Contents

1 QSeaBattle â€” Design Specification 7
1.1 Purpose 7
1.2 Scope and audience 7
1.3 Structure of the specification 8
1.4 Normative vs informative content 8
1.5 Versioning and stability 8
1.6 How to read this document 9
1.7 Repository layout (informative) 9
1.8 Design principle (non-negotiable) 9

2 Algorithms overview (informative) 9
2.1 Purpose 9
2.2 1. Classical assisted algorithm (baseline) 9
2.2.1 Problem 9
2.2.2 High-level idea 10
2.2.3 Algorithm (conceptual) 10
2.3 2. Shared randomness algorithm 10
2.3.1 Resource model 10
2.3.2 Usage rules 10
2.4 3. Deterministic baselines (Simple and Majority) . . 10
2.4.1 Why they exist 10
2.4.2 Simple baseline (conceptual) 11
2.4.3 Majority baseline (conceptual) 11
2.5 4. Lin trainable assisted algorithm 11
2.5.1 Goal 11
2.5.2 Algorithm 11
2.6 5. Pyramid (Pyr) trainable assisted algorithm . . . 11
2.6.1 Motivation 11
2.6.2 Algorithm 11
2.7 6. Training algorithms (summary) 12
2.7.1 Imitation learning 12
2.7.2 Key restriction 12
2.8 Relationship to the rest of the spec 12
2.9 Reader guidance 12

3 Conventions 12
3.1 Purpose 12
3.2 Notation 13
3.2.1 Sizes and shapes 13
3.2.2 Binary values 13
3.3 Terminology 13
3.3.1 Game vs tournament 13
3.3.2 Players and models 13
3.3.3 Shared randomness (SR) 13

3.4 Determinism, seeding, and reproducibility 14
3.5 Channel noise 14
3.6 Contract language (RFC-style) 14
3.7 Interface conventions 14
3.7.1 decide(...) 14
3.7.2 Dtypes 14
3.8 Error handling 14
3.9 Document status 15

4 Class GameLayout **15**
4.1 Purpose 15
4.2 Location 15
4.3 Public attributes 15
4.4 Public methods 16
4.4.1 from_dict(parameters: dict) -> GameLayout . . 16
4.4.2 to_dict() -> dict 16
4.5 Invariants 16
4.6 Failure modes 16

5 Class GameEnv **16**
5.1 Purpose 16
5.2 Location 16
5.3 Public methods 16
5.3.1 reset() -> None 16
5.3.2 provide() -> tuple[np.ndarray, np.ndarray] . . 16
5.3.3 evaluate(shoot: int) -> float 16
5.3.4 apply_channel_noise(comm: np.ndarray) -> np.ndarray 17
5.4 Invariants 17

6 Class Game **17**
6.1 Purpose 17
6.2 Method 17
6.2.1 play() 17

7 Class Tournament **17**
7.1 Purpose 17
7.1.1 tournament() -> TournamentLog 17

8 Class TournamentLog **17**
8.1 Purpose 17
8.1.1 Methods 17

9 Base player interfaces **18**
9.1 Purpose 18
9.2 Class Players 18
9.2.1 Purpose 18

9.2.2 Public methods 18
9.2.3 Invariants 18
9.3 Class PlayerA 18
9.3.1 Purpose 18
9.3.2 Method 18
9.3.3 Contract 18
9.3.4 Default baseline behavior (random) 19
9.4 Class PlayerB 19
9.4.1 Purpose 19
9.4.2 Method 19
9.4.3 Contract 19
9.4.4 Default baseline behavior (random) 19
9.5 Related 19

10 Deterministic and baseline players **19**
10.1 Purpose 19
10.2 Class SimplePlayers(Players) 20
10.2.1 Purpose 20
10.2.2 Behavioral contract 20
10.3 Class SimplePlayerA(PlayerA) 20
10.3.1 Purpose 20
10.3.2 Algorithm (normative) 20
10.3.3 Invariants 20
10.4 Class SimplePlayerB(PlayerB) 20
10.4.1 Purpose 20
10.4.2 Algorithm (normative) 21
10.4.3 Notes 21
10.4.4 Invariants 21
10.5 Class MajorityPlayers(Players) 21
10.5.1 Purpose 21
10.5.2 Behavioral contract 21
10.6 Class MajorityPlayerA(PlayerA) 21
10.6.1 Purpose 21
10.6.2 Algorithm (normative) 21
10.6.3 Invariants 22
10.7 Class MajorityPlayerB(PlayerB) 22
10.7.1 Purpose 22
10.7.2 Algorithm (normative) 22
10.7.3 Invariants 22
10.8 Related 22

11 Assisted players (shared-randomness assisted) **23**
11.1 Purpose 23
11.2 Shared randomness dependency 23
11.3 Class SharedRandomness() 23
11.3.1 Purpose 23

11.4 Class AssistedPlayers(Players) 23
11.4.1 Purpose 23
11.4.2 Location 23
11.4.3 Parameters 23
11.4.4 Behavioral contract 24
11.5 Class AssistedPlayerA(PlayerA) 24
11.5.1 Purpose 24
11.5.2 Behavioral contract (normative) 24
11.5.3 Algorithmic outline (informative) 24
11.6 Class AssistedPlayerB(PlayerB) 24
11.6.1 Purpose 24
11.6.2 Behavioral contract (normative) 24
11.6.3 Algorithmic outline (informative) 25
11.7 Invariants 25

12 Neural players (unassisted) 25
12.1 Purpose 25
12.2 Characteristics 25
12.3 Behavioral constraints 25
12.4 Notes 25

13 Shared randomness resource (non-classical) 26
13.1 Purpose 26
13.2 Location 26
13.3 Parameters 26
13.4 Public interface (conceptual) 26
13.4.1 reset() -> None 26
13.4.2 measurement_a(measurement: array_like) -> np.ndarray 26
13.4.3 measurement_b(measurement: array_like) -> np.ndarray 26
13.5 Behavioral contract (normative) 27
13.5.1 Single-use per party 27
13.5.2 Two-measurement lifecycle 27
13.5.3 Output format 27
13.5.4 Non-classical correlation requirement 27
13.5.5 No information leakage 28
13.6 Failure modes 28
13.7 Notes (informative) 28

