An introduction to DDD, CQRS and Event Sourcing

2021-03-16 / 2021-03-22

Alex Lawrence (@lx_lawrence)

Preface

About me

- Alex Lawrence (@lx_lawrence)
- freelance developer & consultant
- working with DDD since 2012
    - trivial domains & smaller orgs
- working with CQRS & ES since 2013
    - two long-term projects

Mantras for the talk

Concepts > Code > Technologies

No silver bullets, it always depends

Outline

- Domains, Domain Models, Bounded Contexts
- Software Architecture
- Code quality
- Value Objects, Entities, Services
- Domain Events
- Aggregates
- Repositories
- Application Services
- CQRS
- Event Sourcing
- User Interface

Topics out of scope

- Authentication & Authorization
- Comprehensive DDD overview
- Dealing with Legacy Code
- Event Versioning
- Microservices
- Organizational aspects
- Specific technologies

Structure & formatting

- informal diagrams
- example slides have separate background
- code examples use JS/TS (pseudo-code)

Literature

- Domain-Driven Design: Tackling the Complexity in the Heart of Software - Eric Evans
- Implementing Domain-Driven Design - Vaughn Vernon
- DDD, CQRS and Event Sourcing class - Greg Young
- Conference Talks by Greg Young
- Implementing DDD, CQRS and Event Sourcing - Alex Lawrence

Example topic

- gaming platform
  - publishers publish games
  - users open accounts and download client
  - buy games, add them to library
  - play on selected device
  - social network & community features
- USPs
  - publisher: distribution, reach
  - user: one store, cross-device, updates


Domains

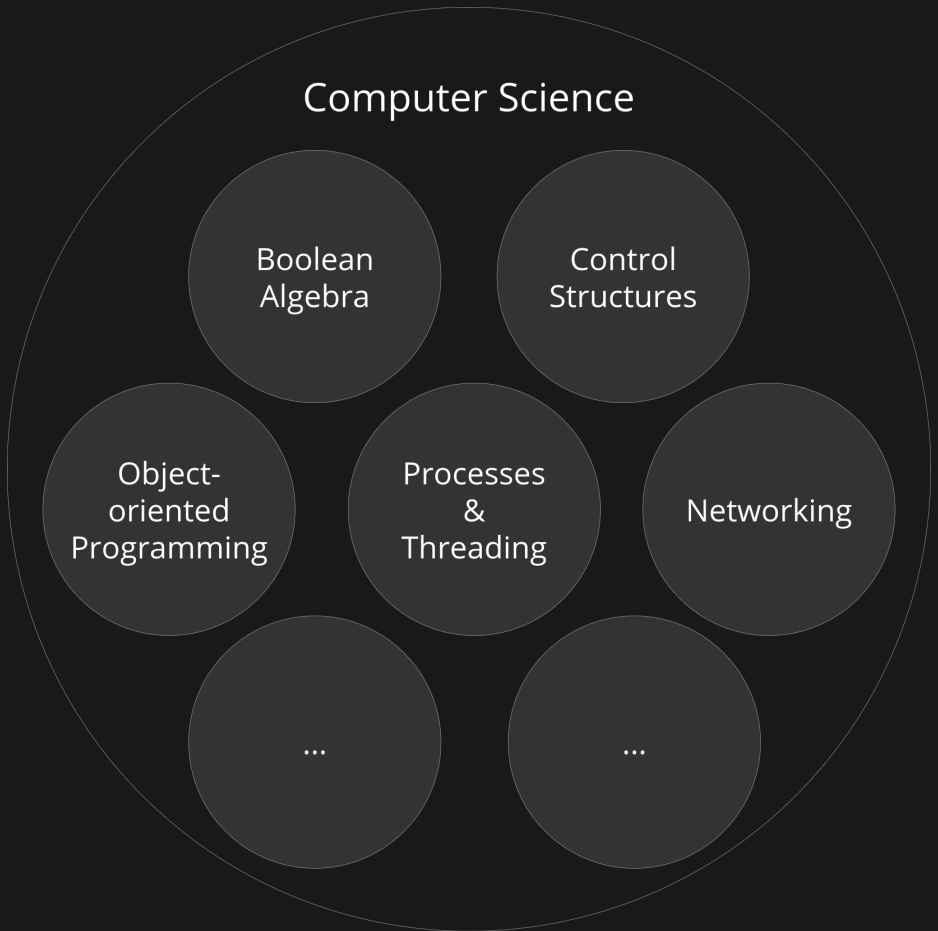
Definition

- in general: area of expertise
- in DDD context
  - topical area a software operates in
  - knowledge space around a problem

Examples

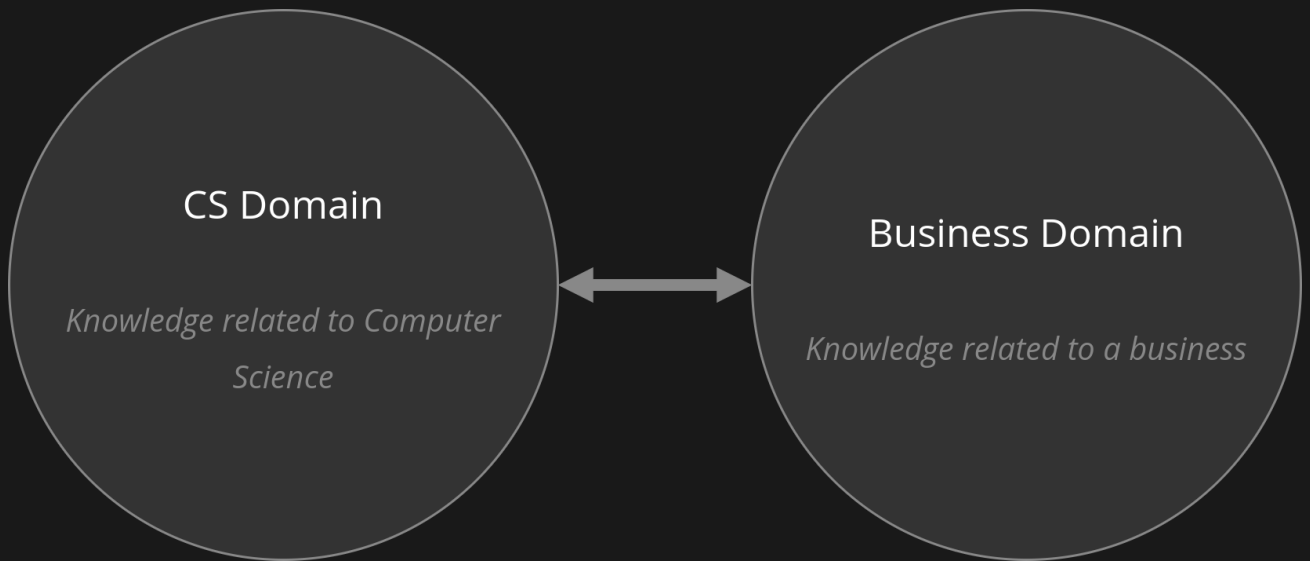
- User Experience
- Project Management
- Human Resources
- Meetings

Computer Science



- developers are experts in CS Domain
- technical knowledge, universally applicable

Technological vs. business



- solutions require business knowledge
- technical focus can be counter-productive
- "everyone" should understand the business

Domain Experts

- viable source of specialized knowledge
- often a secondary role held by PMs / POs
- do not know "everything"

Subdomains

- Domains can enclose multiple knowledge areas
- Domains can also be grouped together
- different types of Subdomains

Core Domain

- area most relevant to problems to solve
- requires most expertise and effort
- key to success & competitive advantage
- every software has one Core Domain

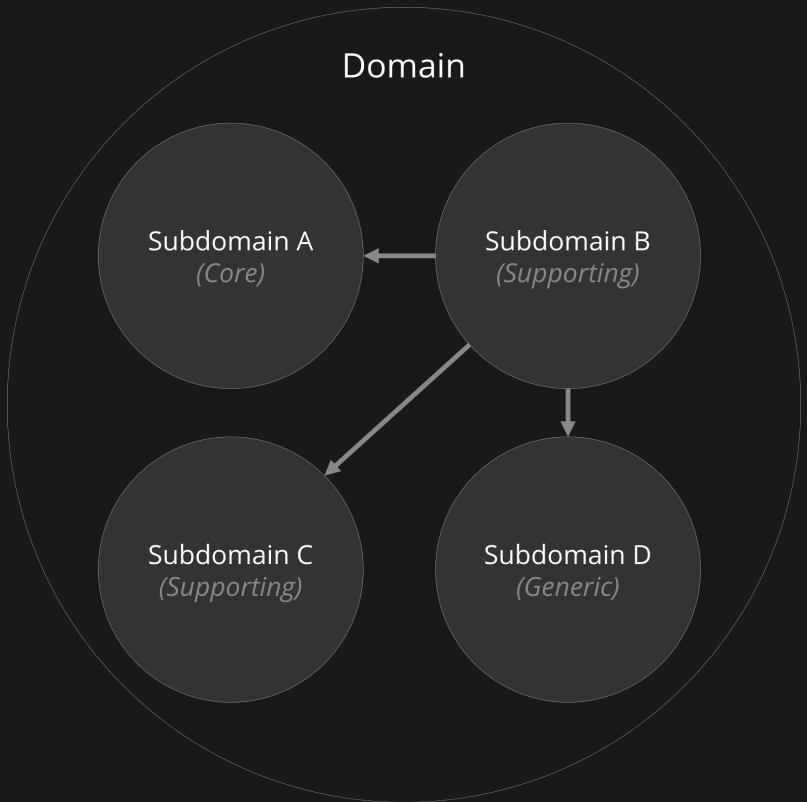
Supporting Subdomain

- partially specific to software Domain
- generic & specialized knowledge
- there can be multiple

Generic Subdomain

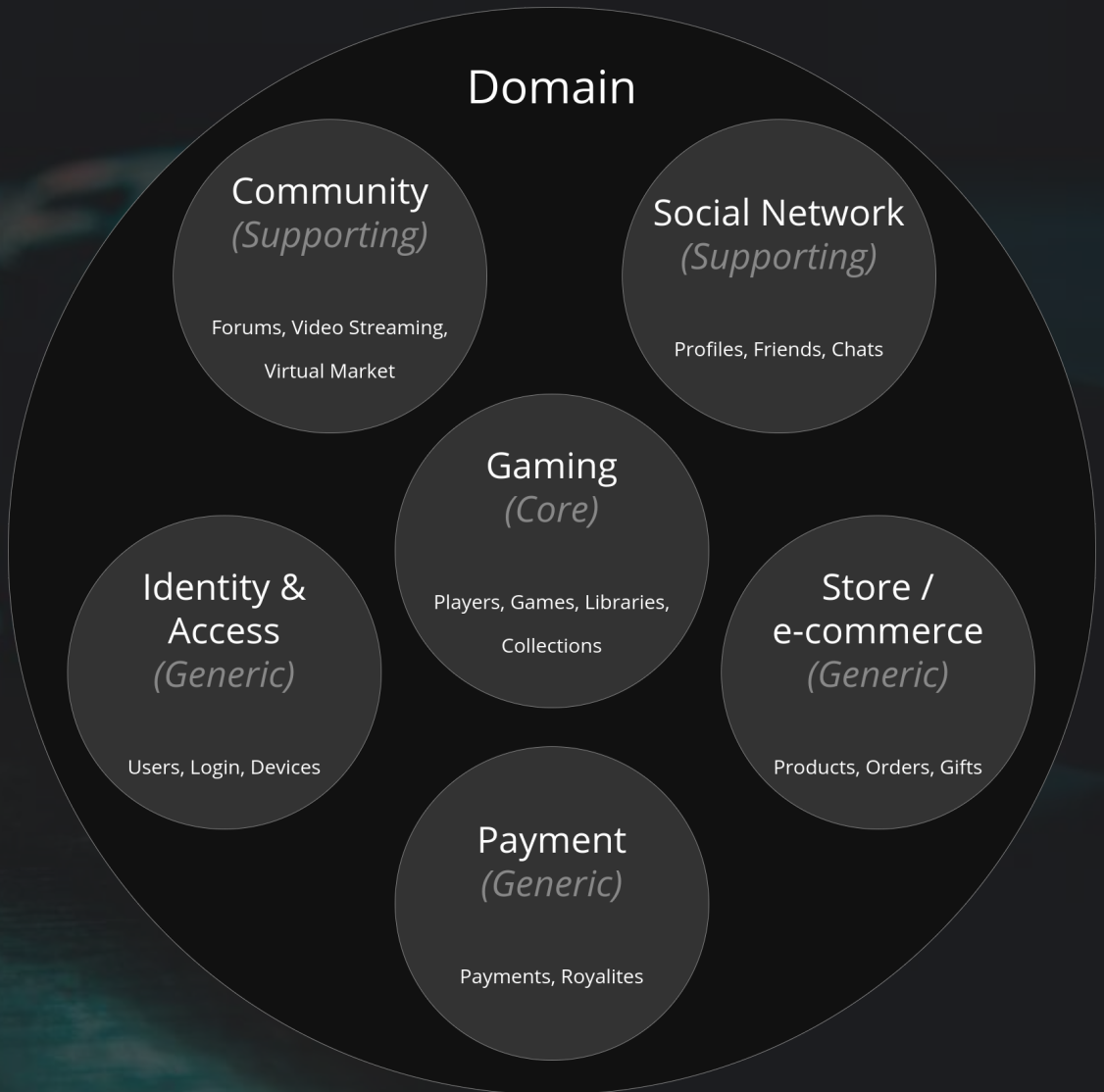
- not specific to software Domain
- there can also be multiple
- outsourcing & third-party software
- common example: Identity & Access

