Hand Me Down Clothing Documentation

Table of Contents

1 Informative Part

1.1 Team

The team is organized into key functional areas with dedicated leads overseeing documentation and requirement completion, authentication, listings, and search and map integration, under the guidance of three project managers.

Managers

- Anthony Martinez
- Alma Pi–eiro
- Jahsyel Rojas

Team 1 - Documentation & Requirements

- Joshua D‡vila (Lead)
- Ojani Figueroa
- Giovanny Garc’a
- Juan Iranzo

Team 2 - Authentication & User Accounts

- Lorenzo PŽrez (Lead)
- Jessy Andœjar
- Gabriel Marrero
- Luis Marrero
- ç ngel Villegas

Team 3 - Listings

- Kevin G–mez (Lead)
- Leanelys Gonz‡lez
- Karina L–pez
- Nicol‡s Rivera
- Jachikasielu Uwakweh

Team 4 - Search & Map Integration

- Jorge De Le–n (Lead)
- Devlin Hahn

- Alejandro Marrero
- Kian Ramos
- Januel Torres

Team 5 - UI/UX & Branding

- Fabiola Torres (Lead)
- Yamilette Alema–y
- Daniella Melero
- Andrea Segarra
- Kenneth Sepœlveda

1.2.1 Current Situation

Landfills and textile waste in Puerto Rico

- High waste volume: Approximately 250 million pounds of clothing and textiles are sent to Puerto Rico◊s landfills annually.
- High landfill rate: Similar to the global and U.S. trends, a very high percentage of discarded textiles, around 85%, end up in landfills, despite being largely recyclable.
- Low recycling rates: Puerto Rico◊s overall recycling rate is notably low, with some reports estimating it to be less than 10%. This is significantly lower than the U.S. national average.
- Overwhelmed landfills: Puerto Rico◊s landfills are facing a serious crisis, with many already at or over capacity. The high volume of textile waste contributes to this problem.

Sources:

- investpr.org
- theenvironmentalblog.org

Poverty

- Overall Poverty: The poverty rate in Puerto Rico is alarmingly high, at around 41.7% as of 2022. This is over three times the U.S. national average.
- Child Poverty: A staggering 54.3% to 57.6% of children under 18 in Puerto Rico live in poverty. This is more than any U.S. state and indicates that a vast number of families cannot afford to consistently provide their children with properly fitting, weather-appropriate clothing and shoes.
- Persistent Poverty: All 78 municipalities in Puerto Rico are classified as "persistent poverty counties," meaning they have maintained a poverty rate of 20% or more for at least 30 years.

Sources:

- centropr.hunter.cuny.edu

- geopoliticaleconomy.com

Homelessness:

- Homeless Population: Recent counts show the homeless population in Puerto Rico to be around 2,096 individuals. For these individuals, clothing is a constant and critical need.

Sources:

- periodismoinvestigativo.com

1.2.2 Need

The purpose of this section is to establish the fundamental needs that motivate the Hand Me Down project, expressed independently of any system-to-be. These needs are grounded in the resale domain and must reflect the concerns of students and families who participate in secondhand exchanges. The articulation of these needs will guide the subsequent development of domain descriptions, requirements, software architecture, and testing activities.

Stakeholders in this domain shall be understood as students and families seeking opportunities for affordable, accessible, and trustworthy secondhand exchanges. Their needs are not for a platform itself, but for solutions to the problems they encounter when attempting to exchange goods in local communities.

The following distinct needs are identified:

- Students and families must have affordable access to secondhand goods that support daily life, education, and well-being.
- Stakeholders must be able to rely on transparent information about the condition and history of pre-owned items.
- Exchanges shall be conducted in a manner that establishes trust, fairness, and safety between participants.
- Opportunities for accessibility and inclusivity must be available so that all families and students, regardless of economic background, will participate in the resale domain without barriers.
- Developers shall have clear requirements, descriptions, and architecture to build upon, since no structured system currently exists to organize this resale context.

These needs form the foundation for further project work. They are deliberately expressed at the domain level, independently of any particular solution, to ensure that subsequent design and implementation activities will remain aligned with the stakeholders underlying motivations.

1.3.1 Scope & Span

Scope

The Hand Me Down project will operate in the broad domain of online resale marketplaces. It will

address the general problem of enabling individuals and communities to exchange secondhand goods in a structured, reliable, and sustainable manner. The scope will cover activities in domain engineering, requirements engineering, and software architecture to ensure a well-founded solution.

The project will emphasize the following areas:

- Domain:: Resale of pre-owned items across categories such as clothing and accesories.
- Requirements:: Identifying user needs related to affordability, sustainability, accessibility, and usability.
- Architecture:: Defining a framework that supports secure and scalable interactions between sellers and buyers.
- Project Activities:: Documentation, validation, and design processes that must accompany implementation.

Span

The span narrows the focus of the Hand Me Down project to specific concerns and audiences within the general resale domain. The platform must primarily serve individuals and families who wish to exchange items affordably, students and young adults seeking budget-friendly goods, and community members interested in sustainable consumption.

The span includes the following project-specific aspects:

- User Interaction:: Individuals must be able to list, browse, and search for secondhand items.
- Categorization:: Items will be organized into categories that facilitate discovery.
- Transaction Support:: The system must provide structured means for creating and viewing listings, including optional prices or donation markers. Negotiation and exchange are arranged outside the platform.
- Trust and Transparency:: Item conditions and relevant metadata must be clearly described to support informed decisions.

1.3.2 Synopsis

This Synopsis provides overview of the Hand Me Down project from the perspective of students and families engaged in secondhand exchanges. It articulates the domain, affordability, accessibility, trust, safety and states that stakeholders must be able to discover, evaluate, and exchange pre-owned goods with transparent information about item condition and history. The project shall be conducted through structured domain acquisition to produce a domain description, a requirements prescription that specifies goals, constraints, and quality attributes with traceability to stakeholder needs and a software architecture that evaluates alternatives and justifies decisions, prototyping where necessary to mitigate risk. Component design and iterative implementation will realize prioritized capabilities while preserving traceability. Verification and validation shall include a test plan that covers functional fitness, usability, and trust/safety concerns, supported by versioned documentation, change control, risk tracking, and metrics.

1.4 Other Activities than Just Developing Source Code

This project will not be limited to writing source code. To satisfy the needs identified for affordability, accessibility, trust/safety, and transparency in the MayagŸez/UPRM context, the team shall execute the following activities in addition to implementation. Each activity is mandatory, tied to a stakeholder need and justified by the current situation.

Domain engineering

The team shall elicit and model the domain of donation and resale of clothing and accessories for students and families (actors, workflows, vocabulary, constraints).

- ¥ Need satisfied: developers must share a precise vocabulary and mental model to preserve accessibility, affordability, and transparency.
- ¥ Current situation: rough sketches are based on team self-observation; no external interviews conducted yet; stakeholder roles are not enumerated; internal terminology seems consistent but remains unvalidated in the field.
- ¥ Contributions to date: condition labels and verification practices were researched; the Sell vs Donate category structure was standardized; initial personas were drafted to ground concepts.
- ¥ Planned outcomes: domain glossary, context diagram, concrete exchange workflows, and domain verification norms (tag photo, full-item photos, defect call-outs).

Requirements engineering

The team shall prescribe goals, user-level functional requirements (listing, browse/search, donation/resale flows, offer/negotiation) and quality attributes (trust/safety, transparency, usability, accessibility) with explicit traceability to needs and acceptance criteria.

- ¥ Need satisfied: developers must clearly understand system functionality and nonfunctional expectations that build trust and reduce effort.
- ¥ Current situation: no consolidated requirements baseline or acceptance criteria exist.
- ¥ Contributions to date: epics were linked to user stories (buyer/seller perspectives) against stakeholder needs; interface requirements at the system boundary were authored; measurable machine requirements (response time, uptime, user capacity) were drafted.
- ¥ Planned outcomes: requirements set with SHALL statements and acceptance criteria; traceability matrix (Need ! Requirement ! Test); nonfunctional thresholds made testable.

Software architecture

The team shall select and justify an architecture addressing security, privacy, modifiability, and campus-scale usage; architectural views and decision records will be maintained.

- ¥ Need satisfied: stakeholders must receive a reliable, maintainable basis for transparent, safe exchanges.
- ¥ Current situation: direction is leaning toward Supabase for authentication (Google sign-in compatibility) with a full custom backend; no UPRM hosting/privacy constraints identified; ADRs and C4 views are not yet written.
- ¥ Contributions to date: search and map integration approaches were evaluated and

geolocation/privacy considerations documented; authentication backends (Firebase vs. Supabase) and session-management implications were analyzed; page-level layouts were produced to inform view composition and navigation flows.

- Planned outcomes: C4 views (context/container), Architectural Decision Records (auth, data, map/search), quality-attribute scenarios, and targeted risk spikes where uncertainty is high.

Component design

The team shall define modules, interfaces, and data contracts to preserve testability and changeability (catalog/search, profiles/auth, exchange/offer, reporting/moderation, Òitems circulatedÓ metrics).

- Need satisfied: maintainable, verifiable components are required to deliver transparency (clear item/condition data) and accessibility (predictable flows).
- Current situation: boundaries and interfaces are only partially documented.
- Contributions to date: the user/auth schema (roles, profile fields, donation/sell history) was initiated and APIs for login/registration/logout were defined; search behavior and map-related data interactions were documented; wireframes and page designs (Homepage, About, Clothes Listing, Individual Item, Favorites, Checkout, Log In/Sign Up, Profile) clarify interface responsibilities and data needs.
- Planned outcomes: module responsibilities, interface specs (inputs/outputs/preconditions/postconditions), initial schemas with migration notes, and example queries.

Implementation planning

The team will establish a delivery roadmap, Definition of Done, contribution standards, and a branching/CI strategy suitable for a student-run service.

- Need satisfied: predictable, reviewable progress must be ensured to realize stakeholder value without regressions.
- Current situation: a branch is created per team with controlled pull requests and merges to main when working; no CI pipeline is in place; review checklist/PR template usage is minimal.
- Contributions to date: the AsciiDoc documentation structure and conventions were established; documentation issues with acceptance criteria were created; a step-by-step Node.js/npm installation and verification guide was produced to bootstrap the development environment.
- Planned outcomes: roadmap with dates/owners, Definition of Done, CONTRIBUTING guidelines, PR template, and CI workflow for lint/tests on pull requests.

Testing and validation

The team shall produce a test plan spanning unit, integration, end-to-end, and usability/acceptance with students and families; trust/safety validations shall be included (e.g., prohibited items policy, condition/fit etiquette).

- Need satisfied: stakeholders must have confidence that behavior and quality attributes match the prescription.
- Current situation: no automated tests exist and the test plan is not started;

usability/acceptance testing is deferred to later milestones; recruitment will be informal (Òask around UPRMÓ); top priority requirements for acceptance scenarios are not selected yet.

- ¥ Contributions to date: machine requirements were expressed in measurable terms (e.g., support ~100 simultaneous users) to anchor performance testing; trust-building practices (verification methods, condition labels) were documented to translate into validation checks.
- ¥ Planned outcomes: test plan, seeded test suites, acceptance scenarios (GivenÐWhenÐThen) mapped to requirements, trust/safety validations, and a defect taxonomy with triage protocol.

Deployment considerations

The team will define dev/staging environments, configuration, seed/reset data, rollback procedures, and a minimal operations runbook for a student-operated service.

- ¥ Need satisfied: availability and safe adoption are necessary for accessibility and affordability benefits to materialize.
- ¥ Current situation: no documented path to deploy, recover, or roll back; backend selection work is informing environment and secrets management but is not yet consolidated in docs.
- ¥ Contributions to date: Docker usage was explored to standardize developer environments and setup was documented; backend evaluations (Supabase/Firebase) inform environment and secret management decisions.
- ¥ Planned outcomes: environment definitions, secrets/config guidance, release/rollback steps, seed scripts, and a basic ops runbook.

Stakeholder liaison and feedback (cross-cutting)

The team shall schedule and document periodic touchpoints with students and families in MayagŸez to validate assumptions early (quotes/anecdotes shall be recorded in ¤2.1.1).

- ¥ Need satisfied: continuous alignment is required to keep requirements correct and trust high.
- ¥ Current situation: no formal liaison cadence is defined; external interviews are not planned at this time; consent/ethics approach is undefined.
- ¥ Planned outcomes: contact cadence, feedback and decision logs, and lightweight consent notes for any future interactions.

Documentation & governance (cross-cutting)

The team shall maintain versioned documentation, change control, risk tracking, and metrics to ensure durable traceability across activities.

- ¥ Need satisfied: traceability is required to justify decisions and onboard contributors without rework.
- ¥ Current situation: a docs index/navigation page is considered established (initial draft); changelog and risk register are not started; project metrics beyond Òitems circulatedÓ are not yet defined.
- ¥ Contributions to date: the docs/ layout and AsciiDoc style were standardized and most documentation issues were created; branding (logo, color palette, typography) was

established to keep artifacts consistent and legible.

- Planned outcomes: docs index and navigation (maintained), changelog, risk register (e.g., technology choice risk, schedule slip, data/privacy misconfiguration), and basic metrics (e.g., items circulated as a primary signal).

1.5 Derived Goals

Derived goals provide higher-level motivations that inform the design of epics and features. Each derived goal links to the epics it directly influences.

DG-1: Promote sustainability literacy and circular practices in MayagŸez The project shall normalize reuse, repair, and responsible disposal behaviors among students and families through donation and resale norms. Supports: [EPIC-1], [EPIC-4] Broader impact: item lifecycles will be extended and textile waste pressure will be reduced.

DG-2: Strengthen community engagement and mutual aid through UPRM-led outreach The project will cultivate equitable sharing practices (donation, fair resale) centered on UPRM as the primary touchpoint. Supports: [EPIC-2], [EPIC-5], [EPIC-6] Broader impact: social capital will increase and households will respond more effectively to clothing and accessory needs.

DG-3: Raise awareness of affordability and access constraints faced by local households The project shall make visible how structured sharing reduces acquisition cost and effort for students and families. Supports: [EPIC-1], [EPIC-3] Broader impact: schools and neighborhood groups will make more informed choices about drives and targeted outreach.