14 SharedRandomnessLayer (differentiable implementation) 28
14.1 Purpose 28
14.2 Location 28
14.3 Inputs / Outputs 28
14.4 Behavioral contract 29
14.5 Recommended modes (informative) 29

14.6 Failure modes 29

15 Trainable assisted models â€” Linear (Lin) 29
15.1 Purpose 29
15.2 Scope 29
15.3 Class LinTrainableAssistedModelA 30
15.4 Purpose 30
15.5 Location 30
15.6 Inputs 30
15.7 Outputs 30
15.8 Internal state 30
15.9 Behavioral contract 30
15.10 Invariants 30
15.11 Class LinTrainableAssistedModelB 31
15.12 Purpose 31
15.13 Location 31
15.14 Inputs 31
15.15 Outputs 31
15.16 Behavioral contract 31
15.17 Invariants 31
15.18 Notes / rationale 31

16 Trainable assisted models â€” Pyramid (Pyr) 32
16.1 Purpose 32
16.2 Conceptual overview 32
16.3 Class PyrTrainableAssistedModelA 32
16.4 Purpose 32
16.5 Location 32
16.6 Inputs 32
16.7 Outputs 33
16.8 Internal state 33
16.9 Iterative structure (normative) 33
16.10 Behavioral contract 33
16.11 Invariants 33
16.12 Failure modes 34
16.13 Class PyrTrainableAssistedModelB 34
16.14 Purpose 34
16.15 Location 34
16.16 Inputs 34
16.17 Outputs 34
16.18 Internal state 34
16.19 Reconstruction procedure (normative) 34
16.20 Behavioral contract 35
16.21 Invariants 35
16.22 Notes / rationale 35

17 Imitation learning for assisted models **35**
17.1 Purpose 35
17.2 Data generation 35
17.3 Behavioral contract 35
17.4 Losses 35
17.5 Invariants 36

18 DRU / DIAL training **36**
18.1 Purpose 36
18.2 Algorithm (informative) 36
18.3 Constraints 36
18.4 Notes 36

19 Reinforcement learning **36**
19.1 Purpose 36
19.2 Supported approaches 36
19.3 Constraints 37
19.4 Notes 37

20 Training considerations for Pyramid (Pyr) models **37**
20.1 Purpose 37
20.2 Training regimes 37
20.3 Behavioral constraints 37
20.4 Diagnostics (recommended) 37
20.5 Invariants 37

21 Utility module: Data generation **38**
21.1 Purpose 38
21.2 Scope 38
21.3 Core utilities 38
21.3.1 generate_measurement_dataset_a(layout, num_samples, seed) 38
21.3.2 generate_measurement_dataset_b(layout, num_samples, seed) 38
21.4 Invariants 38

22 Utility module: Imitation training **38**
22.1 Purpose 38
22.2 Core utilities 39
22.2.1 train_layer(layer, dataset, loss, epochs, metrics=None) 39
22.2.2 train_model(model, datasets) 39
22.3 Invariants 39

23 Utility module: Layer and weight transfer **39**
23.1 Purpose 39
23.2 Core utilities 39

23.2.1 transfer_assisted_model_a_layer_weights(...) . 39
23.2.2 transfer_assisted_model_b_layer_weights(...) . 39
23.3 Behavioral contract 39
23.4 Invariants 40

24 Utility module: Evaluation **40**
24.1 Purpose 40
24.2 Core utilities 40
24.2.1 evaluate_players(players, layout) 40
24.2.2 confidence_interval(mean, std_error) 40
24.3 Contract 40
24.4 Invariants 40

25 Invariants **40**
25.1 Purpose 40
25.2 Game and environment 40
25.3 Communication 41
25.4 Assisted / shared randomness 41
25.5 Trainable assisted models (Lin/Pyr) 41
25.6 Data generation and training 41
25.7 Logging and evaluation 41
25.8 Testability expectation (recommendation) 42

1 QSeaBattle â€” Design Specification

1.1 Purpose

This document is the **authoritative design specification** for the QSeaBattle project. It defines the game mechanics, player models, algorithms, training regimes, and utilities that together constitute the QSeaBattle system.

If any behavior in the codebase contradicts this specification, the code is considered incorrect.

1.2 Scope and audience

This specification is intended for: - Developers implementing or modifying the QSeaBattle codebase - Researchers reviewing the algorithmic and information-theoretic aspects - Future maintainers re-implementing components independently

This is **not** a user tutorial; it is a technical design contract.

1.3 Structure of the specification

The document is organized into the following logical parts:

1. **Algorithms**
   - High-level, informative explanations of the classical assisted strategy, trainable Lin models, and hierarchical Pyr models.
2. **Conventions**
   - Global notation, terminology, and RFC-style contract language.
3. **Game infrastructure**
   - Game layout, environment, game execution, and tournaments.
4. **Players**
   - Abstract player interfaces and concrete player families (deterministic, assisted, neural).
5. **Shared randomness**
   - Classical shared-randomness resources and differentiable variants.
6. **Models**
   - Trainable assisted models (Lin and Pyr) with strict information constraints.
7. **Training**
   - Imitation learning, DRU/DIAL, and reinforcement learning regimes.
8. **Utilities**
   - Dataset generation, training helpers, evaluation, and weight transfer.
9. **Invariants**
   - One-sentence global truths that must always hold.

1.4 Normative vs informative content

- Chapters describing components, interfaces, and constraints are **normative**.
- The Algorithms chapter is **informative** and provides intuition only.
- In case of conflict, **normative chapters take precedence**.

Normative language uses: - **MUST / MUST NOT** - **SHOULD / SHOULD NOT** - **MAY**

1.5 Versioning and stability

This specification evolves with the codebase. Breaking changes to behavior MUST be accompanied by updates to this document.

A frozen PDF export may be tagged as: - *QSeaBattle Specification vX.Y*

1.6 How to read this document

- Start with **Algorithms** for intuition
- Read **Conventions** and **Invariants** carefully
- Then read the chapters relevant to the component you are working on

1.7 Repository layout (informative)

```
QSeaBattle/
â”œâ”€â”€ docs/              # This specification (Markdown source)
â”œâ”€â”€ mkdocs.yml         # Documentation build configuration
â”œâ”€â”€ src/               # Implementation
â”œâ”€â”€ tests/             # Tests (should reference invariants)
â””â”€â”€ notebooks/         # Experiments and analysis
```

1.8 Design principle (non-negotiable)

> Information flow is the primary design constraint.
> Learning may approximate logic, but must never bypass it.