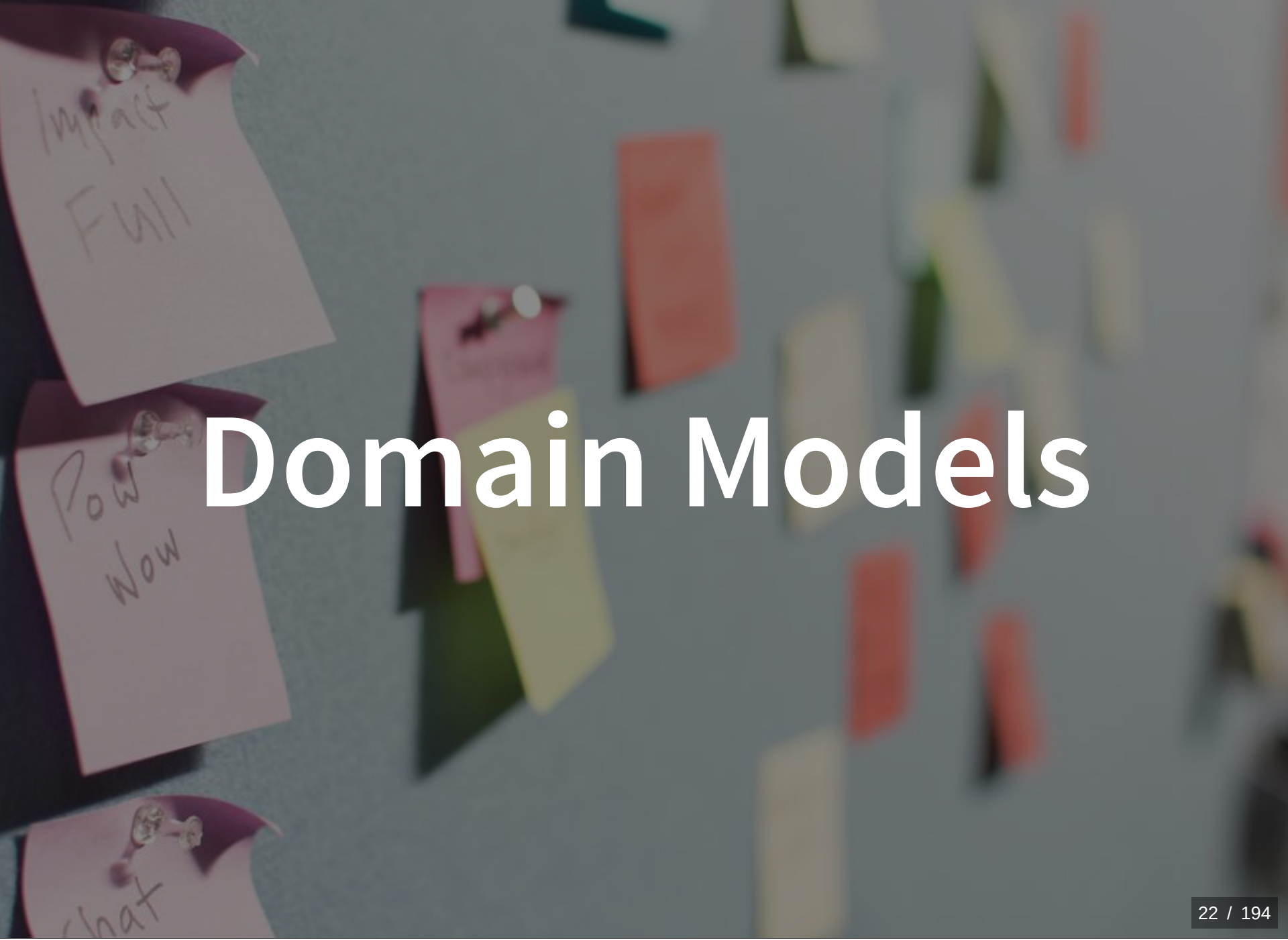
Subdomain identification



- subjective process, no exact rules
- ideally a team effort

Example: Domains (incomplete)





Domain Models

Definition

- abstractions relevant to solving a problem
- Evans: *simplified and [..] structured form of knowledge*
- not the code or some part of it

Structure and components

- targeted abstractions
  - not real world representation
- combination of data and behavior
- emphasize behavior and characteristics
  - use verbs, adjectives, adverbs

Ubiquitous Language

- language of the Domain Model
- valid where model is applicable
- eliminate translation, avoid information loss
- use everywhere, also in the code
- documented as glossary

Example: Language (incomplete)

Term	Context	Meaning
Library	Gaming	games owned by a player
Collection	Gaming	custom subset of library
Streaming	Gaming	playing on another device
Streaming	Community	broadcasting video of playing

Domain Modeling

- creating and refining a Domain Model
- iterative and continuous activity
  - understanding evolves over time
  - insights during implementation
- integrate into overall process
- both formal & informal approaches work

Model representations

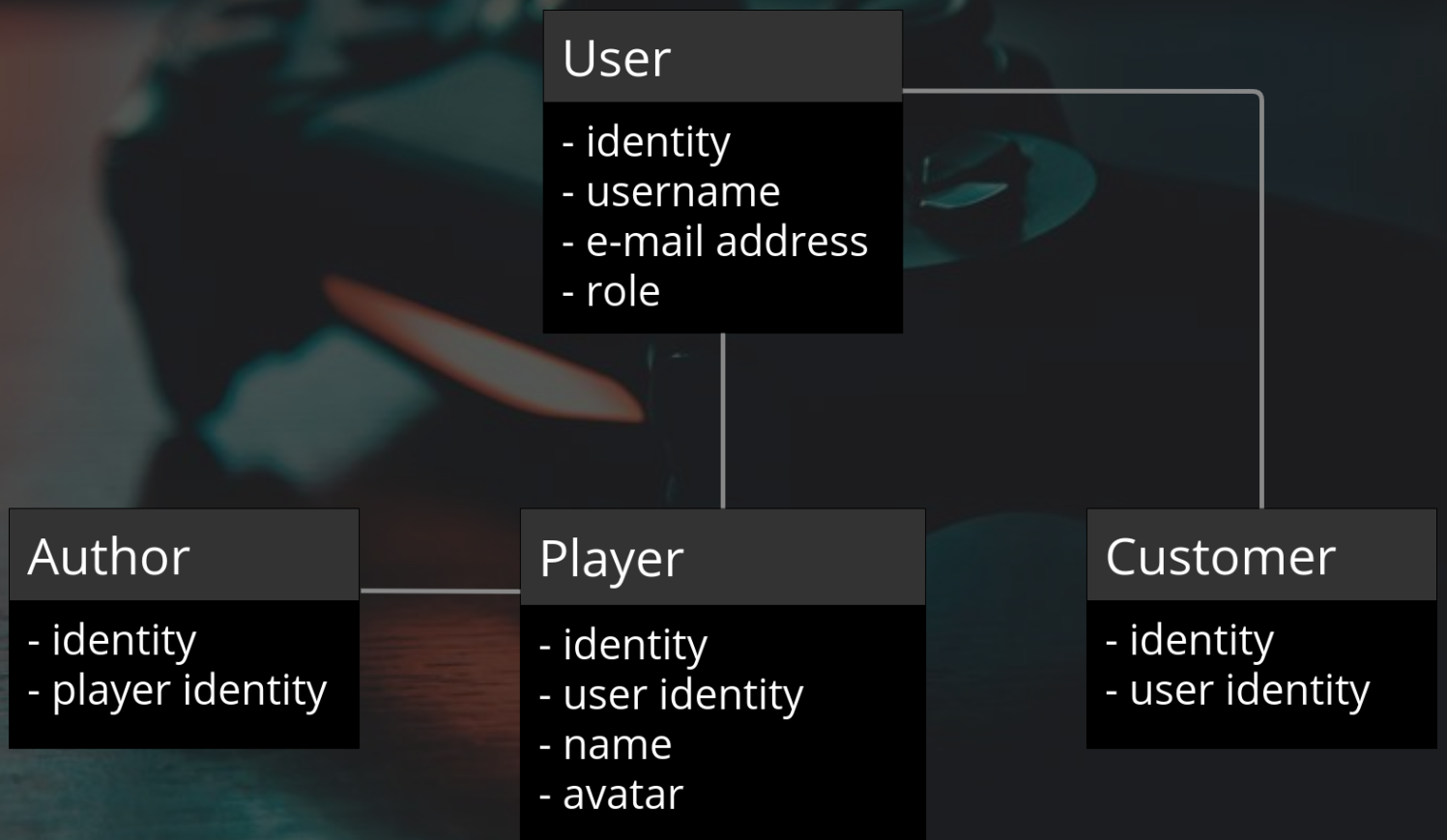
- written text
  - high expressiveness
  - can get too tedious
- diagrams
  - more abstract
  - cheap if informal, otherwise risks overhead
- code experiments
  - tangible
  - may be mistaken for final implementation

Example: Statements

- Player is a User in Gaming context
- Players have Libraries and Wishlists
- Library is a set of player-owned Games
- Wishlist is a set of Games a Player likes
- Publishers publish Games
- Players buy and play Games

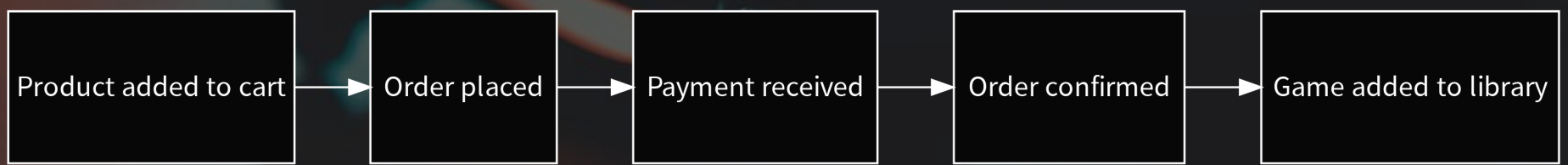
Example: Diagram

Relation of identities



Example: Event Flows

buying a game in the store




Bounded Contexts

Definition

- conceptual area in which a Domain Model is valid
- Vernon: *boundary in which a model exists*
- determines validity of Ubiquitous Language
- often relates to technical decisions
- is not Software Architecture

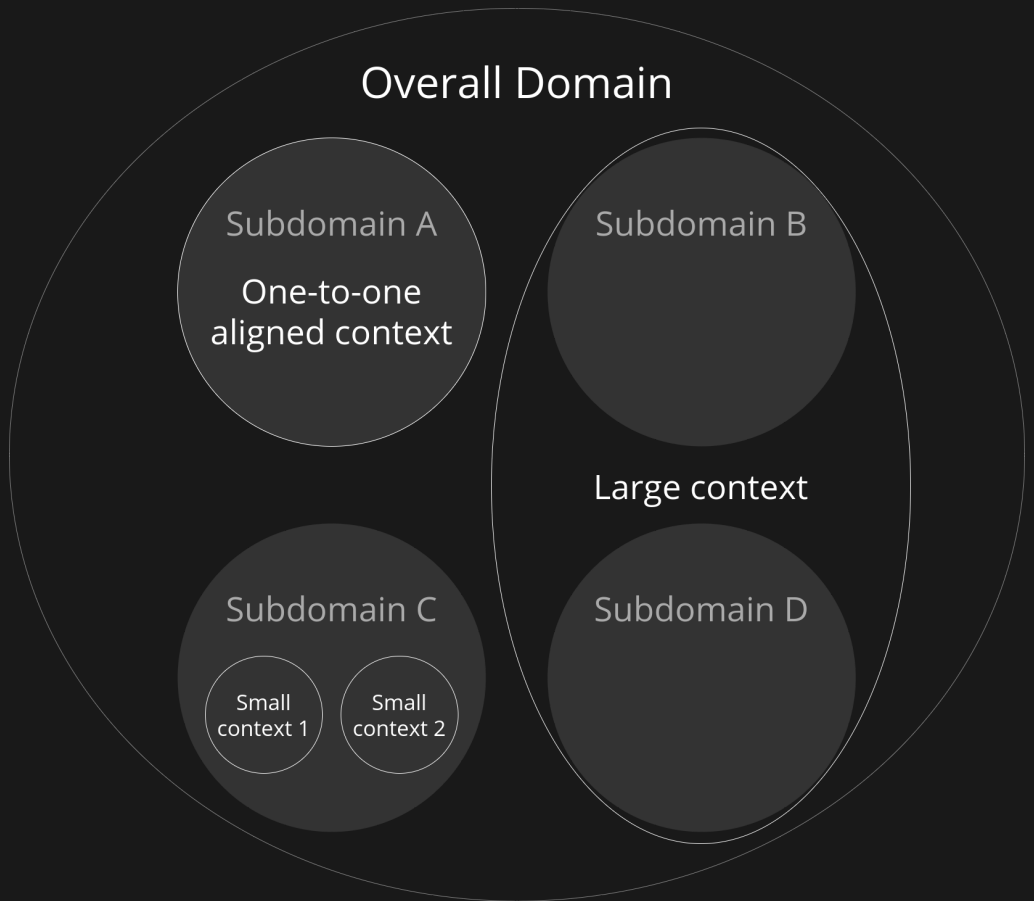
Differentiation of terms

Term	Meaning
Domain	collection of existing knowledge
Domain Model	individually created abstractions
Bounded Context	individually defined boundary

Model and context sizes

- large model in single context
  - ✔️ knowledge uniformity, no ambiguity
  - ❌ higher complexity
  - ❌ more frequent changes & knowledge sync
- smaller models in separate contexts
  - ✔️ lower complexity
  - ✔️ higher expressiveness
  - ⚠ related contexts require integration