These derived goals guide outreach, education, and validation activities alongside the primary objectives.

2.1 Domain Description

The domain of hand-me-down clothing exchange in Puerto Rico is shaped by social, environmental, and economic realities. It exists independently of any digital platform or technical system and can be understood through the people, practices, artifacts, and norms that sustain the circulation of clothing and accessories among students, families, and local communities.

At its core, the domain revolves around two primary actors:

- Sellers: Individuals or households who post garments they no longer need, offering them for reuse or resale.
- Buyers: Individuals who discover these garments through listings and arrange exchanges directly with the seller, outside the system.

The central entity in this domain is the Piece, any individual article of clothing or accessory that circulates between actors. Each Piece carries attributes such as:

- Type: The category of clothing.
- Condition Rating: A measure of quality or usability.

Meanwhile, a Listing is a published representation of a Piece in the platform. Listings move between states through events:

- Listing Published: Occurs when a Seller makes a garment visible to potential Buyers.
- Interest Expressed: Occurs when a Buyer contacts a Seller regarding a listed garment.
- Listing Closed: Occurs when a garment has been exchanged offline or removed from visibility.
- Discard Event: Occurs when a Piece leaves circulation, feeding into the wider Textile Waste Stream.

The domain is sustained by informal practices and behaviors:

- Discovery Flows: The recurring sequence in which garments are listed, browsed, and handed over in person.
- Condition Disclosure Norms: The expectation that Sellers will show tag photos, highlight defects, and represent garments honestly.
- Informal Price Bands: Symbolic or suggested valuations (typically USD $8Đ$15), distinguishing resale from donation or commercial sale.
- Dormant Stock: Clothing stored in homes awaiting redistribution, resale, or disposal.
- Meetup Spots: Semi-public locations (e.g., campus benches, apartment lobbies) chosen for exchanges.
- Ad-hoc Channels: Informal digital venues (e.g., Facebook Marketplace, WhatsApp groups, Instagram stories).
- Trust Cues: Bilingual communication, recognizable names, or clear photos that influence whether an exchange proceeds.
- Seasonal Demand Pulses: Cyclical increases in demand, such as back-to-school or weather-driven surges for uniforms or outerwear.

These elements form an interconnected web of events, actions, and behaviors. Sellers and Buyers rely on ad-hoc digital channels for discovery, agree on terms through direct contact, and meet in semi-public spaces to exchange items offline. Exchanges are underpinned by implicit rules of trust and fairness, while also constrained by larger social and environmental forces such as poverty rates, limited recycling infrastructure, and overflowing landfills.

From a functional perspective, the domain can be represented through abstract operations:

- `publishListing(Piece, Seller, Locale) ! ListingPublished`
- `bookmarkListing(Listing, Buyer) ! ListingBookmarked`
- `expressInterest(Listing, Buyer) ! InterestExpressed`
- `rate(Piece, ConditionRating) ! ConditionRating`
- `submitReview(Seller, Buyer, Review) ! ReviewSubmitted`
- `categorize(Piece, Type) ! Piece`
- `closeListing(Listing) ! ListingClosed`

Each function captures an action that transforms the state of a Listing or Piece within the domain, producing observable changes such as entering circulation, connecting interested parties, or leaving visibility once the exchange occurs offline.

In sum, the domain of secondhand clothing discovery in Puerto Rico is defined by the visibility of Pieces, the actors who list and find them, the events that mark transitions, and the behaviors and norms that make these exchanges trustworthy, affordable, and sustainable. This description provides a foundation for later requirements and design work, while remaining independent of any specific system or implementation.

2.1.1 Domain Rough Sketches

This section documents raw, observable examples of clothing exchange and discovery as they occur in daily life. These anecdotes are early insights and do not interpret or generalize; they simply record what was seen or described.

Example 1: Adrianaôs Search for Affordable Outfits

Adriana, a 20-year-old UPRM student, needs new outfits for an upcoming presentation but cannot afford retail prices. She opens a local Facebook group called ÒUPRM Ropa y Accesoriosó and scrolls through the feed. She spots a post titled ÒBlazer, lightly used, $10ó with photos showing the tag and sleeves. Adriana messages the seller in Spanish: ÒÀTodav'a lo tienes disponible?ó The seller responds quickly and confirms that the blazer is still available. They agree to meet at the main campus entrance the next afternoon. Adriana checks the garment in person, pays in cash, and the seller later marks the post as ÒSold.ó The entire flowÑscrolling, messaging, coordinating, and closingÑhappened outside any dedicated app, relying solely on informal trust and visible cues like clear photos and quick replies.

Example 2: Manuelôs Donation After Semester End

Manuel, a middle school teacher and UPRM alumnus, sorts his closet at the end of the semester. He finds several shirts in good condition but no longer wears them. He takes photos, labels them ÒFree for pickup near UPRM apartmentsó, and posts them in a WhatsApp group where local students trade items. Within hours, a student replies: ÒCan I pick up tomorrow morning?ó Manuel agrees, places the items in a bag labeled ÒFree clothes,ó and leaves them at his apartment lobby. By afternoon, the bag is gone. No money exchanged hands, and no platform intervention was needed Ñ only quick, direct communication and mutual trust.

Observed Raw Facts

- Listings often use informal phrases like Òlightly used,ó Òlike new,ó and Òfree.ó
- Meetups happen in semi-public, familiar places.
- Sellers use photos as proof of honesty (tags, condition, size).
- Prices are symbolic and flexible ($8Đ$15 typical range).
- Listings and messages are bilingual; Spanish often dominates the first contact.
- Exchanges rely on visibility and responsiveness rather than any formal guarantee.

Observed Item Lifecycle

Real-world clothing circulation in the MayagŸez and UPRM community is not linear. The same item may move through several stages repeatedly as different people use it or store it.

Below is a raw-domain sketch of this recurring cycle:

```
Ê      Active Use
Ê           |
Ê           v
Ê       Storage
Ê   (Dormant Period)
Ê           |
Ê           v
Ê Resale or Donation
Ê    (Relisting)
Ê           |
Ê           v
Ê       Exchange
Ê    (In-person)
Ê           |
Ê           v
Ê ! Returns to Active Use
Ê   by a new owner
Ê   (cycle repeats)
```

Items often circulate through several owners. A shirt donated one semester may reappear in a resale group months later, reenter storage, and be listed again by a new owner. This repeating loopÑuse ! store ! relist ! exchange ! new useÑcontinues until the garment is no longer wearable and is ultimately discarded or repurposed.

2.1.2 - Terminology

The following terminology consolidates entities, events, functions, and behaviors in the domain. Each entry specifies the type of concept it represents and the phase in which it is introduced (domain, requirements, design, implementation). This approach avoids circular definitions and ensures alignment with both domain knowledge and system concerns.

Term	Concept Type	Phase Introduced	Definition / Notes
User	Entity	Domain	A person who interacts with the platform that can assume the role of a Seller or Buyer.
Seller	Role	Domain	Social role representing an actor who offers Pieces for reuse or resale.
Buyer	Role	Domain	Social role representing an actor who browses Listings and initiates contact.

Term	Concept Type	Phase Introduced	Definition / Notes
Piece	Entity	Domain	A physical clothing item, existing independently of the system's representation.
Listing	Entity	Design	A system-created representation of a Piece made visible for discovery.
Listing Published	Event	Domain	Instantaneous domain event marking when a Listing becomes publicly visible.
Interest Expressed	Event	Domain	Event occurring when a Buyer initiates contact regarding a Listing.
Listing Closed	Event	Domain	Event reflecting that a Listing is no longer available, either due to exchange or withdrawal.
Condition Rating	Value Object	Domain	Immutable value representing the assessed quality of a Piece (e.g., scale 1–10).
Review	Value Object	Domain	User-generated evaluative artifact created after an offline interaction; not an event itself.
Locale	Setting	Domain	Physical or institutional environment where exchanges occur (campus areas, public spaces).
Type	Value Object	Domain	Taxonomic clothing classification used to categorize Pieces (e.g., shirts, pants, shoes).
rate(Piece, ConditionRating) → ConditionRating	Action	Design	Operation that evaluates or updates the condition rating of a Piece (pure transformation).
publishListing(Piece, Seller, Locale) → ListingPublished	Action	Design	Action that causes the Listing Published event and results in a new system Listing.
expressInterest(Listing, Buyer) → InterestExpressed	Action	Design	Action that triggers the Interest Expressed event by creating an Interest that references the Buyer and Listing (the Seller is obtained through the Listing, not through a direct structural link).
Interest	Entity	Domain	A persistent association created when a buyer shows interest in a listing; the Interest references the Buyer and the Listing involved, with the Seller derived through the Listing.
closeListing(Listing, Seller) → ListingClosed	Action	Design	Action that ends Listing availability and triggers the Listing Closed event.

2.1.3 Ñ Domain Terminology in Relation to Domain Rough

This section explains how the terminology in ¤2.1.2 was derived from the raw observations in the Domain Rough Sketch (¤2.1.1). The goal is to document the transformation from informal descriptions to stable domain concepts. Each term is tagged with its conceptual category (Behavior, Action, Norm, Setting, Entity, Value Object).

Exchange Flow (Behavior)

Students repeatedly described a sequence: discover item ! contact seller ! clarify details ! meet ! exchange. We abstracted this into Exchange Flow, a Behavior pattern representing the typical order of actions in informal garment handoffs.

Condition Disclosure Norm (Norm)

Multiple stories included showing defects, tags, or wear. These reflect a Norm: sellers are expected to disclose condition accurately. It is a social rule, not an action.

Student Resale Price Band (Value Object)

Mentions of Òsymbolic pricesÓ created a recognizable interval of acceptable low-cost pricing. This became Student Resale Price Band, a Value Object representing a shared reference range.

Textile Waste Stream (Environmental Phenomenon)

References to donated, discarded, or reused clothing reflect the end-of-life pathways of garments. We abstracted this into Textile Waste Stream, a phenomenon describing how clothing leaves circulation.

Dormant Stock (State / Entity Condition)

Stories of bags of unused clothes indicated items in limbo. We named this Dormant Stock, representing a Piece that exists physically but is not in active circulation.

Meetup Spot (Setting)

Campus benches, bus stops, and parking lots were repeatedly used as handoff locations. These

---

became Meetup Spot, a Setting where Exchange Flow steps occur.

Ad-hoc Channel (Communication Setting)

Students negotiated via WhatsApp, Marketplace, Instagram stories, etc. We unified these into Ad-hoc Channel, a medium enabling discovery and negotiation.

Trust Cue (Value Object / Signal)

Recognizable names, bilingual communication, and clear photos all served as signals of reliability. These became Trust Cues Ñ interpreted signals aiding decision-making.

Seasonal Demand Pulse (Environmental/Market Behavior)

Demand surges around semester beginnings or weather changes created cyclical patterns. We abstracted this into Seasonal Demand Pulse, describing market-level fluctuations.

These derived terms now serve as the conceptual foundation for the deeper analysis in ¤3.1, ensuring that the vocabulary used in the domain model originates directly from observed practices.

2.1.4 Domain Narrative

In Puerto Rico, especially around university communities like MayagŸez, informal clothing exchange is woven into everyday life. Students, families, and neighbors frequently pass along garments that no longer fit, are no longer needed, or might be useful to someone else. Most of these exchanges begin casually: a photo shared through a group chat, a quick post on social media, or a message asking whether anyone nearby has a specific item available.

People commonly describe the condition of an item, mention its size, and include a picture taken at home or on campus. Those looking for clothing browse these posts whenever they have a free momentÑbetween classes, during breaks at work, or at home in the evening. When something catches their attention, they reach out directly to the person offering it and ask whether it is still available.

If both parties agree, they arrange to meet somewhere familiar and convenient, usually on or near the UPRM campus, at a bus stop, in a parking lot, or outside a classroom building. The handoff is quick and direct. Some exchanges involve a symbolic price, while others are simple gifts. Afterward, people often let their friends or peers know the item has been taken so others do not continue asking about it. In many cases, those who have had a positive experience mention it to others or comment on it in the group where the item was originally shared.

Alongside these exchanges, people sometimes keep track of items they find interesting, ask friends to check certain posts for them, or look at previous interactions whendeciding whether to engage

with someone new. These habits Ñ sharing, searching, meeting, and informal reputation Ñ form the everyday flow of secondhand clothing circulation in the community.

2.1.5 Ñ Events, Actions, and Behaviors

This section contains only events, actions, and the behaviors they compose. Entities, repositories, or structural concepts appear elsewhere and are not repeated here.

Events

Domain events are instantaneous occurrences that record that something of significance has happened in the domain.

Listing Published
: Occurs at the moment a seller makes an item publicly visible.

Interest Expressed
: Occurs when a buyer signals interest in a listing. This marks the moment the expression of interest happens; the corresponding entity is handled elsewhere.

Listing Updated
: Occurs when visible listing attributes (images, title, condition, category, etc.) are modified.

Listing Closed
: Occurs when a seller marks a listing as terminal (sold, donated, or retracted).

Actions

Domain actions are deliberate operations performed by actors that may trigger events.

Publish Item
: A seller provides the information required to make an item visible. *Triggers:* Listing Published

Browse
: A buyer inspects available listings, optionally using filters. *Triggers:* none; browsing is a pure query.

Express Interest
: A buyer initiates contact or signals desire to acquire a listing. *Triggers:* Interest Expressed

Update Listing
: A seller modifies previously posted information. *Triggers:* Listing Updated

Close Listing
: A seller marks the listing as no longer available after an exchange or a withdrawal. *Triggers:*

Listing Closed

Behaviors

A behavior is a recurring pattern consisting of a sequence of actions and the events they produce. Behaviors arise across many instances of the domain and capture the characteristic rhythm of interactions.

1. Listing Lifecycle Behavior

A recurring pattern describing how an item becomes visible, may be refined, and eventually leaves active circulation.

Trigger	A seller decides to make an item available.
Actors	Seller (primary)
Sequence	- Publish Item ! Listing Published - (optional) Update Listing ! Listing Updated - Close Listing ! Listing Closed
Pattern	Items cycle through visibility: made available ! possibly refined ! made unavailable. This cycle may repeat: items may re-enter visibility after a period of dormant storage.