2 Algorithms overview (informative)

2.1 Purpose

This chapter provides a consolidated, **informative (non-normative)** explanation of the core algorithms underlying QSeaBattle. It complements the normative chapters by explaining *how* the system works step by step, without introducing new requirements.

Nothing in this chapter overrides behavioral contracts defined elsewhere.

2.2 1. Classical assisted algorithm (baseline)

2.2.1 Problem

Player A observes a binary field of size n^2.
Player B observes a one-hot gun position.
Only **one communication bit** is allowed.

2.2.2 High-level idea

Use **shared randomness** to correlate partial information held by the players so that Player B can reconstruct the relevant field bit with probability greater than any unassisted classical strategy.

2.2.3 Algorithm (conceptual)

1. Player A groups the field into pairs.
2. For each pair, Player A performs a shared-randomness measurement.
3. Outcomes reduce the field size by half.
4. Steps 1â€“3 repeat until a single bit remains.
5. Player A sends this bit to Player B.
6. Player B performs corresponding measurements in reverse order, guided by the gun position.

This forms a **pyramid reduction / reconstruction scheme**.

2.3 2. Shared randomness algorithm

2.3.1 Resource model

A shared-randomness resource produces correlated binary outcomes (x, y) such that:

[P(x = y) = p_{high}, P(x ≠y) = 1 - p_{high}]

2.3.2 Usage rules

- Exactly one measurement per party
- First measurement is unbiased
- Second measurement is conditionally biased

This abstracts quantum-like correlations while remaining classical.

2.4 3. Deterministic baselines (Simple and Majority)

2.4.1 Why they exist

Deterministic baselines provide: - sanity checks (does the environment work?), - stable regression baselines, - and lower bounds for performance.

They do **not** use shared randomness.

2.4.2 Simple baseline (conceptual)

A minimal deterministic protocol: 1. Player A computes a fixed summary of the field (e.g., parity of a fixed subset) and sends it as comm. 2. Player B maps (gun, comm) through a fixed rule to decide shoot.

This baseline is intentionally simple and implementation-defined, but MUST remain deterministic.

2.4.3 Majority baseline (conceptual)

A stronger heuristic: 1. Player A sends whether the majority of a fixed subset S of the field is 1. 2. Player B uses this as its guess (optionally region-conditioned if comms_size > 1).

2.5 4. Lin trainable assisted algorithm

2.5.1 Goal

Approximate the classical assisted algorithm using minimal-capacity neural networks.

2.5.2 Algorithm

1. Replace fixed measurement logic with a linear measurement layer.
2. Replace fixed combination logic with a linear combine layer.
3. Train both layers using imitation learning from the classical policy.
4. Enforce all information-flow constraints at runtime.

The Lin model performs the same *logical* algorithm as the classical strategy, but learns the mapping from data.

2.6 5. Pyramid (Pyr) trainable assisted algorithm

2.6.1 Motivation

Lin models do not scale efficiently to large n^2.

2.6.2 Algorithm

For $k = \log_2(n^2)$ layers:

Forward (Player A): 1. Apply a trainable measurement layer. 2. Perform one shared-randomness measurement. 3. Combine outputs to halve the field size.

Backward (Player B): 1. Use the gun position to select the relevant subtree. 2. Combine shared-randomness outcomes layer by layer. 3. Apply the communication bit at the final step.

The Pyr algorithm preserves logarithmic depth and strict information control.

2.7 6. Training algorithms (summary)

2.7.1 Imitation learning

1. Generate optimal trajectories using classical assisted players.
2. Train measurement and combine layers to match targets.
3. Validate that trained models obey runtime contracts.

2.7.2 Key restriction

Training may change **weights**, but never: - Communication bandwidth - Information access - Layer structure

2.8 Relationship to the rest of the spec

Chapter	Role
3	Game mechanics
6	Classical coordination
7	Trainable baseline
8	Hierarchical scaling
9	Safe utilities
This chapter	Algorithmic intuition

2.9 Reader guidance

This chapter is intended for: - Readers new to the project - Reviewers evaluating novelty - Future re-implementations

For authoritative behavior, always defer to the normative chapters.

3 Conventions

3.1 Purpose

This document defines **global notation, terminology, and cross-cutting conventions** used throughout the QSeaBattle specifica-

tion. Unless explicitly marked as *informative*, statements in this file are **normative**.

3.2 Notation

3.2.1 Sizes and shapes

- Let field_size = n (board side length). The total number of cells is n^2.
- Vectors representing the full field or gun are **flattened** to length n^2.
- Batch dimension is denoted by B.

Shape conventions - field: (B, nÂ^2) or (nÂ^2,) for a single sample - gun: (B, nÂ^2) or (nÂ^2,) one-hot - comm: (B, m) where m = comms_size (often m=1) - shoot: (B, 1) or scalar {0,1}

3.2.2 Binary values

Unless stated otherwise, binary arrays use values in {0,1}.

3.3 Terminology

3.3.1 Game vs tournament

- A **game** is one execution of the pipeline: generate field+gun â†’ A decides â†’ noise â†’ B decides â†’ reward.
- A **tournament** is a batch of games under identical layout parameters.

3.3.2 Players and models

- **Players** are decision-makers that implement PlayerA.decide and PlayerB.decide.
- **Models** are reusable computational components (e.g., neural models) that may back a player.
- A trainable model may store **per-decision state** (e.g., per-layer outcomes) required for Player B.

3.3.3 Shared randomness (SR)

- **Shared randomness** is a two-party resource used once per layer (Pyr) or once per decision (Lin).
- SR is **not communication**. SR does not increase comms_size.

3.4 Determinism, seeding, and reproducibility

- Any utility that samples randomness (data generation, evaluation) MUST accept a seed parameter or use a reproducible RNG strategy.
- Runtime gameplay MAY be stochastic due to SR and channel noise; evaluation runs SHOULD support deterministic replay via fixed seeds.