Alignment with Subdomains



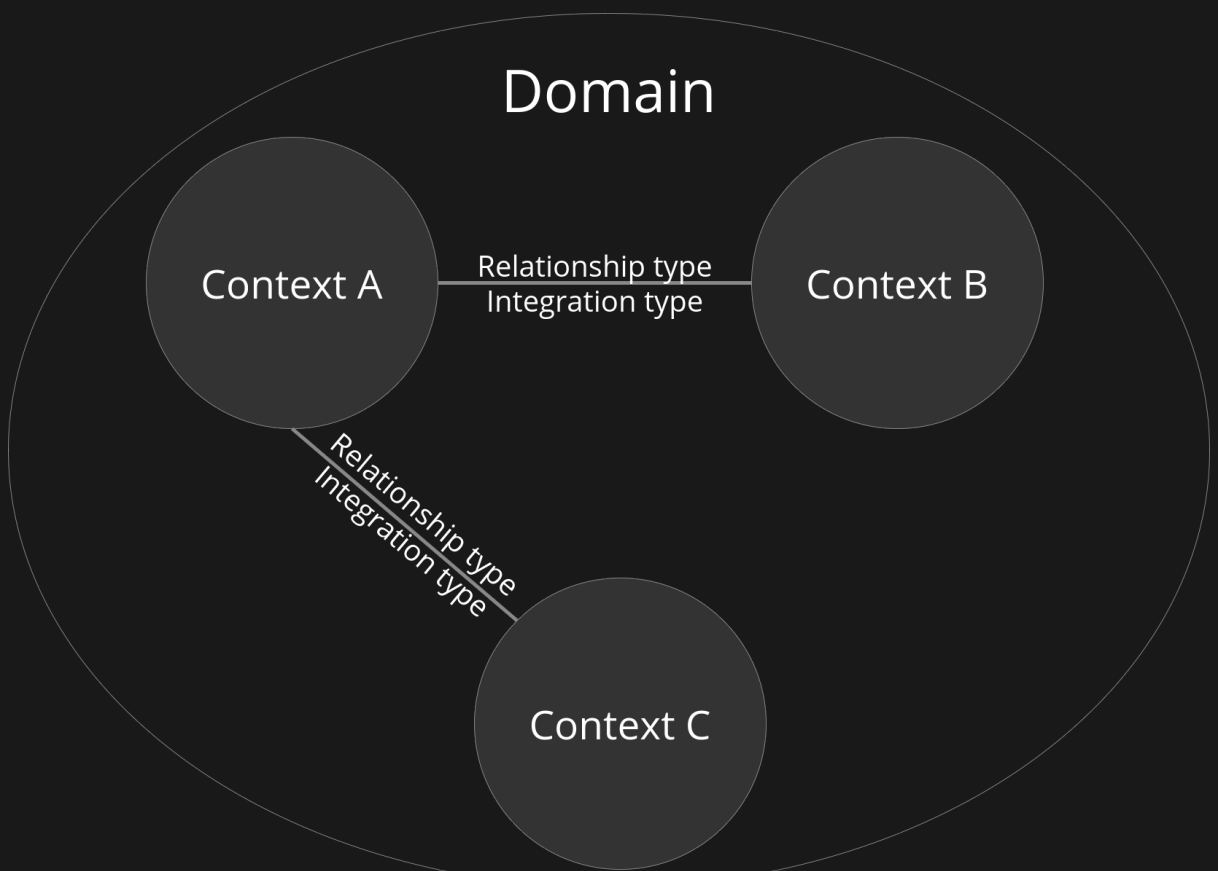
- ideally, contexts and Subdomains align 1-to-1
- some contexts may affect multiple Subdomains

Technological boundaries

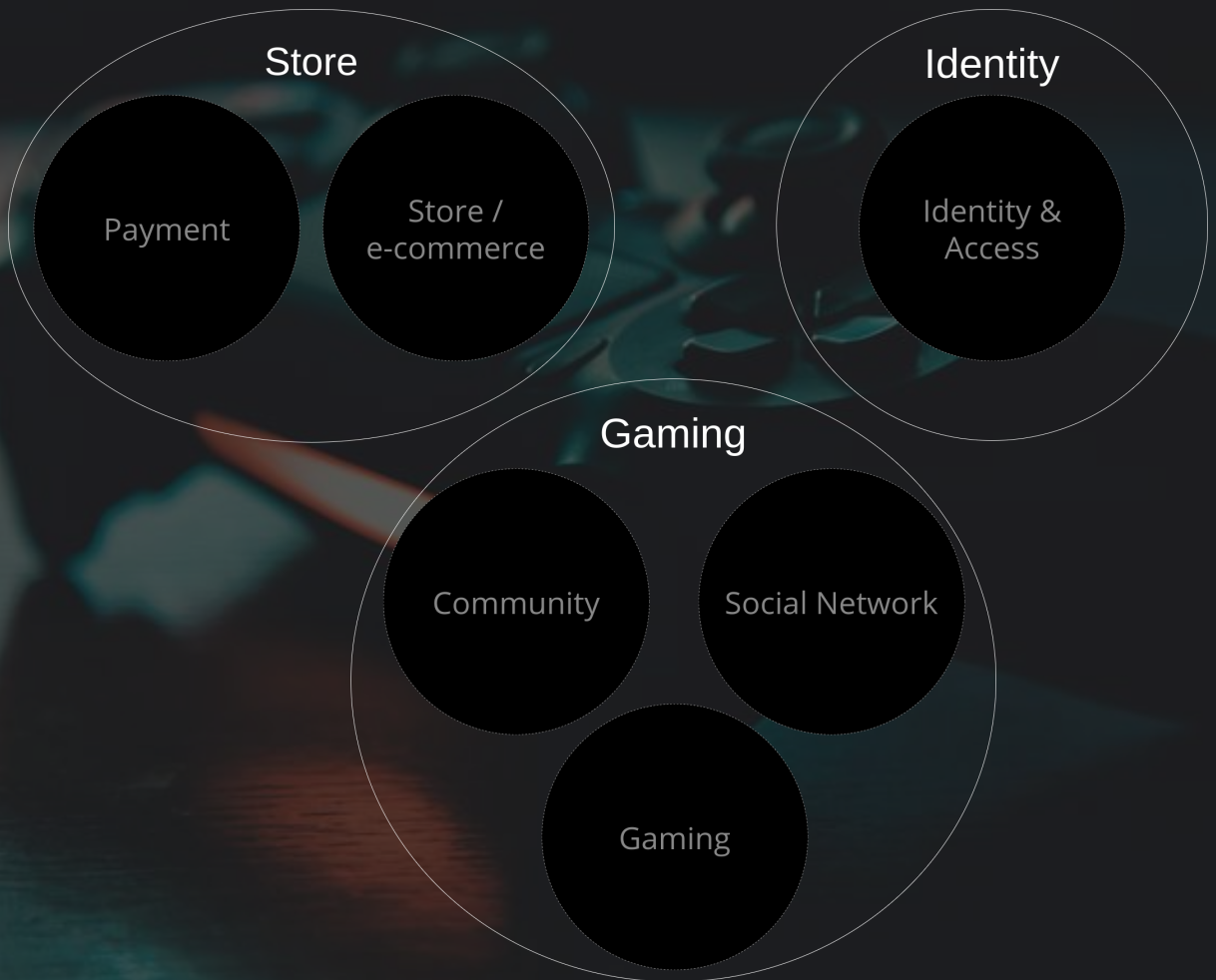
- BCs often align with technological structures
- division can be any kind
  - modules in monolithic software
  - separate programs on host
  - groups of distributed services
- make relationships and integration explicit

Context Maps

visualize Bounded Contexts and their relationships



Example: Bounded Contexts




Software
Architecture

Scope

- architectural patterns that fit well with DDD
- produce modular software with decoupled parts
- difference between system level and service level

Common software parts

- "every" software consists of multiple parts
- distinct responsibilities and relationships
- conceptual parts, different technological options
- for DDD-based software, typically four parts

Domain

- hosts Domain Model implementation
- no outbound dependencies
- Value Objects, Entities, Aggregates, Services
- Domain Events
- persistence-related abstractions

Infrastructure

- technical functionalities
- Vernon: *low-level services for the application*
- may have dependencies and use technologies
- persistence, message exchange, security

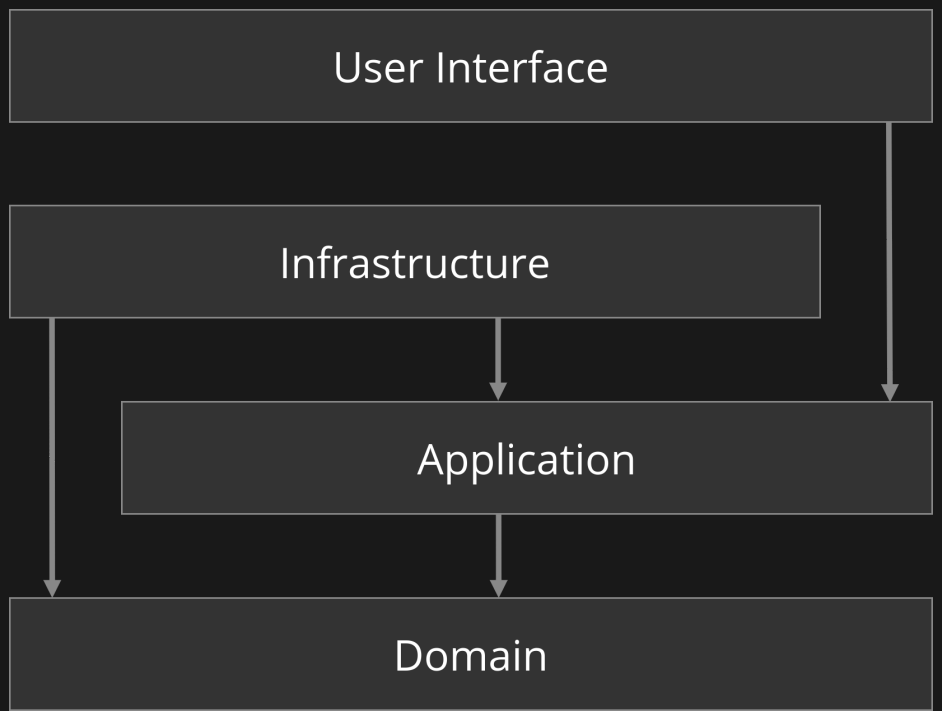
Application

- execution of domain-specific use cases
- utilizes infrastructural functionalities
- works with Domain components
- responsible for transactions and security

User Interface

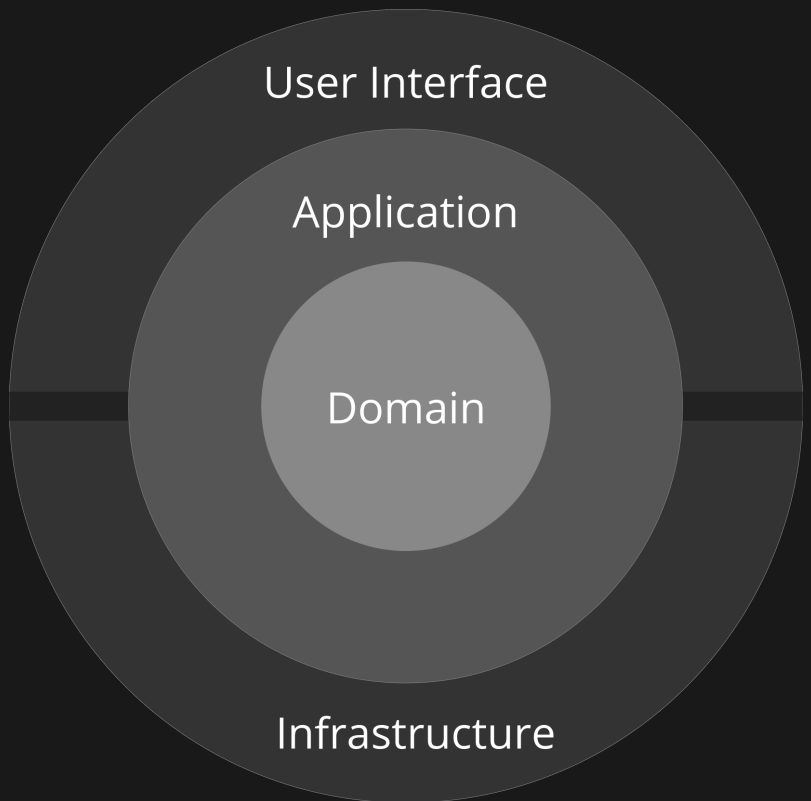
- interface (for humans or machines) to interact
- display information
- validate, accept and translate inputs
- communicate with Application part