2. Discovery Behavior

The recurring pattern by which buyers locate items.

Trigger	Buyer searches intentionally or browses casually.
Actors	Buyer (primary)
Sequence	- Browse - (optional) apply filters or text queries *No event is generated until interest is expressed.*
Pattern	Opportunistic scanning of available items during brief time windows.

3. Interest and Contact Behavior

The interaction pattern when a buyer wishes to proceed toward acquiring an item.

Trigger	Buyer identifies a listing of interest.
Actors	Buyer (primary), Seller (respondent)
Sequence	- Express Interest ! Interest Expressed - Seller provides clarification or confirmation

Pattern	Short informal exchanges that establish availability and next steps.

4. Offline Exchange Behavior

The culminating real-world interaction where the item changes possession.

Trigger	Buyer and seller agree on meeting arrangements.
Actors	Buyer and Seller
Sequence	- Coordinate meeting place and time - Physical handoff - Seller performs Close Listing ！ Listing Closed
Pattern	Face-to-face exchanges in familiar semi-public areas; payment or donation are informal.

5. Post-Exchange Feedback Behavior

Reputation-related actions occurring after the exchange.

Trigger	Listing has been closed following an exchange.
Actors	Buyer (primary)
Sequence	- Buyer optionally provides feedback or a review through supported channels (No required event; feedback patterns vary)
Pattern	Positive interactions are acknowledged publicly; negative ones are avoided or reported informally.

2.1.6 - Function Signatures

Objective

Define domain-level function signatures that describe how actions are carried out, including inputs, outputs, and possible changes in the domain. These signatures reflect the separation between entities, events, and repositories as clarified in professor feedback.

Description

Function signatures specify how actors in the system interact through domain actions. They define the logical relationships between inputs, outputs, and resulting state changes:

- ¥ The name of the function: The action being performed.
- ¥ The input parameters: The information and collaborators required by the action.
- ¥ Output: The type of data or event the function produces.
- ¥ State changes: How the action affects the domain (typically via repositories).

The general format of a function signature is:

- `FunctionName : Input1 >< Input2 >< É ! OutputType`
- Description Ð what the function does.
- Preconditions Ð what must be true or available for the function to execute.
- Postconditions Ð what becomes true afterwards.
- Notes Ð clarifications about scope, limits, or non-covered cases.

In addition to individual entities (Piece, Listing, Seller, Buyer), we use the following domain-level collaborators:

- `ListingRepository` as the domain-level collection that conceptually stores all listings in the system. From this repository we can derive:
- all listings (including closed/archived),
- available listings (listings that are still open and visible to buyers).
- `InterestRepository` as the domain-level collection that stores Interest entities. An Interest connects a Buyer and a Listing and is distinct from the InterestExpressed event.
- `ListingFactory` as the domain-level service responsible for constructing Listing instances from input data (Piece, Seller, Locale, and related metadata) before they are persisted by the ListingRepository.

Browsing and searching operations always work against `ListingRepository`. Contact/interest operations work against `InterestRepository`.

Filter Value Object

A Filter is a value object that represents buyer-defined search constraints. It is immutable and does not depend on `Piece`, `Page`, or UI-layer concepts.

The Filter has the following fields:

Field	Description	Type / Concept
`category`	Desired category of the item. Optional.	Value: Category
`size`	Requested size. Optional.	Value: Size
`gender`	Intended gender classification. Optional.	Value: Gender
`minCondition`	Minimal acceptable condition rating. Optional.	Value: ConditionRating
`maxPrice`	Maximum price the buyer is willing to pay. Optional.	Value: Money
`locale`	Campus/location where the buyer is willing to meet. Optional.	Value: Locale
`textQuery`	Free-text query for title/description matches. Optional.	Value: String

Field	Description	Type / Concept
`onlyActive`	Whether to restrict results to active listings. Defaults to true.	Boolean

The Filter induces a matching predicate:

`matches(f: Filter, p: Listing)` holds iff all present fields in `f` are satisfied by `p`.

This value object is used explicitly by the search-related function signatures below.

Function Signatures

publishListing : Piece >< Seller >< Locale >< ListingRepository ! Either[ValidationError, Listing]

Description

A Seller publishes a Piece in a given Locale, producing and persisting a new Listing.

Preconditions

Seller and Piece exist and are active; Piece has required metadata (title, condition, images).

Postconditions

A ListingFactory constructs a new Listing from the Piece, Seller, and Locale. The new Listing is stored in the ListingRepository as available and the same Listing is returned as part of the result.

Notes

Discovery-only; no in-platform payments. Validation errors (e.g., missing images, invalid locale) are represented as ValidationError on the left side of the Either.

expressInterest : Listing >< Buyer >< InterestRepository ! Either[ContactError, InterestExpressed]

Description

A Buyer signals interest in a Listing, creating a new Interest entity that references the Buyer and the Listing. The InterestExpressed event records the moment this Interest is created.

Preconditions

Listing is available (open); Buyer is authenticated.

Postconditions

A new Interest entity is created and stored in the InterestRepository, referencing the Buyer and the Listing. An InterestExpressed event is emitted/returned to reflect that this Interest has just been created.

Notes

This function uses the InterestRepository to persist the Interest (entity) and the event to record the occurrence in the domain timeline.

rate : Piece >< ConditionRating ! Piece

Description

Assigns or updates a Piece's condition rating.

Preconditions

Piece exists; rating is within the valid scale.

Postconditions

Returns the updated Piece with the new ConditionRating embedded within it.

review : Seller >< Buyer >< Review ! Either[ReviewError, ReviewSubmitted]

Description

Records a review after an offline exchange.

Preconditions

A prior successful exchange exists between the parties.

Postconditions

ReviewSubmitted maintains references to the Listing and the User being reviewed. Listings do not store or contain Reviews structurally; reviews are associated via references.

closeListing : Listing >< Seller ! Either[ClosedError, Listing]

Description

Seller closes an active Listing after an exchange or withdrawal.

Preconditions

Seller owns the Listing; Listing is active.

Postconditions

Listing state becomes terminal (sold/donated/retracted) and the updated Listing is returned. The updated Listing can then be persisted via the ListingRepository.

categorize : Piece >< Type ! Option[Piece]

Description

Assigns a category type to a Piece; returns an updated Piece if the categorization is applicable.

discard : Piece ! Option[Void]

Description

Marks a Piece as inactive and removes it from circulation.

browseAvailableItems : ListingRepository ! Set<Listing>

Description

Returns the set of Listings that are currently available.

Preconditions

ListingRepository is defined.

Postconditions

Returns a (possibly empty) set of available Listings.

Notes

Backend operates solely on active listings for this operation.

findAvailableByFilter : ListingRepository >< Filter ! Set<Listing>

Description

Returns the set of available Listings that satisfy the Filter predicate.

Preconditions

ListingRepository is defined; Filter is valid.

Postconditions

Returns all Listings that are:

- available, and
- matching the Filter (according to `matches(f, p)`).

Notes

Distinguishes Òall listingsÓ from the subset that is currently available and relevant to buyers.

findListingById : ListingRepository >< ListingID ! Option[Listïng]

Description

Retrieves a Listing by identifier from the ListingRepository.

Example Scenario: From Listing to Offline Exchange

1. Seller publishes a listing using `publishListing(Piece, Seller, Locale, ListingRepository) ! Either[ValidationError, Listing]`, and, on success, the new Listing is constructed by ListingFactory, stored in the ListingRepository as available, and returned.
2. Buyer browses available items using `browseAvailableItems(ListingRepository) ! Set<Listing>` or refines results using `findAvailableByFilter(ListingRepository, Filter) ! Set<Listing>`.
3. Buyer expresses interest using `expressInterest(Listing, Buyer, InterestRepository) ! Either[ContactError, InterestExpressed]`, which both persists a new Interest in the InterestRepository and emits the InterestExpressed event.
4. The exchange happens offline.
5. Seller closes the listing using `closeListing(Listing, Seller) ! Either[ClosedError, Listing]`, and the returned Listing reflects the new terminal state (sold/donated/retracted).
6. Buyer leaves a review using `review(Seller, Buyer, Review) ! Either[ReviewError, ReviewSubmitted]`, which records a review referencing the counterpart and the related Listing without embedding the review into the Listing structure.

Closure Under Operations

Closure under operations ensures that every domain-level function defined in this section returns a value that remains within the domain model, preserving the internal consistency of entities, events, value objects, and repositories. This principle guarantees that domain operations do not Òleak Ó values from outside the domain and that all returned types are themselves valid domain constructs.

The following demonstrates closure for each operation:

- ¥ publishListing Input types: `Piece`, `Seller`, `Locale`, `ListingRepository` Output type: `Either[ValidationError, Listing]` Ð Both `ValidationError` and `Listing` are domain-defined concepts. Ð The returned `Listing` is a domain entity created by `ListingFactory` and stored through the `ListingRepository`, maintaining closure.
- ¥ expressInterest Input types: `Listing`, `Buyer`, `InterestRepository` Output type: `Either[ContactError, InterestExpressed]` Ð `InterestExpressed` is a domain event, and `ContactError` is a domain error type. Ð The created `Interest` entity is stored within the domain (`InterestRepository`). Ð No extraneous structures are introduced.
- ¥ rate Input types: `Piece`, `ConditionRating` Output type: `Piece` Ð The updated `Piece` remains the same domain entity, preserving closure.
- ¥ review Input types: `Seller`, `Buyer`, `Review` Output type: `Either[ReviewError, ReviewSubmitted]` Ð Reviews remain domain constructs linked by reference and never embedded into Listings. Ð Returned values are fully internal to the domain vocabulary.
- ¥ closeListing Input types: `Listing`, `Seller` Output type: `Either[ClosedError, Listing]` Ð The returned `Listing` is still the same domain entity, now in a terminal state (sold/donated/retracted). Ð Domain closure is respected because no outside type is introduced.
- ¥ categorize Input types: `Piece`, `Type` Output type: `Option[Piece]` Ð Both `Piece` and `Type` are domain constructs, and `Option` wraps domain values only.
- ¥ discard Input type: `Piece` Output type: `Option[Void]` Ð Returns domain-level void semantics (representing removal from circulation), not system-level types.
- ¥ browseAvailableItems Input: `ListingRepository` Output: `Set<Listing>` Ð `Set` is a domain-level collection of domain entities (`Listing`). Ð No extraneous data structures are introduced.
- ¥ findAvailableByFilter Input: `ListingRepository`, `Filter` Output: `Set<Listing>` Ð `Filter` is a domain value object. Ð Returned values remain domain entities.
- ¥ findListingById Input: `ListingRepository`, `ListingID` Output: `Option[Listing]` Ð Lookup returns only domain types.

In all cases, the domainÕs operations map domain inputs to domain outputs without producing external or implementation-level values (e.g., raw database rows, API records, or UI components). This demonstrates that the domain model is closed under its defined operations, satisfying the formal requirement highlighted in the Milestone 3 guidelines.

2.1.7 Closure Under Operations

Closure Under Operations ensures that every domain action returns an object of the same

conceptual type as the entity being transformed. A function that takes a Listing must return a Listing; a function that updates a Piece must return a Piece. This keeps all transformations inside the domain vocabulary and prevents leaking UI objects, storage formats, or event structures into the domain layer.

Motivation

Earlier versions of the model returned heterogeneous outputs: some functions returned events, others returned partial data, and several returned structures tied to implementation details. These breaks in closure made it difficult to chain operations and prevented the domain from being expressed as a stable algebra of transformations.

This was corrected by ensuring that all operations return either:

- the updated domain entity, or
- a domain-defined error wrapped around the same entity type.

Updated Function Shapes

The following operations now preserve closure and match the function definitions in Section 2.1.6 of the documentation.

- publishListing : Piece >< Seller >< Locale ! Either[ValidationError, Listing]
- closeListing : Listing >< Seller ! Either[ClosedError, Listing]
- rate : Piece >< ConditionRating ! Piece
- browseAvailableItems : ListingRepository ! Set<Listing>
- findAvailableByFilter : ListingRepository >< Filter ! Set<Listing]

Listing-related transformations return Listings, and piece-related updates return Pieces.

Cross-Team Evidence of Closure

- Authentication operations always return the canonical AuthContextValue shape or an AuthError.
- Listing and piece operations always return Piece or Listing entities, never DTOs.
- Location operations always return Location or List<Location>.

This cross-team consistency ensures that closure is preserved throughout the system.

Unified Example of Closure

A typical Listing lifecycle demonstrates closure under operations:

1. A seller publishes an item, returning a Listing.
2. A buyer expresses interest, producing an InterestExpressed event referencing the same Listing.
3. The seller closes the listing, returning the updated Listing.

Events annotate transitions, but the entity remains stable throughout.

Canonical Flow Diagram

```
Piece + Seller + Locale
Ê        |
Ê        v
Ê     Listing (available)

Listing + Buyer
Ê        |
Ê        v
Ê  InterestExpressed (event)
Ê        |
Ê        v
Ê     Listing (same structure)

Listing + Seller
Ê        |
Ê        v
Ê     Listing (closed)
```

Benefits

Enforcing closure under operations provides:

- ¥ predictable chaining of domain functions,
- ¥ compatibility with CommandÐQuery Separation,
- ¥ composability across repositories and domain collections,
- ¥ a stable set of shapes for testing, validation, and requirements traceability.

2.2.1 Epics, Features, and User Stories

Epics describe high-level goals that span multiple sprints and require several coordinated features. Features refine epics into concrete functionality. User stories express these features from the perspective of domain stakeholders.

