● Both players are shown the outcome and taken out of the gameroom back to the main screen ● The leaderboard is updated accordingly

Test cases for UC-2: Challenge include but are not limited to:

Test-case Identifier:	TC-2.01
Use Case Tested:	UC-2: Challenge – main success scenario
Pass/Fail Criteria:	The test passes if the opponent accepts the challenge within the response time limit; otherwise, the test fails
Input Data:	available Opponent's identifier

Test Procedure:	**Expected Result:**
Set Up: Player logs in and sees the list of available opponents (as part of UC-1)	
Step 1. Player invites an opponent from the list	System conveys the invitation to the opponent
Step 2. Opponent player indicates acceptance within the response time limit	System informs both players about the success

Test cases for UC-3: Select Game Variant include but are not limited to:

Test-case Identifier:	TC-3.01
Use Case Tested:	UC-3: Select Game Variant – main success scenario
Pass/Fail Criteria:	The test passes if both players agree to a version of the game after no more than one counteroffer and within the response time limit; otherwise, it fails
Input Data:	offered game version and counter-offered version

Test Procedure:	**Expected Result:**
Set Up: Two players are in a gameroom (as part of UC-1)	

Step 1. Player suggests a game version	System displays the offer to the opponent
Step 2. Opponent player suggests a counteroffer within the response time limit	System displays the counteroffer to the first player
Step 3. Player accepts the counteroffer	System informs both players about the success

Obviously, the above test cases do not provide **coverage** of all alternate scenarios in our use cases. Because alternate scenarios are more complex to implement (and, hence, more likely to have implementation mistakes), it is critical to ensure complete coverage of all identified alternate scenarios. This task is left to the reader as an exercise.

Of course, testing all alternate scenarios does not ensure the complete test coverage. For example, alternate scenarios may occur in many different combinations (along with the main success scenario), and it is practically impossible to test all the combinations.

G.3.6 System Sequence Diagrams

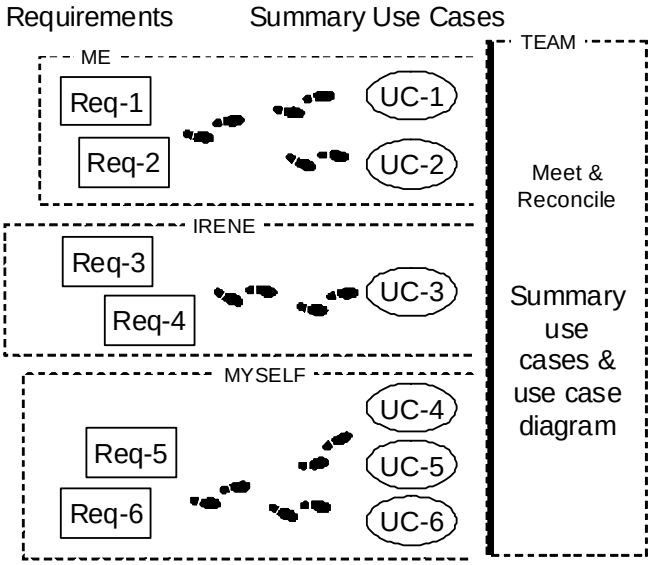
_____ TO BE COMPLETED _____

G.3.7 Risk Management

A mitigation strategy for the denial-of-service attack identified for use case UC-2: Challenge in Section G.3.4 is to use a design (Strategy pattern) that allows easy change of the business policy TTT-BP05 if the problem is observed.

How We Did It & Plan of Work

We found defining the use cases relatively easy and most useful. It helped to enumerate the exact functionality of our program before we started so we knew how to tackle to construction of the system without losing focus and getting misdirected. One problem the group faced time and time again was finding time when everyone could meet up. We managed to meet together after individually defining the summary use cases and jointly created the use case diagram.



At this point, Me suggested *Central Repository* as the architectural style for our system. We continued by deriving the detailed use cases. Figure G-7 shows what happened next. Me and Irene managed to work together on the first three use cases, while Myself veered off and did not finish his assignment. Me and Irene realized early on that they needed to coordinate their work because UC1, UC2, and UC3 are tightly coupled.

Ideally, we would all pull even weight but as the stuff got more complex, the speed of individual efforts became a factor. Delegating important tasks to some of the weaker members would hurt the team's performance and thus most of the tasks were on the shoulders of few team members. Irene insisted that we must make everyone devote equal effort, but Me would not surrender the ownership of the highest-priority use cases. He warned that the success of our project directly depended on UC1: Play Game and he felt most qualified to own this use case.

Me says: A major challenge we faced individually while working as part of a team was determining the overall direction of the project. In this case, one may think that the issue would be conflicting opinions about the features of the program. This was not the case at all. In fact, I felt that it was quite the opposite. The group itself was too apathetic about the approach I put forth for the program and this lack of feedback turned out to hurt us later on. This was because people accepted the ideas without really understanding what they were. When it came time to talk about the project or write parts of the report I found that almost none of what people wrote or talked about matched up with what we had agreed on, or even with each other! And that eventually led to me writing most of the report myself since I was the one that had a good understanding of the idea behind the project since it was my idea after all.

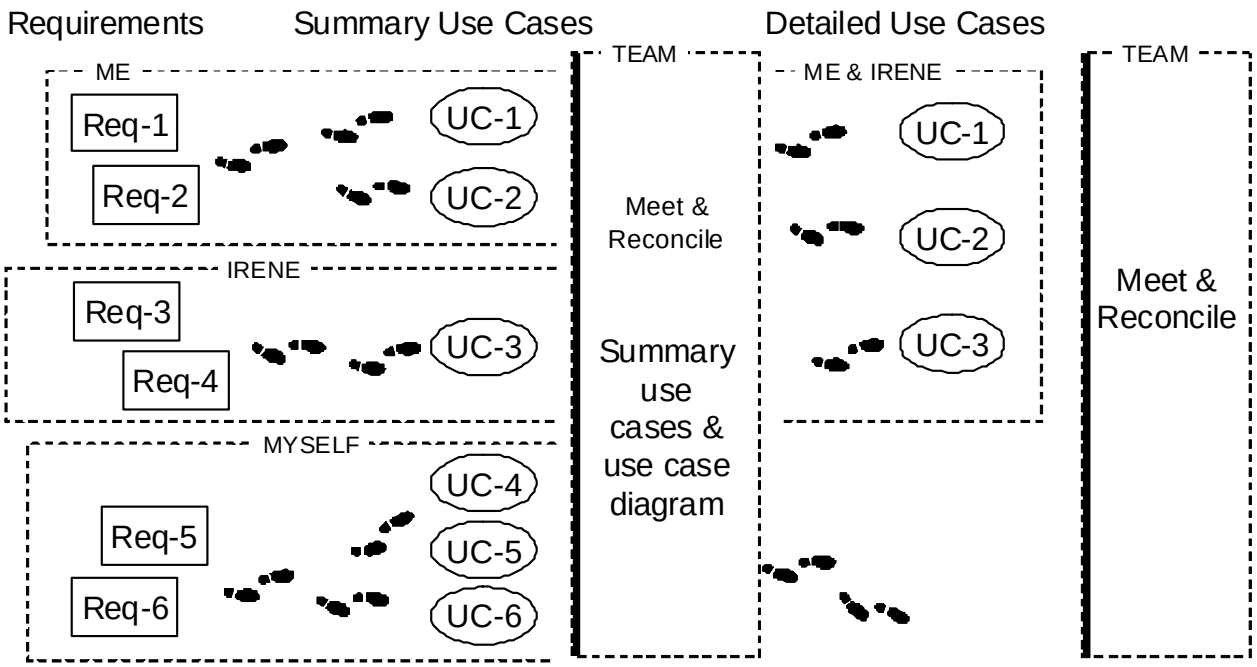


Figure G-7: How teamwork plan from Figure G-3 actually played out. (Continued in Figure G-8.)

Myself says: My biggest challenge from working in a group is how shy I am. This is especially a problem because of the competitive nature of this project. This is because other team members seem to be very aggressive about doing the parts that are of higher priority to the customer. Initially, to be a good sport, I would tell others to take whatever component of the projects that they liked and I would do the rest. This very quickly proved to be a huge mistake since I just couldn't motivate myself to do the residue work. Meanwhile, I did not realize how much work the rest of the team was putting in while I wasn't pulling my weight. It really took a read through their detailed use cases for me to grasp how much I had let my team down. Fortunately, this epiphany had a positive effect. I started to get into the project and did my best to contribute. I took on the responsibility to work on the user interface specification (next section).

Irene says: I have been an active part of different organizations since sophomore year in various leadership roles so I thought I was ready to be team leader. But leading a project like this is very different from any of the positions I had ever held. I compiled the final copy of the project report from everyone's contribution, additionally formatting and editing the document. This often involved redoing the tables and diagrams to try and achieve continuity and solid aesthetics in the report. I like to get my work done as soon as it is assigned so what ended up happening is that I'd work on the first sections of a report without any help and then weeks later when everyone else began to look at it I'd have to explain what I did and tell them how to do their sections.

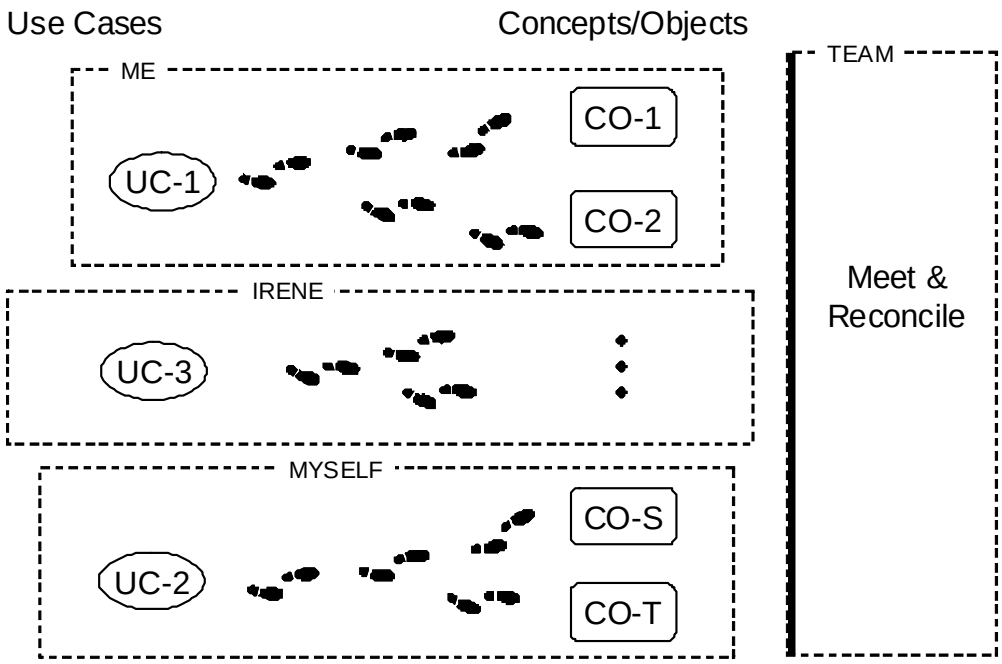


Figure G-8: Plan of work and ownership diagram for the next phase of the project: moving on from use cases to domain analysis. (Continued in Figure G-7.)

Unfortunately, having the report due in sections did not help this much, I was still working on things late on the night before the deadline because most team members didn't realize how much time needed to be put into this project. Given that the report turned out being many pages long, this was a very significant amount of work. I spent a good deal of time rewriting entire sections of the report to reflect our customer's suggestions, for which other team members gave me no credit. This was just extra stressful because I would start doing my best work very early in the semester and my grade was really hurt by people who waited till the night before to start trying to figure out what to do. I have source documents, draft iterations of the project report, and email traffic that would demonstrate the above statements to be true. In the future, we must ensure that the burden of compiling the final copy of the project report is equitably shared.

At this point, we decided that the first three use cases are critical, but very complex and cannot be done by one or two team members. We decided to leave out the supporting use cases for user registration and leaderboard display (UC4–UC6), and focus our resources to the highest-priority work (Figure G-8). Myself objected for being pushed around and wanted to continue working on his original use cases. Irene and Me convinced him that our team would be better off if he just got over it and took ownership of the new components more vigorously than in the past. We have to adapt our plans on the go to achieve the maximum impact. He grudgingly consented and thus we were off to the next phase.

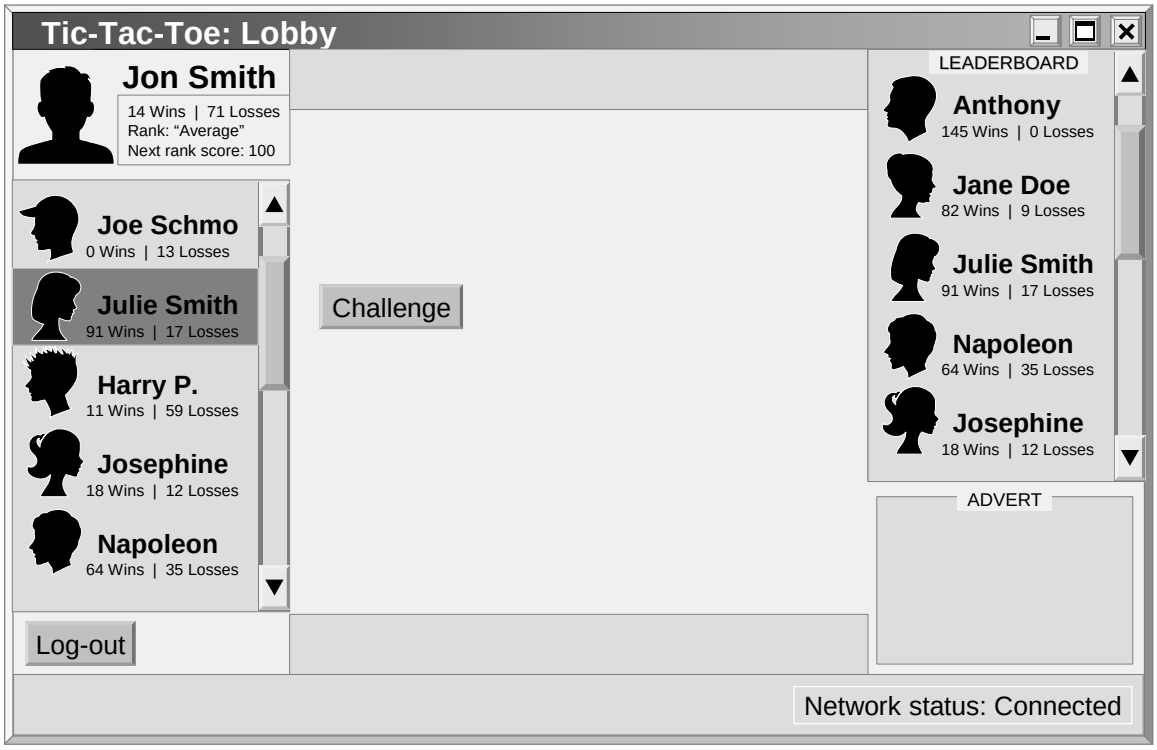


Figure G-9: Preliminary user interface design. Compare to Figure G-2.

G.4 User Interface Specification

Figure G-2 shows an initial sketch provided by our customer, showing how the customer envisioned the on-screen appearance of the User Interface (UI). Given that we learned about each usage scenario from detailed use cases, here we derive a preliminary user interface design. The designs presented in this section bear some resemblance to the customer requirement (Figure G-2), but they also reflect the details of use cases. They are still preliminary, because the application logic, which they are interfacing, is not implemented. Many details of the interface may and likely will change once we start coding.

G.4.1 Preliminary UI Design

For a given use case, show step-by-step how the user enters information and how the results appear on the screen.

Use screen mock-ups and describe exactly what fields the user enters and buttons the user presses. Describe navigational paths that the user will follow.

In case you are developing a graphics-heavy application, such as a video game, this is one of the most important sections of your report.

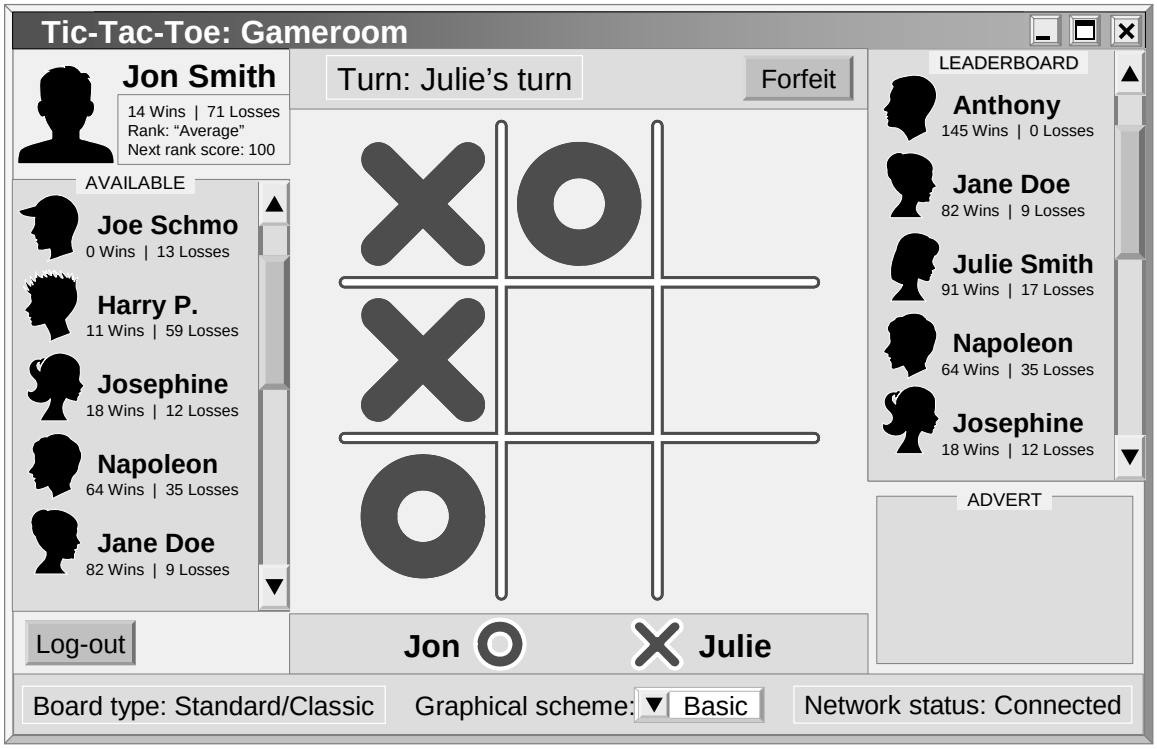


Figure G-10: Preliminary user interface design. Compare to Figure G-2.