3.5 Channel noise

- Channel noise flips each bit of comm independently with probability channel_noise.
- Noise MUST be applied **after** Player A decides and **before** Player B decides.

3.6 Contract language (RFC-style)

The following keywords are normative: - **MUST / MUST NOT**: required for compliance - **SHOULD / SHOULD NOT**: recommended; deviations require justification - **MAY**: optional behavior permitted by the spec

3.7 Interface conventions

3.7.1 decide(...)

- decide methods MUST be pure with respect to their explicit inputs, except for documented internal state needed for coordination (e.g., per-layer outcomes).
- For comms_size == 1, comm MUST still be represented as an array/tensor of shape (B, 1) (not a scalar).

3.7.2 Dtypes

- Numpy arrays are acceptable for environment-level logic.
- TensorFlow tensors are acceptable for trainable models and training utilities.
- A given function MUST document whether it expects NumPy or Tensor inputs if ambiguous.

3.8 Error handling

- Shape mismatches MUST raise ValueError (preferred) rather than silently reshaping.

- Double-use of shared randomness MUST raise an error (typically ValueError or RuntimeError), not silently proceed.

3.9 Document status

- Chapters describing components are **normative**, unless explicitly labeled *informative*.
- docs/algorithms.md is **informative** and does not override contracts.

4 Class GameLayout

4.1 Purpose

Define the immutable parameters that specify a QSeaBattle game configuration, ensuring consistency across environment, players, tournaments, and logs.

4.2 Location

- **Module:** src/Q_Sea_Battle/game_layout.py
- **Class:** GameLayout

4.3 Public attributes

Name	Type	Constraints	Description
field_size	int	power of 2	Board dimension n; total cells n^2
comms_size	int	divides n^2	Number of communication bits
enemy_probability	float	[0,1]	Probability a field cell equals 1
channel_noise	float	[0,1]	Bit-flip probability on comm channel
number_of_games_in_tournament	int	>0	Games per tournament
log_columns	list[str]	–	Columns stored in TournamentLog

4.4 Public methods

4.4.1 from_dict(parameters: dict) -> GameLayout

Create a validated GameLayout from a dictionary.

4.4.2 to_dict() -> dict

Return all layout parameters as a dictionary.

4.5 Invariants

- field_size MUST be a power of two.
- comms_size MUST divide field_size^2.
- All components MUST share the same GameLayout instance.

4.6 Failure modes

- ValueError if constraints are violated.

5 Class GameEnv

5.1 Purpose

Implement the physical rules of QSeaBattle: field generation, gun placement, reward evaluation, and communication noise.

5.2 Location

- **Module:** src/Q_Sea_Battle/game_env.py
- **Class:** GameEnv

5.3 Public methods

5.3.1 reset() -> None

Generate a new random field and gun.

5.3.2 provide() -> tuple[np.ndarray, np.ndarray]

Return (field, gun) as flattened binary arrays.

5.3.3 evaluate(shoot: int) -> float

Return 1.0 if decision matches field value at gun index, else 0.0.

5.3.4 apply_channel_noise(comm: np.ndarray) -> np.ndarray

Apply independent bit flips with probability channel_noise.

5.4 Invariants

- gun MUST be one-hot.
- field.shape == gun.shape == (n^2,)

6 Class Game

6.1 Purpose

Execute a single QSeaBattle game.

6.2 Method

6.2.1 play()

Runs one game: reset, decide A, apply noise, decide B, evaluate.

7 Class Tournament

7.1 Purpose

Run multiple games and collect structured data.

7.1.1 tournament() -> TournamentLog

Run a full tournament.

8 Class TournamentLog

8.1 Purpose

Store per-game data for analysis and training.

8.1.1 Methods

- update(...)
- outcome() -> (mean, std_error)

9 Base player interfaces

9.1 Purpose

Define the abstract interfaces for Player A and Player B, plus the factory interface that returns a matched pair. Unless otherwise specified by a concrete implementation, the **default baseline behavior is random**.

9.2 Class Players

9.2.1 Purpose

Factory and lifecycle manager for paired PlayerA / PlayerB instances.

9.2.2 Public methods

- players() -> tuple[PlayerA, PlayerB]
  Return a matched (player_a, player_b) pair.
- reset() -> None
  Reset any internal per-game state (if applicable).

9.2.3 Invariants

- players() MUST always return a matched A/B pair.
- Both players MUST share the same GameLayout.

9.3 Class PlayerA

9.3.1 Purpose

Observe the field and produce a communication vector.

9.3.2 Method

```
decide(field: np.ndarray, supp: Any | None = None) -> np.ndarray
```

9.3.3 Contract

- Output MUST be a binary array of length comms_size.
- Even when comms_size == 1, output MUST be an array (shape (1,) or (B,1)), not a scalar.
- Player A MUST NOT access gun information.

9.3.4 Default baseline behavior (random)

If a concrete implementation does not specify a different decision rule, PlayerA.decide is interpreted as: - Sample each communication bit independently and uniformly from {0,1}.

This provides a well-defined **random baseline** for debugging and sanity checks.

9.4 Class PlayerB

9.4.1 Purpose

Observe the gun position and received communication, and decide whether to shoot.

9.4.2 Method

```
decide(gun: np.ndarray, comm: np.ndarray, supp: Any | None = None) -> int
```

9.4.3 Contract

- Output MUST be scalar 0 or 1 (or (B,1) for batched implementations).
- Input gun MUST be one-hot.
- Player B MUST NOT access the full field.

9.4.4 Default baseline behavior (random)

If a concrete implementation does not specify a different decision rule, PlayerB.decide is interpreted as: - Sample shoot uniformly from {0,1}, independent of (gun, comm).

9.5 Related

- Implemented by: deterministic, assisted, neural players
- Algorithms: docs/algorithms.md (informative)

10 Deterministic and baseline players

10.1 Purpose

Define simple, non-learning baseline player families used for: - sanity checks, - debugging, - performance lower bounds, - and validating tournament logging.

This chapter specifies two baseline families: 1. **SimplePlayers** (a minimal baseline; Player B includes a stochastic fallback) 2. **MajorityPlayers** (a deterministic heuristic baseline)

10.2 Class SimplePlayers(Players)

10.2.1 Purpose

Factory that returns a matched (SimplePlayerA, SimplePlayerB) pair.