Traceability uses AsciiDoc anchors to link: EPIC ! FEATURE ! REQUIREMENT and back REQUIREMENT ! FEATURE ! EPIC

Epics

Buyer Epics

1. Listing Discovery As a buyer, I want to browse, filter, and search listings so that I can efficiently find items that match my preferences. Implemented by: [FEATURE-1.1], [FEATURE-1.2], [FEATURE-1.3] Supported by Requirements: [REQ-DR1], [REQ-DR2], [REQ-DR3], [REQ-DR4]

1. Saved Listings As a buyer, I want to save listings to revisit them later. Implemented by: [FEATURE-2.1], [FEATURE-2.2] Supported by Requirements: [REQ-DR1]

1. Trust and Transparency As a buyer, I want seller profiles and reviews to make safe choices. Implemented by: [FEATURE-3.1], [FEATURE-3.2], [FEATURE-3.3], [FEATURE-3.4] Supported by Requirements: [REQ-DR3]

Seller Epics

1. Listing Management As a seller, I want to create, update, and close listings. Implemented by: [FEATURE-4.1], [FEATURE-4.2], [FEATURE-4.3], [FEATURE-4.4] Supported by Requirements: [REQ-DR1], [REQ-DR3], [REQ-DR4], [REQ-DR5]

1. Interest Notifications As a seller, I want to be notified when buyers express interest. Implemented by: [FEATURE-5.1], [FEATURE-5.2] Supported by Requirements: [REQ-DR3]

1. Seller Profile & Trust Signals As a seller, I want a minimal profile to help buyers build confidence. Implemented by: [FEATURE-6.1], [FEATURE-6.2] Supported by Requirements: [REQ-DR1], [REQ-DR3]

Features

Features refine epics into implementable slices of functionality.

Buyer Features

1. Filtering (category, size, condition, price/free marker) Supports Epic: [EPIC-1] Requires: [REQ-DR1], [REQ-DR2], [REQ-DR4] Supported by Requirements: [REQ-DR1], [REQ-DR2], [REQ-DR4]

1. Keyword Search Supports Epic: [EPIC-1] Requires: [REQ-DR1], [REQ-DR3] Supported by Requirements: [REQ-DR1], [REQ-DR3]

1. Sorting Options (newest first, alphabetical, condition) Supports Epic: [EPIC-1] Requires: [REQ-DR1] Supported by Requirements: [REQ-DR1]

Saved Listings

1. Bookmark Listing Supports Epic: [EPIC-2] Requires: [REQ-DR1] Supported by Requirements: [REQ-DR1]

1. Persistent Saved Listings Supports Epic: [EPIC-2] Requires: [REQ-DR1] Supported by Requirements: [REQ-DR1]

Trust & Transparency

1. Seller Profile Page Supports Epic: [EPIC-3] Requires: [REQ-DR3] Supported by Requirements: [REQ-DR3]

1. Seller Ratings & Reviews Supports Epic: [EPIC-3] Requires: [REQ-DR3] Supported by Requirements: [REQ-DR3]

1. Review Submission Flow Supports Epic: [EPIC-3] Requires: [REQ-DR3] Supported by Requirements: [REQ-DR3]

1. Reporting Mechanism Supports Epic: [EPIC-3] Requires: [REQ-DR1] Supported by Requirements: [REQ-DR1]

Seller Features

1. Create Listing Form Supports Epic: [EPIC-4] Requires: [REQ-DR1], [REQ-DR4] Supported by Requirements: [REQ-DR1], [REQ-DR4]

1. Upload Multiple Images Supports Epic: [EPIC-4] Requires: [REQ-DR1] Supported by

Requirements: [REQ-DR1]

1. Edit Listing Details Supports Epic: [EPIC-4] Requires: [REQ-DR1], [REQ-DR5] Supported by Requirements: [REQ-DR1], [REQ-DR5]

1. Close Listing (Sold/Donated/Retracted) Supports Epic: [EPIC-4] Requires: [REQ-DR3] Supported by Requirements: [REQ-DR3]

Interest Notifications

1. Instant Notification on Interest Expressed Supports Epic: [EPIC-5] Requires: [REQ-DR3] Supported by Requirements: [REQ-DR3]

1. Notification Center Supports Epic: [EPIC-5] Requires: [REQ-DR3] Supported by Requirements: [REQ-DR3]

Seller Profile & Trust

1. Editable Seller Profile Supports Epic: [EPIC-6] Requires: [REQ-DR3] Supported by Requirements: [REQ-DR3]

1. Seller Dashboard Supports Epic: [EPIC-6] Requires: [REQ-DR1], [REQ-DR3] Supported by Requirements: [REQ-DR1], [REQ-DR3]

User Stories

User stories refine features into small, testable goals from domain actors perspectives.

(UNCHANGED Ñ no cross-linking needed here)

2.2.2 Ñ Personas

Personas are fictional but plausible representations of stakeholders in the domain. They help justify requirements by grounding them in the observed needs, constraints, and behaviors of real people who participate in or are affected by the hand-me-down clothing ecosystem.

Personas are not limited to direct platform users; any stakeholder who influences or benefits from the domain can serve as a persona. Each persona below includes a brief biography, goals, pain points, needs, and the areas of the platform or domain design where they provide insight.

Direct-User Personas

Adriana G–mez Ñ Buyer

- Age: 20
- Occupation: University student on financial aid; part-time campus worker
- Appearance: Dark straight hair, lowÐaverage height, vintage aesthetic
- Personality: Creative, expressive, community-minded

Adriana uses fashion as a form of identity and expression. She frequently seeks unique, older garments but lacks time to visit local thrift stores due to academic and work responsibilities. She depends on digital tools to discover items efficiently.

- Goals: Build an affordable, expressive wardrobe
- Pain Points: Limited local thrift options; time constraints; inconsistent online visibility
- Needs: Trustworthy online listings; discovery tools that surface unique items
- Interaction Pattern: Browses from her phone in short intervals between classes
- Design Relevance: Guides the structure of category filters, listing layout, and search tooling

---

Manuel Torres Ñ Seller

- Age: 35
- Occupation: Middle school teacher
- Appearance: Curly brown hair, average height, casual polos and graphic tees
- Personality: Patient, humorous, practical

Manuel is preparing to move and prefers to give his clothing a second life rather than discarding it. He is motivated by reducing waste and recouping modest value but lacks a simple and reliable way to list items.

- Goals: Streamline the moving process while extending item usefulness
- Pain Points: No convenient channel to sell used clothes locally
- Needs: Straightforward listing creation and editing; low friction workflows
- Interaction Pattern: Creates listings from his work computer during breaks
- Design Relevance: Informs listing creation, editing flows, and status transitions (active ! reserved ! sold/donated/retracted)

Daniela L–pez Ñ Buyer with Sustainability Focus

- Age: 27
- Occupation: Nurse practitioner
- Appearance: Tall, wavy brunette hair, comfortable athleisure style
- Personality: Eco-conscious, outspoken, deliberate decision-maker

Daniela avoids fast fashion and prioritizes durable vintage items. After the closure of her preferred thrift store, she now relies on digital discovery tools to maintain access to quality second-hand clothing.

- Goals: Convenient access to durable, quality garments
- Pain Points: Lack of local sustainable clothing options; declining item quality in major retailers
- Needs: Effective filters to identify high-quality, durable items
- Interaction Pattern: Uses search + filtering extensively
- Design Relevance: Validates the need for category guidance, condition indicators, and strong filtering capabilities

Domain-Level Personas (Indirect Stakeholders)

These roles help justify requirements beyond simple buyer/seller interactions, especially around sustainability, visibility, and community support mechanisms.

Isidra Montoya Ñ Local Thrift Store Owner

- Age: 61
- Occupation: Thrift store owner
- Appearance: Short stature, white pixie-cut hair, practical clothing
- Personality: Warm, welcoming, diligent

Isidra's shop has limited visibility due to its location and her low digital literacy. Community members occasionally upload her shop to the platform's thrift-store map, indirectly increasing her reach.

- Pain Points: Low visibility; limited digital outreach
- Needs: Community-driven discovery mechanisms; intuitive interfaces
- Interaction Pattern: Does not use the platform directly but benefits from being discoverable through it
- Design Relevance: Supports map-related features, community contribution flows, and accessibility-first design considerations

---

Fernando Ruiz Ñ Community Organizer

- Age: 29
- Occupation: Event organizer and nonprofit volunteer
- Appearance: Tall, medium build, short hair dyed blue at the tips
- Personality: Energetic, charismatic, service-oriented

Fernando organizes monthly clothing drives, but donation volume is often insufficient due to low public awareness. When users add his events to the community map, drive participation increases.

- Pain Points: Insufficient visibility for donation events
- Needs: Platforms that amplify donation opportunities
- Interaction Pattern: Indirect beneficiary of user-generated map entries
- Design Relevance: Helps justify donation-specific flows, map event visibility, and clear distinction between resale vs donation in the domain (supports REQ-DR3)

Purpose of Personas

These personas ensure that requirements:

- ¥ reflect the motivations and constraints of real actors,
- ¥ support sustainability and circular-economy practices,
- ¥ consider privacy vs transparency trade-offs,
- ¥ address both direct interactions and indirect community benefits,
- ¥ remain grounded in the actual domain phenomena observed during rough-sketch analysis.

Personas thus serve as anchors connecting the domain understanding to the requirements and later design decisions.

2.2.3 Domain Requirements

DR1: The system must classify every listing under exactly one primary category. Supported by: [FEATURE-1.1], [FEATURE-1.2], [FEATURE-1.3], [FEATURE-2.1], [FEATURE-2.2], [FEATURE-3.4], [FEATURE-4.1], [FEATURE-4.2], [FEATURE-4.3], [FEATURE-6.2] Supports Epics: [EPIC-1], [EPIC-2], [EPIC-3], [EPIC-4], [EPIC-6]

DR2: Categories may have hierarchical subcategories. Supported by: [FEATURE-1.1] Supports Epics: [EPIC-1]

DR3: The system must distinguish between Resale and Free/Donation listings as separate domain concepts. Supported by: [FEATURE-1.2], [FEATURE-3.1], [FEATURE-3.2], [FEATURE-3.3], [FEATURE-4.4], [FEATURE-5.1], [FEATURE-5.2], [FEATURE-6.1], [FEATURE-6.2] Supports Epics: [EPIC-1], [EPIC-3], [EPIC-4], [EPIC-5], [EPIC-6]

See justification in ¤2.2.3: Justification: Why Donation Is Not Equivalent to a Zero-Price Sale.

Justification: Why Donation Is Not Equivalent to a Zero-Price Sale

(Your existing justification remains unchanged.)

DR4: The system shall support (not enforce) item-to-category compatibility by offering suggestions and warnings. Supported by: [FEATURE-1.1], [FEATURE-4.1] Supports Epics: [EPIC-1], [EPIC-4]

DR5: The system must support category evolution. Supported by: [FEATURE-4.3] Supports Epics: [EPIC-4]

2.2.4 Ñ Interface Requirements

Objective

Interface Requirements describe how users interact with the platform through visible UI components and controls. They specify observable states, validation rules, and permissible interactions. These requirements do not prescribe implementation details and do not function as a

user manual. All exchanges between buyers and sellers continue to occur offline.

Interface Requirements

Create Listing Form

- The authenticated user shall be implicitly treated as the seller; no seller-selection field is displayed.
- Required fields shall include: Title, at least one Image, and Category.
- Optional fields may include: Price, Description, Condition, and Size.
- If Price is omitted, the listing shall be marked as Free/Donation to align with REQ-DR3.
- When selecting a Category, the interface may present non-binding suggestions derived from title, description, or images (supports REQ-DR4).
- The Publish control shall remain disabled until all required fields satisfy validation rules.
- Submissions with invalid or incomplete data shall trigger inline validation messages, with focus directed to the earliest failing field.
- On successful submission, the interface shall navigate to the Listing Details view and display a confirmation message.

Listing Details (Seller View)

- When the Listing is Active, editable attributes (Title, Description, Images, Price, Condition) shall be available.
- A status selector shall reflect the domain transitions: Active ! Reserved ! Sold / Donated / Retracted.
- Backward transitions or skipping intermediate states shall not be permitted.
- When a Listing reaches a terminal state (Sold, Donated, Retracted), all editing controls shall be disabled and a corresponding status badge displayed.
- The interface shall present buyer contact requests associated with that Listing in reverse chronological order.

Browse & Search (Buyer View)

- Buyers shall be able to perform keyword search and apply filters for Category, Condition, Size, and Price/Free marker.
- Sorting controls shall permit ordering by Newest, Lowest Price, or Condition Rating.
- Each listing card shall display: Title, Thumbnail, Price/Free indicator, Condition, and Seller Rating (if available).
- Selecting a listing shall open its Listing Details view.

- If the viewer is not the Listing's owner, the interface shall display a Contact Seller action.

Saved Listings (Buyer View)

- Buyers shall be able to mark a listing as Saved or Unsaved from both card and detail views.
- Saved items shall persist across sessions.
- Visual indicators shall reflect saved/unsaved state consistently across browsing and detail views.

Contact Seller / Messaging

- Activating Contact Seller shall open a message composer associated with the selected Listing.
- Upon sending a message, a confirmation shall appear, and the conversation shall be added to the buyer's and seller's message threads.
- Sellers shall receive a notification indicator when a new message arrives.
- Message threads shall be grouped by Listing and ordered chronologically.

Profile & Reviews

- Each user shall have a profile page showing:
- their Listings (Active, Reserved, Sold/Donated/Retracted),
- their Ratings and Reviews,
- minimal profile metadata (e.g., bio or location) as allowed by privacy norms.
- Users may edit their own profile information.
- Reviews shall be read-only for the profile owner and may only be submitted by counterparties after an offline exchange.
- Seller rating summaries (average rating + count) shall be presented prominently.

Accessibility & Feedback

- All interactive controls shall be keyboard-navigable with visible focus indicators.
- Inline validation messages shall be ARIA-live announced for screen readers.
- Toasts and banners shall be non-blocking and dismissible via keyboard.
- All forms shall contain descriptive labels and accessible placeholder text.

Examples (Non-Exhaustive Illustrations)

1. Create Listing: Required fields remain invalid until a Title, Category, and at least one Image are provided; once valid, Publish becomes enabled.
2. Status Transitions: A seller moves a listing from Active ! Reserved ! Sold; editing controls disappear once a terminal state is reached.
3. Contact Seller: A buyer sends a message; a confirmation appears and the seller receives a notification.

Relation to Domain Requirements

- Domain Requirements define what must exist in the domain (e.g., Listings, Saved Listings, Reviews).
- Interface Requirements define how users interact with those domain concepts through observable UI states.
- Each interface rule supports verification of domain behavior by exposing consistent, testable interactions.