Based on the detailed use cases (Section G.3.4) and operational model of the game (Figure G-6), we decide to have two main screens for the game. One screen will support user activities in preparation for the match, which we call the "game lobby" (Figure G-9). The other screen will support user activities during the match, which we call the "gameroom" (Figure G-10).

G.4.2 User Effort Estimation

When estimating the user effort, we assume that the user interface will be implemented as in Figure G-9 and Figure G-10.

Match Setup, in the Lobby

Minimum effort (best-case scenario) needed to successfully set-up a match:

one click to select an opponent + one click to send invitation = 2 mouse clicks

Maximum effort (worst-case scenario) needed to successfully set-up a match:

one click to select an opponent + one click to send invitation + one click to suggest different version + one click to accept a version counteroffer = 4 mouse clicks

This maximum effort is calculated assuming that the first selected opponent will accept the challenge, then will reject the proposed version and submit a counteroffer, which will finally be accepted by the initiating user. A successful match setup may be preceded by one or more unsuccessful attempts, for which the worst-case effort is calculated as follows.

Maximum effort for every unsuccessful set-up attempt:

one click to select an opponent + one click to send invitation + one click to suggest different version + one click to reject a version counteroffer = 4 mouse clicks

Match Playing, in a Gameroom

Every move requires just a single click. (Our system will not support undoing of user actions, as explained in Section @@.)

G.5 Domain Analysis

G.5.1 Domain Model

We first derive the domain model concepts, starting from responsibilities mentioned in the detailed use cases (Section G.3.4). Table G-3 lists the responsibilities and the assigned concepts. The reader should be able to identify the first 11 in the main scenario of UC-1. The next two responsibilities are identified from the alternative scenarios of UC-1. We also realize from the first alternative scenario of UC-2 that we need a queue to line up potential simultaneous invitations, which yields responsibilities R14 and R15.

Concept definitions

The concepts and their responsibilities are discussed below (also see Figure G-11).

Table G-3: Deriving concepts from responsibilities identified in detailed use cases.

Responsibility	Type	Concept
R1: Coordinate activity and delegate work originated from the local player in a way that is compliant with the game operational model (Figure G-6).	D	Controller
R2: Display the game information for the player and dialog messages		Interface
R3: Monitor network connection health and retrieve messages from opponent	D	Communicator
R4: Keep the list of players that are available to play the game	K	Player List
R5: Keep the status of the local player and his/her scores	K	Player Profile
R6: Information about the opponent invitations to play the game	K	Match Invitation
R7: Gameroom information and the game board	K	Gameroom
R8: Randomly assign **X**s or **O**s to the players	D	Communicator
R9: Prompt the player to make the next move	D	Controller
R10: Prevent invalid moves, detect three-in-a-line and declare a winner or detect that the board is filled to end in a draw	D	Referee
R11: Store the updated score of the local player in the database	D	Communicator
R12: Time the player responses	D	Response Timer

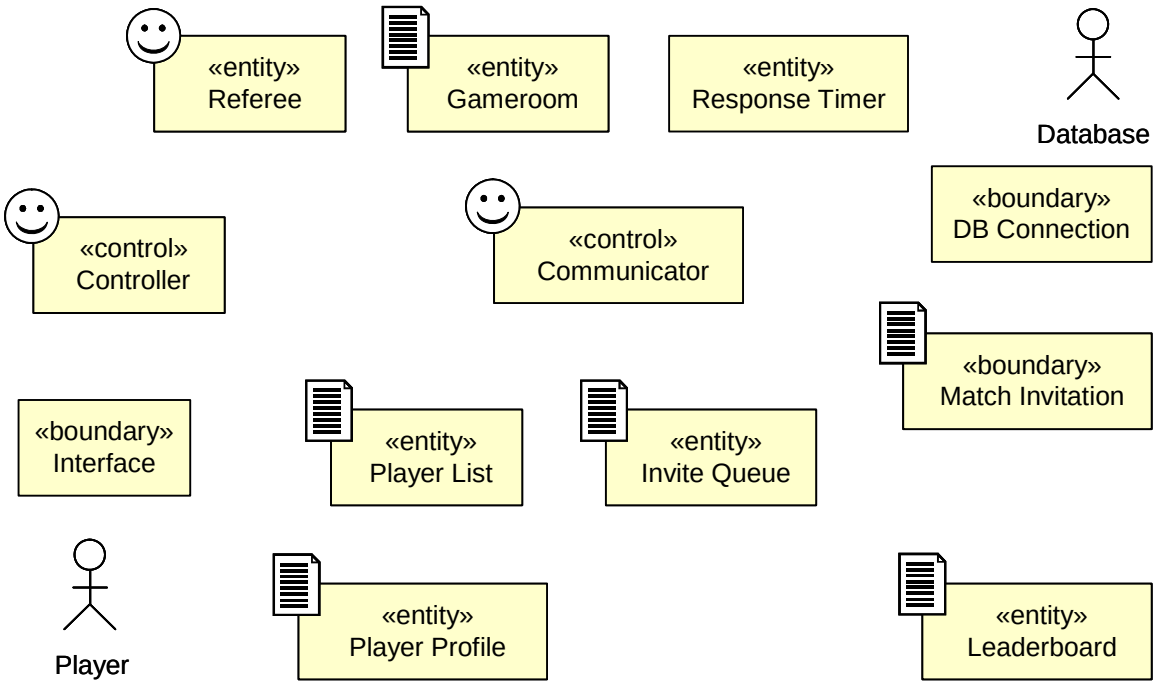


Figure G-11: Concepts of the domain model for the game of tic-tac-toe.

R13: Apply the RESPONSE TIME POLICY and declare the winner	D	Referee
R14: Queue multiple (nearly simultaneous) invitations from other players	K	Invite Queue
R15: Randomly select a challenger from the Invite Queue	D	Communicator
R16: Manage interactions with the database		DB Connection
R17: Match-in-progress information	K	Gameroom
R18: Process actions in reaction to Response Timer timeouts	D	Communicator
R19: Conclude the failed negotiations for selecting a version of the game	D	Communicator
R20: A scoreboard with the current scores of the leading competitors	K	Leaderboard

We realize that our system will receive two types of requests: commands from the local player and messages from the remote opponent. To keep separate these unrelated responsibilities, we introduce two «control» type objects: **Controller** and **Communicator**.

Given the CENTRAL REPOSITORY architectural style adopted in Section G.3, we assume that each user will run his or her application that will connect to the central database. The Communicator keeps track of database updates by other players that are relevant for the local user. The Communicator also stores to the database information from the local player that is relevant to the opponent or any other remote player.

The Communicator uses **DB Connection** interface to ensure separation of concerns: we want to separate database-querying responsibilities from responsibilities of dispatching remote requests and monitoring the network connection health.

Responsibility R8 (randomly assigning **X**s and **O**s) is performed by the Communicator. However, every client runs a Communicator, so which one performs the random assignment or does this need to be negotiated? A simple solution is having the Communicator of the player who sent an invitation to do the random assignment and send it as part of the invitation. Or, the opponent's

Communicator could do the assignment. Alternatively, the client with the greater network address performs the assignment. (In the last option, the database would need to store players' network addresses.) This is a design decision that may need to be changeable, but unlike business policies that are decided by the customer, design decisions are made by the developer.

There are several other responsibilities assigned to the Communicator in Table G-3 and at some point it may be necessary to introduce additional concepts to offload some responsibilities.

Player List is the list of currently available players that is periodically retrieved from the database by the Communicator. **Player Profile** keeps the availability status of the local player, which is why it is marked as an «entity» concept. However, one may argue that it is also a «boundary» concept when representing remote players in Player List. **Match Invitation** is the message sent to an opponent or the message(s) received by opponents.

The **Gameroom** conceptualizes the game session established between two players (after a remote opponent accepts a challenge). We may consider introducing a new concept Match to keep information about the current match (R17); however, we decide against it based on the following reasoning. Given the way we operationalized the game of tic-tac-toe in Section G.3.4, the players can play only a single match in the gameroom. Every new match must be initialized anew. Therefore, the distinction between the Gameroom and Match is apparent. We may also consider introducing a concept GameBoard, but for now, we decide against it because the board state can be represented as an array data structure that, in turn, can be an attribute of the concept Gameroom.

We decide that the Controller should not act as the game referee (R10), because the Controller has function-dispatching responsibilities and game refereeing is a complex and unrelated responsibility. Instead, we introduce the Referee concept.

The **Referee** monitors players' moves in a match, sanctions valid moves, and determines if a move is a winning move (three identical signs, e.g., **X**s, in a line), or a finale condition ("draw"). Of course, each Referee is refereeing only its local player, i.e., the Referee verifies only whether the local user's last move was valid.

The Referee may additionally check if it is impossible for either player to win (because the current board state is such that it is impossible to have a three-pieces- in-a-line) and terminates the match ("draw" outcome) without waiting that all cells on the game board become filled up. In this scenario, one hopes that both Referees will produce the same result, but given the network latencies and communication via the central database, there may be intermittent inconsistencies that need to be considered. This is particularly true when the RESPONSE TIME POLICY needs to invoked because the opponent has not responded within the time limit.

If the players are playing the revenge version, the Referee shall declare a match a win if the local user has three pieces on the board and the next move cannot result in a win for the opponent.

The system shall ensure that if the 9-board version is being played, all but the first move are placed in the empty spaces on the bard corresponding to the square of the previous move.

We will need three specialized Referee types to implement game rules for different variants. One may wonder whether the game "rules" should be explicitly formulated as an attribute or another concept. For example, to recognize a winning condition, the Referee needs to apply simple pattern matching. Given the 3×3 board array $a[i,j]$, a player wins if any of the following is true:

Horizontal three-in-a-line: $a[i,j] = a[i,j+1] = a[i,j+2]$, for $0 \le i \le 3$ and $j = 0$

Vertical three-in-a-line: $a[i,j] = a[i+1,j] = a[i+2,j]$, for $i = 0$ and $0 \le j \le 3$

Diagonal three-in-a-line: $a[i,j] = a[i+1,j+1] = a[i+2,j+2]$, for $i = 0$ and $j = 0$

These rules apply for all three version of tic-tac-toe, with additional rules for revenge and nine-board versions (with a small modification of the rules for the nine-board). At this point, we decide that no additional concepts are needed. Note that this analysis is *not* for the purposes of solution design; rather, its purpose is to decide whether we need additional concepts.

We assume that the **Leaderboard** is only a data container (knowing responsibility), because it does not need to perform any computation. The database already keeps an up-to-date score of each player as part of the player's record. The rank-ordered list of players can be retrieved and sorted by a database query.

Attribute definitions

Attributes of domain concepts are derived in Table G-4. The Controller needs to carry out the policy of not allowing the user to send more than one invitation at a time, so it needs to know if the user is awaiting an opponent's response. Therefore, the Controller has an attribute **isAwaitingOpponentAnswer**. The Controller also has an attribute **isAwaitingOpponentMove** to prevent the local user from moving board pieces before the opponent responds.

The Communicator also needs an attribute **isAwaitingOpponentAnswer** to know if waiting opponent's answer discard requests from other users or requests from the opponent that were generated nearly simultaneously—see the alternative scenarios for UC-2: Challenge and UC-3: SelectGameVariant in Section G.3.4. The reader may find it redundant to keep duplicate information but, at this point, we are only identifying what is needed, *not* optimizing the solution design or implementation.

The Communicator also has an attribute **connectionStatus**, which indicates the network connection health: connected or broken. When the Communicator detects network failure, it informs the Controller, which in turn cancels the ongoing match and closes the gameroom, signals the network failure to player and informs about a possibly forfeited match. The Controller also blocks any actions that require network connectivity, which in our simple version of the game means that the user cannot do anything (the leaderboard state will be stale) except logout. The Communicator will continue monitoring if the network connection becomes restored.

It is not clear why the Controller's attribute **isInGameroom** would be necessary, because the system has other means to keep track if currently the user in the gameroom—for example, if the Gameroom attribute matchStatus is "pending." However, this attribute expresses a needed responsibility and because, during analysis, design optimization is of low priority, we keep it. This attribute's significance will become apparent in Section G.8.1.

The Controller's attribute **isNetworkDown** is related to **connectionStatus**, but the former is used to block user's activity if network is down, while the latter helps the Communicator to monitor the network outage and recovery.

The Referee needs to keep track of whether the local player should make the next move (attribute **isLocalPlayerNext**). For the first move, the Referee needs to know if the local player is assigned

Xs (attribute **isLocalPlayerX**). The reader may conclude that these attributes make the Controller's attribute isAwaitingOpponentMove redundant. We keep it to indicate the existence of a responsibility and avoid optimizing at this stage. The Referee also keeps track if the local user's last move was valid (attribute **isLocalMoveValid**). Finally, after the Referee detects a match end, it notes the winner's identity, so that if the local user won, it can request the Communicator to update the user's score in the Database.

Attribute **boardMatrix** of the concept Gameroom stores the contents of each of the nine cells on the game board. The allowed values of each cell are: empty, an **X**, or an **O**. The Gameroom also maintains the current state of an ongoing match (values: pending, ongoing, or complete).

Table G-4: Deriving the attributes of concepts in Table G-3 from responsibilities identified in detailed use cases (Section G.3.4).

Responsibility	Attribute	Concept
R21: Know if local user is awaiting opponent's answer, to prevent actions except viewing leaderboard or logout	isAwaitingOpponentAnswer	Controller
R22: Know if local user is in the gameroom	isInGameroom	
R23: Know if local user is awaiting opponent's move	isAwaitingOpponentMove	
R24: Know if network connection broken to block actions	isNetworkDown	
R25: Player's identity or name	ID / name	Player Profile
R26: Player's cumulative score	score	
R27: Player's status (idle, available, engaged, invisible)	status	
R28: Identity of the invitation sender	inviter	Match Invitation
R29: Identity of the invitation recipient	invitee	
R30: Invitation status (pending, accepted, declined)	status	
R31: Store the contents of the 9 cells on the game board	boardMatrix	Gameroom
R32: Match present state: none, pending, ongoing, complete	matchStatus	
R33: Indicate if the local player is assigned **X**s	isLocalPlayerX	Referee
R34: Indicate if the local player goes next	isLocalPlayerNext	
R35: Indicate the validity of the local player's last move	isLocalMoveValid	
R36: Identity of the match winner, none in case of a draw	winnerID	
R37: Time to count down to zero	duration	Response Timer
R38: Identity of the current opponent	opponentID	Communicator
R39: Know if waiting opponent's answer discard requests from other users or requests from the opponent that were generated nearly simultaneously	isAwaitingOpponentAnswer	
R40: Watch network connection for health or expected messages	connectionStatus	
R41: Rank-ordered list of currently top scoring players	playerRankList	Leaderboard
R42: When was the leaderboard last updated	updateTime	
R43: Network address of the relational database	dBnetworkAddress	DB Connection

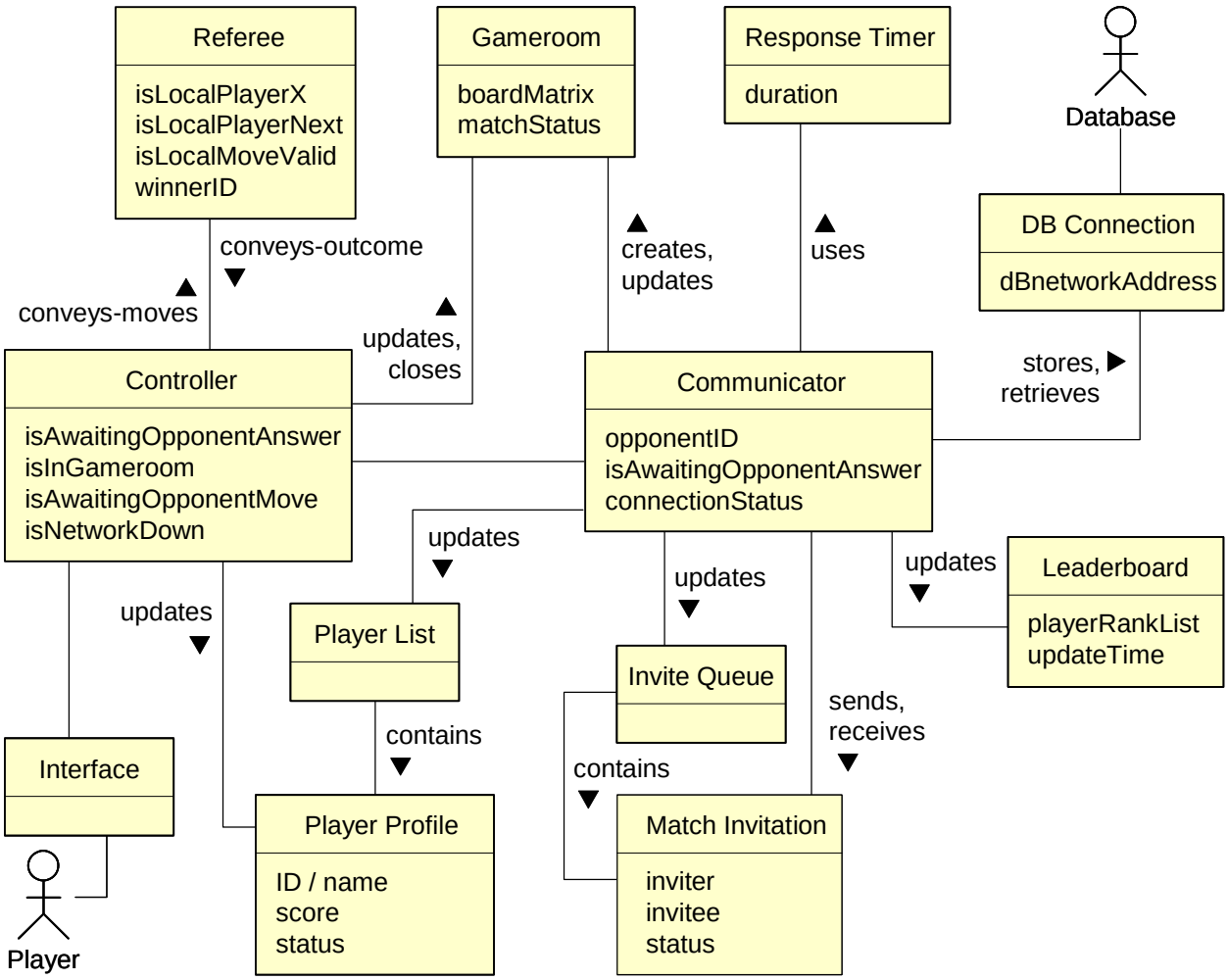


Figure G-12: Domain model diagram for the distributed game of tic-tac-toe.