Layered Architecture



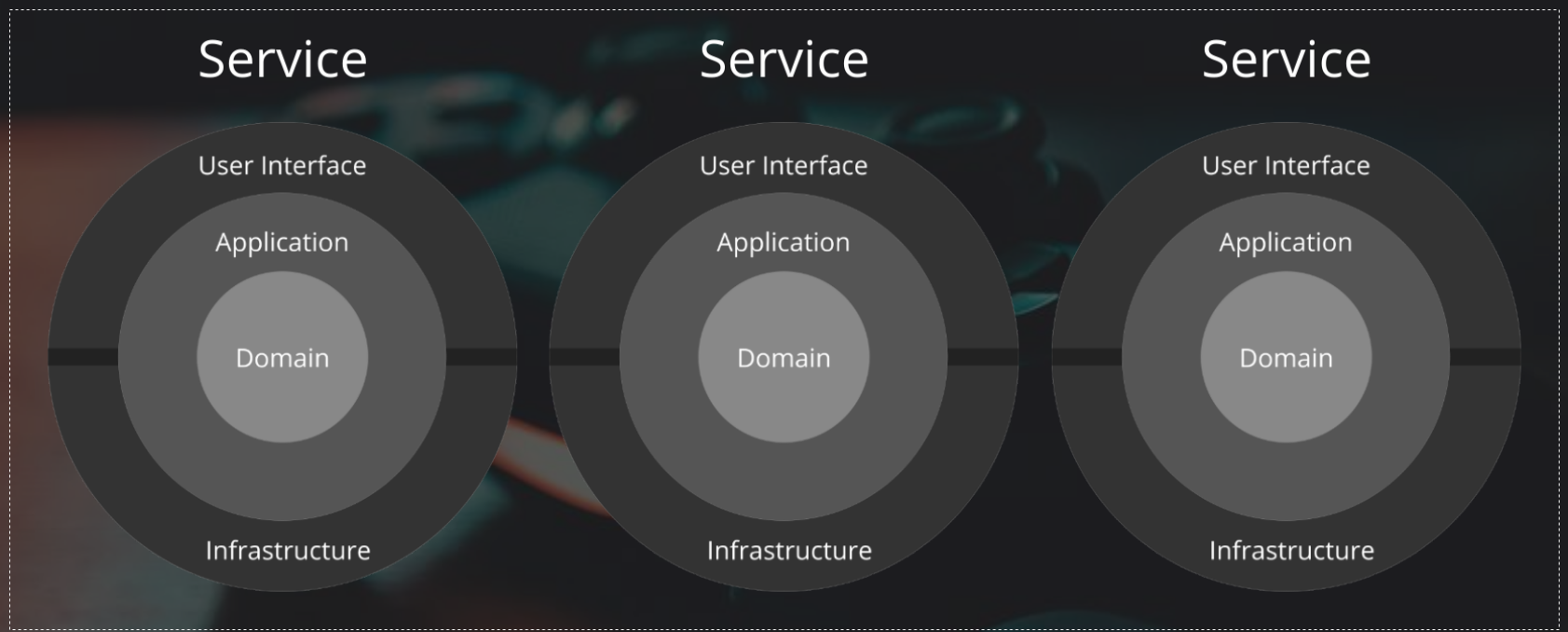
- layers must only depend on themselves & below
- invert dependency direction with abstractions

Onion Architecture



- outer layers depend on inner layers
- again, dependency inversion

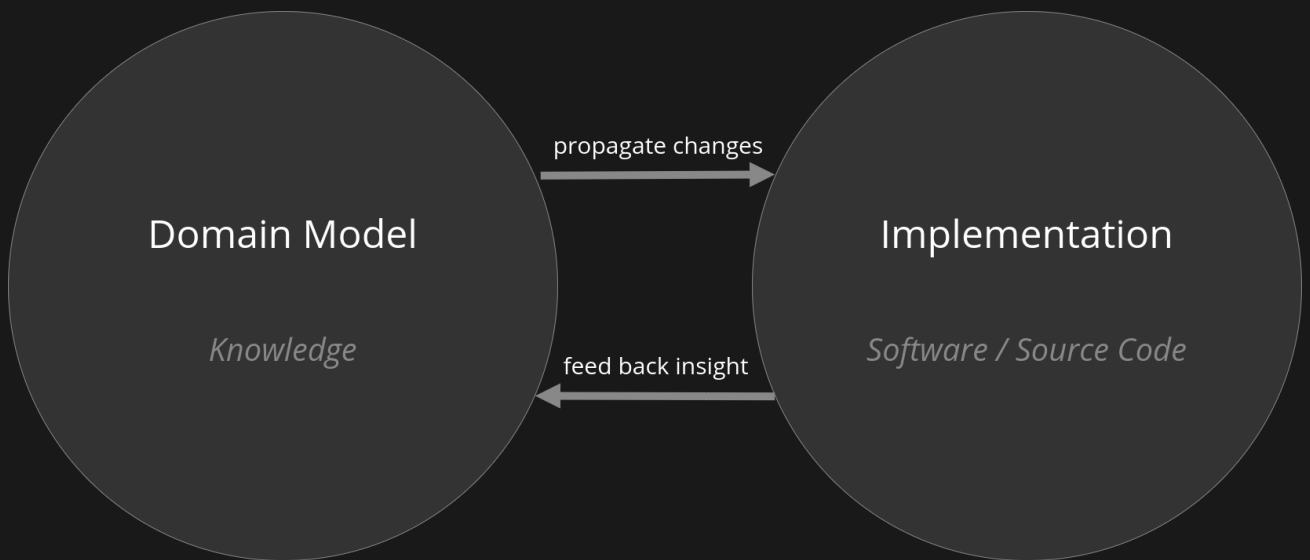
Example: Software Architecture



- units can all employ same architecture
- units can have different architecture
- it depends™


Code quality

Model binding



- model changes require code adjustments
- insights during implementation let model evolve

Readability

- model code as textual knowledge expression
- enables domain discussions based on code
- simplifies knowledge transfer
- avoids unnecessary additional complexity

Example: Readability

```
const calc = (a, b) => {
  const p = 100 / Math.max(a.length, b.length);
  return a.reduce((s, x) => s + b.includes(x), 0) * p;
};
```

vs.

```
const getMatchingScore = (genresA, genresB) => {
  const maximum = Math.max(genresA.length, genresB.length);
  const pointsPerMatch = 100 / maximum;
  const matches = genresA.filter(g => genresB.includes(g));
  return matches.length * pointsPerMatch;
};
```

calculate similarities in preferred genres

Dealing with dependencies

- focus on own behavior, exclude foreign details
- Dependency inversion
  - instead of one part depending on another, both depend on abstraction
  - at runtime, abstraction is substituted with concrete functionalities

Example: Dependency injection (1)

```
class Chat {
  #languageFilter = new TermFilter(); #messages = [];

  writeMessage(author, message) {
    this.messages.push({author,
      message: languageFilter.filter(message)});
  }
}
```

chat depends directly on foreign component

Example: Dependency injection (2)

```
class Chat {
  #languageFilter; #messages = [];

  constructor({languageFilter}) {
    this.#languageFilter = languageFilter;
  }

  writeMessage(author, message) {
    this.#messages.push({author,
      message: this.#languageFilter.filter(message)});
  }
}

const languageFilter = new TermFilter(); // or any other
const chat = new Chat({languageFilter});
```

Command-Query-Separation

- functions are either command or query
- commands modify state, no return value
- queries return data, do not change state
- universally useful, with exceptions

Example: CQS (1)

```
class GamingDevices {
  #devices = {};

  // is executed from unverified device
  registerNewDevice(id) {
    if (this.#devices[id]) throw new Error();
    const verificationCode = generateId();
    this.#devices[id] = {verificationCode, verified: false};
    return verificationCode; // gets sent via e-mail
  }

  // is executed through e-mail link
  confirmNewDevice(id, verificationCode) { /* .. */ }
}
```

what if verification code must be resent?

Example: CQS (2)

```
class GamingDevices {
  #devices = {};

  registerNewGamingDevice(id) {
    if (this.#devices[id]) throw new Error();
    const verificationCode = generateId();
    this.#devices[id] = {verificationCode, verified: false};
  }

  getVerificationCode(id) {
    return this.#devices[id].verificationCode;
  }

  confirmNewDevice(id, verificationCode) { /* .. */ }
}
```


Value Objects,
Entities, Services

Context

- tactical patterns for design and implementation
- patterns can be applied independently

Value Objects

- describe aspect without conceptual identity
- are defined by their attributes
- Evans: *care about only for what they are, not who or which they are*
- should be used extensively

VO characteristics

- Conceptual Whole
  - enclose attributes and behavior
- Equality by Values
  - VOs are equal when attributes match
  - matter of equality, not identity
- Immutability
  - no change after initial assignment
  - free of side effects

Example: Value Objects (1)

```
class Name {
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
    Object.freeze(this);
  }

  equals(name) {
    return this.firstName == name.firstName &&
      this.lastName == name.lastName;
  }

  change(newNameComponents) {
    return new Name({firstName: this.firstName,
      lastName: this.lastName, ...newNameComponents});
  }
}

const name = new Name('Alex', 'Lawrence');
const newName = name.change({firstName: 'Alexander'});
```

Example: Value Objects (2)

```
type Currency = 'EUR' | 'USD' /* .. */;
type Country = 'Germany' | 'USA' | /* .. */;

type Money = {
  value: number,
  currency: Currency,
};

type Price = {
  [country in Country]: Money
};

const price: Price = {
  USA: {value: 10, currency: 'USD'},
  Germany: {value: 10, currency: 'EUR'}
};
```

Entities

- unique elements with conceptual identity
- also related attributes and behavior
- lifecycle throughout which they change
- act as Value Containers
    - use Value Objects wherever possible

Identities

- must be permanent, constant, unique
  - uniqueness may have validity scope
- natural vs. artificial identifiers
  - natural: existing attributes
  - artificial: dedicated value
- identifier generation
  - persistence technology vs. application code
- recommendation: UUID version 4
  - 3a7fd777-e486-4b90-9dd4-ee1531e42172
  - assume generateId() for examples

Entity-to-Entity Relationships

- enclose other Entities
  - enclosed parts do not exist on their own
- identifier references
  - referenced parts exist on their own
- dedicated Value Objects
  - additional information about foreign parts
- favor ID references for bidirectional relation

Example: Entities (1)

```
class Game {
  id; title; description; #genres = [];

  constructor({id, title, description}) {
    Object.defineProperty(this, 'id', {value: id});
    this.title = title;
    this.description = description;
  }

  addGenre(genre) { this.#genres.push(genre); }

  isOfGenre(genre) { return this.#genres.includes(genre); }
}

const tetris = new Game({id: generateId(),
  title: 'Tetris', description: 'falling blocks'});

tetris.title = 'Tetris (Original)';
tetris.addGenre('puzzle');
```

Example: Entities (2)

```
type Game = {
  readonly id: string,
  title: string,
  description: string,
  systemRequirements: SystemRequirements,
};

type SystemRequirements = {
  minimum: SystemSpecs,
  recommended: SystemSpecs,
};

type SystemSpecs = {
  os: OperatingSystem,
  cpu: string,
  ram: string,
  graphics: string
};

type OperatingSystem = 'linux' | 'macos' | 'windows';
```

Example: Entities (3)

```
class Publisher { // encloses Entities
  id; #games = new Map();

  addGame(game) { this.#games.set(game.id, game); }
}
```

```
class Publisher { // references identifiers
  id; #gameIds = [];

  addGame(id) { this.#gameIds.push(id); }
}
```

```
const summary = (game) => ({id: game.id, title: game.title});
class Publisher { // dedicated Value Object, needs sync
  id; #gameSummaries = new Map();

  addGame(summary) {
    this.#gameSummaries.set(summary.id, summary);
  }
}
```

Domain Services

- computations without domain-specific state
- functionalities with external dependencies

Example: Domain Services

```
const calculateAverageGameRating = (reviews) => {
  const sum = reviews.reduce(
    (sum, review) => sum + review.score, 0);
  return sum / reviews.length;
};
```

Invariants

- constraint that must satisfied during lifecycle of affected components
- Vernon: *state that must stay transactionally consistent*
- difference to validation
  - invariants prevent invalid state
  - validation subjects may be invalid

Example: Invariants

```
const MAX_GENRES = 5;

class Game {
  id; title; description; #genres = [];

  constructor({id, title, description}) { /* .. */ }

  addGenre(genre) {
    if (this.#genres.length >= MAX_GENRES - 1)
      throw new Error(`No more than ${MAX_GENRES} genres`);
    this.#genres.push(genre);
  }

  get genres() { return this.#genres.slice(); }
}
```

Domain Events

Definition

- something that happened in a Domain
- domain knowledge about specific occurrence
- Evans: *full-fledged part of Domain Model*
- facilitates communication, integration

Event-driven Architecture

- architectural style
- loosely coupled notification exchange
- reduce/avoid direct dependencies
- increase scalability and resilience
- style is independent of Domain Events

Naming conventions

- event type should describe what happened
- expressive and unambiguous
- use respective Ubiquitous Language
- subject + executed action in past tense

Example: Event types

- OrderPlaced
- PaymentReceived
- GameAddedToLibrary
- ChatMessageWritten

Structure and content

- immutable attributes without functions
- consists of data and metadata
- specialized data
  - event type
  - identity of affected subject
  - further custom data
- metadata
  - event identifier
  - time of occurrence
  - further meta information

Example: Event structure

```ts
type Event<Type extends string, Data> = {
  id: string,
  type: Type,
  data: Data,
  metadata: {creationTime: number} & Record<string, unknown>,
};

type UserCreatedEvent = Event<'UserCreated', {
  gameId: string,
  libraryId: string,
  gameTitle: string,
}>;

type ChatMessageWrittenEvent = Event<'ChatMessageWritten', {
  chatId: string,
  message: string,
}>;
```