Justification

These requirements ensure that UI-visible behavior remains predictable, accessible, and aligned with domain semantics. They maintain a clear boundary between domain logic and interaction flow while supporting user trust, transparency, and usability.

Testing Plan

- Verify validation logic, required fields, and control enable/disable states.
- Confirm that transitions to terminal states disable editing as required.
- Ensure Saved Listings persist correctly across sessions.
- Test keyboard navigation and screen reader output for compliance.
- Validate notification behaviors following message exchanges.

2.2.5 Machine Requirements

Objective

The web server must support reliable and efficient operation of a React + JavaScript clothing discovery platform. To avoid ambiguity, all load-related terms (e.g., simultaneous, average, peak) are defined before the requirements.

These definitions are referenced throughout this section (see Load & Timing Definitions (Forward Reference)), and the usage scenarios (see User Mix Scenarios (With Frequency Rationale)) include rationales for expected frequencies.

Load & Timing Definitions (Forward Reference)

To eliminate ambiguity, the following timing windows are used consistently:

- Simultaneous Users: Users who perform actions within the same 10-second activity window (e.g., search, filter, view listing).
- Average Concurrent Users: Users active within a 1-minute operational window, representing typical daytime traffic.
- Peak Concurrent Users: Users active within the same 10-second window during high-demand periods (e.g., semester start, donation drives).
- Response Time (Average): The mean latency over a rolling 5-minute window under typical load.
- Response Time (Peak): The 95th-percentile latency during peak concurrent user activity.
- Uptime Window: Measured over a calendar month, consistent with industry operational standards.

These definitions are applied in all performance, reliability, and scalability requirements below.

Requirements

Performance

- The system shall return search results within 2 seconds on average, where Òaverageó means

the rolling 5-minute mean latency (see Load & Timing Definitions (Forward Reference)).

- ¥ Under peak load (150 simultaneous users within a 10-second window), the system shall maintain a maximum response time of 4 seconds for 95% of requests.
- ¥ The system shall support 200 simultaneous browsing users, performing actions within a 10-second window.
- ¥ Listing creation and updates (metadata + images) shall complete within 3 seconds on average, excluding client-side compression.

Reliability

- ¥ The server shall maintain 99.7% uptime per calendar month, allowing ² 2.1 hours of unplanned downtime.
- ¥ Core operations (create, edit, save, close listing) shall fail for ² 0.1% of requests due to server errors.
- ¥ Scheduled maintenance shall not exceed 3 hours/month and must be announced ³ 48 hours in advance.

Scalability

- ¥ The system shall scale to 500 concurrent users performing mixed actions while keeping average response time ² 3.5 seconds.
- ¥ The database shall support 50,000 active listings with ² 20% increase in baseline response time.
- ¥ Architecture must allow vertical and horizontal scaling without major redesign.

User Mix Scenarios (With Frequency Rationale)

These scenarios model the expected real-world activity distribution. Frequencies are based on observations from UPRM informal reselling groups, typical campus marketplace usage, and usage spikes during academic transitions.

1. Scenario 1: Browsing Listings (Å 60% of active users) Rationale: Most platform interaction is searching, scrolling, filtering, similar to Facebook Marketplace usage patterns.
2. Scenario 2: Creating Listings (Å 20%) Rationale: Listing creation is less frequent; students typically post items in small batches (closet cleanouts, moving periods).
3. Scenario 3: Messaging / Contact Seller (Å 15%) Rationale: Only a fraction of browsing leads to contact. Matches expected marketplace conversion rates (~10Ð20%).
4. Scenario 4: Administrative or Moderation Tasks (Å 5%) Rationale: Admin/moderator actions occur sporadically, typically only during reported issues or verification checks.

These percentages are used to model load distribution during performance testing and to validate server capabilities under normal and peak conditions.

Justification

Defining timing windows and usage frequencies explicitly ensures that Òsimultaneous,Ó Òpeak,Ó and Òaverage Ó conditions become testable, measurable, and reproducible. Forward references (Load & Timing Definitions (Forward Reference), User Mix Scenarios (With Frequency Rationale)) help reviewers locate definitions quickly and avoid ambiguity.

This structure aligns the platform with operational standards used in lightweight marketplaces such as OfferUp, Depop, and Facebook MarketplaceÑespecially during high-demand periods (semester start, back-to-school, donation drives).

2.3 Implementation

Objective

Describe the overall architecture and design of the Hand Me Down application, showing how the React frontend, JavaScript logic, and Supabase backend (database, storage, auth) work together as an integrated system. Mockups and implementation fragments are included only when they clarify architectural decisions. This section distinguishes explicitly between Implemented functionality and Planned Ñ Not Yet Implemented functionality, as required by Milestone 3.

Description

The application is composed of a React frontend, a set of JavaScript application services, and Supabase components for persistence, authentication, and storage. The subsections below indicate which parts are implemented today and which remain planned for future milestones.

Frontend (React)

Implemented

- Browsing and filtering of active listings (connected to Supabase).
- Basic filtering (category, price/free marker, and other simple fields).
- Saving/unsaving listings (with Supabase persistence).
- Upload Page with:
  - image upload to Supabase Storage,
  - required-field validation,
  - successful insertion into Supabase.
- Map Page with:
  - rendering via Leaflet,
  - ESRI tile-layer imagery (metadata is provided externally by ESRI),
  - working marker display and clustering,
  - implemented route preview (OSRM).
- Basic profile editing and authentication screens.
- Styling through CSS modules or CSS-in-JS.

Planned Ñ Not Yet Implemented

- Full Seller Dashboard (only partially completed for demo purposes).
- Advanced filtering tied directly to the Filter Value Object (VO); current filtering exists, but spec-driven filters are planned.
- Messaging and contact flows.
- Review submission and display.
- Notification center for ÒInterest ExpressedÓ events.
- Extended editing flows on the Upload Page.
- Map metadata overlays (map currently relies solely on ESRIÕs imagery tiles).

Backend Logic (JavaScript Functions)

Implemented

- Input validation for listing creation (following CQS validators).
- Image uploads via Supabase Storage.
- Authentication integration (sign up, login, logout, session restore).
- CRUD operations for Listings and Pieces.
- Saving/unsaving listings.
- Listing closure (seller transitions listing to a closed state).
- Domain invariants (required fields, ownership checks) enforced in backend flows.
- Routing strategy implementation using OSRM + Strategy Pattern abstraction.

Planned Ñ Not Yet Implemented

- Messaging and InterestExpressed workflow.
- Review submission rules (requires closed listing).
- Advanced Filter VO ! composable Specification Pattern integration.
- Moderation/reporting workflows.
- Additional routing strategies (Google Maps, Mapbox, custom shortest-path).

Database & Storage (Supabase)

Implemented

- Profiles table (name, email, reputation score).
- Listings table (title, description, category, size, condition, price/free marker, status).
- Saved Listings relation table.
- Donation-center locations (map uses these coordinates).
- Supabase Storage for image uploads.
- Supabase Auth for secure authentication.
- RLS policies for profile and listing access control.

Planned Ñ Not Yet Implemented

- Review + rating table logic (tables exist, workflow not implemented).
- RLS rules for messaging and review flows.
- Extended geospatial metadata layers (beyond ESRI tiles).

External Services

Implemented

- Leaflet.js for map rendering.
- ESRI tile server for high-quality satellite imagery and basemap tiles.
- OSRM routing provider for path previews.

Planned Ñ Not Yet Implemented

- Geocoding services (address ! coordinate conversion).
- Additional routing providers (Google Maps, Mapbox).
- Metadata overlays on the map (relying on ESRI tile imagery for now).

Architecture Diagram

The diagram is included because it highlights how implemented frontend, backend, and Supabase components communicate. It also clarifies extension points for planned features, such as messaging, review workflows, and additional routing strategies.

1. Architecture Overview
    a. React Frontend
        $ Implements primary user-facing flows (browse, filter, save, upload).
        $ Prototype-only components (e.g., messaging UI) appear for architectural completeness.
        $ Planned components will reuse the same application-service boundaries.
    b. JavaScript Logic (Application Layer)
        $ Implements validation, image-upload flow, CRUD operations, and closure logic.

- $ Provides abstraction for routing through a pluggable Strategy Pattern.
- $ Planned extensions include expressing interest, advanced filtering, and moderation.

c. Supabase Services

- $ Database tables persist listings, profiles, saved relationships, and donation centers.
- $ Storage manages listing images.
- $ Auth manages sessions, secure access, and RLS enforcement.
- $ Planned enhancements include messaging constraints and review linkage.

d. Data Flow

- $ Requests travel from React ! Application Services ! Supabase.
- $ Responses update React state with the most recent domain data.
- $ System intentionally supports offline exchanges only; no payments or mediated transactions.

Layer Mapping (Clean Architecture / DDD Alignment)

- ¥ Domain Layer Core domain concepts (Listing, Piece, User, Review), invariants, and key events (ListingPublished, InterestExpressed, ListingClosed).
- ¥ Application Layer Implemented use cases: `publishListing`, `saveListing`, `browseAvailableItems`. Planned use cases: `expressInterest`, `review`, advanced filtering, messaging.
- ¥ Interface / Adapters Layer React screens, forms, prototypes, and data-adapter modules for calling application services.
- ¥ Frameworks / Drivers Layer Supabase (Database, Storage, Auth), Leaflet, OSRM, ESRI tiles, HTTP clients.

2.3.1 Selected Fragments of the Implementation

This section provides selected implementation fragments that complement the architectural description in Section 2.3. To satisfy Milestone 3 requirements, fragments are divided into:

1. Implemented Fragments Ð Functionality already built and connected to Supabase.
2. Planned Ñ Not Yet Implemented Fragments Ð Architectural prototypes illustrating intended domain mappings, not functional code.

Screenshots appear only where they clarify domain flows, state transitions, or architectural decisions. They are not used as a user manual.

Implemented Fragments

Home Page (Implemented UI Fragment)

Caption: Included because it shows the implemented data-binding architecture: the Home Page retrieves active listings from Supabase and renders them in React state. This illustrates how UI components consume application-level services such as `browseAvailableItems`.

The implemented Home Page: - fetches active listings from Supabase, - renders them in an infinite/scrollable feed, - links to other implemented features (upload page, browse view).

This fragment demonstrates the separation between UI rendering, application-service calls, and Supabase queries.

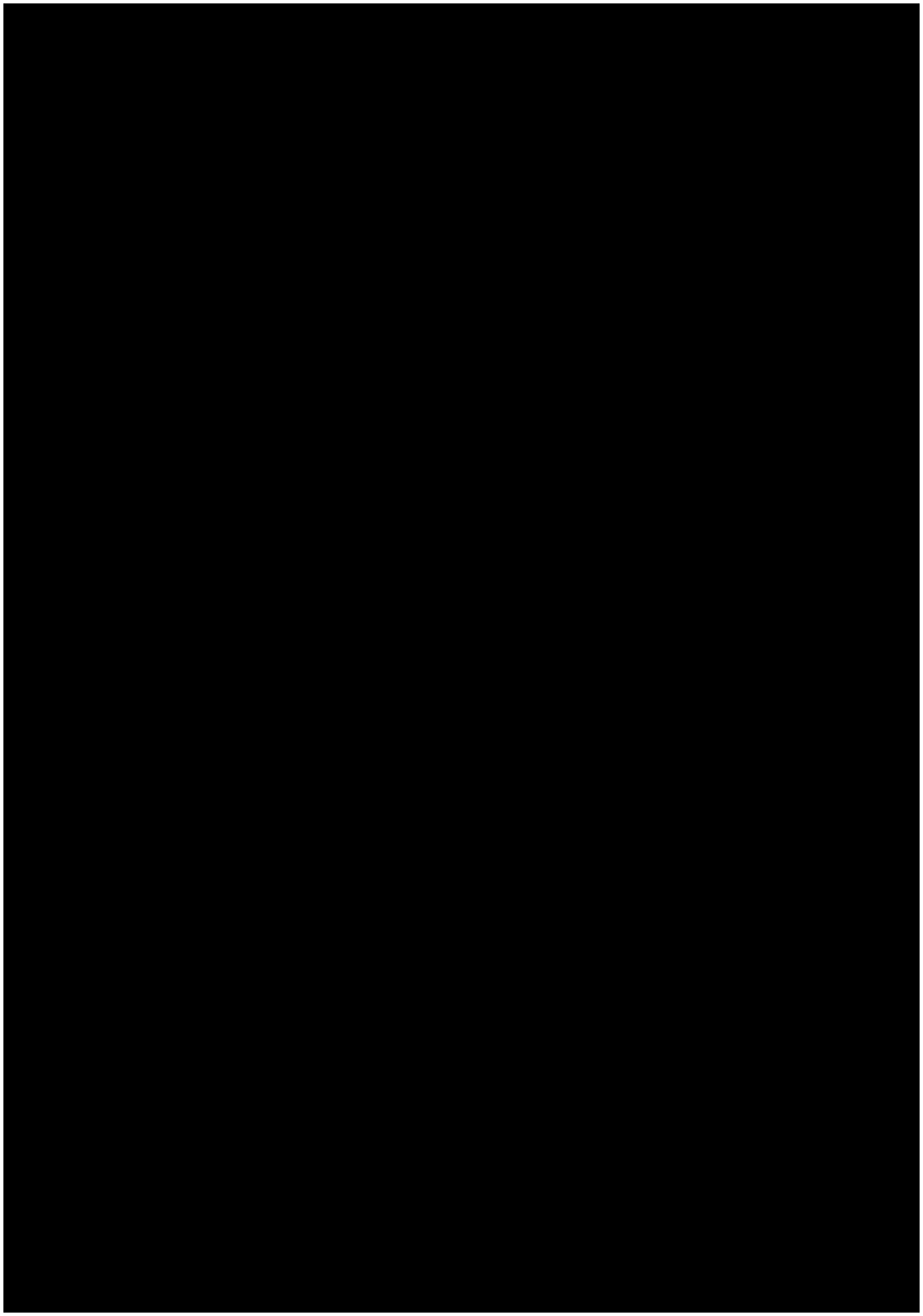
Clothing Listing Page (Implemented UI Fragment)

Caption: Included because it illustrates the implemented end-to-end flow from Supabase data retrieval ! React rendering ! filter/sorting logic. This fragment validates the correctness of the listing-retrieval architecture.