10.2.2 Behavioral contract

- SimplePlayers.players() MUST generate instances of SimplePlayerA and SimplePlayerB.
- The family MUST respect communication bandwidth: output length equals layout.comms_size.

10.3 Class SimplePlayerA(PlayerA)

10.3.1 Purpose

Send the values of the first m field cells directly, where m = layout.comms_size.

10.3.2 Algorithm (normative)

Let m = comms_size. Let field be the flattened field vector of length nÂ^2.

1. Read the first m entries: field[0:m].
2. Output comm = field[0:m].

10.3.3 Invariants

- MUST NOT access gun information.
- MUST output exactly m bits.

10.4 Class SimplePlayerB(PlayerB)

10.4.1 Purpose

If the gun points at one of the first m cells, use the communicated value for that cell. Otherwise, fall back to a simple prior-based stochastic guess using layout.enemy_probability.

10.4.2 Algorithm (normative)

Let m = comms_size. Let gun be one-hot of length nÂ^2. Let idx = argmax(gun).

1. If idx < m:
    - Output shoot = comm[idx].
2. Else:
    - Sample shoot ~ Bernoulli(layout.enemy_probability).

10.4.3 Notes

This baseline is **not fully deterministic** due to the stochastic fallback in step 2.

10.4.4 Invariants

- MUST NOT access full field information.
- Output MUST be binary (0 or 1).
- If idx < m, the output MUST equal the communicated bit for that index.

10.5 Class MajorityPlayers(Players)

10.5.1 Purpose

Factory that returns a matched (MajorityPlayerA, MajorityPlayerB) pair implementing a majority heuristic over partitions.

10.5.2 Behavioral contract

- MajorityPlayers.players() MUST generate instances of MajorityPlayerA and MajorityPlayerB.
- The family MUST respect communication bandwidth: output length equals layout.comms_size.
- The family MUST be deterministic given inputs.

10.6 Class MajorityPlayerA(PlayerA)

10.6.1 Purpose

Split the flattened field into m pieces and send one majority bit per piece.

10.6.2 Algorithm (normative)

Let m = comms_size. Let N = nÂ^2 be the flattened field length.

1. Partition indices {0, 1, ..., N-1} into m contiguous blocks:
   - Block j contains indices from start = floor(j*N/m) up to (but excluding) end = floor((j+1)*N/m).
2. For each block j:
   - Let ones_j = sum(field[i] for i in block j).
   - Let zeros_j = |block j| - ones_j.
   - Set comm[j] = 1 if ones_j >= zeros_j, else comm[j] = 0.
3. Output comm (length m).

10.6.3 Invariants

- MUST NOT access gun information.
- MUST output exactly m bits.
- MUST be deterministic given field.

10.7 Class MajorityPlayerB(PlayerB)

10.7.1 Purpose

Determine which subset (block) the gun points to and return the corresponding communicated majority bit.

10.7.2 Algorithm (normative)

Let m = comms_size. Let N = nÂ^2. Let idx = argmax(gun).

1. Compute the block id j such that idx lies in block j under the same partition rule used by MajorityPlayerA.
2. Output shoot = comm[j].

10.7.3 Invariants

- MUST NOT access full field information.
- Output MUST be binary.
- MUST be deterministic given (gun, comm).

10.8 Related

- Base interfaces: docs/players/base.md
- Game parameters: docs/game/GameLayout.md

11 Assisted players (shared-randomness assisted)

11.1 Purpose

Define Player A and Player B strategies that use a shared-randomness resource to outperform unassisted baselines under strict communication constraints.

This chapter specifies the assisted-player family and its dependency on the shared randomness contract.

11.2 Shared randomness dependency

Assisted players MUST use a resource that satisfies:

- docs/shared_randomness/shared_randomness.md (normative)

The resource may be implemented as: - a pure Python class (SharedRandomness), or - a differentiable Keras layer (SharedRandomnessLayer),

but the externally observable behavior MUST comply with the same contract.

11.3 Class SharedRandomness()

11.3.1 Purpose

Reference class name for the shared randomness resource used by assisted players.

The authoritative behavior is defined in docs/shared_randomness/shared_randomness.md.

11.4 Class AssistedPlayers(Players)

11.4.1 Purpose

Factory and manager for assisted (AssistedPlayerA, AssistedPlayerB) pairs that share correlated resources.

11.4.2 Location

- **Module:** src/Q_Sea_Battle/assisted_players.py
- **Class:** AssistedPlayers

11.4.3 Parameters

Name	Type	Constraints
p_high	float	[0,1]
layout	GameLayout	comms_size == 1 (recommended baseline)

11.4.4 Behavioral contract

- Communication bandwidth MUST be respected (comms_size bits).
- Assisted players MUST NOT treat shared randomness as extra communication.
- Per game, the shared randomness resources MUST be reset.

11.5 Class AssistedPlayerA(PlayerA)

11.5.1 Purpose

Observe the field and compute comm using shared randomness according to a fixed hierarchical reduction scheme.

11.5.2 Behavioral contract (normative)

- MUST NOT access gun information.
- MUST consume shared randomness in the same sequence that Player B will mirror.
- MUST produce exactly comms_size bits (typically 1).

11.5.3 Algorithmic outline (informative)

- Reduce the field iteratively (pairwise grouping) while consuming one SR measurement per reduction stage.
- Store per-stage outcomes needed for Player B reconstruction.

11.6 Class AssistedPlayerB(PlayerB)

11.6.1 Purpose

Use gun position, received comm, and shared randomness outcomes to reconstruct the relevant field bit.

11.6.2 Behavioral contract (normative)

- MUST NOT access full field information.
- MUST mirror Player Aâ€™s SR consumption order (one SR measurement per stage).

- Gun input MUST remain one-hot throughout any internal transformations.
- Output MUST be binary at inference time.

11.6.3 Algorithmic outline (informative)

- Traverse the reduction hierarchy guided by gun position.
- Combine SR outcomes and comm to compute shoot.

11.7 Invariants

- For the standard assisted protocol: comms_size == 1 and nÂ^2 is a power of two.
- Each SR resource is measured at most once per party per game.