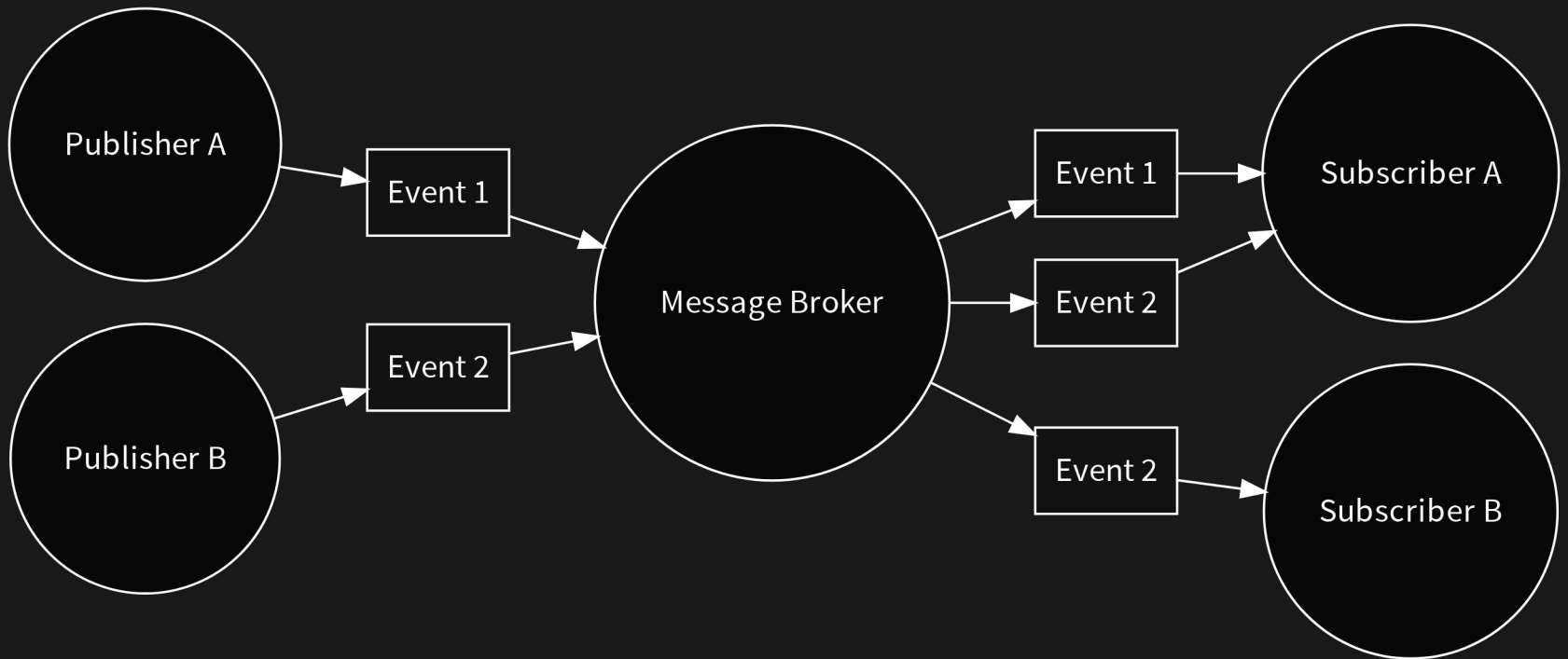
Event creation

- Domain Events originate from Domain Layer
- custom data must be provided upon creation
- metadata should be provided from the outside
- use configurable factory / dependency injection

Example: Event creation

```
const Event = function (type, data) {
  Object.assign(this, {
    id: Event.idGenerator(), type,
    data, metadata: Event.metadataProvider(),
  });
};
Event.idGenerator = generateId;
Event.metadataProvider = () => ({creationTime: Date.now()});

// Domain Layer
const event = new Event(
  'ChatMessageWritten', {chatId, message: 'Hi!'});
/* {
  id: ' 489e17ba-357e-411e-974c-04cdadb76aa2',
  type: 'ChatMessageWritten',
  data: {
    chatId: 'b829c57a-0947-4f4e-b3e6-6937f5defcbf',
    message: 'Hi!',
  },
  metadata: {creationTime: 1615567680504},
} */
```

Event distribution



- Publish-Subscribe pattern
- message broker as central component
- no coupling between producers and consumers

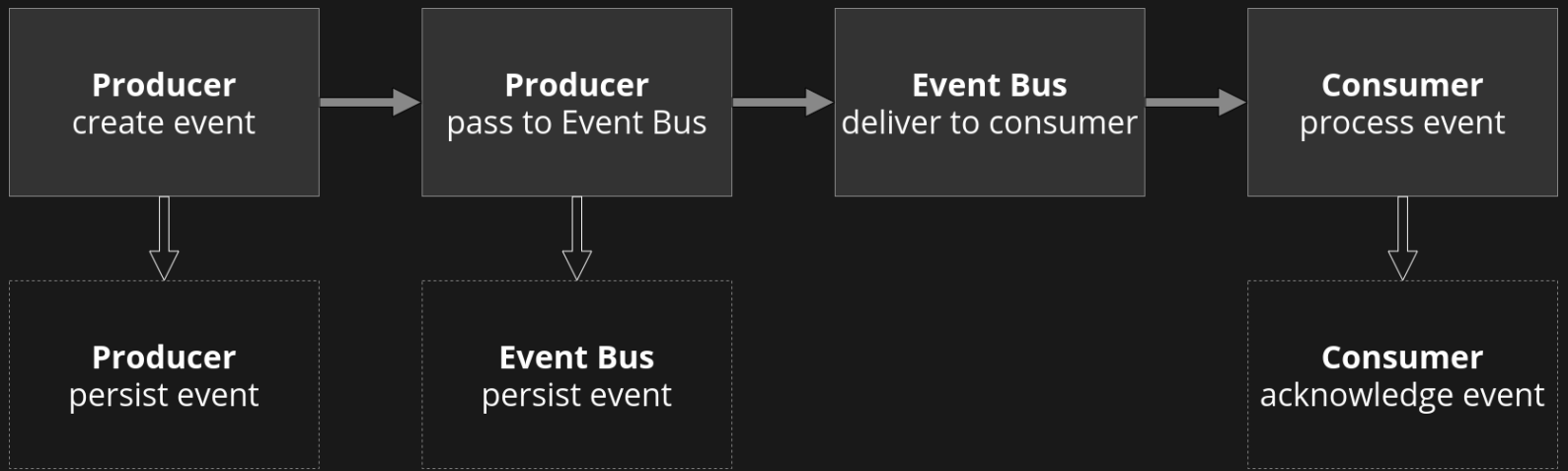
Example: Event distribution

```
class User {
  constructor({id, username, emailAddress, name}) {
    /* state assignment */
    const {firstName, lastName} = name;
    eventBus.publish(new Event('UserCreated', {
      userId: id, username, emailAddress, firstName, lastName
    }));
  }
  /* .. */
}
```

```
eventBus.subscribe('UserCreated', async (event) => {
  const {emailAddress, firstName, lastName} = event.data;
  await sendEmail(emailAddress, {
    subject: 'Welcome to our gaming platform!',
    message: `Great to have you, ${firstName} ${lastName}!`,
  });
});
```

Domain should not know of Event Bus (infrastructure)

Delivery guarantees



- crucial actions and IPC require guarantees
- use persistent queues and acknowledgement
- "at least once" guarantee (duplicates may occur)
  - duplicate detection
  - idempotent processing
- avoid order guarantees

Example: Delivery guarantees

```
eventBus.subscribe('UserCreated', async (event, ack) => {
  const {emailAddress, firstName, lastName} = event.data;
  await sendEmail(emailAddress, { /* .. */ });
  ack();
});
```

```
const processedUsers = [];
eventBus.subscribe('UserCreated', async (event, ack) => {
  const {emailAddress, firstName, lastName} = event.data;
  if (processedUsers.includes(event.data.userId)) return;
  await sendEmail(emailAddress, { /* .. */ });
  processedUsers.push(event.data.userId);
  ack();
});
```

```
eventBus.subscribe('UserCreated', async (event, ack) => {
  const {userId, emailAddress} = event.data;
  await createOrUpdatePlayer({userId, emailAddress});
  ack();
});
```


Aggregates

Definition

- transactional consistency boundary
- encloses related model components
- Evans: *unit for the purpose of data changes*
- essential for persistence with concurrent access
- no specific technical constructs

Transactions

- treat multiple steps as inseparable procedure
- ACID principles apply
  - Atomicity
  - Consistency
  - Isolation
  - Durability

Structure and access

- combination of Entities and Value Objects
- one Entity is Aggregate Root
- other parts may have local identity
- root acts as facade and ensures consistency

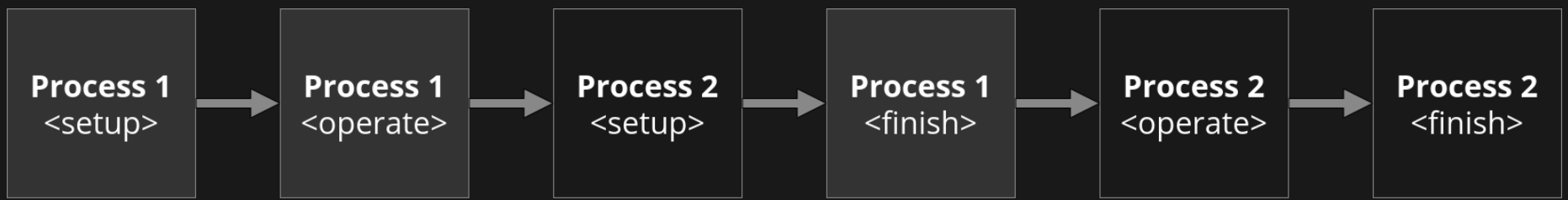
Example: Structure & access (1)

```
class Notification {
  id; content; #isRead = false;

  constructor({id, content}) {
    Object.defineProperty(this, 'id', {value: id});
    Object.defineProperty(this, 'content', {value: content});
  }

  markAsRead() { this.#isRead = true; }

  get isRead() { return this.#isRead; }
}
```

Example: Structure & access (2)

```
class NotificationCenter {
  #notifications = new Map();

  addNewNotification(id, content) {
    const notification = new Notification({id, content});
    this.#notifications.set(id, notification);
  }

  markNotificationAsRead(id) {
    this.#notifications.get(id).markAsRead();
  }

  markAllNotificationsAsRead() {
    Array.from(this.#notifications.values())
    .forEach(notification => notification.markAsRead());
  }

}
```

Concurrency



- execute multiple tasks in overlapping periods
- requires threading or event loop
- increase in performance
- persistence implies concurrency

Concurrency Control

- isolate resource representation
- prevent overwrites
- pessimistic approach
  - exclusive resource access
  - simple, but affects performance
- optimistic approach
  - concurrent resource access
  - updates must be based on latest version
  - better performance, more complex

Example: Concurrency (1)

```
const writeMessage = async (chatId, authorId, message) => {
  const chat = await chatStorage.load(chatId);
  chat.writeMessage({authorId, message});
  await chatStorage.save(chat);
};

// example use case
await Promise.all([
  writeMessage(chatId, author1Id, "Hey! What's up?"}),
  writeMessage(chatId, author2Id, 'Wanna play something?'}),
]);
console.log(await chatStorage.load(chatId));
```

one of the messages will not be persisted

Example: Concurrency (2)

```
let queue = Promise.resolve();
const writeMessage = (chatId, authorId, message) {
  queue = queue.then(async () => {
    const chat = await chatStorage.load(chatId);
    chat.writeMessage({authorId, message});
    await chatStorage.save(chat);
  });
  return queue;
};

await Promise.all([
  writeMessage(chatId, author1Id, "Hey! What's up?"}),
  writeMessage(chatId, author2Id, 'Wanna play something?'}),
]);
```

- second call only starts after first finished
- exclusive access

Design considerations

- align with specific Domain Model concept
- as small as possible
- parts that share invariant inside same Aggregate
- persistence performance matters
- aggregate relationships via identifier references
    - consistency boundary cannot contain others

Example: Design considerations

- invariant viewpoint
  - 1 review per player per game
  - game and reviews belong in same Aggregate
- concurrency viewpoint
  - review changes should not affect each other
  - games and reviews are separate Aggregates
- size / performance viewpoint
  - most reviews for a game on Steam: ~235.000
  - game and reviews are best saved separately

question: is there really an invariant?

Eventual Consistency

- information across transactional boundaries
  - identical, derived, following actions
- non-transactional change synchronization
  - can cause staleness
  - should be fast
  - must be reliable
- sync strategy
  - query for changes
  - send notifications (with Domain Events)

Example: Consistency (1)

- Aggregates: Publisher, Game
- Publisher encloses game summary Value Objects

```
class Publisher {
  id; #gameSummaries = new Map();

  addGame({id, title}) {
    this.#gameSummaries.set(id, {id, title});
  }

  updateGame({id, title}) {
    this.#gameSummaries.set(id, {id, title});
  }
}
```

rare use case: what if game title must change?