This implemented component: - loads available listings through Supabase queries, - displays essential listing metadata (title, main image, free/resale marker), - supports client-side sorting and basic filtering (category, free/price).

Advanced filter-composition (Filter VO + Specification Pattern) is planned, but basic filtering shown here is implemented.

Item Upload Flow Ñ Implemented Logic



Caption: Included because it visualizes the implemented control flow: client-side validation ! Supabase Storage upload ! listing insertion. This supports the domain function `publishListing` and documents the actual data movement across layers.

Implemented steps: 1. Authenticated user opens the creation form. 2. Required fields validated using CQS validators. 3. Images uploaded to Supabase Storage. 4. Listing inserted into Supabase with `status = active`.

This flow enforces domain invariants but intentionally performs no transaction or payment logic.

Planned Ñ Not Yet Implemented Fragments

(These prototypes illustrate the intended architectural integration of domain concepts. They are not implemented and contain no production logic.)

Item Detail Page (Planned UI Fragment)

Caption: This prototype shows how multiple domain objects (Listing, Seller, Review summary) will be aggregated into a single detail view. It is included solely to illustrate architectural fit for planned use cases such as `expressInterest` and `review`.

Planned integrations: - listing metadata display, - seller information panel, - interest/contact actions linked to `InterestExpressed`, - review summary once review logic is implemented.

No backend logic exists yet.

Map Page (Planned UI Fragment)

Caption: These prototypes illustrate the architectural role of the Map feature: React ! Leaflet ! donation-center data from Supabase. ESRI imagery tiles provide basemap layers; metadata overlays are not implemented. These fragments exist strictly for architectural documentation.

The planned Map architecture includes: - Leaflet for geospatial rendering, - donation-center locations retrieved from Supabase, - optional geocoding + alternate routing strategies (planned).

No advanced metadata overlays or enhanced routing providers exist yet.

Summary

This section now: - clearly distinguishes between Implemented and Planned fragments, - uses screenshots solely for architectural justification, - removes any user-manual language, - accurately reflects what the current system supports versus what remains planned for future milestones.

This aligns fully with Milestone 3 expectations and professor feedback.

2.3.2 Application of Techniques

This subsection presents the concrete techniques applied by the team during Milestone 3, demonstrating how the project evolved from scattered, duplicated logic to modular, intention-revealing, and maintainable structures. The examples below come from real improvements implemented by the Map, Auth, and Backend teams. Each example includes a before/after transformation aligned with Domain-Driven Design principles and the architectural rules described

in Section 2.2 Architecture & Layering.

Specification Pattern: From Scattered Conditionals to Composable Rules

Before

Filtering logic in the Map and Listing domains was originally written using inline conditionals inside controllers, leading to duplication and tight coupling. Each filter Ñ such as ÒNear Me,Ó ÒOpen Now,Ó or ÒCategoryÓ Ñ was evaluated manually:

```
function filterResults(listings, filters) {
Ê return listings.filter(listing => {
Ê   let ok = true;

Ê   if (filters.nearMe) {
Ê     const dist = calculateDistance(listing.location, filters.userLocation);
Ê     if (dist > filters.radiusMeters) ok = false;
Ê   }

Ê   if (filters.openNow) {
Ê     const hour = new Date().getHours();
Ê     if (!(listing.openingHour <= hour && hour <= listing.closingHour)) {
Ê       ok = false;
Ê     }
Ê   }

Ê   if (filters.category) {
Ê     if (listing.category !== filters.category) ok = false;
Ê   }

Ê   return ok;
Ê });
}
```

Problems:

- ¥ Controller-level business logic.
- ¥ Repeated conditions in multiple files.
- ¥ Adding a new filter required editing several modules.
- ¥ Hard to test filtering rules independently.
- ¥ No clear mapping to the Filter Value Object documented in the domain model.

After

Filters were refactored into Specification objects, each exposing a single, meaningful rule through a shared interface:

```
export interface Specification<T> {
Ê isSatisfiedBy(candidate: T): boolean;
}
```

Examples of concrete Specifications:

```
export class NearMeSpecification implements Specification<Listing> {
Ê constructor(private userLocation, private radiusMeters: number) {}

Ê isSatisfiedBy(listing: Listing): boolean {
Ê   const dist = calculateDistance(listing.location, this.userLocation);
Ê   return dist <= this.radiusMeters;
Ê }
}

export class OpenNowSpecification implements Specification<Listing> {
Ê constructor(private currentHour: number) {}

Ê isSatisfiedBy(listing: Listing): boolean {
Ê   return (
Ê     listing.openingHour <= this.currentHour &&
Ê     this.currentHour <= listing.closingHour
Ê   );
Ê }
}
```

Specifications can be composed:

```
export class AndSpecification<T> implements Specification<T> {
Ê constructor(
Ê   private left: Specification<T>,
Ê   private right: Specification<T>
Ê ) {}

Ê isSatisfiedBy(candidate: T): boolean {
Ê   return (
Ê     this.left.isSatisfiedBy(candidate) &&
Ê     this.right.isSatisfiedBy(candidate)
Ê   );
Ê }
}
```

The call site becomes clean and intention-revealing:

```
const specs: Specification<Listing>[] = [];

if (filterVO.nearMe) {
```

```
Ê specs.push(new NearMeSpecification(userLoc, filterVO.radius));
}

if (filterVO.openNow) {
Ê specs.push(new OpenNowSpecification(currentHour));
}

const result = listings.filter(listing =>
Ê specs.every(spec => spec.isSatisfiedBy(listing))
);
```

Benefits:

- ¥ Encapsulated filtering rules instead of scattered conditionals.
- ¥ Easy to add new filters without modifying controllers.
- ¥ Clear mapping to the Filter VO.
- ¥ Fully testable, composable domain rules.

Strategy Pattern: Interchangeable Routing Providers

Before

Routing was tightly coupled to Leafletů s OSRMv1 provider. Controllers directly instantiated and configured the OSRM router:

```
async function getRoute(origin, destination) {
Ê const router = L.Routing.osrmv1({
Ê   serviceUrl: "https://router.project-osrm.org/route/v1",
Ê });

Ê return new Promise((resolve, reject) => {
Ê   router.route([origin, destination], (err, routes) => {
Ê     if (err) reject(err);
Ê     else resolve(routes[0]);
Ê   });
Ê });
}
```

Problems:

- ¥ Hard-coded provider (OSRMv1).
- ¥ No ability to switch to other routing engines (e.g., Google Maps, Mapbox, custom).
- ¥ Difficult to test routing behavior in isolation from Leaflet.
- ¥ Violated the OpenÐClosed Principle: adding a new routing policy required changing existing controller code.

After

A routing abstraction was introduced following the Strategy Pattern taught in lecture:

```
export interface RoutingStrategy {
Ê getRoute(origin: LatLng, destination: LatLng): Promise<RouteResult>;
}
```

Current concrete strategy:

```
export class OSRMRoutingStrategy implements RoutingStrategy {
Ê private router = L.Routing.osrmv1({
Ê   serviceUrl: "https://router.project-osrm.org/route/v1",
Ê });

Ê async getRoute(origin: LatLng, destination: LatLng): Promise<RouteResult> {
Ê   return new Promise((resolve, reject) => {
Ê     this.router.route([origin, destination], (err, routes) => {
Ê       if (err) reject(err);
Ê       else resolve(routes[0]);
Ê     });
Ê   });
Ê }
}
```

Planned strategies (documented as extension points, not yet implemented):

- `GoogleMapsRoutingStrategy`
- `MapboxRoutingStrategy`
- `CustomShortestPathRoutingStrategy`

Controller usage becomes independent of the concrete routing engine:

```
const routingStrategy: RoutingStrategy = new OSRMRoutingStrategy();
const route = await routingStrategy.getRoute(origin, destination);
```

Benefits:

- Decouples controllers from a specific routing provider.
- Enables future routing engines by implementing the same interface.
- Simplifies testing by allowing mock strategies.
- Aligns with the Strategy Pattern as presented in the course and with the project's extensibility goals.

Intention-Revealing Interfaces: Centralizing Authentication Logic

Before

Each protected page manually performed Supabase authentication and profile retrieval, mixing concerns and duplicating logic:

```
import { supabase } from "@/lib/supabaseClient";

export async function load({ redirect }) {
Ê const { data: { user }, error: userError } = await supabase.auth.getUser();
Ê if (!user || userError) {
Ê   throw redirect(302, "/login");
Ê }

Ê const { data: profile, error: profileError } = await supabase
Ê   .from("profiles")
Ê   .select("*")
Ê   .eq("id", user.id)
Ê   .single();

Ê if (profileError) {
Ê   throw redirect(302, "/login");
Ê }

Ê return { user, profile };
}
```

Problems:

- ¥ Authentication and profile logic duplicated in many routes.
- ¥ Slight variations in profile queries between pages.
- ¥ Harder to maintain consistent behavior when auth rules change.
- ¥ The main intention (Òget authenticated user with profile or redirectÓ) was obscured by boilerplate.

After

The Auth team introduced intention-revealing helpers, matching the lectureÕs emphasis on services with clear intent and side-effect-free functions where possible:

```
export async function getAuthUserWithProfile(supabase) {
Ê const { data: { user }, error: userError } = await supabase.auth.getUser();
Ê if (!user || userError) {
Ê   return { user: null, profile: null };
Ê }

Ê const { data: profile, error: profileError } = await supabase
```

```
Ê   .from("profiles")
Ê   .select("*")
Ê   .eq("id", user.id)
Ê   .single();

Ê if (profileError) {
Ê   return { user: null, profile: null };
Ê }

Ê return { user, profile };
}
```

At the call site, the logic becomes shorter and intention-revealing:

```
const { user, profile } = await getAuthUserWithProfile(supabase);
if (!user) {
Ê throw redirect(302, "/login");
}

return { user, profile };
```

For reactive components, a hook centralizes subscription to auth state:

```
export function useSupabaseAuth() {
Ê const supabase = useSupabase();
Ê const auth = ref({ user: null, loading: true });

Ê supabase.auth.onAuthStateChange((_event, session) => {
Ê   auth.value = {
Ê     user: session?.user ?? null,
Ê     loading: false,
Ê   };
Ê });

Ê return auth;
}
```

Typical usage:

```
const auth = useSupabaseAuth();

if (auth.value.loading) {
Ê return "Loading...";
}

if (!auth.value.user) {
Ê router.push("/login");
```

```
}
```

Benefits:

- Intention-revealing: the purpose of each helper is clear at the call site.
- Centralized logic: changes to authentication or profile retrieval occur in one place.
- Consistency: all pages rely on the same profile shape and redirect behavior.
- Testability: helpers can be tested separately from page code.
- Alignment with lectures: matches the emphasis on side-effect-free, intention-revealing services at boundaries.

Assertions and Domain Invariants

As discussed in the Supple Design lecture (slides 25Ð31), assertions help document and enforce the invariants that must always hold before and after domain operations. They make assumptions explicit, reduce ambiguity, and support the correctness of aggregates and services. In this project, assertions are incorporated into key operations across the Authentication, Backend, and Map subsystems to guarantee stability and prevent invalid states.

Authentication Operation: signUp(firstname, lastname, email, password)

The `signUp` flow creates a new user within the authentication subsystem. Assertions clarify the conditions under which the operation is valid and the guarantees it provides afterward.

Preconditions - `email` is syntactically valid. - `password` meets the security policy (³ 8 characters, includes number or symbol). - `email` is not already associated with an existing account. - `firstname` and `lastname` are non-empty strings.

Postconditions - A new User account exists with the provided email. - The password is securely hashed (never stored in plain text). - A valid session or authentication credential is created. - The new user is considered authenticated immediately after sign-up.

This mirrors the lecture guidance that authentication workflows benefit from explicit pre- and post-conditions to avoid ambiguous states (e.g., Òuser created but not authenticatedÓ).

Backend Operation: createPiece(pieceData)

The `createPiece` operation instantiates a new `Piece` entity and enforces domain rules regarding categories, conditions, and ownership.

Preconditions - The seller initiating the operation is authenticated and valid. - `category` must match one of the allowed domain categories. - Provided attributes must be compatible with the category (e.g., shoes must include `shoeSize`; accessories must not include a size field). - `condition` % {new, like-new, lightly-used, used, heavily-used}. - `description` length does not exceed the domainÕs maximum.

Postconditions - A new Piece object exists and is associated with the seller. - The Piece receives a unique `pieceID`. - All fields (`category`, `condition`, `attributes`, `description`) match the validated input. - The Piece enters the `unlisted` state and is not part of any Listing yet.

These assertions document the invariant that a Piece must always enter the system in a valid, unlisted, domain-compliant state.

Map Operation: nearMe(listings, userLocation)

The `nearMe` operation returns all listings within a fixed radius (8km) of the specified user location or map-selected point. Assertions ensure geospatial correctness and consistent results.

Preconditions - `userLocation` contains valid latitude and longitude values. - `listings` is a non-null, iterable collection. - Each listing must include a valid meetup location or seller coordinate.

Postconditions - Returned listings are only those within 8km of `userLocation`. - No listing outside the radius appears in the result. - Ordering of listings is preserved unless explicitly sorted afterward. - The function has no side effects (does not modify listings or persist data).

By capturing the geospatial invariants explicitly, the Map subsystem aligns with the lecture principle that system behavior should never rely on hidden assumptions.

Summary

These assertions clarify the systemÕs behavioral expectations and match the lecture recommendations for maintaining model integrity through explicit invariants. Each subsystemÑAuth, Backend, and MapÑuses assertions to ensure operations transition between valid states, prevent malformed inputs, and reinforce domain rules. Together, they contribute to a more predictable, verifiable, and maintainable design.

CommandÐQuery Separation (CQS) and Side-Effect-Free Validators

CommandÐQuery Separation (CQS) was applied in the Auth subsystem to distinguish pure, side-effect-free validation from mutating commands that change authentication state or persist data. This refactor simplifies reasoning about behavior, improves testability, and aligns with the course emphasis on supple design.