Association definitions

Associations of domain concepts are derived in Table G-5. The Communicator creates the Gameroom after receiving challenge acceptance from an opponent. The Communicator also updates the gameboard with opponent's moves. Note, however, that the Controller updates the gameboard with local player's moves. Because the Controller is associated with the Referee and will be the first to hear about a finished match, the Controller will close the Gameroom when the match is over.

We do not show that the Controller has an association with Match Invitation. Although the Controller will receive the local user's selection of the opponent to challenge, we assume that the Controller will pass this request on to the Communicator that, in turn, will generate the Match Invitation. The Communicator will inform the Controller about the network connection health. Because this association between the Controller and the Communicator is complex and involves different information exchanges, it is not named and shown in Table G-5.

We assume that only valid moves will be communicated across the network to the opponent's system. The Referee conveys the move validity back to the Controller, which, in turn, asks to the Communicator to send it remotely. Therefore, the Referee and the Communicator are not directly associated.

Note that the Controller is associated with Player□Profile to update the local player's status. This association is because the Controller learns from the Referee when the match finished, so it closes the Gameroom, which makes the local player available for the next opponent. The Controller may also receive requests from the player to make him "invisible."

When considering the associations for Response Timer, we realize that we have not specified whether the timer times the local user's response or the opponent's response. If former, it would probably best be associated with the Controller. In this case, if the local user does not respond or move a piece on the board within the response time limit, the Controller would decide that the user lost the match and ask the Communicator to record it the database. However, we realize that we also have a policy that the user who loses the network connection also loses the match, see TTT-BP04 in Section G.3.4. This cannot be implemented if the local user has no connection to the database. Therefore, we decide that the timer will time the opponent. As a result, Response Timer is associated with the Communicator. If the Communicator does not receive the opponent's response before the timer expires, it will declare the local user winner and update the user's score in the database.

Table G-5: Deriving the associations of concepts listed in Table G-3.

Concept pair	Association description	Association name
Controller ↔ Referee	Controller conveys to Referee the local user's move and Referee conveys the evaluation outcomes to Controller	conveys-move, conveys-outcome
Controller ↔ Gameroom	Controller updates gameboard with local player's moves	updates, closes
Controller ↔ Player Profile	Controller updates Player Profile to reflect the local player's current status	updates
Communicator ↔ DB Connection	Communicator stores requests from the local user to database (via DB Connection) and retrieves requests from opponents	stores, retrieves
Communicator ↔ Gameroom	Communicator creates Gameroom and updates gameboard with opponent's moves	creates, updates
Communicator ↔ Player List	Communicator updates Player List with currently available opponents	updates
Communicator ↔ Match Invitation	Communicator sends Match Invitation to a selected opponent and receives invitations from other opponents	sends, receives
Communicator ↔ Invite Queue	Communicator updates Invite Queue with invitations received from other opponents	updates
Communicator ↔ Response Timer	Communicator uses Response Timer to time opponent's response and implement RESPONSE TIME POLICY	uses
Referee ↔ Communicator	Referee asks Communicator to update the local player's score if he won a match	update-score

The complete domain model diagram is shown in Figure G-12. The concept ornaments are omitted for clarity, and the reader should refer to Figure G-11 for additional details.

Use Case	PW	Domain Concepts: Controller	Interface	Communicator	Player List	Player Profile	Match Invitation	Gameroom	Referee	Response Timer	Invite Queue	DB Connection	Leaderboard
UC1	16	X	X	X	X			X	X	X		X	
UC2	4	X	X	X	X		X	X		X	X	X	
UC3	7	X	X	X				X	X	X		X	
UC4	1	X	X	X		X						X	
UC5	1	X	X	X								X	
UC6	2	X	X	X								X	X
Max PW		16	16	16	16	1	4	16	16	16	4	16	2
Total PW		31	31	31	20	1	4	27	23	27	4	31	2

Figure G-13: Traceability matrix mapping the use cases to domain concepts. (Continued from Figure G-5 and continues in Figure X.)

Traceability matrix

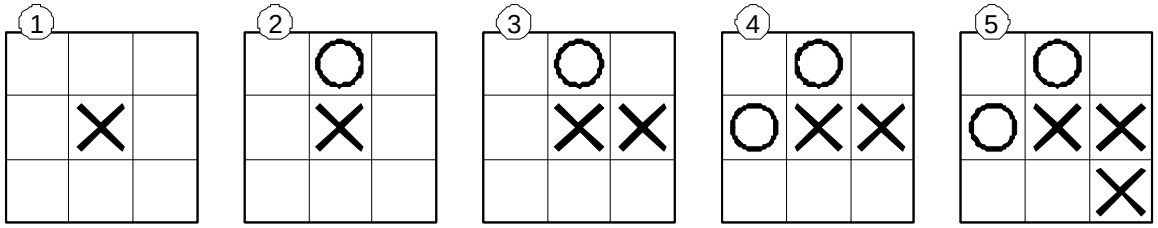
Figure G-13 shows how the system use cases map to the domain concepts. This matrix is derived based on the assignment of responsibilities to concepts in Table G-3. The responsibilities, in turn, originated from the use cases.

G.5.2 System Operation Contracts

Should be provided only for the operations of the fully-dressed use cases elaborated in Section 3.c), for their system operations identified in Section 3.d).

G.5.3 Mathematical Model

As with any strategy game, it helps to know some strategies to win the game of tic-tac-toe or at least to force a draw. For example, the **O**s player must always respond to a corner opening with a center mark, and to a center opening with a corner mark. Otherwise, the player who makes the first move (the **X**s player) will always have an advantage. Here is an example match:



After the fifth move, the **X**s player has sealed his victory. No matter where the **O**s player moves next, the **X**s player will win. Show that the **O**s player could have avoided this situation if his first move was a corner mark.

Therefore, the player going second is always on the defensive and may never get a chance to win when playing with a player that knows what he or she is doing. The Wikipedia page contains more discussion on the game strategies (http://en.wikipedia.org/wiki/Tic-tac-toe).

To help make the game more fair, our system will always randomly designate the players to play either **X**s or **O**s and the players will not be allowed to choose their pieces (Section G.3.4). The reader may wish to relax this constraint under certain scenarios. For example, for non-standard versions of the game of tic-tac-toe, such as revenge or nine-board, or when both players achieve certain expertise level (based on their standing on the leaderboard), the system would allow the players to choose their pieces.

How We Did It & Plan of Work

The least important technique to our group, to our detriment, was effort estimation. This often caused significant logistic problems because very little time was left to effectively collaborate. There was always significant "bottlenecking" between stages of our project. In other words, portions that are dependent on others cannot be completed until their dependencies are sufficiently completed.

Irene says: My group habitually waited until the last minute to do their parts. I tried my best to get my portions done as early as I could but when I was restricted by a "bottleneck" stage and would attempt to go ahead and finish, the member would get very upset because they were afraid of loosing ownership of their components.

Me says: My prior experience was only small programs. Analyzing the system provided much better understanding of the software we were expected to implement. It is hard to implement a program when you don't fully understand it. I believe that not everyone in the team understood the operational model that I came up with for the game and just went with it without voicing their opinions. Then, when the time came to actually write the report on how it worked, almost everyone in the team had a different opinion on how it worked. Thus, several parts of the report did not really match up at first. I had to revise a lot of parts to match what the initial idea of the project represented, but I could not catch everything as the report was fairly extensive and I already had many responsibilities in the group as it was.

Irene says: This was due to miscommunication within our group since a majority of our meetings were quick due to conflicts with each member's schedule. This issue led to a majority of individual work until we have a meeting date to combine each individual member's work. This harshly reduced individual member's work quality since they were doing it based on any concepts and idea of how the system should work during our group meeting.

Myself says: The biggest challenge of working with a team was communication and work distribution. It was hard to stay in touch, which was mostly due to our individual busy schedules. This made it hard to properly distribute the work. I wasn't always sure who was working on what, so occasionally some of us would have each done the same part of the project. There were definitely (and unfortunately) weak moments and bitterness that we had to face during situations when we could not reach to a clear consensus on an idea or where one would feel short up to one's standards.

While much of the content of the reports was helpful, I found a lot of it tedious. It detracted from the more important aspects like improving the application and marketing. The amount of planning that had to go into the project felt like overkill for something of the tic-tac-toe game size, however it did expose us to a myriad of techniques. Among the techniques we learned were gathering and formulating a comprehensive set of requirements, deriving use cases, and translating those into a domain model.

Me is a very bright individual, but he's most concerned with the programming aspect of things. He wasn't the slightest bit interested in writing up reports. However, for the first report, after Irene and I worked through what we had done and prepared it for submission, Me finally appeared a few hours before submission, after not answering phone calls for days and pretty much redid most sections of the report with what he felt was better for the project. Obviously, this was good for the report, but it caused quite a hassle for Irene and me because we had to prepare a final product to send you in a timely manner. Once we got our grade back, we realized that we lost a lot of points for things that Me was assigned to do and never even did (mostly pertaining to the Domain Analysis section). I wasn't pleased, but I know the process isn't perfect, so I just sucked it up and knew I had to do better on the next reports.

G.6 Design of Interaction Diagrams

We know that software design cannot be gotten "right" first time around; we need iteratively to refactor the design until it converges to a satisfactory quality or we run out of time. We start by deriving an initial design ("first iteration"). Then we evaluate the initial design and introduce some improvements.

G.6.1 First Iteration of Design Sequence Diagrams

Figure G-14 shows the main loop for the Communicator and Controller. The Controller periodically accesses the central database to retrieve any messages for the local user. By default, it needs to refresh the list of available players and the leaderboard. In addition, the local user might have already challenged an opponent and is awaiting an answer. Alternatively, the local player might have been challenged by a remote player. If the local player received a challenge but does not respond within the response time, the Communicator will automatically bring up the initial screen.

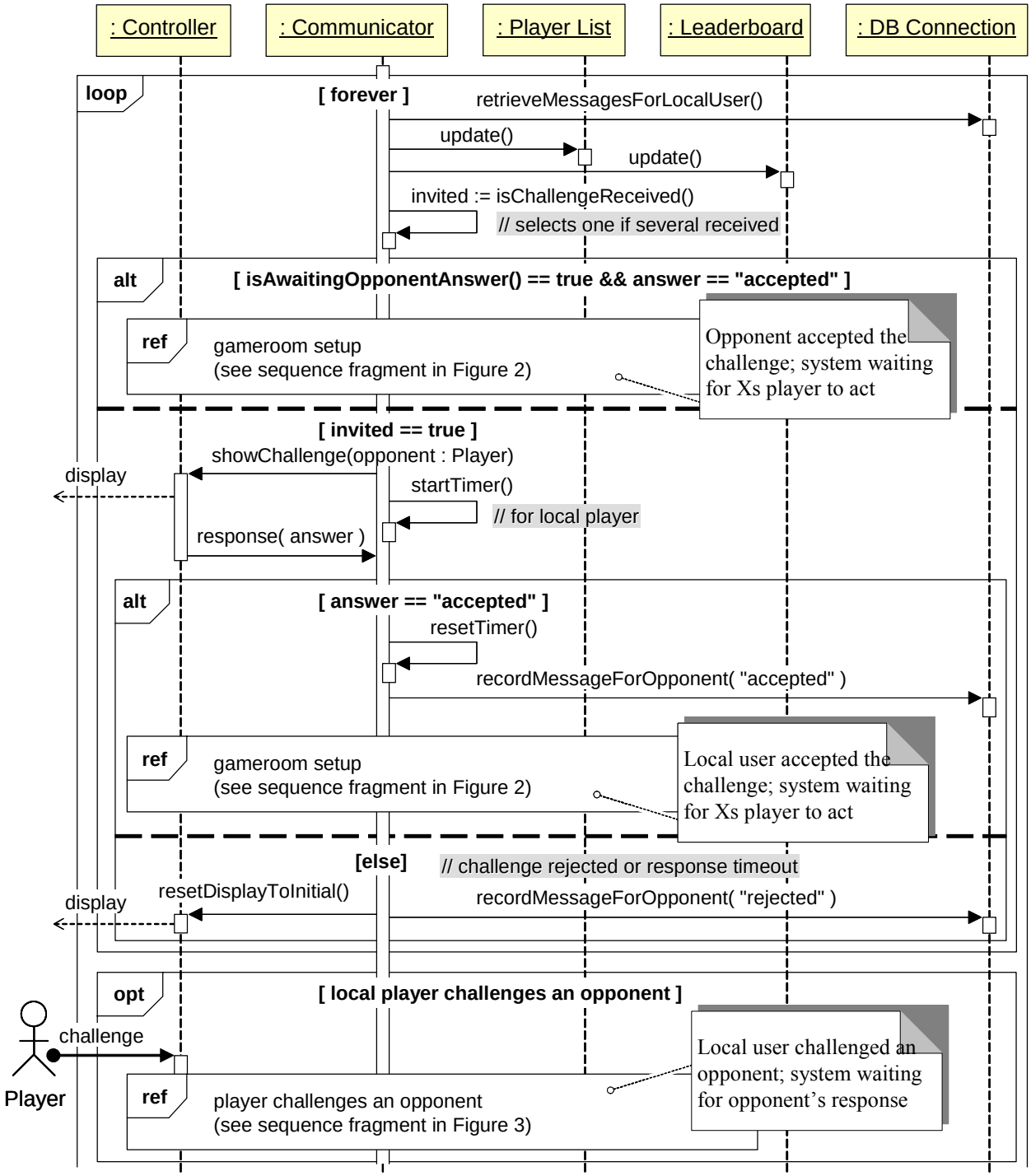


Figure G-14: Sequence diagram for the main loop in the game of tic-tac-toe. See Figure G-15 and Figure G-16 for the [ref] interaction fragments.

In Figure G-14, the main loop breaks down the possible interactions well. The user can send out a challenge, accept a challenge, or decline a challenge. The "forever" loop allows the system to wait idle until the local or remote player decides to send an invitation.

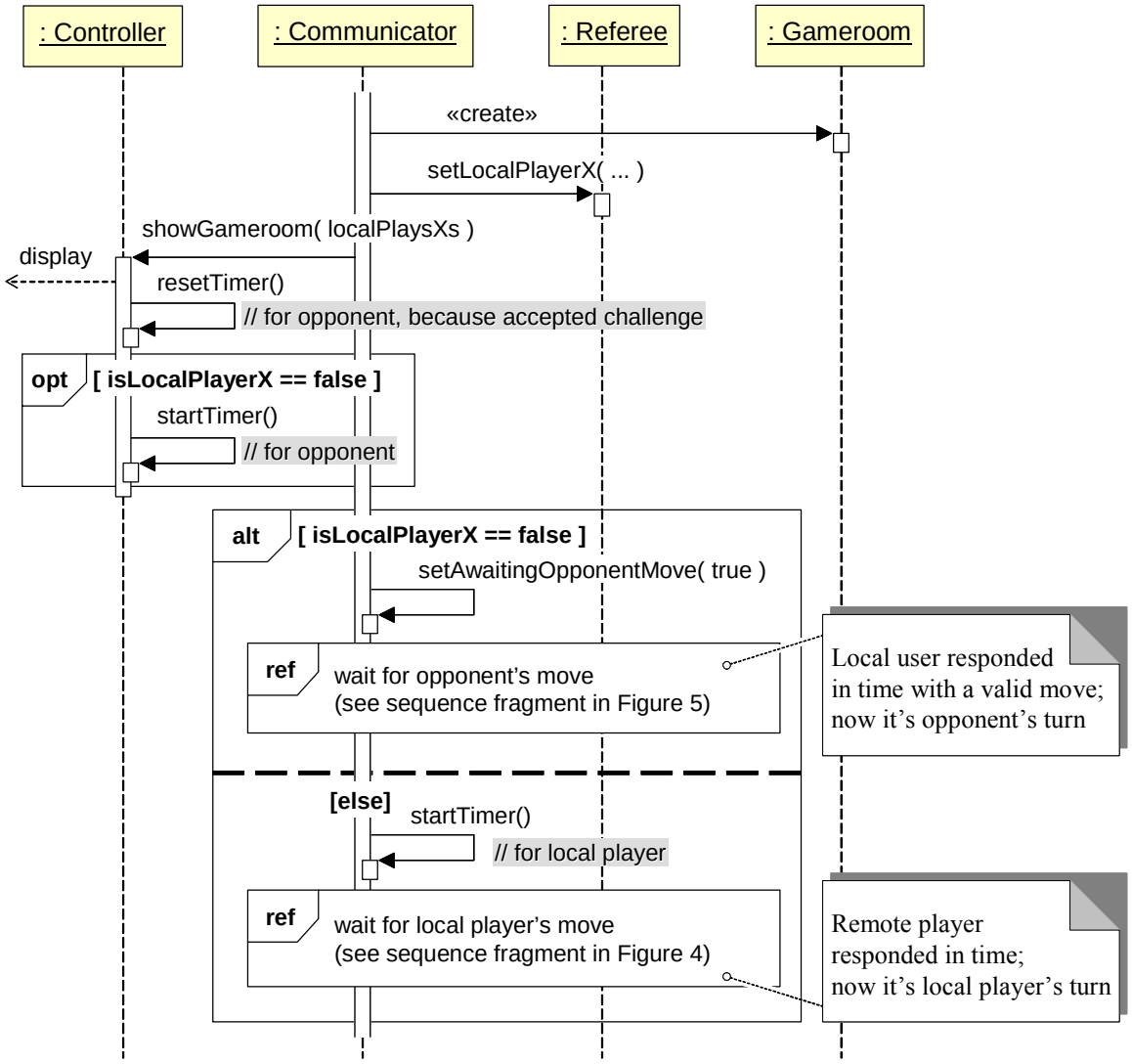


Figure G-15: Sequence diagram for the gameroom setup in the game of tic-tac-toe. See the [ref] interaction fragments in Figure G-17 and Figure G-18.

Figure G-15 shows a fragment of the sequence diagram that executes when a challenged player accepts the invitation. (It may be that either a remote opponent accepted the challenge by the local user, or the local user accepted the challenge from a remote player, see Figure G-14.) The Communicator first calls the method `setLocalPlayerX()` on the Referee. Recall from Section G.3.4 that the player assignment to **X**s and **O**s is performed randomly by the Communicator that sends a match invitation. If the local user challenged a remote opponent, then the local Communicator performed the assignment before sending the invitation. Alternatively, if the local user accepted a remote challenge, then the local player's designation was received in the invitation. The Communicator creates a new Gameroom and asks the Controller to show it.

The Controller also resets the response timer for the opponent that might have been set when the invitation was sent to an opponent (see Figure G-16).