Example: Consistency (2)

```
class Game {
  /* .. */
  changeTitle(title) {
    this.#title = title;
    eventBus.publish(new Event('GameTitleChanged', {
      gameId: this.id,
      publisherId: this.publisherId,
      title,
    }));
  }
  /* .. */
}
```

```
eventBus.subscribe('GameTitleChanged', async (event) => {
  const {publisher, gameId, title} = event.data;
  const publisher = await publisherStorage.load(publisherId);
  publisher.updateGame({id: gameId, title});
  await publisherStorage.save(publisher);
});
```


Repositories

Definition

- persistence in meaningful way to Domain
- interfaces express Domain Model knowledge
- incorporate Ubiquitous Language
- encapsulate infrastructural details
- 1-to-1 alignment with Aggregates

Design and implementation

- technology-agnostic interface
- base functionality: save Aggregate, load by id
- custom queries
  - find Aggregates based on criteria
  - compute results
- differentiate between objects and data
- model components must accept and expose data

Example: Repositories

```
interface Repository<Entity> {
  save(entity: Entity): Promise<void>;
  load(id: string): Promise<Entity | null | undefined>;
}

interface GameRepository extends Repository<Game> {
  findGamesWithGenre(genre: string): Promise<Game[]>;
  findGamesOfPublisher(publisherId: string): Promise<Game[]>;
}
```

- interfaces express domain-specific aspects
- implementation can use any storage technology

```
class FilesystemRepository implements Repository<T> { }
class PostgreSQLRepository implements Repository<T> { }
class MongoDBGameRepository implements GameRepository { }
```

Optimistic Concurrency

- allow concurrent reads and writes
- verify actuality of write requests
- requests based on old status produce conflict
- implementation: timestamps vs. versions
- version can be attribute of Aggregate Root

Example: Optimistic Concurrency (1)

```
class ChatRepository {
  async save(chat) {
    await this.#storage.execute(async (transaction) => {
      const {version} =
        await transaction.load('chat', chat.id);
      if (version != chat.baseVersion)
        throw new Error('concurrency');
      await transaction.save('chat', chat.id, chat);
    });
  }

  async load(id) { return await this.#storage.load(id); }
}
```

Example: Optimistic Concurrency (2)

```
const writeMessage = async (chatId, authorId, message) => {
  const chat = await chatRepository.load(chatId);
  chat.writeMessage({authorId, message});
  await chatRepository.save(chat);
};

try {
  await Promise.all([
    writeMessage(chatId, author1Id, "Hey! What's up?"}),
    writeMessage(chatId, author2Id, 'Wanna play?'}),
  ]);
} catch (error) {
  console.log(error);
}
```

Concurrency conflicts

- design Aggregates towards transactional success
- frequent conflicts can hint to modeling issues
- action opportunities
  - report conflict
  - automatic retry
  - custom resolution mechanism

Example: Concurrency conflicts

automatic retry:

```
const writeMessage = async (data, attempt = 0) => {
  const {chatId, authorId, message} = data;
  const chat = await chatRepository.load(chatId);
  chat.writeMessage({authorId, message});
  try {
    await chatRepository.save(chat);
  } catch (error) {
    if (error.message == 'conflict' && attempt < 5) {
      await new Promise(resolve => setTimeout(resolve, 15));
      await writeMessage(data, attempt + 1);
    }
    else throw error;
  }
};
```

Persistence and event publishing

- persisted state is source of truth
- separate event creation and publishing
- collect new events inside model components
- publish new items after persisting
    - publishing guarantee

Example: Event publishing

```
class Order {
  /* .. */
  confirm() {
    eventBus.publish(new Event('OrderConfirmed', {
      orderId: this.id,
      userId: this.userId,
      items: this.#items,
    }));
  }
}
```

```
eventBus.subscribe('OrderConfirmed', async (event) => {
  const {userId, items} = event.data;
  const player = await playerRepo.findByUserId(userId);
  const library = await libraryRepo.load(player.libraryId);
  items.filter(item => item.type == 'game')
  .map(g => g.gameId).forEach(id => library.addGame(id));
});
```

what if publishing fails?

Transactional outbox

- persist new events together with state
  - one transaction
  - separate event collection/table
- wait for new persisted events and publish
  - mark or delete after publishing
  - "at least once" guarantee

Example: Outbox (1)

```
class Order {
  #newEvents = [];

  get newEvents() { return this.#newEvents.slice(); }

  confirm() {
    this.#newEvents.push(new Event('OrderConfirmed', {
      orderId: this.id,
      userId: this.userId,
      items: this.#items,
    }));
  }
}
```

Example: Outbox (2)

```
class OrderRepository {
  async save(order) {
    await this.#storage.execute(async (transaction) => {
      await transaction.save('order', order.id, order);
      for (const event of order.newEvents)
        await transaction.save('event', event.id, event));
    });
  }
}
```

```
const publishNewEvents = async() => {
  await storage.execute(async (transaction) => {
    const events =
      await transaction.find('event', {published: false});
    for (const event of events) {
      await eventBus.publish(event);
      await transaction.update(
        'event', event.id, {published: true});
    }
  });
};
```

Application Services

Application Layer

- execute use cases
- manage transactions
- handle security
- implemented as Application Services
  - dedicated vs. categorized style

Use case scenarios

typical scenarios

- load component, execute actions, persist change
- create component, persist data
- load component, (compute), return result

atypical scenarios

- load multiple, perform interaction, persist change
- perform stateless computation, return result

Example: Use case scenarios

```
class ForumApplicationServices {
  async addDiscussion({discussionId, forumId, title}) {
    await this.#discussionRepository.save(
      new Discussion({id: discussionId, forumId, title}));
  }

  async writePost(data) {
    const {discussionId, authorId, postId, content} = data;
    const discussion =
      await this.#discussionRepository.load(discussionId);
    discussion.writePost({authorId, postId, content});
    await this.#discussionRepository.save(discussion);
  }

  async getDiscussionCount({forumId}) {
    return await
      this.#discussionRepository.countDiscussions(forumId);
  }
}
```

Transactions

- one service should affect one transaction
- not create or modify more than one Aggregate
- possibility of interruption risks corrupted state
- execute secondary functionalities deferred

Processes

- use case that incorporates multiple transactions
- implementation approaches
    1. single service affects multiple transactions
    2. consumer orchestrates steps
    3. steps are connected via notifications
        - orchestration vs. choreography

Example: Processes (1)

```
class ForumApplicationServices {
  async archiveForum({forumId}) {
    const forum = await this.#forumRepository.load(forumId);
    forum.archive();
    await this.#forumRepository.save(forum);
    const discussions =
      await this.#discussionRepository.findByForum(forumId));
    for (const discussion in discussions) {
      discussion.lock();
      await this.#discussionRepository.save(discussion);
    }
  }

  async lockDiscussion({discussionId}) { /* .. */ }
}
```

worst case: forum archived, some discussions locked

Example: Processes (2)

```
class ForumApplicationServices {
  async archiveForum({forumId}) {
    const forum = await this.#forumRepository.load(forumId);
    forum.archive();
    await this.#forumRepository.save(forum);
  }

  async lockDiscussion({discussionId}) {
    const discussion =
      await this.#discussionRepository.load(discussionId);
    discussion.lock();
    await this.#discussionRepository.save(discussion);
  }
}
```

```
await services.archiveForum({forumId});
Promise.all((await services.findDiscussions({forumId})).map(
  (discussionId) => services.lockDiscussion({discussionId})))
```

responsibility shifted to consumer

Example: Processes (3)

```
class Forum {
  #isArchived = false; #newEvents = [];

  archive() {
    this.#isArchived = true;
    this.#newEvents.push(
      new Event('ForumArchived', {forumId: this.id});
  }
}
```

```
class ForumApplicationServices {
  async archiveForum({forumId}) {
    const forum = await this.#forumRepository.load(forumId);
    forum.archive();
    await this.#forumRepository.save(forum);
  }
}
```

Example: Processes (4)

```
class ForumDomainEventHandlers {

  constructor({eventBus, discussionRepository}) {
    this.#discussionRepository = discussionRepository;
    eventBus.subscribe('ForumArchived',
      this.handleForumArchivedEvent.bind(this));
  }

  async handleForumArchivedEvent({data: {forumId}}, ack) {
    const discussions =
      await this.#discussionRepository.findByForum(forumId);
    await Promise.all(discussion.map(async (discussion) => {
      discussion.lock();
      await this.#discussionRepository.save(discussion);
    }));
    ack();
  }
}
```

guaranteed process (at least once)

Cross-cutting concerns

- Application manages aspects across use cases
- security, validation, error handling, logging, etc.
- favor abstract and unified integration

Example: Cross-cutting concerns

```
const LoggingProxy = function(target) {
  const get = (target, property) => {
    return async (...args) => {
      const result = await target[property](...args);
      console.log(`executed ${target}`);
      console.log(`arguments: ${args}, result: ${result}`);
      return result;
    }
  };

  return new Proxy(target, {get});
}
```

```
const services = new ApplicationServices({/* .. */});
const servicesWithLogging = new LoggingProxy(services);
```

Notes on authentication and authorization

- infrastructural concerns
- authentication as middleware
- authorization as abstract interface

Example: Authentication

```
const AuthenticationProxy = function(target) {
  const get = (target, property) => {
    return async (data, metadata) => {
      const {authentication: {userId}} = metadata;
      if (!await authentication.isAuthenticated(userId))
        throw new Error('not authenticated');
      return await target[property](data, metadata);
    }
  };

  return new Proxy(target, {get});
}
```

```
const services = new ApplicationServices({/* .. */});
const servicesWithAuth = new AuthenticationProxy(services);
```

Example: Authorization

```
class ForumApplicationServices {
  constructor({authorization, collaboratorRepository}) {
    this.#authorization = authorization;
    this.#collaboratorRepository = collaboratorRepository;
  }

  async makeCollaboratorAdmin(data, metadata) {
    const {collaboratorId} = data;
    const {authentication: {userId}} = metadata;
    await this.#authorization.verify(userId, 'forum/admin');
    const collaborator =
      await this.#collaboratorRepository.load(collaboratorId)
    collaborator.isAdmin = true;
    await this.#collaboratorRepository.save(collaborator);
    await this.#authorization.grant(
      collaborator.userId, 'forum/admin');
  }
}
```

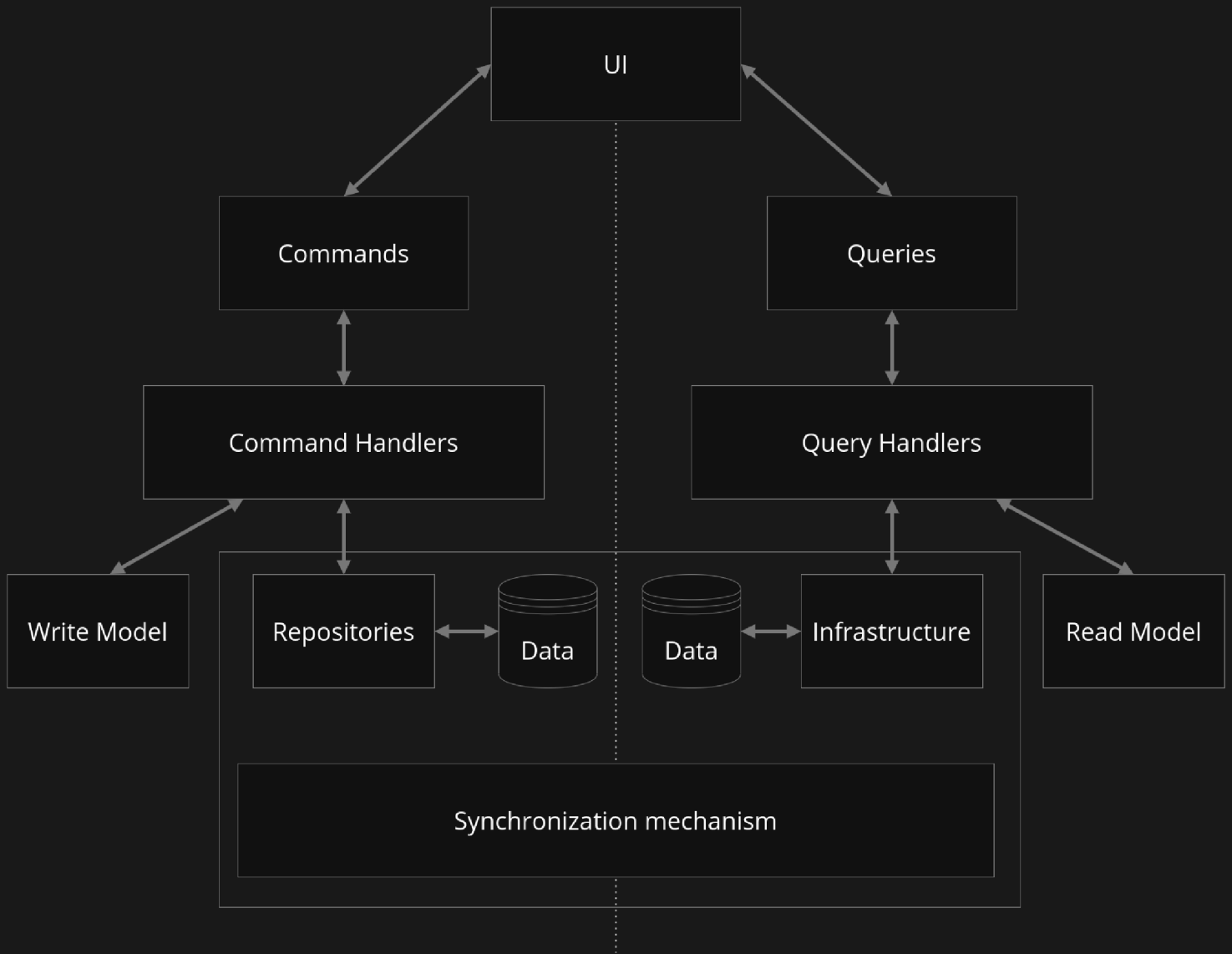
⚠ grant/revoke is separate transaction


CQRS

Overview

- architectural pattern
- separate write and read side
- differences in data, consistency, performance
- allows individual scaling
- optionally, separate storages for write and read
    - this talk assumes separate storages

High-level picture



Write and Read Model

- model implementation split into two
- Domain Model may reflect separation
- persistence is split accordingly

Write Model

- actions that cause state changes
- Aggregate with only commands
- Repository with basic operations
- transactions, consistency, invariants
- operate on current state

Read Model

- data and structures for querying
- may be de-normalized, derived and aggregated
- concrete components or structural description
- persistent or in-memory
- expected to be eventually consistent