12 Neural players (unassisted)

12.1 Purpose

Define unconstrained neural-network-based players that do not use shared randomness. These players serve as comparison baselines against assisted strategies.

12.2 Characteristics

- Communication limited to comms_size
- No shared randomness
- Fully trainable via reinforcement learning or supervised learning

12.3 Behavioral constraints

- Player A MUST NOT access gun information
- Player B MUST NOT access full field information
- Communication bandwidth MUST be respected

12.4 Notes

Neural players may violate classical assisted performance bounds, but serve as useful empirical baselines.

13 Shared randomness resource (non-classical)

13.1 Purpose

Define a two-party shared-randomness resource that produces **non-classical correlations** between Player A and Player B outcomes, without increasing communication bandwidth.

This specification defines the **behavioral contract** of shared randomness. Multiple implementations (e.g., a pure Python class and a Keras layer) MUST satisfy the same contract.

13.2 Location

- **Python implementation (example):** src/Q_Sea_Battle/shared_randomness.py â€” class SharedRandomness
- **Differentiable implementation (example):** src/Q_Sea_Battle/shared_randomness_layer.py â€” class SharedRandomnessLayer

13.3 Parameters

Name	Type	Constraints	Description
length	int	> 0	Number of correlated outcome bits produced per measurement
p_high	float	[0, 1]	Correlation strength parameter

13.4 Public interface (conceptual)

13.4.1 reset() -> None

Reset per-game state so the resource can be used again.

13.4.2 measurement_a(measurement: array_like) -> np.ndarray

Player A performs a measurement and receives a binary outcome vector of length length.

13.4.3 measurement_b(measurement: array_like) -> np.ndarray

Player B performs a measurement and receives a binary outcome vector of length length.

> The measurement input is a *measurement choice* (a â€œsettingâ€) and MAY be encoded as a binary vector, integer index, or tensor â€” but it MUST be documented by the implementation and used consistently by both parties.

13.5 Behavioral contract (normative)

13.5.1 Single-use per party

- Each party MUST call its measurement method at most once per resource instance per game.
- Calling measurement_a twice, or measurement_b twice, MUST raise an error.

13.5.2 Two-measurement lifecycle

- The resource supports up to two measurements total (one per party) between resets.

13.5.3 Output format

- Outcomes MUST be binary arrays in {0,1} of shape (length,) (or (B, length) for batched/tensor implementations).

13.5.4 Non-classical correlation requirement

Let x be Player Aâ€™s outcome, y Player Bâ€™s outcome (bit-wise).
The joint distribution MUST satisfy:

- P(x_i = y_i) = p_high
- P(x_i â‰ y_i) = 1 - p_high

for each bit index i, **conditional on both parties using the same measurement choice**.

If measurement choices differ, the implementation MUST still: - return valid binary outcomes, and - produce correlations that are a documented function of both measurement choices.

> This clause is what makes the resource â€œnon-classicalâ€ in this project: the correlation structure is an explicit, tunable contract, and not restricted to classical shared coins.

13.5.5 No information leakage

- The resource MUST NOT depend on field, gun, or any game secrets.
- It MUST NOT encode additional side channels beyond its specified outputs.

13.6 Failure modes

- ValueError (recommended) if a party measures twice.
- RuntimeError if used without reset across games (implementation-dependent).

13.7 Notes (informative)

Two compliant implementations are common: 1. **Pure Python**: explicit sampling of correlated bits with the required joint distribution. 2. **Keras layer**: a differentiable proxy used during training that matches the same marginal/correlation behavior.

14 SharedRandomnessLayer (differentiable implementation)

14.1 Purpose

Provide a differentiable implementation of the **Shared randomness resource (non-classical)** contract, to enable gradient-based training (imitation, DRU/DIAL, RL variants) while preserving runtime semantics.

This layer is an **implementation** of the shared randomness contract; it does not define new behavior.

14.2 Location

- **Module:** src/Q_Sea_Battle/shared_randomness_layer.py
- **Class:** SharedRandomnessLayer

14.3 Inputs / Outputs

Implementations MAY support: - Unbatched (length,) inputs and outputs - Batched (B, length) inputs and outputs - TensorFlow tensors throughout

Outcomes MUST be interpretable as binary at runtime: - During training: may use continuous relaxations (e.g., logits, straight-through estimators) - During inference/evaluation: MUST produce discrete {0,1} outcomes (or be replaced by Python SharedRandomness)

14.4 Behavioral contract

The layer MUST satisfy all requirements from: - docs/shared_randomness/shared_randomness.

Additionally: - The differentiable relaxation MUST NOT increase the number of communicated bits. - Any temperature/annealing schedule MUST be documented and controlled by training utilities.

14.5 Recommended modes (informative)

- mode="train": differentiable relaxation (logits / sigmoid / straight-through)
- mode="eval": discrete sampling consistent with p_high

14.6 Failure modes

- Shape mismatch MUST raise ValueError.

15 Trainable assisted models â€” Linear (Lin)

15.1 Purpose

Provide a trainable approximation of the classical assisted strategy using linear neural network layers, preserving all behavioral contracts defined for assisted players.

15.2 Scope

This chapter specifies the **Lin** family of trainable assisted models. These models replace fixed logical operations with learned linear mappings while maintaining the same information-theoretic constraints.

15.3 Class LinTrainableAssistedModelA

15.4 Purpose

Approximate the classical AssistedPlayerA strategy using a single linear measurement layer and a linear combine layer.

15.5 Location

- **Module:** src/Q_Sea_Battle/lin_trainable_assisted_models.py
- **Class:** LinTrainableAssistedModelA

15.6 Inputs

Name	Shape	Description
field	(B, nÂ^2)	Binary field vector

15.7 Outputs

Name	Shape	Description
comm	(B, 1)	Communication bit

15.8 Internal state

- measurements_per_layer: list of tensors
- outcomes_per_layer: list of tensors

15.9 Behavioral contract

- Exactly **one measurement** is performed per decision.
- Exactly **one communication bit** is produced.
- The model **MUST NOT** use gun information.
- Shared randomness is accessed via a differentiable proxy.

15.10 Invariants

- Input length nÂ^2 **MUST** be a power of two.
- comms_size == 1.