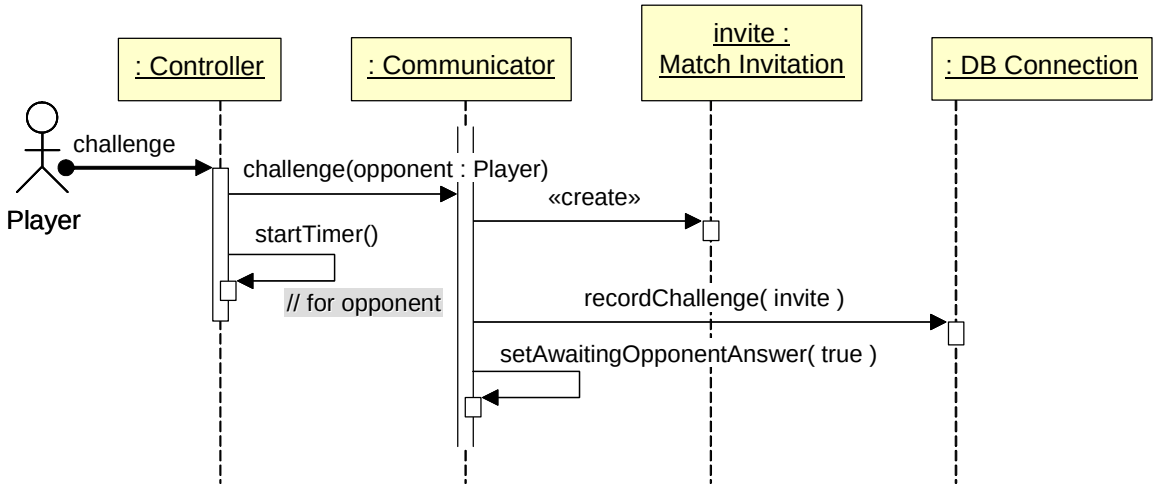


Figure G-16: Sequence diagram for the local player to challenge an opponent to play a match of the game of tic-tac-toe. This [ref] interaction fragment is part of the main loop Figure G-14.

If the local player is assigned **O**s, then the Controller starts the response timer for the opponent and the local player is allowed only the following actions: view leaderboard, logout, or forfeit the just started match. The remote player (in this case assigned **X**s) can suggest a different version of the tic-tac-toe game or move a piece on the board.

For the sake of simplicity, the initial design does not show the case when the players negotiate a different version for tic-tac-toe. We will add this case in subsequent iterations.

In Figure G-18, in the method `opponentMove()`, the Referee implementation will use a system timer to time the local player's response. If the local player fails to respond within the response time limit, he loses the match, the (local) gameroom is closed and the player is brought to the initial screen.

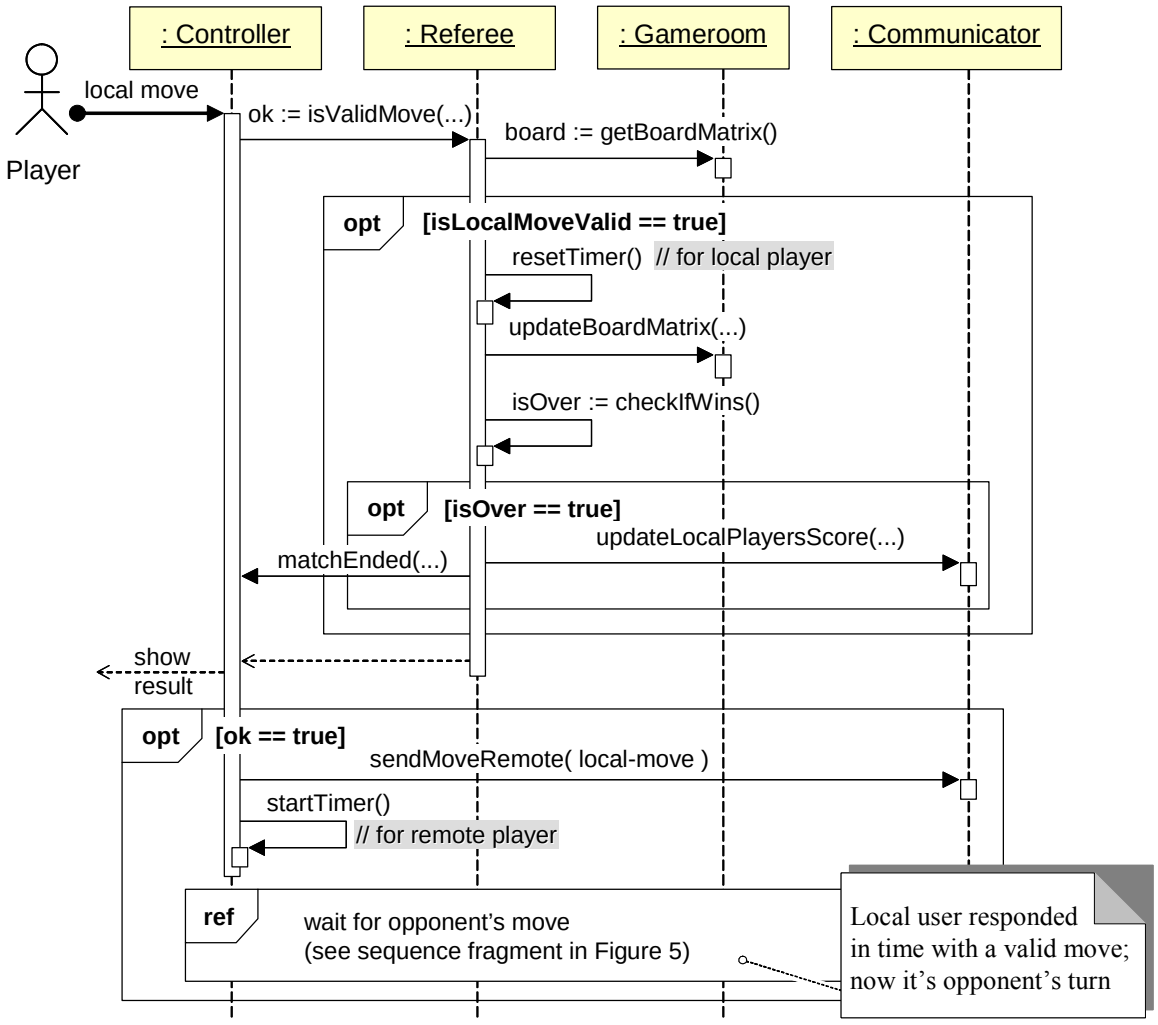


Figure G-17: Sequence diagram for local player's move in the game of tic-tac-toe. Compare to Figure G-18 that shows the [ref] interaction fragment.

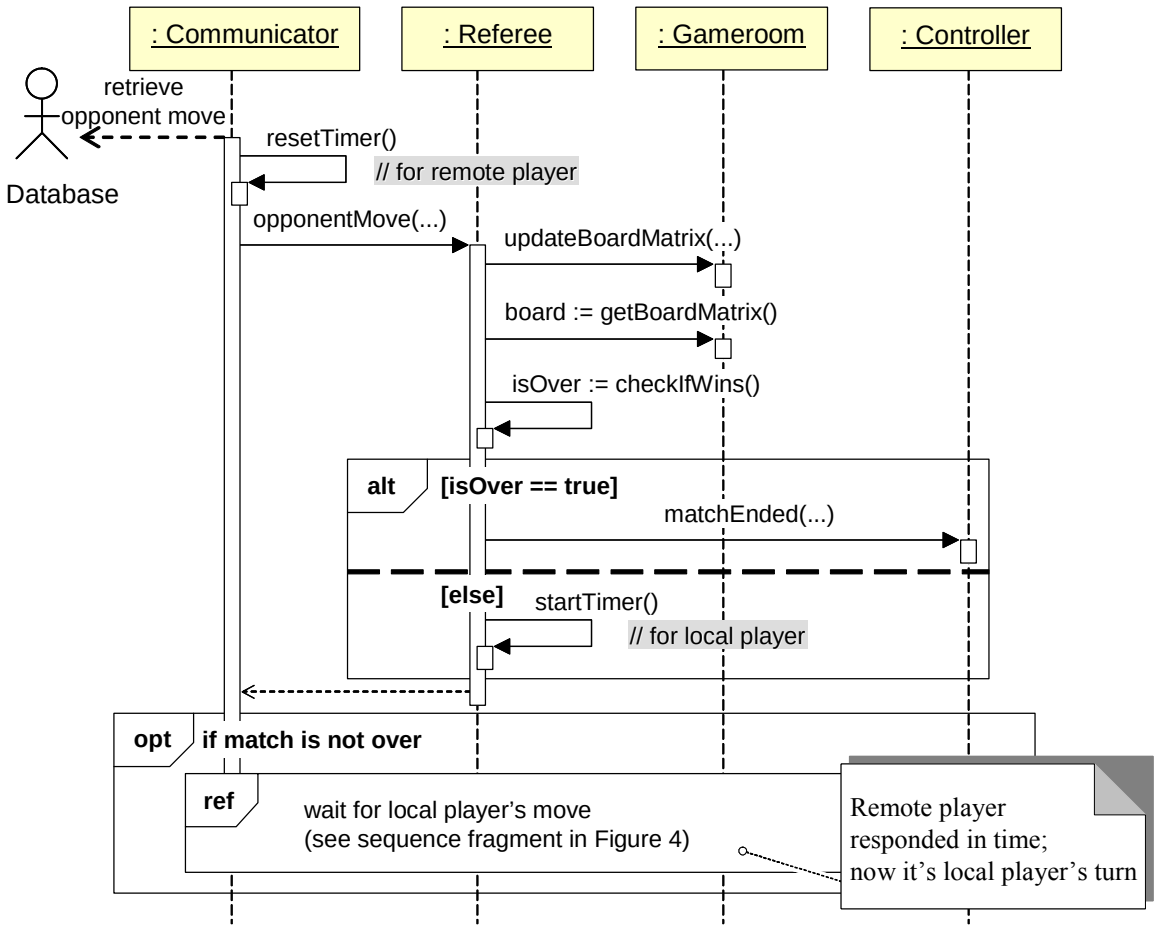
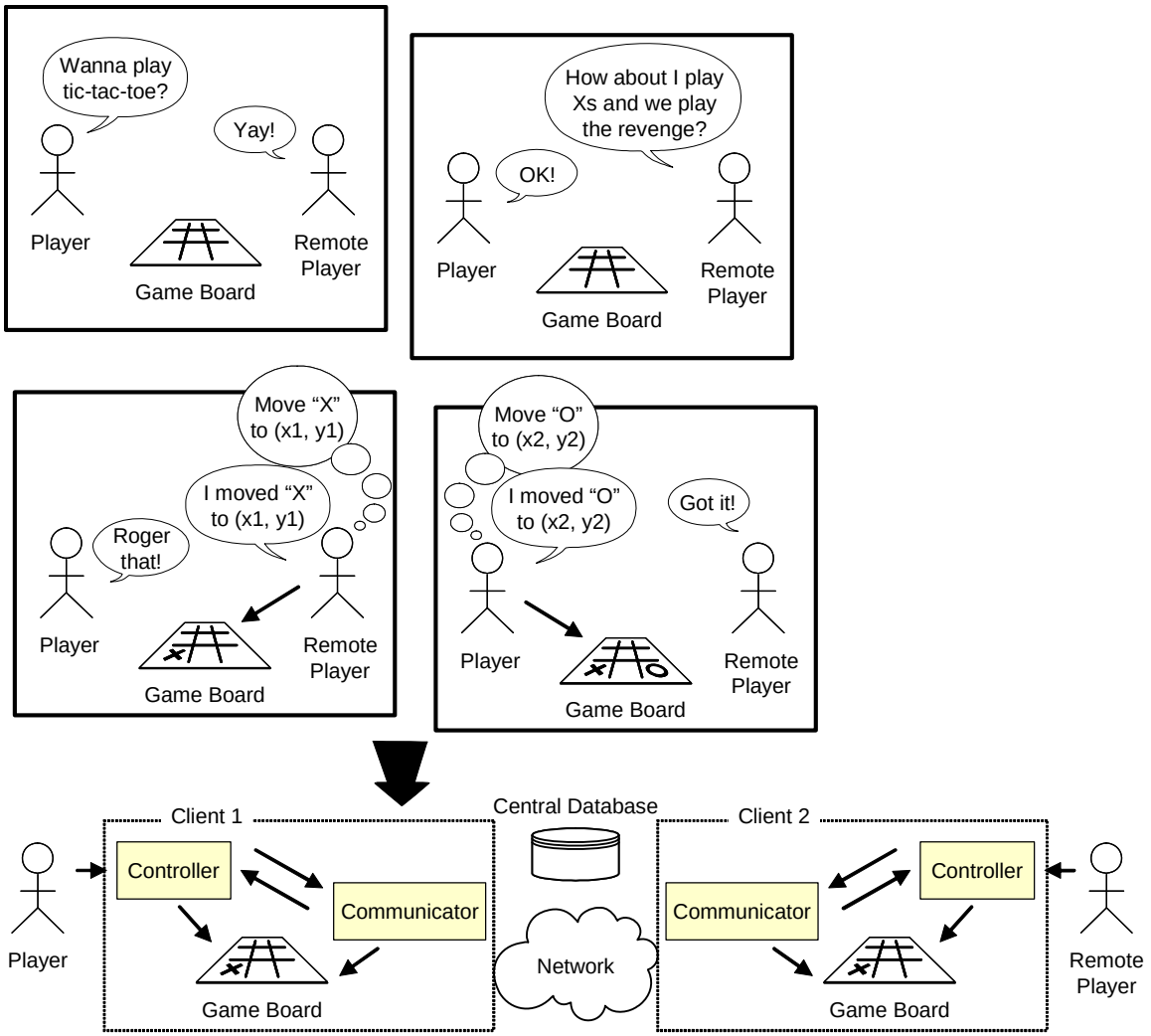


Figure G-18: Sequence diagram for remote player's move in the game of tic-tac-toe. Compare to Figure G-17 that shows the [ref] interaction fragment.

G.6.2 Evaluating and Improving the Design

This section evaluates the above initial design and introduces some improvements. A key task in this section will be to compile the responsibilities of classes from the initial design and look for overloaded classes or imbalances in responsibility allocation. Further improvements will be considered in Section G.9 by applying design patterns.

To better understand the system that we are designing, we draw this storyboard for the game.



Our system-to-be must implement the virtual gameboard, and support players' interaction with the gameboard and communication with one another. We could have had a single Controller to orchestrate the work of other objects, but that would make the Controller too complex because of too many responsibilities. Our initial design offloads at least some responsibilities to the Communicator. Here is what these to key objects are doing:

Controller:

- Allowing the local player to interact with the local gameboard
- Allowing the local player to interact with the remote gameboard (via the Communicator)
- Allowing the local player to communicate with the remote player (via the Communicator), such as challenge, negotiate version, etc.

Communicator:

- Allowing the remote player to interact with the local gameboard (via the Communicator)

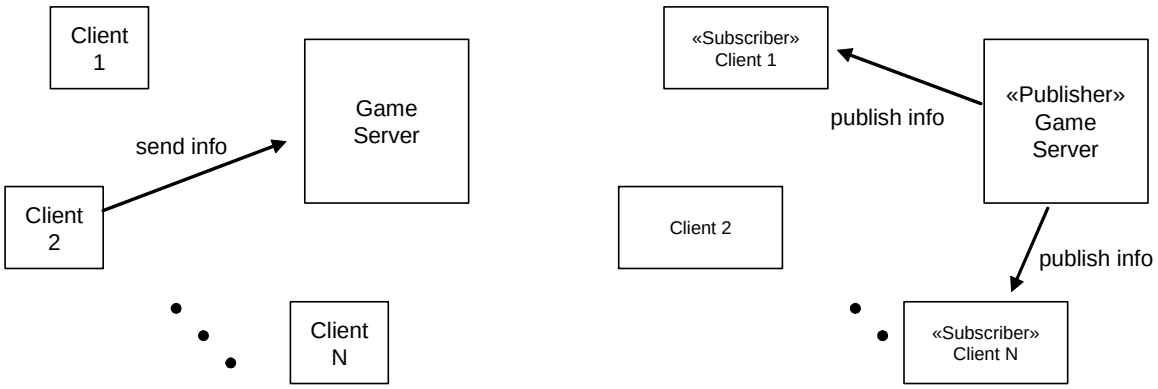


Figure G-19: Client-Server architectural style for the distributed game of tic-tac-toe, where the clients ("game consoles") connect to the central server. When a client wishes to send information to other clients, it first sends it to the server (left figure), which publishes this information to appropriate clients (right figure).

• Allowing the remote player to communicate with the local player (via the Controller), such as challenge, negotiate version, etc.

• Conveying the local player's actions (received from the Controller) remotely

This allocation still gives many responsibilities to each of these objects, but the question is how to offload these responsibilities to other objects without making the design even more complex. There are different ways to subdivide the communication, such as

• "lobby communication" that takes place before a match starts (represented by the top row in the above storyboard), and

• "in-game communication" during the match (represented by the middle row in the above storyboard).

However, it is not clear what structural improvement is gained by such division.

For example, if Communicator were to be divided, then there would be an overhead of multiple objects communicating with each other to accomplish what is not accomplished by Communicator alone.

On the other hand, we may consider splitting the Controller into the part that handles the local player's moves and the part that displays the remote player's moves. This intervention is better motivated as part of introducing the Model-View-Controller design pattern, as explained later.

One of the problematic aspects of the given design is it being based on the central-repository architectural style. The Communication must periodically poll the database for updates relevant to the local player, which may be inefficient. In addition, because of a finite interval between polls, messages may be delivered with a delay. To minimize the response latency, each client would need to poll the database very frequently, say twice per second. If there are many users in the system, this frequent polling will introduce high load on the database server and will require a powerful computer to cope. Finally, polling when noting new happens wastes network bandwidth.