Example: Write Model

```
class Review {

  id; gameId; text; #rating;

  constructor({id, gameId, text = '', rating}) {
    Object.defineProperty(this, 'id', {value: id});
    Object.defineProperty(this, 'gameId', {value: gameId});
    this.rating = rating;
    this.text = text;
  }

  set rating(rating) {
    if (rating < 0 || rating > 5) throw new Error('invalid');
    this.#rating = rating;
  }

}
```

Example: Read Model

```
class GameRatings {

  gameId; #ratings = new Map(); #averageRating = 0;

  constructor({gameId}) {
    Object.defineProperty(this, 'gameId', {value: gameId});
  }

  addOrUpdateRating(reviewId, rating) {
    this.#ratings.set(reviewId, rating);
    this.#averageRating = Array.from(this.#ratings.values())
    .reduce((sum, rating) => sum + rating, 0);
  }

  get averageRating() { return this.#averageRating; }

}
```

Read Model synchronization

- separate storages necessitate synchronization
- write side changes must propagate to read side
- non-transactional, asynchronous, reliable
- complex transformations are Domain concerns

Synchronization approaches

- push updates from write side
  - ✔️ simple
  - ❌ write side depends on read side
- lazily compute read data (+ caching)
  - ✔️ only compute when needed
  - ⚠ performance, information access
- event notifications trigger read updates
  - ✔️ write side stays independent
  - ✔️ sync mechanism is reliable
  - ⚠ increased complexity

Example: Push updates

```
const createReview = async (data) => {
  const {reviewId, gameId, text, rating} = data;
  const review = new Review({reviewId, gameId, text, rating});
  await reviewRepository.save(review);
  const ratings = gameRatingsStorage.load(gameId);
  ratings.addOrUpdateRating(reviewId, rating);
  await gameRatingsStorage.save(ratings);
};

const updateReviewRating = async ({reviewId, rating}) => {
  const review = await reviewRepository.load(id);
  review.rating = rating;
  await reviewRepository.save(review);
  const ratings = gameRatingsStorage.load(review.gameId);
  ratings.addOrUpdateRating(reviewId, rating);
  await gameRatingsStorage.save(ratings);
};
```

operations affect multiple transactions

Example: Compute on demand

```
const createReview = async (data) => {
  const {reviewId, gameId, text, rating} = data;
  await reviewRepository.save(
    new Review({reviewId, gameId, text, rating}));
};

const updateReviewRating = async ({reviewId, rating}) => {
  const review = await reviewRepository.load(reviewId);
  review.rating = rating;
  await reviewRepository.save(review);
};

const getAverageRating = async (gameId) => { // add cache
  const reviews = await reviewRepository.findByGame(gameId);
  return reviews.reduce(
    (sum, review) => sum + review.rating, 0);
};
```

Example: Event notifications (1)

```
class Review {
  /* .. */
  set rating(rating) {
    if (rating < 0 || rating > 5) throw new Error('invalid');
    this.#rating = rating;
    this.#newEvents.push(new Event('ReviewRatingSet',
      {reviewId: this.id, gameId: this.gameId, rating}));
  }
}
```

```
eventBus.subscribe('ReviewRatingSet', async ({data}) => {
  const ratings = await gameRatingsStorage.load(data.gameId);
  ratings.addOrUpdateRating(data.reviewId, data.rating);
  await gameRatingsStorage.save(gameRatings);
});
```

```
const getAverageRating = async (gameId) =>
  (await gameRatingsStorage.load(gameId)).getAverageRating();
```

guaranteed process

Commands and Queries

- intent to execute use case
- commands trigger actions, queries retrieve data
- naming
  - describe use case to execute
  - combine action and subject in present tense
- structure
  - data: type name, custom data
  - metadata: id, timestamp, authentication

Example: Commands/Queries

```ts
type Message<Type extends string, Data> = {
  id: string,
  type: Type,
  data: Data,
  metadata: {
    creationTime: number,
    authentication: Record<string, unknown>,
  } & Record<string, unknown>,
};

type CreateReviewCommand = Message<'CreateReview', {
  reviewId: string,
  gameId: string,
  text: string,
  rating: 0 | 1 | 2 | 3 | 4 | 5,
}>;

type GetAverageRatingQuery = Message<'GetAverageRating', {
  gameId: string,
}>;
```

Commands vs. Events

	Commands	Events
Purpose	Instruction to act	Description of occurrence
Naming	Present tense	Past tense
Communication	One-to-one	One-to-many
Possible actions	Accept/Reject	Acknowledge

- what happened vs. what should happen
- there are no one-way Commands

Command and Query Handlers

- form of Application Services (same rules apply)
- Command Handlers
  - load Write Model, execute action, persist
  - do not return data
- Query Handlers
  - retrieve data, (transform), return
  - free of side effects

Example: Handlers

```
class GameCommandHandlers {
  async updateReviewRating(command) {
    const {reviewId, rating} = command.data;
    const review = await this.#reviewRepository.load(reviewId);
    review.rating = rating;
    await this.#reviewRepository.save(review);
  }
}
```

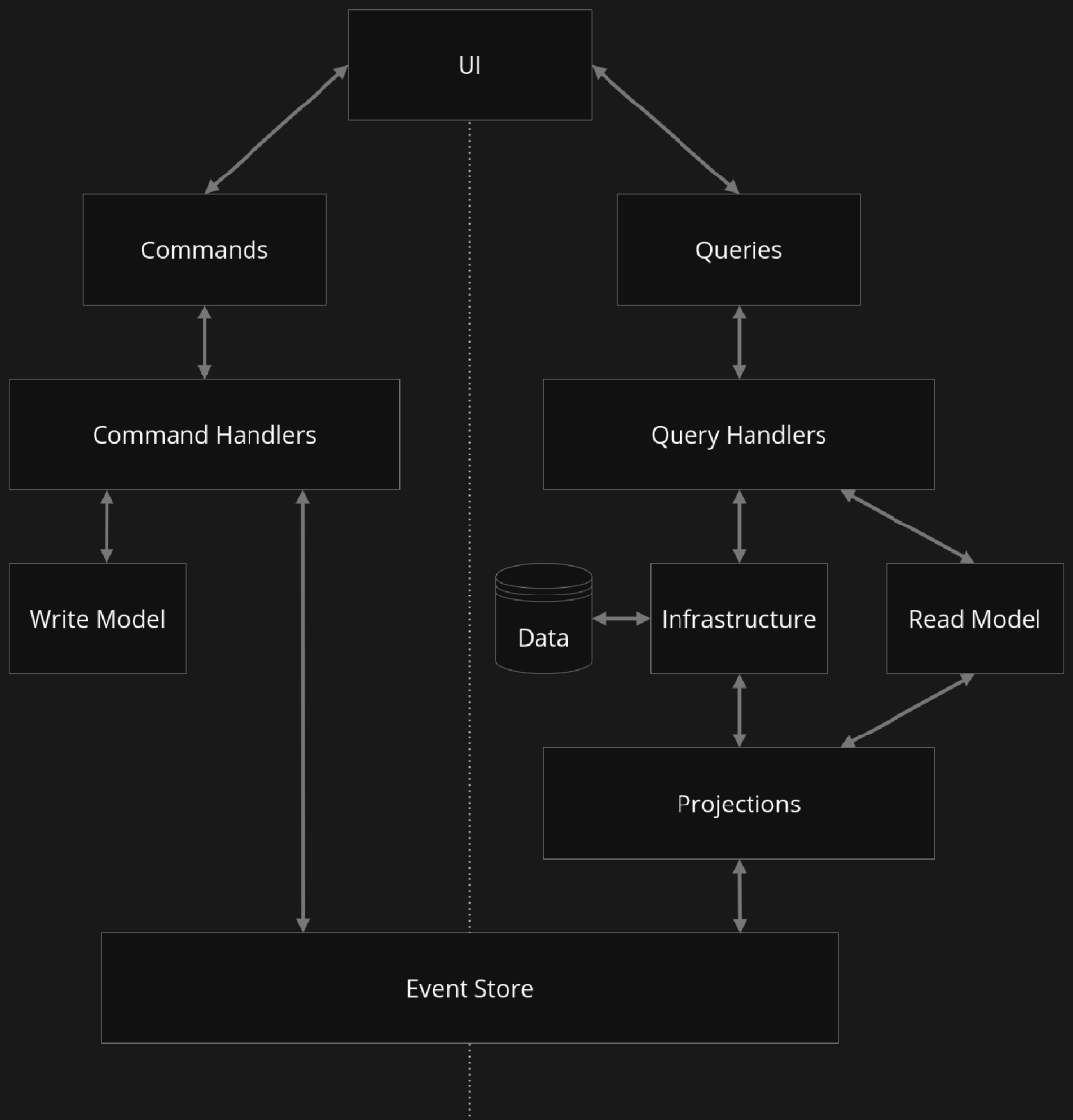
```
class GameQueryHandlers {
  async getAverageRating(query) {
    const {gameId} = query.data;
    const gameRatings =
      await this.#gameRatingsStorage.load(gameId);
    return gameRatings.getAverageRating();
  }
}
```

Event Sourcing

Overview

- architectural pattern
- state is sequence of immutable change events
- emphasizes intentions and behavior
- Young: *time becomes an important factor*
- derive read data from events, even retroactively
- requires CQRS
- apply to selected areas, treat as internal detail

Overall flow



Relation to Domain Events

- similar recommendations on structure & content
- should have no derived or extraneous data
- conceptual differences
  - Domain Events are for notifications
  - ES events are state transitions
- mechanism for translating into Domain Events
  - filter out private events
  - perform transformation / enrichment

Event-sourced Write Model

- requirements
  - offer state state rebuild/update from events
  - behavioral operations produce events
- object-oriented style
  - upon construction, rebuild state from events
  - behavioral functions push events into list
  - as side effect, state is updated internally
- functional style
  - behavior and state are separated
  - state rebuild/update as left-fold
  - behavioral functions return new events

Example: Write Model (w/o ES)

```
class Wishlist {
  id; playerId; #games = [];

  constructor({id, playerId}) {
    Object.defineProperties(this, {
      id: {value: id}, playerId: {value: playerId}});
  }

  addGame(gameId) {
    if (this.#games.includes(gameId)) throw new Error();
    this.#games.push(gameId);
  }

  removeGame(gameId) {
    this.#games.splice(this.#games.indexOf(gameId), 1);
  }

  getGames() { return this.#games.slice(); }
}
```

Example: Write Model (OOP/1)

```
class Wishlist {
  #id; #games = []; newEvents = [];

  create({wishlistId, playerId}) {
    this.#applyNewEvent(new Event('WishlistCreated',
      {wishlistId, playerId}));
  }

  addGame(gameId) {
    if (this.#games.includes(gameId)) throw new Error();
    this.#applyNewEvent(new Event('GameAddedToWishlist',
      {wishlistId: this.#id, gameId}));
  }

  removeGame(gameId) {
    this.#applyNewEvent(new Event('GameRemovedFromWishlist',
      {wishlistId: this.#id, gameId}));
  }
}
```

Example: Write Model (OOP/2)

```
class Wishlist {
  #id; #games = []; newEvents = [];

  constructor(events = []) {
    events.forEach(event => this.#applyEvent(event));
  }
  /* .. create(), addGame(), removeGame() .. */

  #applyEvent({type, data}) {
    if (type == 'WishlistCreated') this.#id = data.wishlistId;
    if (type == 'GameAddedToWishlist')
      this.#games.push(data.gameId);
    if (type == 'GameRemovedFromWishlist')
      this.#games.splice(this.#games.indexOf(data.gameId), 1);
  }

  #applyNewEvent(event) {
    this.newEvents.push(event);
    this.#applyEvent(event);
  }
}
```

Example: Write Model (OOP/3)

```
const wishlist = new Wishlist();
wishlist.create({wishlistId, playerId});
wishlist.addGame(game1Id);

const events = wishlist.newEvents;

const rebuiltWishlist = new Wishlist(events);
rebuiltWishlist.addGame(game2Id);
rebuiltWishlist.removeGame(game1Id);
```

Example: Write Model (FP/1)

```
const applyEvents = (s, evts) => evts.reduce(applyEvent, s);

const applyEvent = (state, {type, data}) => {
  if (type == 'WishlistCreated') return {id: data.wishlistId};
  if (type == 'GameAddedToWishlist') return {...state,
    games: state.games.concat([data.gameId])};
  if (type == 'GameRemovedFromWishlist') return {...state,
    games: state.games.filter(id => id != data.gameId)};
  return {...state};
};

const createWishlist = ({wishlistId, playerId}) =>
  [Event('WishlistCreated', {wishlistId, playerId})];

const addGame = ({wishlistId, games}, gameId) => {
  if (games.includes(gameId)) throw new Error();
  return [Event('GameAddedToWishlist', {wishlistId, gameId}))];
};

const removeGame = ({wishlistId}, gameId) =>
  [Event('GameRemoveFromWishlist', {wishlistId, gameId}))];
```

Example: Write Model (FP/2)

```
let state = {};

const createEvents = createWishlist({wishlistId, playerId});
state = applyEvents(state, createEvents);

const add1Events = addGame(state, game1Id);
state = applyEvents(state, add1Events);

const add2Events = addGame(state, game2Id);
state = applyEvents(state, add2Events);

const remove1Events = addGame(state, game1Id);
state = applyEvents(state, remove1Events);
```

Event Store

- persistence of event-sourced state
- base functionality: save events, load in order
- categorized in event streams per Aggregate
- optional (version-based) optimistic concurrency
- write side: rebuild state, append items
- read side: build Read Model