Before: Mixed Validation and Side Effects in a Single Flow

Originally, several Auth-related flows combined input validation, Supabase calls, and navigation/redirects into a single function. A typical Òsign upÓ handler both validated inputs and performed mutations:

```
export async function handleSignUp(formData, supabase, router) {
Ê const email = formData.get("email") as string;
Ê const password = formData.get("password") as string;
Ê const firstname = formData.get("firstname") as string;
Ê const lastname = formData.get("lastname") as string;

Ê // Inline validation
Ê if (!email || !email.includes("@")) {
Ê   throw new Error("Invalid email.");
Ê }
Ê if (!password || password.length < 8) {
```

```
Ê   throw new Error("Password too short.");
Ê }

Ê // Auth + profile creation (side effects)
Ê const { data, error } = await supabase.auth.signUp({
Ê   email,
Ê   password,
Ê   options: {
Ê     data: { firstname, lastname },
Ê   },
Ê });

Ê if (error) {
Ê   throw error;
Ê }

Ê router.push("/dashboard");
}
```

Problems:

- ¥ Queries (validation) and commands (mutations) were mixed together.
- ¥ The function had multiple responsibilities: validation, auth side effects, and navigation.
- ¥ Unit testing required mocking Supabase and routing even when only validation was under test.
- ¥ Intent was blurred: the what (validate vs. create) and the how (Supabase, router) were tangled.

After: Pure Validators + Explicit Command Handlers

The Auth Team refactored validation concerns into pure functions that take data and return validation results without side effects. Mutating logic was isolated in dedicated command handlers.

Pure validators (queries):

```
export function validateEmail(email: string): string[] {
Ê const errors: string[] = [];
Ê if (!email) {
Ê   errors.push("Email is required.");
Ê } else if (!email.includes("@")) {
Ê   errors.push("Email must contain '@'.");
Ê }
Ê return errors;
}

export function validatePassword(password: string): string[] {
Ê const errors: string[] = [];
Ê if (!password || password.length < 8) {
Ê   errors.push("Password must be at least 8 characters.");
Ê }
Ê if (!/\d|\W/.test(password)) {
```

```
Ê   errors.push("Password must include a number or symbol.");
Ê }
Ê return errors;
}

export function validateProfileFields(firstname: string, lastname: string): string[] {
Ê const errors: string[] = [];
Ê if (!firstname.trim()) {
Ê   errors.push("First name is required.");
Ê }
Ê if (!lastname.trim()) {
Ê   errors.push("Last name is required.");
Ê }
Ê return errors;
}
```

These functions:

- ¥ Have no side effects (no Supabase calls, no router calls).
- ¥ Always return the same output for the same input.
- ¥ Are easy to unit-test in isolation.

Command handler (mutation):

```
export async function handleCreateAccount(formData, supabase, router) {
Ê const email = formData.get("email") as string;
Ê const password = formData.get("password") as string;
Ê const firstname = formData.get("firstname") as string;
Ê const lastname = formData.get("lastname") as string;

Ê const errors = [
Ê   ...validateEmail(email),
Ê   ...validatePassword(password),
Ê   ...validateProfileFields(firstname, lastname),
Ê ];

Ê if (errors.length > 0) {
Ê   return { ok: false, errors };
Ê }

Ê const { data, error } = await supabase.auth.signUp({
Ê   email,
Ê   password,
Ê   options: {
Ê     data: { firstname, lastname },
Ê   },
Ê });

Ê if (error) {
```

```
Ê   return { ok: false, errors: [error.message] };
Ê }

Ê router.push("/dashboard");
Ê return { ok: true, errors: [] };
}
```

Here:

- ¥ The command (`handleCreateAccount`) is clearly responsible for doing things:
- ¥ calling Supabase (auth side effect),
- ¥ triggering navigation.
- ¥ All query-style logic (validation) is delegated to pure functions.
- ¥ The handlerÔs return structure (`{ ok, errors }`) makes behavior explicit and testable.

Benefits of the CQS Refactor

- ¥ Clear Separation of Responsibilities Validators answer questions (ÒIs this input valid?Ó) without changing state, while commands modify state or trigger navigation.
- ¥ Improved Testability Pure validators can be tested without mocking Supabase or routing. Command handlers can be tested with targeted integration or mocking.
- ¥ Reduced Side Effects Fewer functions can unexpectedly cause persistence or navigation; side effects are localized to command functions.
- ¥ Easier Reasoning and Maintenance Developers can quickly see where mutations happen and reuse validators across multiple forms or flows.
- ¥ Alignment with Course Guidance The refactor demonstrates CommandÐQuery Separation by drawing a firm line between ÒaskingÓ (validators) and ÒdoingÓ (commands), while keeping domain rules explicit and intention-revealing.

Additional CQS Example: signIn (Command) vs getUser (Query)

The Auth subsystem also illustrates CommandÐQuery Separation through a clear distinction between commands that change authentication state and queries that only read it. Here we document the evolution of the login flow from a mixed command/query function into a clean split between `signIn` (command) and `getUser` (query).

======= Before: Login Flow Mixing Command and Query Responsibilities

Initially, a single function handled both:

- ¥ reading current authentication state
- ¥ performing a login attempt (mutation)
- ¥ deciding navigation behavior

```
export async function handleSignIn(formData, supabase, router) {
```

```
Ê const email = formData.get("email") as string;
Ê const password = formData.get("password") as string;

Ê const { data: { user: existingUser } } = await supabase.auth.getUser();
Ê if (existingUser) {
Ê   router.push("/dashboard");
Ê   return;
Ê }

Ê const { data, error } = await supabase.auth.signInWithPassword({
Ê   email,
Ê   password,
Ê });

Ê if (error) {
Ê   throw new Error(error.message);
Ê }

Ê router.push("/dashboard");
}
```

Problems:

- ¥ Mixed responsibilities: querying current user and performing sign-in in the same function.
- ¥ Harder to reuse: any flow that only needed current user information still depended on login logic.
- ¥ More complex tests: tests had to mock both the query (`getUser`) and the command (`signInWithPassword`) for this one function.
- ¥ Weaker intent: it was not obvious from the API which parts read state and which parts mutated it.

======= After: Explicit Query (`getAuthUser`) and Command (`signInCommand`)

The logic was refactored into:

- ¥ a query function `getAuthUser` that only reads current authentication state
- ¥ a command function `signInCommand` that performs sign-in and produces side effects

Query (no side effects):

```
export async function getAuthUser(supabase) {
Ê const { data: { user }, error } = await supabase.auth.getUser();

Ê if (error) {
Ê   return { user: null, error };
Ê }

Ê return { user, error: null };
```

```
}
```

Command (mutation + navigation):

```
export async function signInCommand(
Ê email: string,
Ê password: string,
Ê supabase,
Ê router
) {
Ê const { data, error } = await supabase.auth.signInWithPassword({
Ê   email,
Ê   password,
Ê });

Ê if (error) {
Ê   return { ok: false, error: error.message };
Ê }

Ê router.push("/dashboard");
Ê return { ok: true, error: null };
}
```

The form handler then uses the query and command explicitly:

```
export async function handleSignIn(formData, supabase, router) {
Ê const email = formData.get("email") as string;
Ê const password = formData.get("password") as string;

Ê const { user } = await getAuthUser(supabase);
Ê if (user) {
Ê   router.push("/dashboard");
Ê   return;
Ê }

Ê const result = await signInCommand(email, password, supabase, router);
Ê if (!result.ok) {
Ê   throw new Error(result.error ?? "Unable to sign in.");
Ê }
}
```

Benefits of this CQS refactor:

- Explicit intent: `getAuthUser` clearly communicates that it only reads the current user, while `signInCommand` clearly acts to change authentication state.
- Improved reuse: `getAuthUser` can be reused anywhere current user information is needed without pulling in sign-in logic.

- Easier testing: query and command logic can be unit-tested independently with focused mocks.
- Consistent CQS story: together with the side-effect-free validators and command handlers documented earlier, this example strengthens the project's adherence to Command–Query Separation in the Auth subsystem.

Stand-Alone Auth Provider: Centralizing Supabase SDK Dependency

To reduce coupling and follow the "stand-alone class" guidance from the Supple Design lecture, the Auth Team refactored how components access authentication. Instead of each component depending directly on the Supabase Auth SDK, all SDK usage is now centralized in a stand-alone provider (`SupabaseAuthProvider`) and a stable hook (`useSupabaseAuth`). This section documents the before/after transformation and how it reduces dependencies.

Before: Components Depending Directly on Supabase Auth SDK

Originally, multiple components imported and used the Supabase client directly. Each component was responsible for calling the SDK, handling session changes, and interpreting authentication state.

Example (simplified):

```
import { supabase } from "@/lib/supabaseClient";
import { onMounted, ref } from "vue";

export default {
  setup() {
    const user = ref(null);
    const loading = ref(true);

    onMounted(async () => {
      const { data: { user: currentUser } } = await supabase.auth.getUser();
      user.value = currentUser ?? null;
      loading.value = false;

      supabase.auth.onAuthStateChange((_event, session) => {
        user.value = session?.user ?? null;
      });
    });

    async function handleSignOut() {
      await supabase.auth.signOut();
      user.value = null;
    }

    return { user, loading, handleSignOut };
  },
};
```

Problems:

¥ Each component depended directly on the Supabase Auth SDK.

¥ Auth state subscription logic was duplicated across components.

¥ Changing auth provider (or how sessions are handled) required editing many files.

¥ Testing components required mocking the Supabase client in multiple places.

¥ The intent (ÒI need auth stateÓ) was obscured by low-level SDK calls.

After: SupabaseAuthProvider and useSupabaseAuth as Stand-Alone Abstractions

The team introduced a stand-alone provider component and a shared hook to encapsulate SDK calls. Components no longer depend on the Supabase SDK; instead, they depend only on a stable interface defined by `useSupabaseAuth`.

Provider component (stand-alone class/component):

```
import { supabase } from "@/lib/supabaseClient";
import { defineComponent, provide, ref, onMounted } from "vue";

const AuthSymbol = Symbol("AuthContext");

export const SupabaseAuthProvider = defineComponent({
Ê name: "SupabaseAuthProvider",
Ê setup(_, { slots }) {
Ê   const user = ref(null);
Ê   const loading = ref(true);

Ê   onMounted(async () => {
Ê     const { data: { user: currentUser } } = await supabase.auth.getUser();
Ê     user.value = currentUser ?? null;
Ê     loading.value = false;

Ê     supabase.auth.onAuthStateChange((_event, session) => {
Ê       user.value = session?.user ?? null;
Ê     });
Ê   });

Ê   async function signOut() {
Ê     await supabase.auth.signOut();
Ê     user.value = null;
Ê   }

Ê   const value = { user, loading, signOut };

Ê   provide(AuthSymbol, value);

Ê   return () => slots.default ? slots.default() : null;
Ê },
});

export function useSupabaseAuth() {
```

```
Ê const ctx = inject(AuthSymbol);
Ê if (!ctx) {
Ê   throw new Error("useSupabaseAuth must be used within SupabaseAuthProvider.");
Ê }
Ê return ctx;
}
```

Component usage after refactor:

```
import { useSupabaseAuth } from "@/auth/SupabaseAuthProvider";

export default {
Ê setup() {
Ê   const { user, loading, signOut } = useSupabaseAuth();

Ê   return {
Ê     user,
Ê     loading,
Ê     handleSignOut: signOut,
Ê   };
Ê },
};
```

Key changes:

- All Supabase Auth SDK calls (getUser, onAuthStateChange, signOut) are centralized in `SupabaseAuthProvider`.
- Components depend only on the stable stand-alone abstraction `useSupabaseAuth`.
- If the project changes auth providers or session handling, only the provider implementation must be updated.

Diagram: Stand-Alone Auth Provider Reducing Dependencies

```
@startuml
interface AuthInterface {
Ê +user
Ê +loading
Ê +signOut()
}

class SupabaseAuthProvider {
Ê -supabaseClient
Ê +provide(AuthInterface)
}

class ComponentA
class ComponentB
```

```
SupabaseAuthProvider ..> AuthInterface
ComponentA --> AuthInterface : uses via useSupabaseAuth()
ComponentB --> AuthInterface : uses via useSupabaseAuth()

AuthInterface <|.. SupabaseAuthProvider
@enduml
```

Dependency Reduction and Benefits

- Reduced Dependencies: Components no longer import or configure the Supabase client directly; they depend only on `useSupabaseAuth`, a stand-alone abstraction.
- Centralized SDK Usage: All low-level SDK calls are located in a single provider file, making changes to the auth backend or session behavior localized.
- Improved Testability: Components can be tested by mocking `useSupabaseAuth` instead of mocking the entire Supabase client. This yields simpler, more targeted tests.
- Clearer Intent: At the component level, the code now expresses intent (ÒI need auth state and signOutÓ) rather than implementation details (Òcall supabase.auth.getUser() and manage listenersÓ).
- Alignment with Milestone 3 and Lectures: The `SupabaseAuthProvider` behaves as a stand-alone class that shields the rest of the system from low-level dependencies, satisfying the Milestone 3 requirement for dependency reduction and reflecting the stand-alone class guidance from the Supple Design lecture.

Unresolved directive in sections/section2/section.adoc - include::subsections/2.3.3.adoc[]

Unresolved directive in sections/section2/section.adoc - include::subsections/2.3.4.adoc[]

Unresolved directive in sections/section2/section.adoc - include::subsections/2.4.adoc[]

Unresolved directive in sections/section2/section.adoc - include::subsections/2.5.adoc[]

Unresolved directive in sections/section2/section.adoc - include::subsections/2.5.1.adoc[]

Unresolved directive in sections/section2/section.adoc - include::subsections/2.5.2.adoc[]

3.1 Concept Formation and Analysis

This section documents how the domain concepts were derived from the raw observations in 2.1.1 Ð Domain Rough Sketch and the everyday practices described in 2.1.4 Ð Domain Narrative. It explains how concrete, context-specific details were abstracted into entities, events, roles, value objects, actions, and behaviors. The goal is to show a clear and verifiable path from observation to domain model.