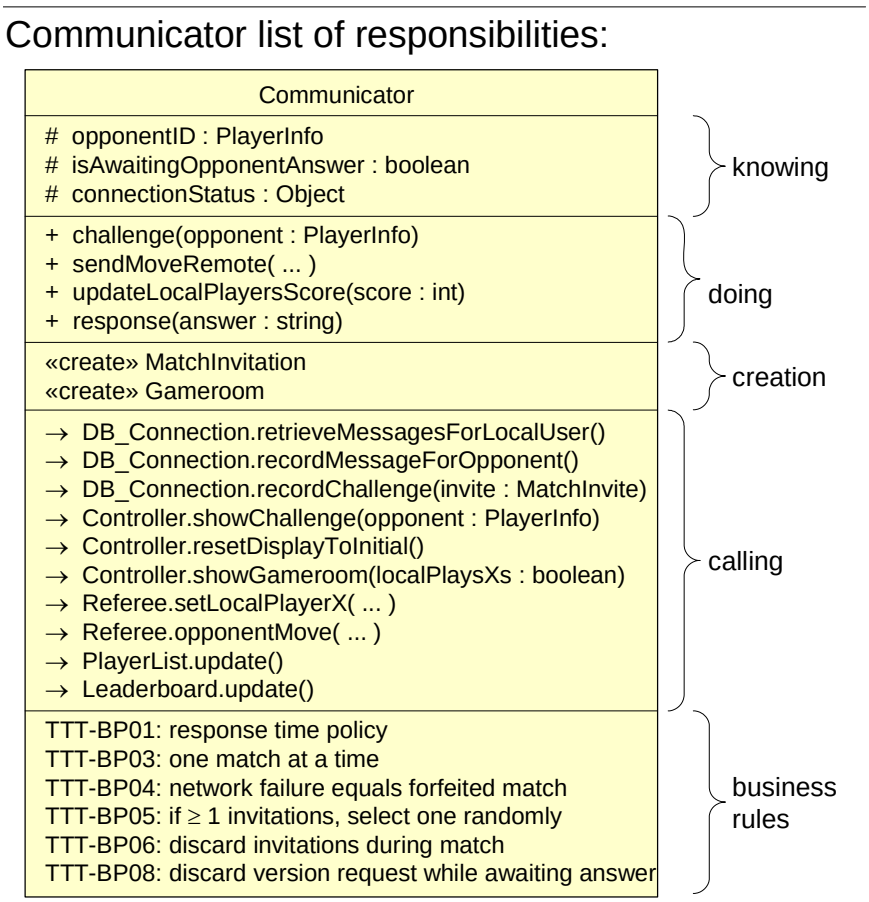


Figure G-20: List of responsibilities for the Communicator class.

An alternative to central repository is to have the Client-Server architectural style, where the clients ("game consoles") connect to the server (Figure G-19). Every time a new player logs in the system, the server would notify all players already logged in about the new player. Therefore, server "pushes" or "publishes" the relevant information instead of having the clients to "pull" or "poll" the database for information. Similarly, other relevant information could be published to clients. This design can be considered a *distributed* Publish-Subscribe. Different player groups (in the case of tic-tac-toe it is player pairs) will be organized as different publishers and subscribers.

This architectural style avoids polling the database. However, now we need a game server that is always running and awaiting new clients to connect. Another architectural style that the reader might wish to consider is Peer-to-Peer, where the player clients directly communicate with each other, without server mediation.

Based on design sequence diagrams from the first iteration (Section G.6.1), we can compile the lists of responsibilities for key objects in the system. As shown in Figure G-20, Communicator has large number of methods (doing responsibilities that let other objects tell it what to do) and even large number of calling responsibilities, to tell other objects what to do. The former characteristic may indicate low cohesion. The latter characteristic indicates high coupling.

The following table summarizes the responsibilities of different objects in our preliminary design:

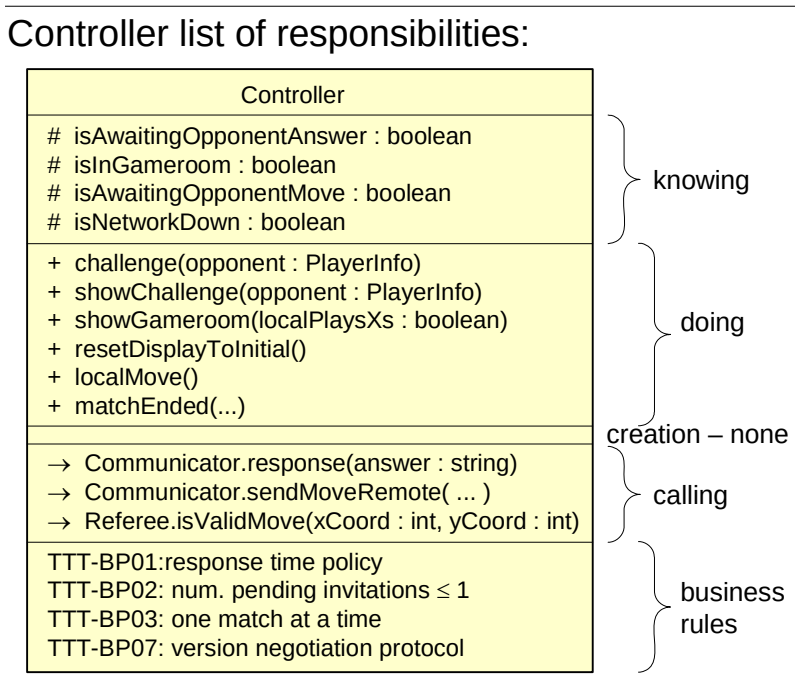


Figure G-21: List of responsibilities for the Controller class.

Responsibility type	Controller	Communicator	Referee	Gameroom	DB Connection	Player List	Invite Queue	Match Invitation	Leader board
Knowing	4	3							
Doing	6	4							
Creation	–	2							
Calling	3	10							
Business policies	4	6							
TOTAL	17	25							

As seen, Communicator is assigned disproportionally large number of responsibilities.

Some smaller issues with the initial design include:

● The Gameroom has the attribute matchStatus, but the Referee is given a method checkIfWins() which whether a move wins and returns a Boolean value isOver (see Figure G-17 and Figure G-18). It appears that the responsibility of computing and memorizing the match status is spread across two different objects (Gameroom and Referee), which indicates poor cohesion of these objects.

● The Communicator is assigned all functions related to communicating with the remote player, except one, which is saving the score of the local player after a match is finished. See Figure G-17 where the Referee calls `updateLocalPlayersScore()` on Communicator. One may argue that Communicator should take all responsibilities for database access. However, in this case it is not clear that there is any advantage of passing this information through Communicator instead of having the Referee directly call DB Connection. The advantage of the latter solution would be that it increases Communicator's cohesion in the sense that it would then deal only with

communicating with the remote player. This is particularly important if in the future the system will be extended with new features that would make the database interaction more complex.

● In Figure G-17, when the local player moves a piece this action is immediately processed and, if valid, forwarded to the opponent. An alternative is to allow the player to "preview" his or her planned move. Only when the player confirms the move, e.g., by clicking the button "Apply" would the move be committed and sent to the opponent.

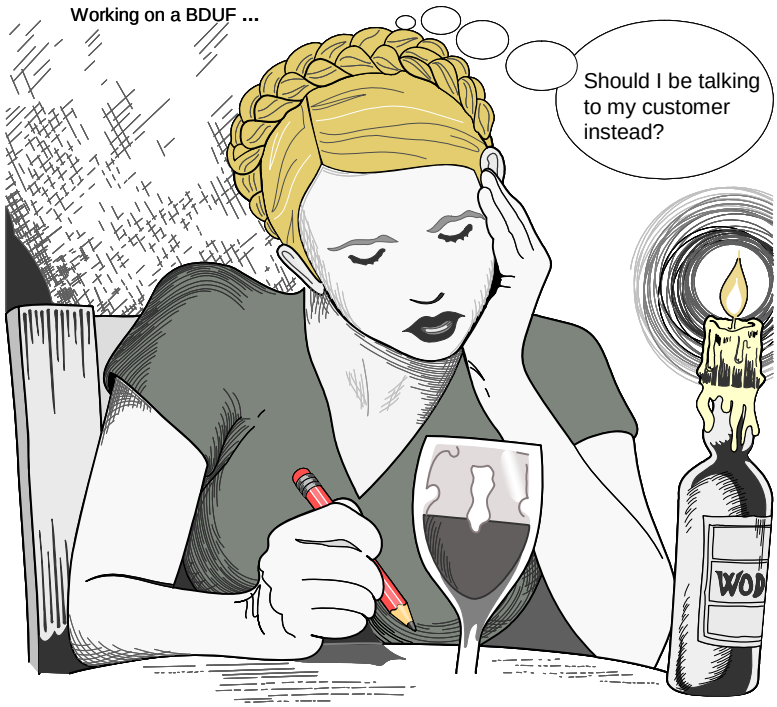
G.7 Class Diagram and Interface Specification

_____ TO BE COMPLETED _____

How We Did It & Plan of Work

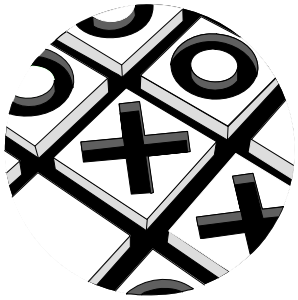
We underestimated how long it would take to create the system designs. To add to the frustration, the UML diagramming software used to generate the figures in the report. was very unwieldy.

Working with a team has been a unique experience. We learned about weaknesses and strength of certain team members and how to create an efficient distribution of work. The benefits from working in a group include the ability to bounce ideas back and forth to create a larger, more coherent idea and knowing there is someone to help you when you don't know what to do or are stuck with a part of the project along with helping others.



Myself says: A technique that was useless to our group was the concept of project estimation, so we often failed to meet certain planned milestones. We simply accomplished more in a short span to compensate for failing to meet

a certain milestone at a date. However, I understand the use of milestones is very important in the workplace as well as deadlines. Many products end up sacrificing quality when having to rush to make up what they failed to accomplish within a certain time frame.



G.8 Unit Tests and Coverage

G.8.1 Deriving the Object States

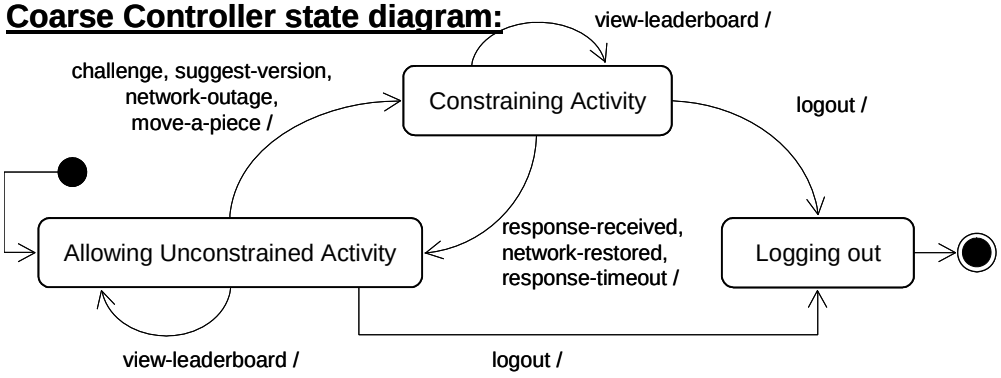
To derive the object states, recall that an object *state* is defined as constraints on the values of object's attributes. From Figure G-11, we see that there are three objects with "doing" responsibilities in our system: Controller, Communicator, and Referee. We need to determine their states because objects with "knowing" responsibilities are essentially passive information containers. As such, they are unlikely to contain conditional logic statements; they will most likely contain simple accessor methods for getting or setting attribute values. A possible exception is the Gameroom object, but we will not consider it for now. Objects DB Connection and Interface are «boundary» objects that interact with external actors and will need to be tested when the external actors will be available, during design and implementation.

Consider the attributes of the Controller, which are concerned with constraining the user's actions while awaiting the opponent's response. When awaiting the opponent's response, the system should disallow all actions except viewing the leaderboard or logout. The Controller essentially needs to implement the operational model shown in Figure G-6. We define these states of the Controller:

Allowing Unconstrained Activity — user can perform any action allowed in the given context

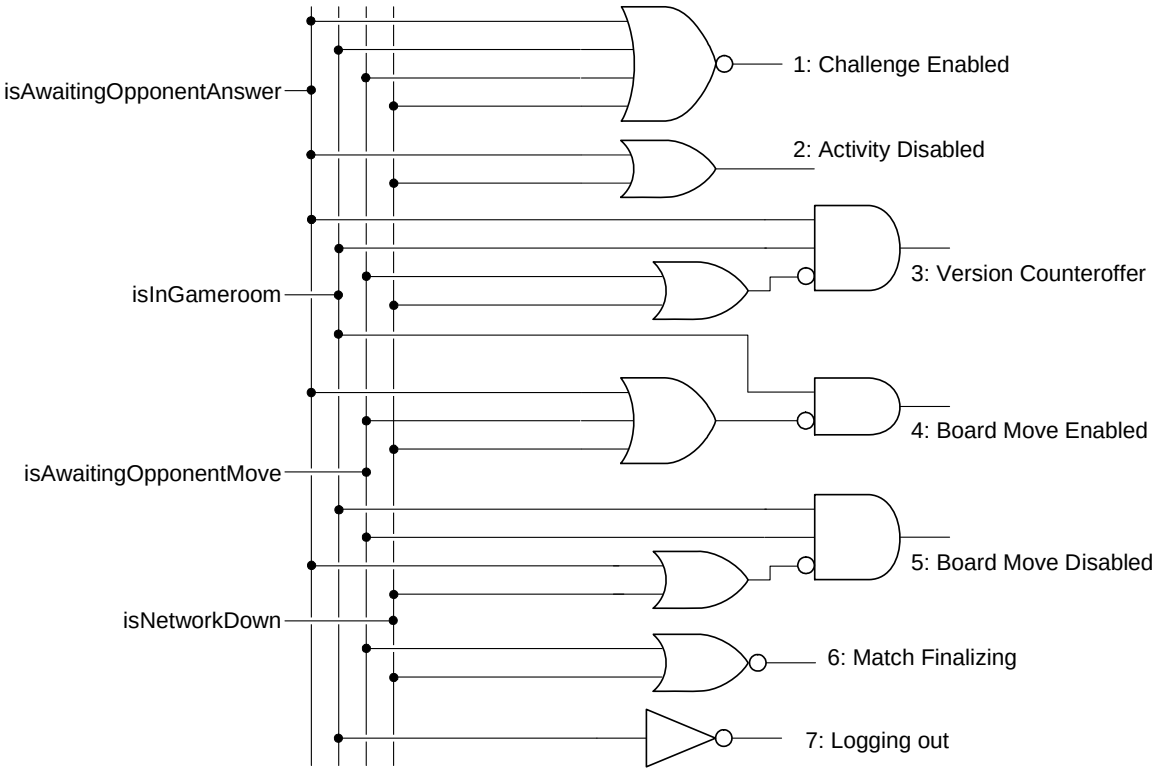
Constraining Activity — user's actions are constrained

Coarse Controller state diagram:



We realize that it is difficult or impossible to define precisely the above coarse states by attribute constraints. For example, how to know when the setup is completed and show the game board? How to know that players do not wish to negotiate a different game version from the default one?

Given these difficulties, we decide that we need a refined state diagram for the Controller. Because the Controller attributes are all Boolean variable, we can represent the states using this logical circuit (also see Figure G-22):



Note that we assume that if the user is not in the gameroom, he or she cannot be awaiting opponent's move, so we do not specify this attribute in such states. Similarly, if isAwaitingOpponentMove is true, then we do not need to ask if isInGameroom. As hinted in Section G.5.1, here we realize that the attribute isInGameroom is necessary and the different stages of the game setup could not be distinguished without introducing this attribute.

The above logical circuit is somewhat insufficient for representing the Controller states in the sense that state 7: *Logging out* overlaps with several other states in attribute values, such as states 1: *Challenge Enabled* and 6: *Match Finalizing*. The difference is that while 7: *Logging out* the local system is blocked for user input and will shut down after a needed "housekeeping."

The Controller state machine diagram is shown in Figure G-22. Note the diamond-shaped *choice pseudostate* on the transition from state 2: *Activity Disabled* for the "response-received" event. To avoid clutter, we use the same pseudostate for the transition from state 1: *Challenge Enabled* upon receiving an invitation/challenge. This pseudostate is part of UML notation used to emphasize that a Boolean condition determines which transition is followed. In our case, the Controller transitions to state 4: *Board Move Enabled* if the local player is assigned **X**s; otherwise, the local player is assigned **O**s and the Controller transitions to state 5: *Board Move Disabled*. The Controller states are defined in the following table:

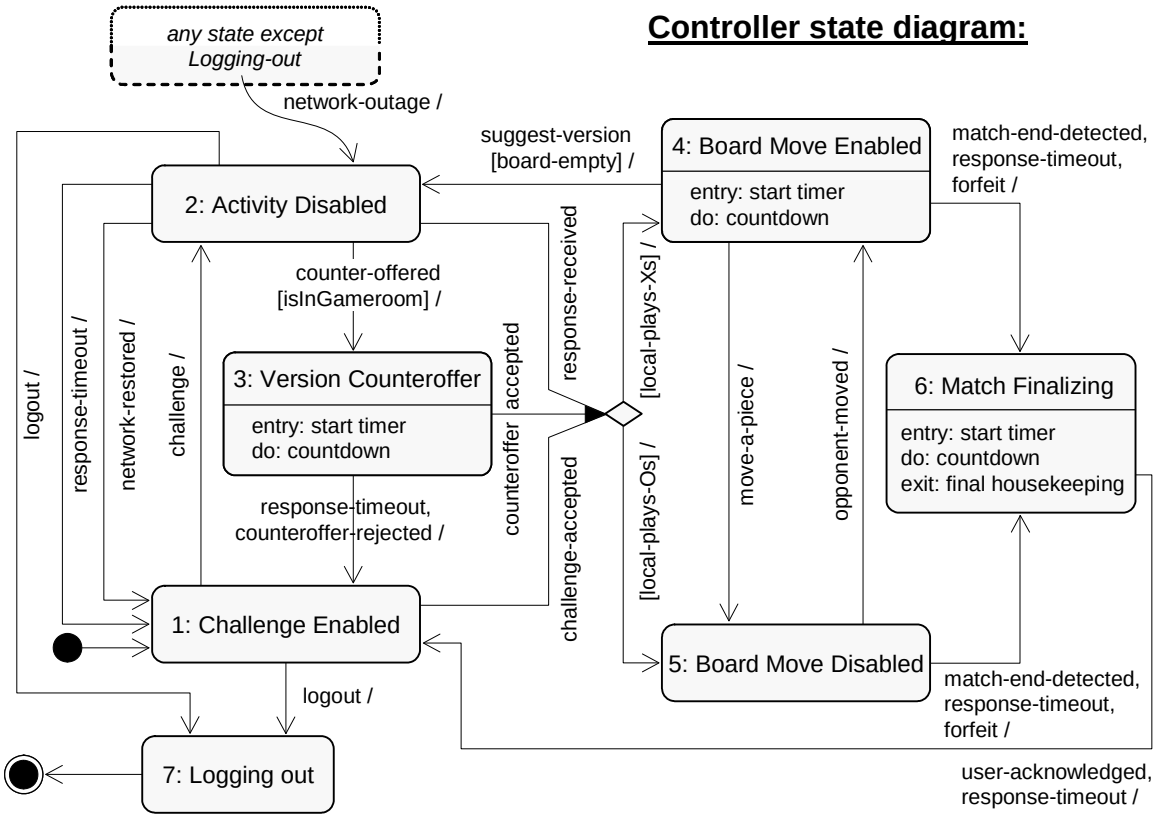


Figure G-22: State machine diagram for the Controller class of the game of tic-tac-toe. Note that transition actions are omitted, but at some point will need to be specified.