15.11 Class LinTrainableAssistedModelB

15.12 Purpose

Approximate the classical AssistedPlayerB strategy using linear layers to reconstruct the relevant field bit.

15.13 Location

- **Module:** src/Q_Sea_Battle/lin_trainable_assisted_models.py
- **Class:** LinTrainableAssistedModelB

15.14 Inputs

Name	Shape	Description
gun	(B, nÂ^2)	One-hot gun vector
comm	(B, 1)	Communication bit

15.15 Outputs

Name	Shape	Description
shoot	(B, 1)	Binary decision

15.16 Behavioral contract

- Exactly **one measurement** is performed per decision.
- Output **MUST** be binary at inference time.
- The model **MUST NOT** access the field directly.

15.17 Invariants

- Gun input **MUST** be one-hot.
- Decision depends only on (gun, comm, shared randomness).

15.18 Notes / rationale

The Lin models serve as the minimal trainable baseline. They are intentionally capacity-limited to: - Test learnability under strict information constraints - Provide a reference point for deeper architectures (Pyr)

All violations of classical-assisted invariants are considered specification errors.

16 Trainable assisted models â€” Pyramid (Pyr)

16.1 Purpose

Define a hierarchical, multi-layer trainable assisted model that scales the classical assisted strategy to large fields by iteratively reducing the problem size while preserving all assisted-player contracts.

The Pyramid (Pyr) models generalize the Lin models by composing **repeated measurementâ€“combine stages** in a strictly structured hierarchy.

16.2 Conceptual overview

Let the field have size n^2, with $n^2 = 2^k$. The Pyr architecture performs k iterations.
At each iteration: - The effective field size is halved - One shared-randomness measurement is consumed - Intermediate outcomes are stored for later reconstruction

Exactly **one communication bit** is produced at the final layer.

16.3 Class PyrTrainableAssistedModelA

16.4 Purpose

Approximate the classical AssistedPlayerA pyramid strategy using trainable neural layers, producing a single communication bit from a large field.

16.5 Location

- **Module:** src/Q_Sea_Battle/pyr_trainable_assisted_models.py
- **Class:** PyrTrainableAssistedModelA

16.6 Inputs

Name	Shape	Description
field	(B, nÂ^2)	Binary field vector

16.7 Outputs

Name	Shape	Description
comm	(B, 1)	Communication bit

16.8 Internal state

The model **MUST** store the following per decision:

- measurements_per_layer[i]: measurement tensor at layer i
- outcomes_per_layer[i]: shared-randomness outcomes at layer i

where: - Layer 0 operates on size n^2 - Layer $i+1$ operates on size $n^2/2^{i+1}$

16.9 Iterative structure (normative)

For layer $i = 0, \ldots, k-1$:

1. **Measurement layer**
   - Input size: $n^2/2^i$
   - Output size: $n^2/2^{i+1}$
2. **Shared randomness**
   - One measurement is performed
   - Outcome size: $n^2/2^{i+1}$
3. **Combine layer**
   - Inputs: measurement output + shared randomness
   - Output: reduced field for next layer

At the final layer, the reduced field has size 1 and is emitted as comm.

16.10 Behavioral contract

- Exactly **one shared-randomness measurement per layer**
- Exactly **one communication bit total**
- No access to gun information
- No skipping or reordering of layers

16.11 Invariants

- Input size **MUST** be a power of two
- Number of layers = $\log_2(n^2)$
- Lengths of measurements_per_layer and outcomes_per_layer **MUST** equal number of layers

16.12 Failure modes

- ValueError if input size is not a power of two
- RuntimeError if layers are executed inconsistently

16.13 Class PyrTrainableAssistedModelB

16.14 Purpose

Reconstruct the relevant field bit using hierarchical outcomes, gun position, and a single communication bit.

16.15 Location

- **Module:** src/Q_Sea_Battle/pyr_trainable_assisted_models.py
- **Class:** PyrTrainableAssistedModelB

16.16 Inputs

Name	Shape	Description
gun	(B, nÂ^2)	One-hot gun vector
comm	(B, 1)	Communication bit

16.17 Outputs

Name	Shape	Description
shoot	(B, 1)	Binary decision

16.18 Internal state

- outcomes_per_layer[i]: outcomes received from Player A
- Optional learned reconstruction layers per hierarchy level

16.19 Reconstruction procedure (normative)

For layer $i = k-1, \dots, 0$:

1. Use gun position to select the relevant subtree
2. Combine:
    - Corresponding shared-randomness outcome
    - Current reconstruction state
    - Communication bit (only at final stage)

The final output is a scalar decision.

16.20 Behavioral contract

- Exactly **one shared-randomness measurement per layer**
- Gun input **MUST** remain one-hot throughout reconstruction
- Output **MUST** be binary at inference time

16.21 Invariants

- Reconstruction depth equals measurement depth
- No access to full field information

16.22 Notes / rationale

The Pyr architecture: - Scales logarithmically with field size - Mirrors the classical pyramid protocol - Makes information flow explicit and inspectable

Any deviation from the layer structure constitutes a **spec violation**.

17 Imitation learning for assisted models

17.1 Purpose

Train Lin (and later Pyr) assisted models to imitate the optimal classical assisted strategy using supervised learning.

17.2 Data generation

- Generate (field, gun) samples using GameEnv
- Generate optimal (comm, shoot) targets using classical AssistedPlayers

17.3 Behavioral contract

- Training data **MUST** be generated from a spec-compliant classical policy.
- Training **MUST NOT** introduce additional information channels.

17.4 Losses

- Binary cross-entropy for communication bit
- Binary cross-entropy for shooting decision

17.5 Invariants

- Dataset inputs and targets **MUST** match the shapes defined in Chapter 7.
- Training **MUST NOT** alter the runtime contracts of the models.

18 DRU / DIAL training

18.1 Purpose

Specify training using Discretizeâ€“Regularize Units (DRU) and Differentiable Inter-Agent Learning (DIAL) for communication-constrained agents.

18.2 Algorithm (informative)

- Replace hard binary communication with differentiable proxies
- Regularize outputs toward discrete values
- Anneal regularization during training

18.3 Constraints

- Runtime communication MUST remain discrete
- DRU/DIAL MUST NOT increase communication bandwidth
- Learned policies MUST satisfy all player contracts

18.4 Notes

DRU/DIAL is optional and experimental in QSeaBattle.