From Observation to Abstraction

The rough sketch stories revealed a recurring structure across different individuals: someone makes an item visible to others, potential takers discover it, a brief conversation follows, and the

parties meet in person to complete the exchange. Although the exact tools vary (group chats, social media posts, direct messaging), the underlying phenomena remain consistent.

Several recurring patterns emerged:

- People show items using posts or photos that function as temporary “public signals.”
- Interested parties initiate contact with short, informal messages asking if the item is still available.
- Agreement is reached by clarifying small details (size, condition, pickup spot).
- The exchange occurs at a familiar physical location.
- After the handoff, the item is no longer advertised, and the original post is often updated or removed.
- Positive or negative experiences influence future interactions through informal reputation.

Because these behaviors appeared across multiple narratives, the team abstracted them into stable domain concepts rather than anecdotal exceptions.

Consolidating Entities

The physical objects (“shirts,” “blazer,” “backpack,” etc.) were generalized into the entity Piece, representing any clothing item circulating in the community.

In contrast, the posts, messages, or shared images that made these items visible were abstracted into Listings—representational artifacts distinct from the Pieces they describe. This distinction was necessary because Listings have their own lifecycle: they can appear, be updated, be ignored, or be removed independently of the Piece.

This separation mirrors what happens in real exchanges: the representation moves through visibility states while the underlying item simply exists.

Identifying Events and Actions

Certain moments in the narratives corresponded to discrete transitions:

- when an item becomes publicly visible
- when someone expresses interest
- when the availability of the item changes

These transitions were abstracted into Events such as Listing Published, Interest Expressed, and Listing Closed. Each event marks an instantaneous change observed in participant behavior.

The human activities that lead to these events—posting an item, contacting someone, removing a post—became Actions. This separation allowed the model to differentiate between what people do and what changes occur in the domain as a result.

Behavioral Norms and Implicit Practices

The rough sketches contained implicit social rules that shape exchanges even though they are rarely stated explicitly.

Several of these norms recurred across different stories:

- Condition Disclosure Norm Ñ participants often show wear, defects, or tags before exchanging items.
- Meetup Practice Ñ exchanges consistently occur in familiar, visible locations, often near the UPRM campus.
- Trust Cues Ñ people rely on conversational tone, mutual acquaintances, bilingual communication, or recognizable names.
- Student Resale Price Band Ñ symbolic or low-cost pricing appeared repeatedly in student-to-student transactions.
- Exchange Flow Ñ the behavioral pattern Òshare ! discover ! contact ! meetÓ formed a recognizable structure across all accounts.
- Reuse Loop Ñ items sometimes re-enter circulation after a period of use or storage, showing that clothing may undergo multiple listing cycles.

These implicit behaviors were made explicit to accurately reflect the lived domain and to ensure future requirements align with local social practices rather than imposing new ones.

Roles, Settings, and Value Objects

Observations showed that participants shift roles depending on context. Someone who gives away an item one week may seek another the next. These shifting positions were abstracted as the roles Buyer and Seller, not as fixed categories of people.

Exchanges consistently took place at physical locations such as campus entries, bus stops, or parking lots. These stable spatial anchors were captured as Settings under the term Locale.

Repeated references to size, type, and condition formed the basis for Value Objects such as Type, Condition Rating, and other descriptive attributes that help people assess suitability quickly.

Iterative Refinement

As concepts solidified, the team rechecked them against the narratives to ensure that each abstraction described something observable without assuming any future technical system. Concepts related to visibility, interest, trust, representations, and reuse were reshaped until they represented domain truths rather than system intentions.

This iterative process ensured that the domain model remained faithful to real-world behaviors while providing a structured foundation for the requirements and design work in Section 2.2.

In summary, the concept formation process bridges concrete stories and abstract structures. It documents how repeated behaviors, roles, events, and norms were distilled into a coherent domain model that guides subsequent architectural and design decisions.

3.2 Validation and Verification

Validation and verification ensure that the Hand Me Down platform meets stakeholder needs, maintains internal consistency across modules, and satisfies measurable quality standards defined in the requirements. These activities are carried out continuously and collaboratively across all sub-teamsÑDocumentation & Requirements, Authentication, Listings, Map/Search, and UI/UXÑ to preserve traceability from stakeholder needs through implementation and testing.

Validation

Domain and Requirements Validation

Validation of terminology and requirements was performed iteratively through internal documentation reviews. Each update to ¤¤ 2.1 Ð 2.3 was examined to confirm that domain concepts such as Piece, Listing, Condition Disclosure Norm, and Listing Closed remained coherent and aligned with stakeholder needs (¤ 1.2.2). All requirements now employ definitive Òshall/mustÓ statements instead of uncertain language (Òmay,Ó ÒaimsÓ) to ensure they are directly testable. No external stakeholder validation sessions have yet occurred; these will take place in later milestones to confirm usability and trust cues with students and families.

Scenario Walkthroughs

Walkthroughs were conducted for the publishing process, tracing data flow from form submission to Supabase storage. No inconsistencies were found. Editing, closing, and saving workflows are planned for final-phase validation once their implementations are complete.

Category and Condition Refinement

Internal validation led to refinement of the category taxonomy and condition-rating scales, resolving ambiguities from the initial model. Terminology was standardized to Donated Piece and Sold Piece to ensure semantic consistency across documentation and database layers.

Search and Map Validation

The Map & Search module was validated by cross-checking UI results against the actual data stored in Supabase. Each search query correctly displayed only listings matching its attributes, and filter walkthroughs confirmed accurate behavior for ÒTops,Ó ÒBottoms,Ó and ÒAll.Ó Usability was validated through manual inspection: map markers are accurately pinned, display complete popup information (name, address, hours), and maintain expected cluster and static behaviors. Search relevance and marker accuracy confirm correct data binding between UI and repository functions.

Authentication Validation

Supabase Auth integration was validated by confirming that users can sign up, log in, and log out successfully without confusion. The authentication flow follows Supabase and OWASP recommendations for secure web login. Manual tests were executed in Google Chrome, Firefox, and Brave on desktop devices. Error messages were displayed properly when login, signup, or logout failed. Mobile testing is scheduled for the next milestone.

UI/UX Validation

Informal validation was performed through manual walkthroughs and internal demos. Team members and classmates interacted with the interface to identify confusing or redundant

elements and provided feedback on button labeling and visibility of listing-creation steps. Adjustments were applied iteratively, improving label clarity and layout alignment. Accessibility was validated manually through color-contrast and tab-navigation checks. Primary text and buttons met readability standards, and form elements followed a logical tab order. Semantic HTML was verified for buttons and labels, while full ARIA labeling and automated accessibility audits are planned for the next milestone.

Planned Stakeholder Validation

A short validation session with student volunteers will be scheduled before the final milestone to evaluate clarity of interface terminology, filter usability, and trust indicators such as condition labels and safety guidance.

Verification

Traceability and Acceptance Criteria

A preliminary Need ! Requirement ! Test mapping exists conceptually and will be formalized in `/docs/tests/traceability.adoc` for milestone 3. Each requirement includes measurable acceptance conditions: Đ Interface Requirements define button-state validation, inline error messaging, and toast feedback. Đ Machine Requirements specify response-time thresholds (² 2 s average; ² 4 s peak) and uptime ³ 99.7 %. Đ Domain Requirements link directly to test cases verifying classification and visibility logic.

Unit Testing

Manual unit tests were completed for publishListing(), validating data verification, database insertion, and frontend feedback. All manual tests passed successfully. Additional automated unit tests for closeListing() and editListing() are scheduled for the next milestone. Authentication unit tests validated correct sign-up, login, and logout behavior, while error handling displayed expected feedback messages when failures occurred.

Integration Testing

End-to-end tests confirmed correct UI-to-database behavior: information entered in the listing form propagates through backend validation and is stored in Supabase as expected. Integration with Authentication (user ID linkage) will be completed in the final milestone. The Search module's repository functions (`PieceRepository.getPieces()` and `filterPieces()`) were validated indirectly through accurate data synchronization between Supabase queries and the rendered results. Authentication privacy constraints were verified through role-based access control using Supabase Row Level Security (RLS) policies implemented in issue #301. Interface behaviors Ñ such as disabled Publish buttons until form validation passes, visible toast messages, and navigation flow correctness Ñ were manually verified against the Interface Requirements. Visual consistency was confirmed across browsers.

Load and Performance Testing

Preliminary manual observations show average search responses in Å 1 second and listing creation times under 0.5 seconds Ñ both within the defined machine-requirement limits. Formal automated load testing using k6 will be added to simulate concurrent usage (150 Đ 500 users) and confirm scalability benchmarks. Authentication latency and API response times will also be measured in the next milestone.

Data Validation and Security Checks

Map-coordinate rendering logic filters out invalid or non-finite latitude/longitude values, preventing off-map markers. All markers are non-draggable, ensuring location data remains immutable in the UI. RLS policies in Supabase protect user records by restricting read/write access based on authentication state and role. UI components were visually validated across major browsers (Chrome, Edge) to ensure consistent layout, iconography, and branding defined in the global style guide.

Continuous Verification

A GitHub Actions workflow will execute linting and unit-test jobs on pull requests to maintain consistent quality and prevent regressions once automated tests are in place.

End-to-End Feature Verification

The following table summarizes the implemented features across subsystems.

Subsystem	Implemented Features	Source
Authentication (Team 2)	Signup, login, logout, persistent sessions, profile editing, protected routes	Team 2 Auth Summary
Backend (Team 3)	Create Piece, Create Listing, Publish Listing, Close Listing (transactional), Saved Listings, Filtered Listing Retrieval	Team 3 Backend Summary
Map (Team 4)	Map rendering, marker clustering, routing providers, modals, location info, route preview	Team 4 Map Summary
UI (Team 5)	Home ! Browse ! Listing ! Interest flow, Auth screens, Profile screens, Map screens, Saved Listings UI, loading/error states	Team 5 UI Summary

The following acceptance tests demonstrate end-to-end behavior across multiple subsystems. All tests use the recommend GivenĐWhenĐThen (GWT).

Authentication Flow

Test A1 Đ Successful Login

Given the user has a valid email and password When they submit the login form Then the system returns an authenticated session And the user is redirected to the Home screen And protected content becomes accessible

Test A2 Đ Session Restoration

Given a previously authenticated session exists When the app is reopened Then the Auth subsystem restores the session And the user is taken directly to Home without re-entering credentials

Test A3 Đ Profile Update

Given the user is authenticated When they update their profile information Then the backend persists the new profile And the UI reflects the updated data

Listing Lifecycle

Test L1 Đ Create ! Publish Listing

Given the seller is authenticated And they have created a Piece When they create a Listing and publish it Then the backend stores the Listing And it appears in the browse screen And Map markers update accordingly

Test L2 Đ Close Listing (Transactional)

Given a Listing is active When the seller closes the Listing Then the backend transitions the Listing atomically to a closed state And it no longer appears in active browsing And the UI shows the closed status on the sellerůs Listing screen

Test L3 Đ Save a Listing

Given a buyer is authenticated When they save a listing Then it appears under Saved Listings And the backend reflects the saved state

Map Interaction

Test M1 Đ Load Map With Markers

Given the user opens the Map screen When the Map provider initializes Then all active Listings are shown as markers And marker clustering activates for dense areas

Test M2 Đ View Location Info Modal

Given markers are visible When the user taps a marker Then a modal appears with Listing details And route options become available

Test M3 Đ Start Route Preview

Given a Listing location is selected When the user chooses a routing mode Then the routing provider builds a path And the route preview renders on the Map

UI Navigation Flows

Test U1 Đ Home ! Browse ! Listing

Given the user is authenticated When they navigate to Browse And select a Listing Then the Listing detail screen appears And the UI renders all Listing data returned by the backend

Test U2 Ð Browse ! Filters ! Results

Given the Browse screen is open When the buyer applies valid filters Then the backend returns matching Listings And the screen updates with the filtered results

Test U3 Ð Saved Listings Screen

Given the user has saved Listings When they navigate to Saved Listings Then all saved items appear with correct metadata And tapping one navigates to its detail page

Outcomes

Ð Documentation, domain model, and requirements were aligned and validated through internal review cycles. Ð Listing-publication backend passed all manual unit tests, achieving < 0.5 s creation time. Ð Search and map functionalities were validated against Supabase data, loading results in Å 1 s on average. Ð Map markers were verified for nine sample donation centers. Ð Authentication features (sign up, login, logout) were validated across major desktop browsers with secure RLS policies. Ð UI elements and flows passed internal usability and accessibility checks; no critical issues were reported. Ð Category and condition-rating systems were refined for accuracy and uniformity. Ð Traceability structure and automated CI testing are established for completion in the final milestone.

Together, these validation and verification activities confirm that the system concepts are sound, the current implementation behaves as specified, and measurable criteria are in place to ensure the platform remains reliable, scalable, and aligned with stakeholder expectations as development continues.

Logbook

Person	Sections worked on
1uismar33r0	2.3.1 (#148)
Alma-pineiro	2.3.1 (#148)
JoshDG03	1.3.1 (#190), 2.1 (#190, #214), 2.1.1 (#190, #214), 2.1.2 (#214), 2.1.3 (#214), 2.1.4 (#214, #216), 2.1.5 (#182, #190, #214, #217, #369), 2.1.6 (#190, #214, #445, #446, #475), 2.2.1 (#190, #214, #447), 2.2.2 (#214), 2.2.3 (#214, #449), 2.2.4 (#190, #214, #450), 2.2.5 (#190, #214, #242, #451, #475), 2.3 (#190, #214, #452, #475), 2.3.1 (#214, #452, #475), 2.3.2 (#470, #471, #472, #473), 3.2 (#242)
JuanIranzo	1.5 (#438), 2.1.1 (#443), 2.1.2 (#439), 2.1.4 (#441), 2.1.5 (#442, #444), 2.1.6 (#218, #434), 2.2.2 (#356)
Ojani	2.3 (#247), 2.3.1 (#247), 3.2 (#246)
angelvillegas1	2.2.5 (#242)
leanelys	2.3.2 (#332)