State of Controller	Definition
1: Challenge Enabled	NOT (isAwaitingOpponentAnswer OR isInGameroom OR isAwaitingOpponentMove OR isNetworkDown)
Description: This is the initial state: the user is allowed only to challenge an opponent, view leaderboard, or logout; the user may also passively wait to receive a challenge from a remote user	
2: Activity Disabled	isAwaitingOpponentAnswer OR isNetworkDown
Description: During the gameroom setup, the user enters this state after challenging an opponent or after suggesting a different game version	
3: Version Counteroffer	isInGameroom AND isAwaitingOpponentAnswer AND NOT (isAwaitingOpponentMove OR isNetworkDown)
Description: While awaiting an answer to a game version offer, the local user receives a different version counteroffer	
4: Board Move Enabled	isInGameroom AND NOT (isAwaitingOpponentAnswer OR isAwaitingOpponentMove OR isNetworkDown)
Description: In the gameroom, the user is allowed to move a pieces on the board or to suggest a different game version, provided that the guard condition is met (no move has been made and the board is still empty); once the match starts, the game version cannot be changed	
5: Board Move Disabled	isInGameroom AND isAwaitingOpponentMove AND NOT (isAwaitingOpponentAnswer OR isNetworkDown)
Description: In the gameroom, the user has made a move and is awaiting the opponent's move.	
6: Match Finalizing	NOT (isInGameroom OR isNetworkDown)

Description: The system detected a match end (either "win" or "draw") or one of the players forfeited the match; the system signals the match end and waits for player's acknowledgement, erases the screen, and closes the gameroom and brings players back to the main screen; network connection is required to store the updated scores for both players in the database	
7: Logging out	NOT isInGameroom
Description: Logout is simple because any scores from played matches would have been already stored	

In Figure G-22, the event "challenge-accepted" can occur in state 1: *Challenge Enabled* if the local user accepts a challenge from a remote user, and in state 2: *Activity Disabled* if a remote opponent accepts a challenge by the local user. Both events transition the Controller to state 4 or state 5, depending on whether the local user is assigned **X**s or **O**s, respectively. This decision is indicated by the guard conditions emanating from the diamond-shaped choice pseudostate.

In state 3: *Version Counteroffer* and state 4: *Board Move Enabled*, the system is timing the local user for response, which may appear redundant because the remote opponent's Communicator will also time our user's response. The reason for doing this is to know when the remote system will detect response timeout and declare the opponent winner, so that the local system can automatically close the gameroom and default to state 1: *Challenge Enabled*. On the other hand, the response-timeout event in state 2: *Activity Disabled* and state 5: *Board Move Disabled* will be generated by the local Communicator, which is timing the remote opponent's response.

Note that only the **X**s player can start game-version negotiation, because the transition labeled "suggest-version [board-empty] /" emanates from state 4: *Board Move Enabled*. We make this choice to avoid further complexity in the system. For example, if we allowed the **O**s player to suggest a version, the following scenario could occur. Assume that the local user is assigned **X**s and he makes a move, but at the same time, the opponent (being assigned **O**s and awaiting a move) suggests a different version. A version offer will arrive while the **X**s player is awaiting the **O**s move and then we need to decide how to handle such a scenario. Instead, we do not allow the **O**s player to make version offer, but he can still make a counteroffer after receiving a version suggestion. For this purpose, we need slightly to modify the business rule TTT-BP07 defined in Section G.3.4.

At this point, it becomes clear that the Controller is very complex and may need to be split into several objects. One may suspect that it is because of being assigned too many responsibilities, but this was not apparent in Section G.5.1, when the responsibilities were assigned. For now, we leave this issue aside, but we keep in mind that the Controller will need special attention.

The Communicator deals with communication with other players. It starts in the Monitoring state, where is monitoring the network health and retrieving other information of interest from the database, such as the latest player availability list and the leaderboard.

The Communicator states are defined as follows (see Figure G-23):

Communicator State	**Definition**
1: Monitoring	NOT isAwaitingOpponentAnswer
Description: The initial and default state of monitoring the network health and relevant database updates	
2: Waiting For Response	isAwaitingOpponentAnswer AND NOT (connectionStatus = "disconnected")
Description: Waiting for response from the opponent; response timer counting down	

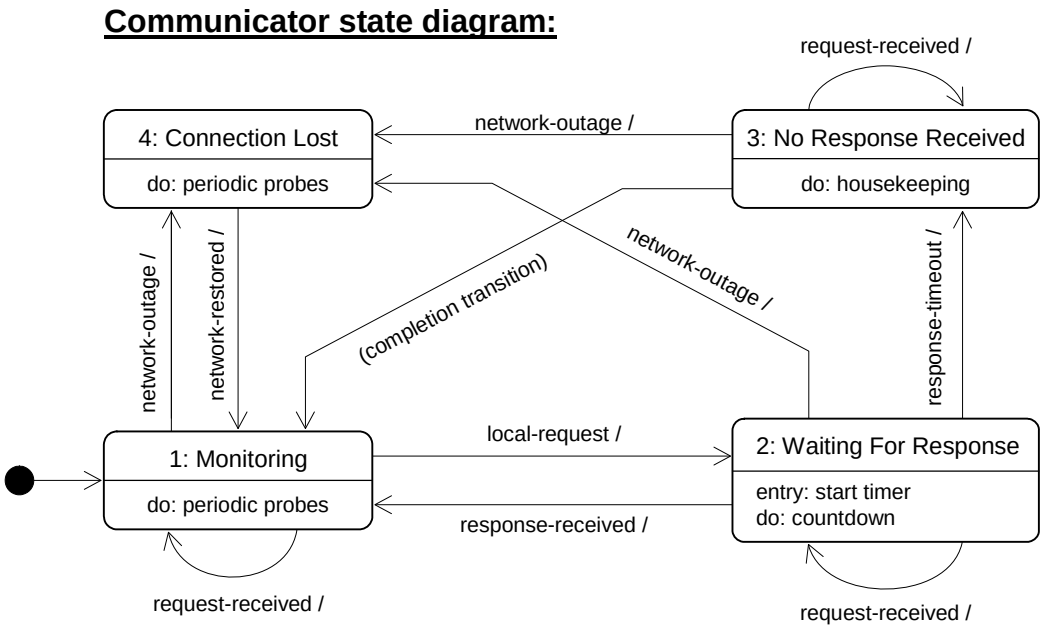


Figure G-23: State diagram for the Communicator class of the game of tic-tac-toe.

3: No Response Received	isAwaitingOpponentAnswer AND NOT (connectionStatus = "disconnected")
Description: Response timer timed out before receiving the opponent's response; do the necessary "housekeeping" and upon completion transition to the default monitoring state	
4: Connection Lost	connectionStatus = "disconnected"
Description: Logout is simple because any scores from played matches would have been already stored	

Although we assume that state 1: *Monitoring* is the initial state, the network connection may be already down at the time when the user logged in. In this case, the system will immediately transition to state 4: *Connection Lost*.

State 2: *Waiting For Response* and state 3: *No Response Received* are indistinguishable in terms of attribute values. The difference is that in state 2: *Waiting For Response* there is also response timer counting down.

Note that one might consider introducing a state of Communicator for situations when it receives a remote request and the local user is expected to respond within a response timeout. However, because this is already responsibility of the opponent's Communicator, we decide against duplicating the responsibilities. Remote requests are shown explicitly as self-transitions on all states in Figure G-23 (except for state 4: *Connection Lost*), because there are actions associated with these events that must be performed. We assume that the Communicator will not deal with local requests in state 4: *Connection Lost* because the Controller will know that the network is down and will not issue requests to the Communicator.

The Referee essentially makes three types of decisions:

- Is the local player next to move?
- Is the local player's last move valid?

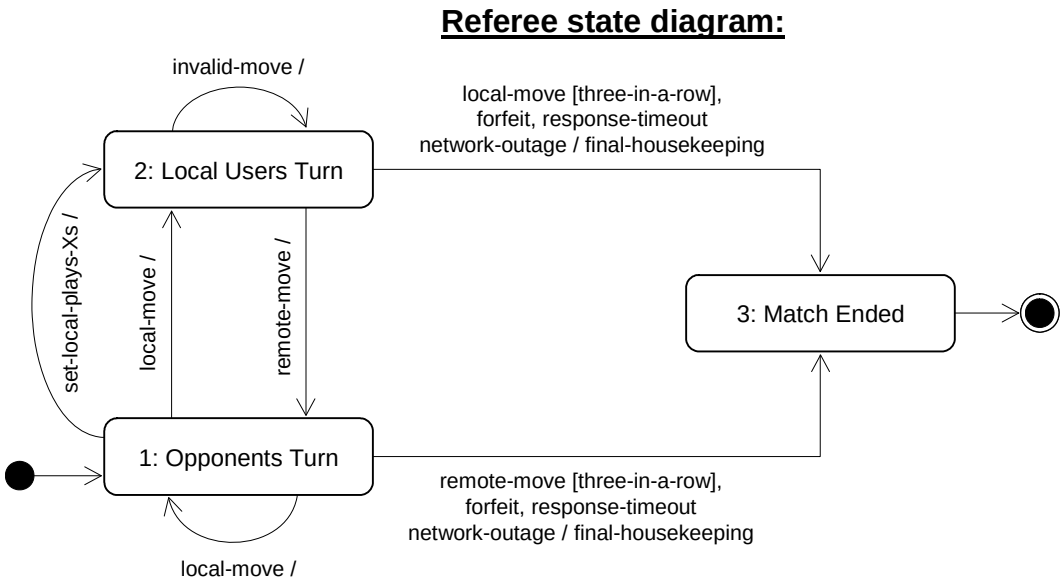


Figure G-24: State diagram for the Referee class of the game of tic-tac-toe.

• Is the match finished because three-in-a-row or some policy invocation?

The turn decision is decided for the first move based on whether the local player is assigned **X**s or **O**s; for subsequent moves, the players alternate by turns.

The validity decision is based on the rules of the game of tic-tac-toe. If the move is invalid, the player is given another chance to move.

When match end is detected, no additional moves are allowed (unless it is the revenge version of tic-tac-toe), and the system needs to do some "housekeeping" activities before closing the gameroom. The Referee state diagram is shown in Figure G-24. The initial state is 1: *Opponent's Turn*, because the attribute isLocalPlayerX is by default initially set as FALSE. This attribute may be set to TRUE by random assignment of **X**s or **O**s at the start of a match and does not change the value during a match; therefore, it is not considered for defining the Referee sates. Recall that in Section G.5.1 we decided that responsibility R8 in Table G-3 (randomly assigning **X**s and **O**s) is performed by the Communicator, when an opponent is challenged.

The Referee states are defined as follows (see Figure G-24):

State of Referee	Definition
1: Opponent's Turn	NOT isLocalPlayerNext
Description: The opponent is allowed to move a piece on the board; the local user is blocked	
2: Local User's Turn	NOT isLocalPlayerNext
Description: The local user is allowed to move a piece on the board; the opponent is presumably blocked	
3: Match Ended	gameroom reference equals nil
Description: Terminal state of a match; the gameroom is closed and Referee is waiting for a new match	

Note that Referee's state 3: *Match Ended* (Figure G-24) is linked to Controller's state 6: *Match Finalizing* (Figure G-22), and in most cases the Referee will cause the event that will make the Controller transition into its state 6.

G.8.2 Events and State Transitions

The events and legal transitions between the states of several objects are shown in Figure G-22 to Figure G-24. By reading these figures, one can see that only certain state sequences are possible.

Legal state sequences for the Referee (Figure G-24):

```
1, 3                    (local user is by default Os player)
{1, 2}, 3
1, {2, 1}, 3
1, 2, 3                 (local user is set as Xs player)
1, {2, 1}, 3
1, 2, {1, 2}, 3
```

where the curly braces symbolize an arbitrary number of repetitions of the enclosed states. For example, the second line says that a sequence of Referee states such that an arbitrary number of repeated transitions from state 1: *Opponent's Turn* to state 2: *Local User's Turn*, back to state 1, etc., ending with state 3: *Match Ended* is legal for the Referee.

Legal state sequences for the Communicator (Figure G-23):

```
1, 4, 1, ...
1, 2, 1, ...
1, 2, 3, 1, ...
1, 2, 4, 1, ...
1, 2, 3, 4, 1, ...
```

Because the Communicator does not have a terminal state, none of the sequences is finished.

Legal state sequences for the Controller (Figure G-22) are a bit more complicated to determine. We start by defining the following state sub-sequences:

```
A:  {1,2}       local player unsuccessfully challenges different opponents
B:  A,4         an opponent accepts and local player is assigned Xs
C:  A,5         an opponent accepts and local player is assigned Os
D:  B,2         local player is assigned Xs and players are negotiating game version
E:  D,3,4       different game version agreed, local player is Xs
F:  D,3,5       different game version agreed, local player is Os
G:  {4,5}       sequence of board moves, starting with the local player
H:  {5,4}       sequence of board moves, starting with the opponent
I:  6,1         match finalizing
J:  4,I         match finished after the local player moved
K:  5,I         match finished after the opponent moved
```

We use the above sub-sequences to compose the following composite sequences:

```
X:  A,1 | D,1 | D,3,1 | W,I | B,K | C,J | B,H,K | C,G,J
W:  B | B,H | C | C,G | E | E,H | F | F,G
Y:  W | W,6 | D | Z | Z,6
Z:  B,5 | C,4 | B,H,5 | C,G,4
```

where the vertical line | symbolizes the "or" operation. "X" represents all legal sequences that lead to state 1, after visiting at least one other state. "W" represents all legal sequences that lead to state 4: *Board Move Enabled*, where it is the local player's turn. "Y" represents all states in which a network failure may occur and the Controller will next transition to state 2: *Activity Disabled*.

Finally, legal state sequences for the Controller are as follows:

```
1,7 | A,7 | X,7 | X,2,7 | Y,2,7 | Y,2,X,7 | Y,2,X,2,7
```

G.8.3 Unit Tests for States

Ideally, the unit tests should check that the object exhibits only the legal sequences of states and not the illegal state sequences. This may be feasible for simple state machines, such as that of the Referee (Figure G-24). However, the Controller has potentially infinite number of both legal and illegal state sequences, as seen in Section G.8.2. This is why we take a practical approach of covering all states at least once and all valid transitions at least once. We start by writing unit tests to cover all identified states at least once (i.e., each state is reached in at least one test case).

In Section G.8.1, we decided that objects with "knowing" responsibilities have trivial states because these objects are unlikely to contain conditional logic statements. Testing these objects is simple by calling their accessor methods for getting or setting attribute values. Objects DB Connection and Interface are «boundary» objects that interact with external actors and their testing plan will be made when their actors will be available, during design and implementation. Based on Figure G-11, we will describe the plan for unit testing of the three objects with "doing" responsibilities: Controller, Communicator, and Referee.

We start with the Referee because it has the simplest state machine (Figure G-24) and has a dependency (or, association) only with the Controller and Communicator (see Section G.5.1). By examining Referee's and Controller's responsibilities, we conclude that the Referee will likely be called by the Controller, not the other way around. The Referee responsibilities include: "Referee asks Communicator to update the local player's score if he won a match." Therefore, we will need only one stub for the Referee unit testing: the Communicator Stub. We examine the legal state sequences in Section G.8.2 to determine which sequences will cover all identified states at least once. The simplest sequence is for the case when the local user is set as **X**s player: `1, 2, 3`, which covers all Referee's states.

Assuming that we will be using the Java programming language and JUnit as our test framework, here is a pseudocode for a test case that checks if the Referee correctly transitions to state 2: *Local User's Turn* when the local user is assigned to play **X**s.

Listing G-1: Example test case for the Referee class.

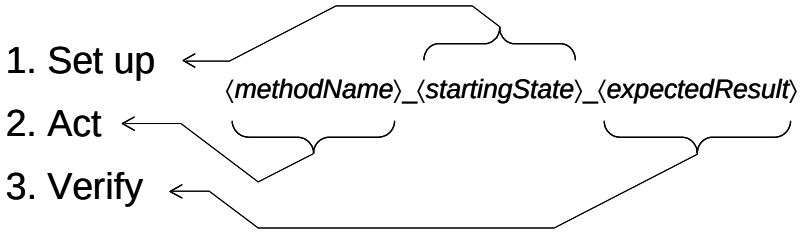
```
public class RefereeTest {
  // test case to check that state 2 is visited
  @Test public void
    setLocalPlayerX_opponentsTurn_toLocalUsersTurn() {
```

```
    // 1. set up
    Referee testReferee = new Referee( /* constructor params */ );

    // 2. act
    testReferee.setLocalPlayerX(true);

    // 3. verify
    assertEqual(testReferee.isLocalPlayerX(), true);
    assertEqual(testReferee.isLocalPlayerNext(), true);
  }
}
```

Recall the notation for the test case method in Listing G-1 (see Section 2.7.3):



In Listing G-1, the tested method is the Referee method `setLocalPlayerX()`, the object is starting in state 1: *Opponent's Turn*, and the expected result is that the Referee will transition to state 2: *Local User's Turn*. Thus the test case method name `setLocalPlayerX_opponentsTurn_toLocalUsersTurn()`.