19 Reinforcement learning

19.1 Purpose

Specify reinforcement-learning-based training regimes for players, primarily for baseline comparison.

19.2 Supported approaches

- Policy gradients
- Self-play
- Actorâ€“critic methods

19.3 Constraints

- RL MUST respect all information-flow constraints
- Reward signals MUST come exclusively from GameEnv
- Exploration MUST NOT leak additional information

19.4 Notes

Reinforcement learning is not the primary focus of QSeaBattle, but provides useful empirical benchmarks.

20 Training considerations for Pyramid (Pyr) models

20.1 Purpose

Specify additional constraints and recommendations for training Pyramid assisted models.

20.2 Training regimes

- Supervised imitation (from classical pyramid policy)
- Layer-wise pretraining (optional)
- End-to-end fine-tuning (allowed but constrained)

20.3 Behavioral constraints

- Training **MUST NOT** introduce cross-layer shortcuts
- Shared-randomness usage **MUST** remain one-per-layer
- Communication bandwidth **MUST** remain 1 bit

20.4 Diagnostics (recommended)

- Per-layer loss monitoring
- Consistency checks between adjacent layers
- Ablation of individual layers

20.5 Invariants

- Runtime behavior **MUST** match Chapter 8 contracts
- Learned weights **MUST NOT** change input/output interfaces

21 Utility module: Data generation

21.1 Purpose

Provide deterministic, spec-compliant utilities for generating datasets used in training and evaluation, without introducing any additional information channels.

21.2 Scope

These utilities are used exclusively for **offline data generation**. They MUST NOT be used at runtime during actual games or tournaments.

21.3 Core utilities

21.3.1 generate_measurement_dataset_a(layout, num_samples, seed)

Generate training samples for Player A measurement layers.

Inputs - layout: GameLayout - num_samples: int - seed: int

Outputs - Dataset of (field, meas_target)

Contract - Targets MUST be generated using classical AssistedPlayerA - No gun information is accessible

21.3.2 generate_measurement_dataset_b(layout, num_samples, seed)

Generate training samples for Player B measurement layers.

Contract - Targets MUST be generated using classical AssistedPlayerB - Field information MUST NOT be accessible

21.4 Invariants

- Dataset shapes MUST match model interfaces
- Random seeds MUST make generation reproducible

22 Utility module: Imitation training

22.1 Purpose

Define helper utilities to train trainable assisted models using supervised imitation learning, while preserving all runtime behavioral contracts.

22.2 Core utilities

22.2.1 train_layer(layer, dataset, loss, epochs, metrics=None)

Train a single neural layer in isolation.

Contract - The utility MUST NOT alter layer input/output signatures - The utility MUST NOT introduce cross-layer information flow

22.2.2 train_model(model, datasets)

Orchestrate training of full Lin or Pyr models.

Contract - Training MUST respect per-layer constraints (Chapter 7, 8) - Runtime behavior MUST remain unchanged after training

22.3 Invariants

- Training utilities MUST NOT modify game logic
- All randomness MUST be controlled and logged

23 Utility module: Layer and weight transfer

23.1 Purpose

Provide safe utilities for transferring trained weights from standalone layers into composite assisted models.

23.2 Core utilities

23.2.1 transfer_assisted_model_a_layer_weights(...)

Copy trained Player A layers into a composite model.

23.2.2 transfer_assisted_model_b_layer_weights(...)

Copy trained Player B layers into a composite model.

23.3 Behavioral contract

- Only weights may be transferred
- Model topology MUST remain unchanged
- Any shape mismatch MUST raise an error

23.4 Invariants

- Post-transfer model MUST satisfy original spec contracts

24 Utility module: Evaluation

24.1 Purpose

Provide helpers for evaluating trained players using tournaments and aggregating results.

24.2 Core utilities

24.2.1 evaluate_players(players, layout)

Run a tournament and return performance statistics.

24.2.2 confidence_interval(mean, std_error)

Compute confidence intervals for reported results.

24.3 Contract

- Evaluation MUST NOT alter player state
- Evaluation MUST be reproducible given fixed seeds

24.4 Invariants

- Results MUST correspond to exactly the games played

25 Invariants

25.1 Purpose

This page lists **global invariants**: one-sentence truths that MUST hold across the QSeaBattle codebase. These are intended to be: - easy to audit during code review, - easy to test, - and stable over time.

25.2 Game and environment

- The field and gun vectors MUST have identical shape (nÂ^2,) (or (B, nÂ^2) batched).
- The gun vector MUST be one-hot: gun.sum() == 1.

- Channel noise MUST be applied only to comm, never to field or gun.
- Rewards MUST be deterministic given (field, gun, shoot).

25.3 Communication

- Communication bandwidth is exactly comms_size bits per game.
- If comms_size == 1, communication MUST still be represented as shape (B, 1) (never a scalar).

25.4 Assisted / shared randomness

- Shared randomness MUST NOT be treated as an extra communication channel.
- Each shared-randomness resource MAY be measured at most once by each party.
- For pyramid strategies, exactly one shared-randomness measurement is consumed per layer per party.

25.5 Trainable assisted models (Lin/Pyr)

- Player A models MUST NOT access gun information.
- Player B models MUST NOT access the full field information.
- Any learned model MUST preserve the same input/output interfaces as the classical policy it approximates.
- Model state required for reconstruction (e.g., per-layer outcomes) MUST be captured during Player Aâ€™s decision and used by Player B; it MUST NOT be recomputed using hidden information.

25.6 Data generation and training

- Imitation-learning targets MUST be produced by a spec-compliant classical policy.
- Training utilities MUST NOT introduce new information paths (no shortcuts, no extra inputs).
- Weight transfer utilities MUST NOT change topology; only parameter values may be copied.

25.7 Logging and evaluation

- Each tournament log row MUST correspond to exactly one game.

- Evaluation utilities MUST NOT mutate player behavior beyond documented resets.

25.8 Testability expectation (recommendation)

- Every invariant above SHOULD have at least one unit test or property test that would fail if the invariant is violated.