State transitions occur because of events. Events are conveyed to objects by calling their methods. This is why we need to design unit test cases to test if methods are causing proper state transitions. We know that methods will be derived at the design stage, so at this point we will guess what methods will need to do and given them names. The Referee will be told when the players make moves and its key responsibilities stated in Section G.5.1 (Table Table G-3) are: prevent invalid moves, detect match ending ("win" or "draw"), and apply the RESPONSE TIME POLICY. Based on the design of interaction diagrams (Sections G.6 through G.???) and, based on this design, we know that the Referee will need the following methods:

```
public interface Referee {
    // sets attributes isLocalPlayerX and isLocalPlayerNext
    public void setLocalPlayerX(boolean localPlaysXs);

    // arbitrates local player's move
    public boolean isValidMove(int xCoordinate, int yCoordinate);

    // notifies about opponent's move; sets timer for local user
```

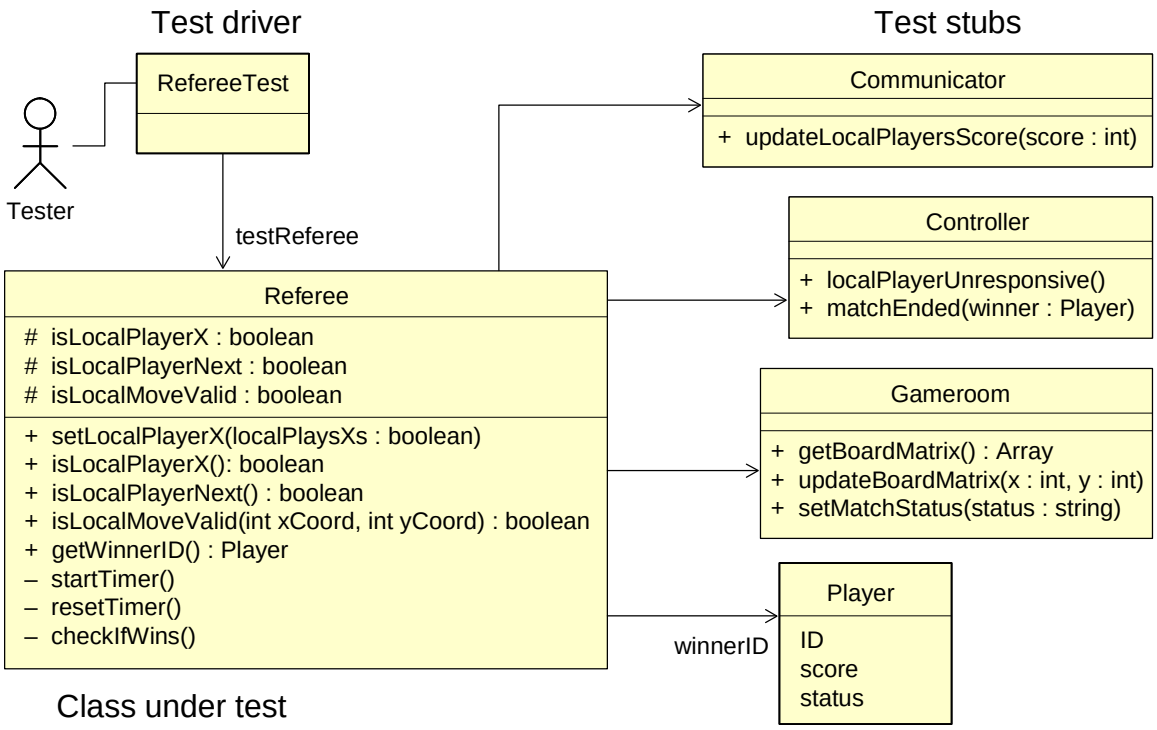


Figure G-25: Test driver and stubs for testing the Referee class of the game of tic-tac-toe.

```
    public void opponentMove(int xCoordinate, int yCoordinate);
}
```

Here we realize that we missed one association in Figure G-12: the Referee will retrieve the board state from the Gameroom to check for valid moves and update it accordingly. Another missed association is between the Referee and the Player, because the Referee uses the Player object argument to invoke the Controller's method `matchEnded()`, as seen below.

In the method `opponentMove()`, the Referee implementation will use a system timer to time the local player's response. If the local player fails to respond within the response time limit, he loses the match, the (local) gameroom is closed and the player is brought to the initial screen.

The Referee will need to call the following methods on the Controller:

```
public interface Controller {
    // notification that local player did not respond in time
    public void localPlayerUnresponsive();

    // notification that the match ended and who won, if any
    public void matchEnded(Player winner);
}
```

The Referee will need to call the following methods on the Communicator:

```
public interface Communicator {
    // notification that local player did not respond
    public void updateLocalPlayersScore(int score);
}
```

The complete arrangement for testing the Referee class is shown in Figure G-25. Listing G-2 shows two test cases that cover all three states of the Referee. For the second test case, we prepare the game board so that the next move of the local player (assuming she plays **X**s) will result in a three-in-a-row board configuration.

Listing G-2: Test cases for the Referee class.

```
public class RefereeTest {
 // test case checks that state 2 is visited (copied from Listing G-1)
  @Test public void
    setLocalPlayerX_opponentsTurn_toLocalUsersTurn() {

    // 1. set up
    Referee testReferee = new Referee( /* constructor params */ );

    // 2. act
    testReferee.setLocalPlayerX(true);

    // 3. verify
    assertEqual(testReferee.isLocalPlayerX(), true);
    assertEqual(testReferee.isLocalPlayerNext(), true);
  }

  // test case to check that state 3 is visited from state 2
  @Test public void
    isLocalMoveValid_localUsersTurn_matchEnded() {

    // 1. set up
    Player localPlayer = new Player( ... );
    Gameroom gameroomStub = new Gameroom( ... );
    Referee testReferee = new Referee(localPlayer, gameroomStub, ...);
    testReferee.setLocalPlayerX(true);
    gameroomStub.updateBoardMatrix(0, 0, "X");
    gameroomStub.updateBoardMatrix(0, 1, "O");
    gameroomStub.updateBoardMatrix(1, 1, "X");
    gameroomStub.updateBoardMatrix(1, 0, "O");

    // 2. act
    boolean ok = testReferee.isLocalMoveValid(2, 2);

    // 3. verify
    assertEqual(ok, true);
    assertEqual(testReferee.isLocalPlayerX(), true);
    assertEqual(testReferee.isLocalPlayerNext(), false);
    assertEqual(testReferee.getWinnerID(), localPlayer);
  }
}
```

I leave it to the reader to write the unit test cases for the Controller and Communicator.

G.8.4 Unit Tests for Valid Transitions

Here we need to write the unit tests to cover all valid transitions at least once. Figure G-24 in Section G.8.1 shows that there are seven valid transitions between the Referee's three states. We need to design test cases that will cause the Referee to cover all seven transitions.

How We Did It & Plan of Work

We had already written about 100 pages of documentation on exactly what our program was going to accomplish and exactly how it was going to do each task. But because the programming languages were new to us, we didn't always follow the plans that we wrote.

Me says: Essentially, by far the most difficult and frustrating aspect was keeping everyone on track and working together. This heavily restricted the amount and degree of work that our group produced. Even at this point, there is still coding that people were supposed to do that has not been shared with me so it is hard to determine what they will decide to implement because it never seems to be what we have agreed on. I know that situations can always be worse but I really feel that at almost every point, I faced an undue amount of frustrating uncertainty about the state of our work.

Myself says: The second report rolled around and we knew we had to get things done. Once again, Irene and I started sending emails about starting the report. We started the report and once again Me was like a ghost. He disappeared and reappeared towards the very end of the submission process, going crazy and fixing up a number of different things as well as adding additional diagrams and text to the report. Once again, good, but very annoying timing and not convenient when Irene and I would like to get things ahead of time.

Me says: I didn't feel as though any of the concepts learned were not helpful in advancing my knowledge of software development because they were all useful in the production and logical creation of software. Although this may be true, I found that many of these software engineering concepts are tedious and take a lot of time. Some of them take more time than they are worth in my opinion. For example, OCL Contracts are useful in showing all of the preconditions, postconditions, and invariants; however, they take a lot of time to enumerate and write, simply when code for the software can be written with these invariants in mind. In a sense, there are other principles that are more useful that also provide this information.

We realized that our project required a strong understanding of network protocols, which at the start of the semester none of us was familiar with and had to quickly learn. Personally, the biggest challenge for Myself was learning to code with multiple new languages in such a short span of time.

There wasn't always an opportunity for every group member to contribute evenly and when it came to spreading the work equally or giving it to the fastest coder, we often chose the latter route.

Myself says: Me is a very advanced programmer that knows a wide variety of programming languages and started to get even more cocky (like this much wasn't enough!) when the time

came to implement the project. He was, obviously, was very excited to get working on things related to the demo because he could show off his programming skills.

Me says: Irene and I spent time in the computer lab working on the demo. Myself showed up at the lab and sat there playing his Playstation portable while Irene and I were coding. Myself eventually left after doing nothing, and later reappeared to "check in on us" and just sat there for a couple minutes, observing what we were doing and left… effectively doing nothing.

We expected everything would work well together when put together. But, when we put all components together nothing worked, it was also due to our lack of experience on building large software. We didn't write out the exact detailed specification initially for each part. So each member would have a different way on implementing each part, then none would work at all together. Many of the topics covered in this class about specific programming methods I had already covered in previous classes. Nonetheless, there were a few topics on the actual design of the project that aided us greatly with our approaches. Concepts such as low cohesion and loose coupling gave us insight about the design of various classes within our program. This in turn made our program operate in a very organized and easy to analyze, (and therefore debug), manner.

G.9 Refactoring to Design Patterns

Developing a distributed game of tic-tac-toe may appear relatively simple, so the design presented in Section G.6 may be considered adequate and the use of patterns may seem to complicate the design unnecessarily. However, we must keep in mind that the given design is quite incomplete and many basic functions are unfinished. In addition, it does not support different variants of the game of tic-tac-toe. Therefore, although the given design is simple, we do not know how complex it will be when completed. Therefore, we consider the merits of employing different design patterns.

G.9.1 Roadmap for Applying Design Patterns

This section explains our plan for applying design patterns to improve the current system design.

G.9.2 Remote Proxy Design Pattern

One way to think about the Communicator is that it is a *remote proxy* for the Controller object of the remote player. In this way, the local Controller (and other local objects, such as Referee) has an illusion of exchanging messages directly with the remote Controller by interacting with the remote Controller's proxy: the local Communicator.

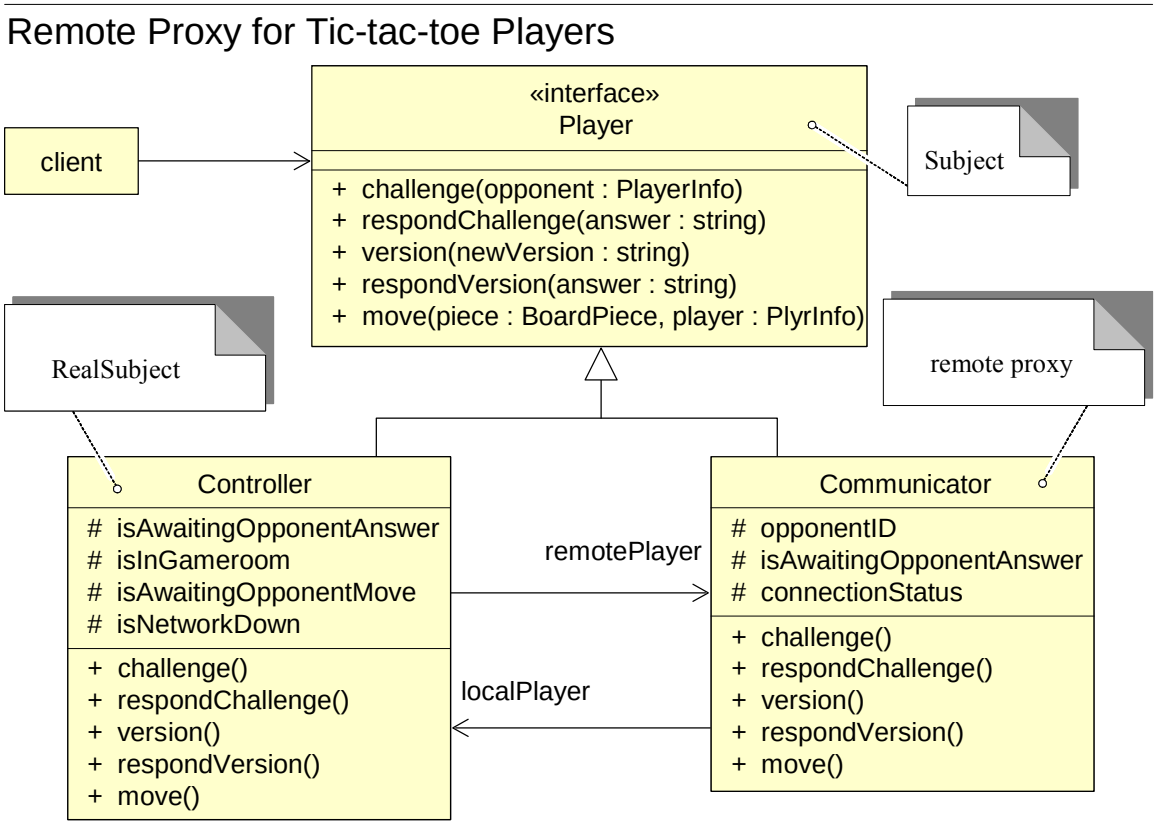
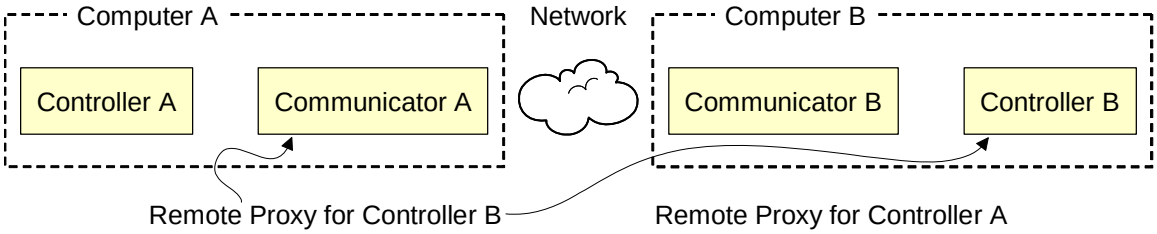


Figure G-26: Communicator and Controller classes implement the same interface (Player) that abstracts the common behaviors of a player, regardless of whether he is local or remote.



To make the remote proxy pattern more prominent, we redesign the Communicator's interface, so that Controller and Communicator implement the same interface. We obtain the class diagram shown in Figure G-26. Here, the Player interface should not be confused with PlayerInfo (Figure G-12), which is just a passive container of player information. The local player is represented by the Controller and the remote player is represented by its proxy, the Communicator. The rationale for the choice of methods in player interface will become apparent later, when we introduce the State design pattern. These methods correspond to the events that are delivered to the players.

As shown in Figure G-26, both Controller and Communicator maintain references to each other (named remotePlayer and localPlayer, respectively). Recall from Section 5.4 that the real subject and remote proxy (stub) need references to each other to allow method calls.

The client class for the Controller is a user interface class. The client class for the Communicator is a class that will subscribe to database events and dispatch the relevant updates directly to Communicator, Player List, or Leaderboard.

G.9.3 Publish-Subscribe Design Pattern

Above we already considered a distributed Publish-Subscribe pattern to improve the system responsiveness.

Referee could act as subscriber for board moves from two different publishers: Controller and Communicator. However, the Referee arbitrates only the local move because the remote Referee already arbitrated its player's move and let it be sent to this client. Therefore, the Referee could check the source of the notification.

On the other hand, the Referee is the source of events after arbitrating the local move—this outcome is of interest to both Controller and Communicator (and possibly Leaderboard, too). It is not a good idea to have interleaved publishers and subscribers to communicate by publishing and subscribing to each other.

G.9.4 Command Design Pattern

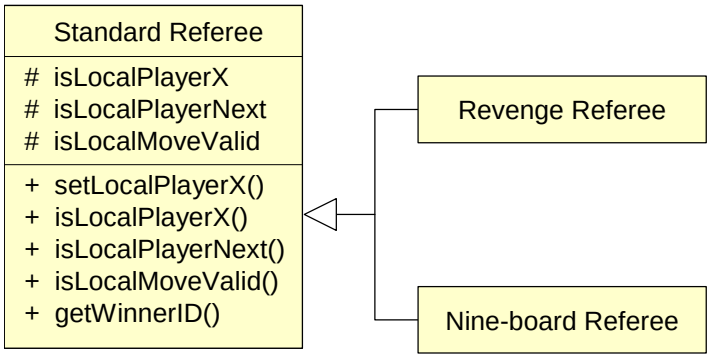
A key action in this system that may be considered for the Command pattern is to update the game board. Command helps to articulate processing requests and encapsulates any pre-processing needed before the method request is made.

The Command pattern may be more broadly useful if we decide to use a Web-based architectural pattern. In this case the browser-based client will initiate a servlet to call its method `service()` to process the client's request. See Section 5.2.1 for more details.

One may also wish to include the undo/redo capability, so the player can undo an accidental move. However, we should keep in mind that undo/redo in a distributed system is much more complex to support than in a standalone system. For example, once the local player made a move this information is sent to the remote opponent who will be allowed to make his move. If meanwhile the local player performs undo of the last move, then this action will cause confusion for the opponent who might already have made his own move. A simpler solution is to use two-stage commit with "preview," as mentioned earlier, so that players can avoid accidental moves.

G.9.5 Decorator Design Pattern

One may think that the Referee may be a good candidate for using the Decorator pattern. The Referee for the standard tic-tac-toe would then be decorated with additional rules for the revenge or nine-broad versions. This can be done, however, it would require that exactly one decorator is added for each version in a very specific order. I feel that this is exactly what class inheritance offers, and in this case, class inheritance would be my preferred choice:



G.9.6 State Design Pattern

We know from Section G.8.1 that the three "doer" objects (Controller, Communicator, and Referee) have relatively complex state machines. Therefore, it may be useful to consider employing the State design pattern to externalize the state of these objects. Before we redesign the classes, we compile the state tables for different objects. The following two tables show the states and events for the Controller object. (Note that two events are missing to complete the table: *user-acknowledged* in state 6: *Match Finalizing*, and *network-restored*.)

Current State \ Next State / Output Action	challenge	challenge-response	version	version-response
1: Challenge Enabled	2: Activity Disabled display the status	[GC1] 4: Move Enabled [GC2] 5: Move Disabled [GC3] 1: Challenge Enabled		
2: Activity Disabled		[GC1,GC2] show gameroom [GC3] show initial screen		[GC5] 4: Move Enabled [GC6] 5: Move Disabled [GC7] 3: Version Counteroffer [GC5, GC6] back-to-gameroom
3: Version Counteroffer				[GC5] 4: Move Enabled [GC6] 5: Move Disabled [GC8] 1: Challenge Enabled [GC8] show initial screen [GC5, GC6] back-to-gameroom
4: Board Move Enabled			[GC4] 2: Activity Disabled show initial screen	
5: Board Move Disabled				5: Board Move Disabled
6: Match Finalizing				
7: Logging out				

Guard Conditions:

GC1: challenge-response="accepted" & local-plays-Xs
GC2: challenge-response="accepted" & local-plays-Os
GC3: challenge-response="rejected"
GC4: board-empty
GC5: version-response="accepted" & local-plays-Xs
GC6: version-response="accepted" & local-plays-Os
GC7: version-response="accepted"
GC8: version-response="rejected"

Current State \ Next State / Output Action	move-piece	response-timeout	forfeit	network-outage
1: Challenge Enabled				2: Activity Disabled display warning
2: Activity Disabled		1: Challenge Enabled show initial screen		2: Activity Disabled display warning
3: Version Counteroffer		1: Challenge Enabled show initial screen		2: Activity Disabled display warning
4: Board Move Enabled	[GC9] 5: Board Move Disabled [GC10] 6: Match Finalizing [GC9] update board [GC10] process game won	6: Match Finalizing process game won	6: Match Finalizing process game lost	2: Activity Disabled display warning
5: Board Move Disabled	[GC11] 4: Board Move Enabled [GC12] 6: Match Finalizing [GC11] update board [GC12] process game lost	6: Match Finalizing process game won	6: Match Finalizing process game lost	2: Activity Disabled display warning
6: Match Finalizing		1: Challenge Enabled show initial screen		2: Activity Disabled display warning
7: Logging out				

Guard Conditions:

GC9: local-player-move & valid-move
GC10: local-player-move & ending-move
GC11: opponent-move
GC12: opponent-move & ending-move

The state tables for the Communicator and Referee are left to the reader as exercise.

Based on the above state table, we derive the class interface for the LocalPlayerState and RemotePlayerState (Figure G-27). Both of these states implement the Player interface derived earlier for the Remote Proxy pattern (Figure G-26). The LocalPlayerState and RemotePlayerState are *abstract classes*, which is why their names are italicized in Figure G-27. The methods of LocalPlayerState correspond to the events that can occur on the Controller object that are listed in the above state table for Controller. As seen, most of the Controller and Communicator attributes from Figure G-26 are dropped because their value is replaced by a state object, as explained next.

State Design Pattern for Tic-tac-toe Players built on Remote Proxy

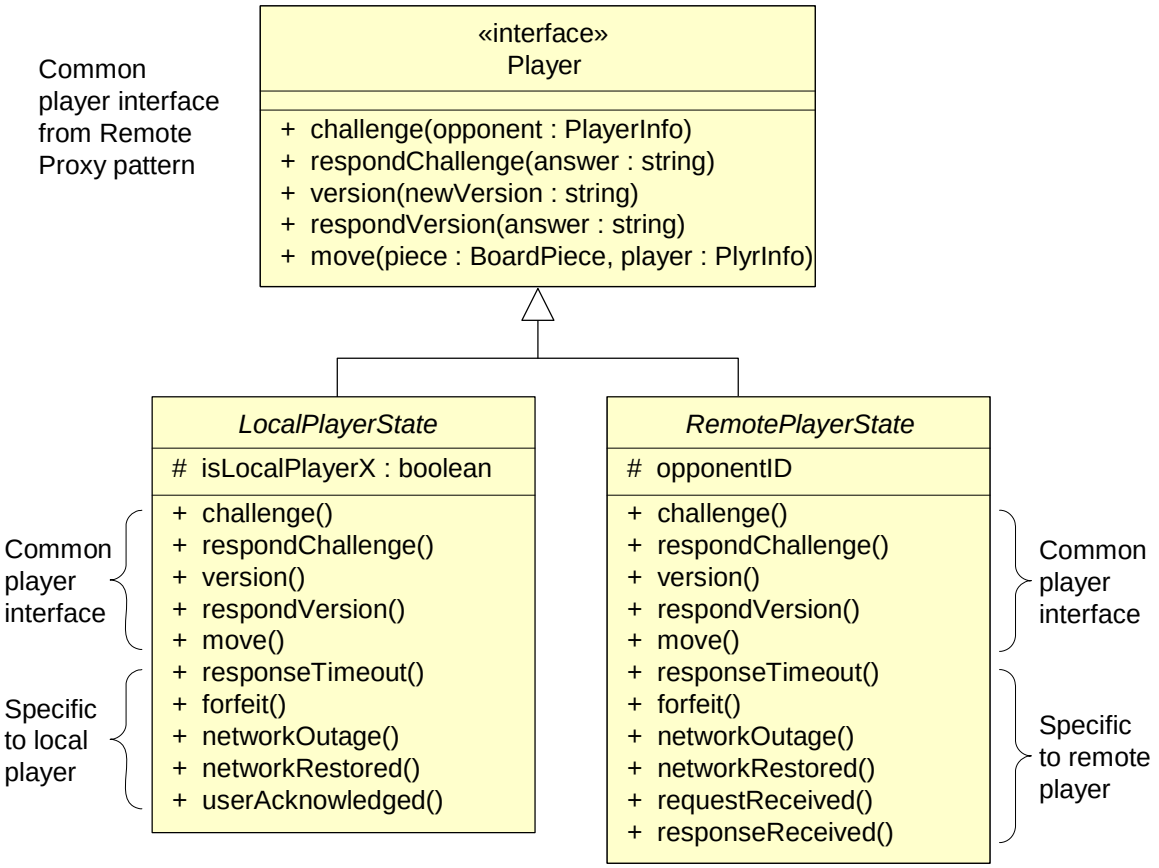


Figure G-27: Class diagram for State classes of Controller and Communicator obtained by extending the Remote Proxy pattern from Figure G-26. This class diagram is completed in Figure G-28.

The abstract base State classes from Figure G-27 are extended by concrete states in Figure G-28. These classes externalize the states for Controller and Communicator.

Listing G-3: The Player interface, and LocalPlayerState and RemotePlayerState base classes (see Figure G-27 for the class diagram).

```
public interface Player {
   public void challenge(PlayerInfo opponent);
   public void respondChallenge(String answer);
   public void version(String newVersion);
   public void respondVersion(String answer);
   public void move(BoardPiece piece, PlayerInfo player);
}
```

```
public abstract class LocalPlayerState implements Player {
   protected Controller context;

   // constructor
```

```
   public LocalPlayerState(Controller context) {
      this.context = context;
   }
   // event handlers:
   public void challenge(PlayerInfo opponent) { }
   public void respondChallenge(String answer) { }
   public void version(String newVersion) { }
   public void respondVersion(String answer) { }
   public void move(BoardPiece piece, PlayerInfo player) { }
   public void responseTimeout(Player unresponsivePlayer) { }
   public void forfeit(Player loser) { }
   public void networkOutage() { }
   public void networkRestored() { }
   public void userAcknowledged() { }
}
```

```
public abstract class RemotePlayerState implements Player {

   protected Communicator context;
   protected PlayerInfo opponentID;

   // constructor
   public RemotePlayerState(Communicator context) {
      this.context = context;
   }
   // event handlers:
   public void challenge(PlayerInfo opponent) { }
   public void respondChallenge(String answer) { }
   public void version(String newVersion) { }
   public void respondVersion(String answer) { }
   public void move(BoardPiece piece, PlayerInfo player) { }
   public void responseTimeout(Player unresponsivePlayer) { }
   public void forfeit(Player loser) { }
   public void networkOutage() { }
   public void networkRestored() { }
   public void requestReceived(Object message) { }
   public void responseReceived(Object message) { }
}
```

The context class (Controller or Communicator) simply dispatches an incoming event to its current state object to handle the event. Listing G-4 shows the context classes. Note that all conditional logic in the context classes has disappeared because of applying the State pattern.

Listing G-4: The Controller and Communicator context classes (Figure G-28 shows the class diagram).

```
public class Controller {

   LocalPlayerState currentState; // field has package-wide visibility

   public void challenge(PlayerInfo opponent) {
      currentState.challenge(opponent);
   }
   public void respondChallenge(String answer) {
```

```
      currentState.respondChallenge(answer);
   }
   public void version(String newVersion) {
      currentState.version(newVersion);
   }
   public void respondVersion(String answer) {
      currentState.respondVersion(answer);
   }
   public void move(BoardPiece piece, PlayerInfo player) {
      currentState.move(piece, player);
   }
   public void responseTimeout(Player unresponsivePlayer) {
      currentState.responseTimeout(unresponsivePlayer);
   }
   public void forfeit(Player loser) {
      currentState.forfeit(loser);
   }
   public void networkOutage() {
      currentState.networkOutage();
   }
   public void networkRestored() {
      currentState.networkRestored();
   }
   public void userAcknowledged() {
      currentState.userAcknowledged();
   }
}
```

```
public class Communicator {

   RemotePlayerState currentState; // field visible package-wide

   public void challenge(PlayerInfo opponent) {
      currentState.challenge(opponent);
   }
   public void respondChallenge(String answer) {
      currentState.respondChallenge(answer);
   }
   public void version(String newVersion) {
      currentState.version(newVersion);
   }
   public void respondVersion(String answer) {
      currentState.respondVersion(answer);
   }
   public void move(BoardPiece piece, PlayerInfo player) {
      currentState.move(piece, player);
   }
   public void responseTimeout(Player unresponsivePlayer) {
      currentState.responseTimeout(unresponsivePlayer);
   }
   public void forfeit(Player loser) {
      currentState.forfeit(loser);
   }
   public void networkOutage() {
      currentState.networkOutage();
   }
   public void networkRestored() {
```

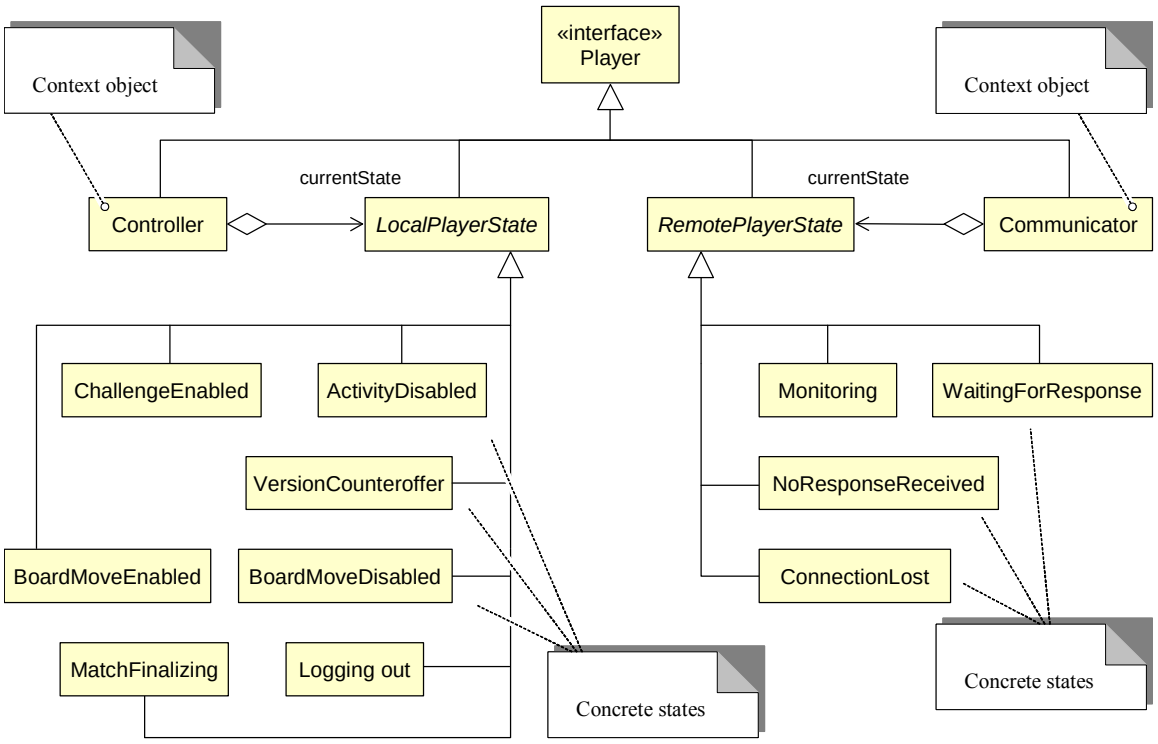


Figure G-28: Class diagram that combines two design patterns, Remote Proxy and State, for the distributed game of tic-tac-toe.

```
      currentState.networkRestored();
   }
   public void requestReceived(Object message) {
      currentState.requestReceived(message);
   }
   public void responseReceived(Object message) {
      currentState.responseReceived(message);
   }
}
```

We show the code for only one state of the Controller in Listing G-5. Based on the above state table for the controller, we know that only three events are handled in the Controller state 1: *Challenge Enabled*.

Listing G-5: Concrete state class ChallengeEnabled for the Controller context.

```
public class ChallengeEnabled extends LocalPlayerState {
   public void challenge(PlayerInfo opponent) {
      // display the status: opponent challenged

      // set the next state of the context: Activity Disabled
      context.currentState = ...
   }
   public void respondChallenge(String answer) {
      if (answer.equals("accepted") && isLocalPlayerX) {
         // set the next state of the context: Move Enabled
```

```
            context.currentState = ...
         }
         else if (answer.equals("accepted") && ! isLocalPlayerX) {
            // set the next state of the context: Move Disabled
            context.currentState = ...
         }
         else if (answer.equals("rejected") {
            // set the next state of the context: Challenge Enabled
            context.currentState = ...
         }
         // perhaps should also handle else case for anything...
      }
      public void networkOutage() {
            // set the next state of the context: Activity Disabled
         context.currentState = ...
      }
}
```

As seen, the events that are not relevant for the given state are not defined—the corresponding methods are inherited from the abstract base state class.

G.9.7 Model-View-Controller (MVC) Design Pattern

The Model-View-Controller pattern is useful for implementing different interaction and visualization techniques for the system data (the so-called "model" part of MVC). We already discussed splitting the original Controller object into the part that handles the local player's moves and the part that displays the remote player's moves. These parts correspond to the "controller" part of MVC and the "view" part of MVC, respectively. A partial class diagram for the "model" part of MVC is shown in Figure G-29. It is partial because the model should include Match Invitation, Invite Queue, and all other "knowing" objects.

Note that GameBoard and its derived classes implement the *Composite* design pattern. The *Composite* pattern allows for treating a group of objects in the same way as a single instance of an object. The intent is to "compose" objects into tree structures to represent part-whole hierarchies. In our case, NineBoard is composed of nine StandardBoards.

As shown in the class diagram in Figure G-29, the nine-board version of tic-tac-toe should not be represented as a single board with $9 \times 9 = 81$ pieces, because to enforce the rules of this game we need to know the identity of the sub-boards. We do not need a special attribute to identify each sub-board because the nine boards are created once for a match and we can enforce a convention that the first three sub-boards represent the first row, the second three the second row, and the last three the third row.

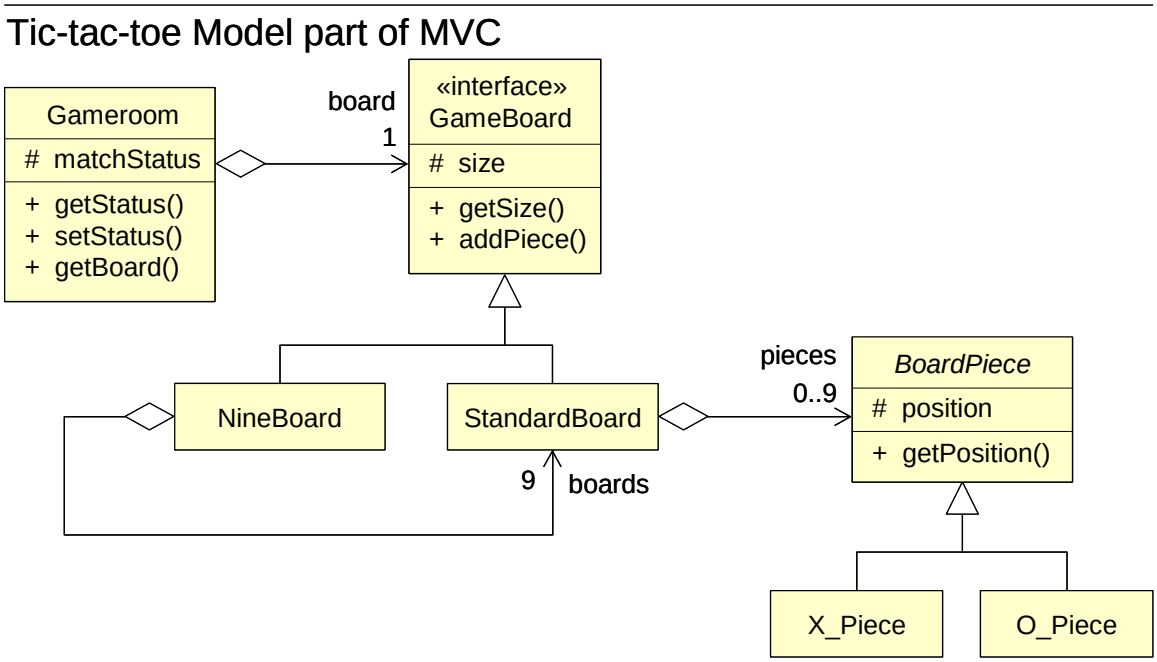


Figure G-29: Class diagram for classes that constitute the Model part of the Model-View-Controller design pattern for the game of tic-tac-toe.

G.10 Concurrency and Multithreading

_____ TO BE COMPLETED _____

How We Did It & Plan of Work

The project took a tremendous amount of effort from all of the team members who cared. There were several occasions that initial ideas had to be trashed halfway through and reworked completely in order to deal with the obstacle at hand, which then led the majority of documentation having to be redone by me to reflect the changes. This was frustrating because we could not move forward with the project because all the previous documentation was now simply invalidated and no longer accurate because we did not account for problems running out of memory or because the algorithm was not performing the way it was supposed to. This gave little room for error in order to get a decent grade. I feel that our schedule did not account for these types of mistakes that can hinder a group's progress and in turn hurt a group's grades for things students could not account for beforehand.

The techniques that were the most useful to the group were undoubtedly working together in the labs every step of the way, especially when coding. This way we could get our program to display and function exactly as was intended. Although later on we found that certain ideas we had before for the functionality of our project were a little more complicated than we had initially thought, at least we could agree on it as a group and there was no more confusion about the functionality of our program amongst our members, (the members that showed up consistently that is…as we had one that did not participate very much and rarely showed up to any meetings).

Me says: This class has taught me that you need one page of documentation for every line of code. Documentation is key to software projects. I came into this class expecting a lot of coding, and have come out really understanding what a software engineer is and why documentation is important. I do feel that we did a lot of documentation, maybe more than needed, versus the real world.

One of the things that I found to be the least valuable in our project specifically was the design patterns because it would have been very nice to know them at the beginning of designing the application in order to incorporate them without a very large time overhead. Also, the implementation that we had started long before knowing about these concepts could not be adapted to include many design patterns as it became inefficient to do so. A few patterns/techniques described in the book were a common sense type of thing for me. I have used these patterns when coding but never thought of them as a technique.

Irene says: This class teaches you how to deal with different types of people, but you might pull out a few chunks of hair first. It has definitely opened my eyes to the reality of software engineering. Working in a team can be a nightmare, and I say that not only because of my experiences, but because of the qualms of many of the other groups as well. When you work with a bunch of incompetent people, without a doubt the development of your software will be a terrible hardship. Trying to get members to realize what they need to do and how they should do it was sadly the most challenging and hardest part of the entire semester. I will adopt a pace of work at the outset, regardless of the others, that allows me to reduce dependence on them.

The techniques that were least useful to our group surprisingly was splitting up the work and working on it gradually. Although this seems as one of the best things to do, it turned out to work against us in the long run. This was because of procrastination and just general lack of effort from certain members. This in turn put more work on the people that were willing to shoulder the responsibilities at the end when deadlines approached and the other team members' work was still not complete or was of poor quality. I turned out to be one of those people and the additional responsibilities and sheer magnitude of work especially for the reports was not pleasant at all.