Example: Event Store (1)

```
const stream = `wishlist/${wishlistId}`;

const createEvents = createWishlist({wishlistId, playerId});
await eventStore.save(stream, events);

const {events, currentVersion} = await eventStore.load(stream);
let state = applyEvents({}, events);

const add1Events = addGame(state, game1Id);
state = applyEvents(state, add1Events);
const add2Events = addGame(state, game2Id);
state = applyEvents(state, add2Events);

await eventStore.save(stream, add1Events.concat(add2Events),
  {expectedVersion: currentVersion});

await eventStore.load(stream, {startVersion: currentVersion});
```

Example: Event Store (2)

stream	no	event
wishlist/1	1	WishlistCreated(wishlistId)
wishlist/1	2	GameAddedToWishlist(wishlistId, gameId)
wishlist/1	3	GameAddedToWishlist(wishlistId, gameId)
wishlist/1	4	GameRemovedFromWishlist(wishlistId, gameId)
wishlist/2	1	WishlistCreated (wishlistId)

Stream subscriptions

- treat streams as continuous event source
- useful for maintaining Read Models
- allow custom starting versions
- optional server-side cursors

Example: Stream subscriptions

```
const handleEvent = async (event) => {
  if (event.type == 'GameAddedToWishlist') {
    const {title, publisherId} =
      await gameStorage.load(event.data.gameId);
    const publisher = await publisherStorage.load(publisherId);
    await sendEmail(publisher.emailAddress, {
      subject: 'Wishlist changes',
      message: `Someone added ${title} to their wishlist`,
    });
  }
});

eventStore.subscribe(
  `wishlist/${wishlistId}`, handleEvent, {startVersion});
```

subscription is only for one wishlist

Grouped streams

- group of multiple (Aggregate) streams
- arbitrary fixed order
- event streams per event type
- global event stream

Example: Stream subscriptions

```
const handleEvent = async (event) => {
  if (event.type == 'GameAddedToWishlist') {
    const {title, publisherId} =
      await gameStorage.load(event.data.gameId);
    const publisher = await publisherStorage.load(publisherId);
    await sendEmail(publisher.emailAddress, {
      subject: 'Wishlist changes',
      message: `Someone added ${title} to their wishlist`,
    });
  }
});

// global stream
eventStore.subscribe('$global', handleEvent);
// grouped stream by event type
eventStore.subscribe('type/GameAddedToWishList', handleEvent);
```

subscription works for all wishlists

Read Model Projection

- consume event streams to update Read Model
- on-demand projection
  - retrieve events, process, return result
  - typically transient data
- continuous projection
  - continuously update read data
  - track version to skip processed events
  - transient or persistent data

Example: On-demand projection

```
const calculateLargestWishlist = (events) => {
  const wishlistSizeById = {};
  events.forEach(event => {
    if (event.type == 'WishListCreated')
      wishlistSizeById[event.data.wishlistId] = 0;
    if (event.type == 'GameAddedToWishList')
      wishlistSizeById[event.data.wishlistId]++;
    if (event.type == 'GameRemovedFromList')
      wishlistSizeById[event.data.wishlistId]--;
  });
  const wishlistSizes = Object.entries(wishlistSizeById)
    .map([id, size] => ({id, size}));
  wishlistSizes.sort((a, b) => b.size - a.size);
  return wishlistSizes[0];
};

const {events} = await eventStore.load('$global');
calculateLargestWishlist(events);
```

Example: Continuous projection

```
class WishlistReadModelProjection {
  constructor({eventStore, readModels}) {
    eventStore.subscribe('$all', this.handleEvent.bind(this));
  }

  handleEvent({type, data: {wishlistId}}) {
    if (!wishlistId) return;
    if (type == 'WishListCreated')
      readModels[wishlistId] = {games: []};
    if (type == 'GameAddedToWishList')
      readModels[wishlistId].games.push(data.gameId);
    if (event.type == 'GameRemovedFromWishlist') {
      const games = readModels[wishlistId].games;
      games.splice(games.indexOf(data.gameId), 1);
    }
  }
}

const readModels = {};
new WishlistReadModelProjection({eventStore, readModels});
```

Example: Persistent projection

```
class WishlistReadModelPersistentProjection {
  constructor({eventStore, storage}) {
    this.activate = async () => {
      eventStore.subscribe('$all', (e) => this.handleEvent(e),
        {startVersion: await this.#storage.load('version')});
    }
  }

  async handleEvent({type, data: {wishlistId: id}}) {
    if (!id) return;
    let list = await this.#storage.load(`wishlist/${id}`);
    if (type == 'WishListCreated') list = {games: []};
    if (type == 'GameAddedToWishList')
      list.games.push(data.gameId);
    if (event.type == 'GameRemovedFromList')
      list.games.splice(list.games.indexOf(data.gameId), 1);
    await this.#storage.save(`wishlist/${id}`, list);
    await this.#storage.save('version', event.number);
  }
}
```

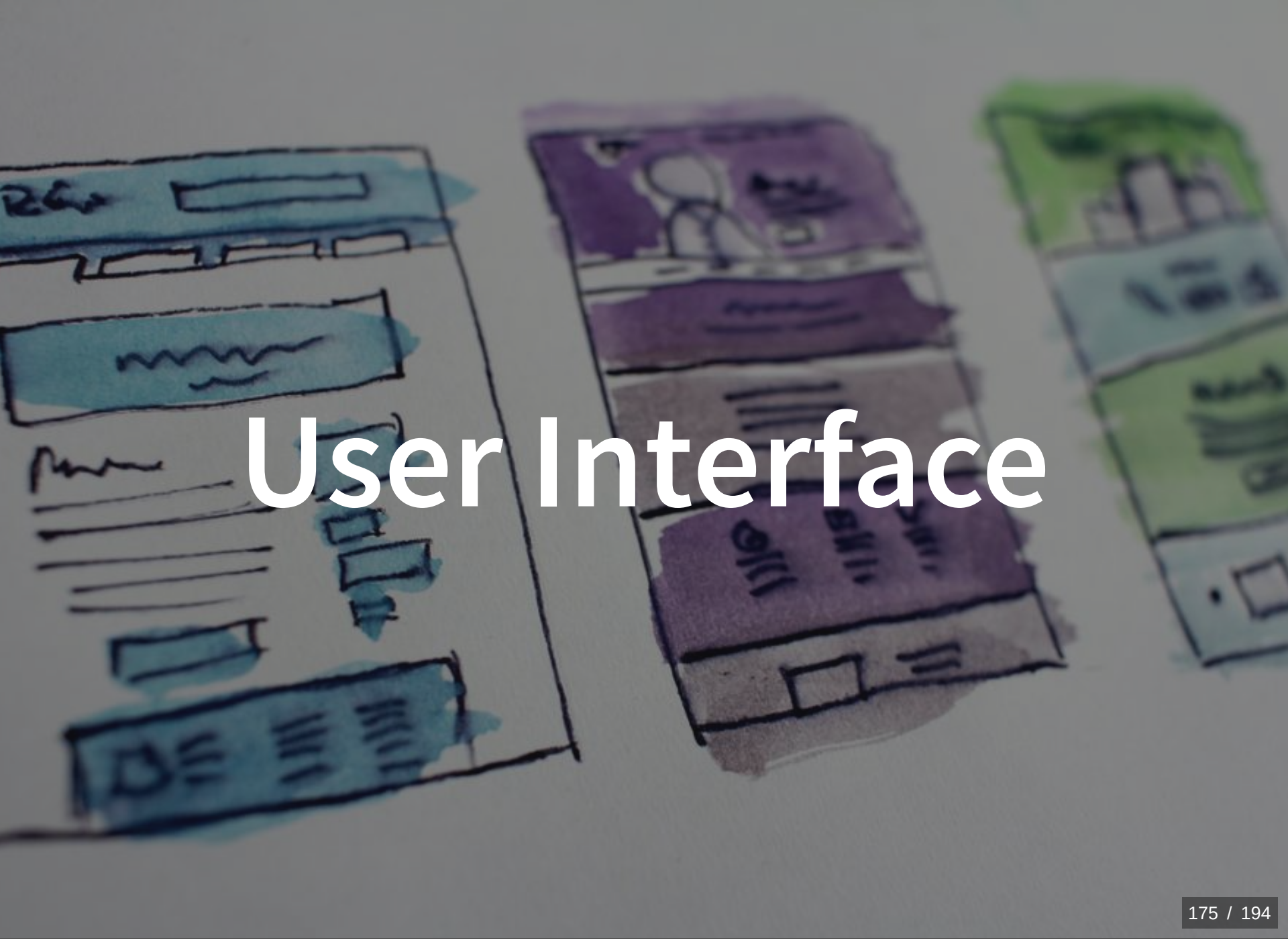
Domain Event publishing

- Event-Driven Architecture on system level
- Event Sourcing for individual areas
- combining both worlds
  - subscribe to Event Store
  - transform into Domain Events
  - forward to Event Bus
  - track processed version

Example: Event publishing

```
class DomainEventPublisher {
  constructor({eventStore, eventBus, publicEventTypes}) {
    this.#eventStore = eventStore;
    this.#eventBus = eventBus;
    this.#publicEventTypes = publicEventTypes;
  }

  async activate() {
    const processedVersion = Number(
      await readFile('version', 'utf-8').catch(() => '0');
    this.#eventStore.subscribe('$global', async event => {
      if (this.#publicEventTypes.includes(event.type))
        await this.#eventBus.publish(event);
      await writeFile('version', `${event.number}`);
    }, {startVersion: processedVersion + 1});
  }
}
```

at least once publishing


User Interface

Responsibilities

- display information
- validate and process inputs
- request behavior execution, report results
- for web: mostly HTML, CSS, JavaScript

Task-based UI

- emphasize domain-specific aspects
- often implicit when applying DDD
- guide user towards use cases
- send Commands and Queries

Example: not Task-based

```
<h1>Edit user</h1>
<form>
  <input name="id" type="hidden">
  <input name="e-mail" type="text">
  <input name="password" type="password">
  <input type="submit">
  <div class="message"></div>
</form>
```

```
const form = document.querySelector('form');
form.addEventListener('submit', async event => {
  event.preventDefault();
  await fetch('/user/edit', {
    method: 'POST',
    body: new FormData(form),
  });
  form.querySelector('.message').innerHTML = 'Success!';
});
```

all changes are treated as the same action

Example: Task-based

```
<h1>User</h1>
<h3 class="e-mail">mail@example.com (<a href="#">edit</a>)</h3>
<h4><a href="#">Update current password</a></h4>
</div>
```

```
const sendCommand = (command) => fetch('/user', {
  method: 'POST', body: JSON.stringify(command),
});

sendCommand({
  id: generateId(), type: 'UpdateEmailAddress',
  data: {name: 'mail@alex-lawrence.com'}, metadata: {/* .. */},
});
sendCommand({
  id: generateId(), type: 'ChangePassword',
  data: {password: hashedPassword}, metadata: {/* .. */},
});
```

individual use cases executed separately

Optimistic UI

- improve perceived performance
- premature success indication
- simulate updates to read data
- dealing with errors
  - display message
  - reload UI
  - offer compensating actions

Example: Optimistic UI

```
writeMessageElement.addEventListener('submit', async () => {
  const content = writeMessageElement.elements.content.value;
  chatElement.insertAdjacentHTML(
    'beforend', `<div>${content}</div>`);
  const response = await sendWriteCommand(content);
  if (response.status === 200) return;
  const errorElement = document.createElement('div');
  errorElement.innerHTML = 'Error. <a href="#">Retry?</a>';
  chatElement.appendChild(errorElement);
  errorElement.addEventListener('click', async () => {
    errorElement.setAttribute('hidden', 'true');
    const newResponse = await sendWriteCommand(content);
    if (newResponse.status === 200) errorElement.remove();
    else errorElement.removeAttribute('hidden');
  });
});
```

Reactive Read Models

- expose continuous source of information
- incorporate current data and future updates
- useful for reactive User Interfaces
- for web: Server-Sent Events

Example: Reactive Read Models

- scenarios
  - players want to see their friends status
  - players want to see what friends are playing
- event-sourced player Aggregate
- events
  - PlayerWentOnline
  - PlayerStartedPlaying
  - PlayerStoppedPlaying
  - PlayerWentOffline

Event forwarding

- consume event stream in User Interface
- ✔️ UI maintains projection and (persistent) data
- ⚠ potential network traffic overhead
- ⚠ internal events are exposed to outside
- ⚠ requires stream access rights

Example: Event forwarding

```
class PlayerQueryHandlers {
  async handleStreamPlayerEvents(data) {
    const stream = new Readable({objectMode: true, read() {}});
    eventStore.subscribe(`player/${data.playerId}`,
      (event) => stream.push(event));
    return stream;
  }
}
```

```
const eventSource = new EventSource(`/player/${playerId}`);
eventSource.addEventListener('message', async (message) => {
  const {type, data} = JSON.parse(message.data);
  if (type == 'PlayerWentOnline') status.innerHTML = 'online';
  if (type == 'PlayerWentOffline')
    status.innerHTML = 'offline';
  if (type == 'PlayerStartedPlaying')
    playing.innerHTML = `Plays ${await getTitle(data.gameId)}`;
  if (type === 'PlayerStoppedPlaying') playing.innerHTML = '';
});
```

Data events

- continuously transmit full Read Model
- server maintains projection and data
- events trigger Read Model updates
- updated Read Model is sent to UI
- ✔️ low complexity in UI
- ⚠ concerns are spread across layers
- ⚠ network traffic overhead

Example: Data events (1)

```
class PlayerQueryHandlers {
  async handleStreamPlayerData(data) {
    const stream = new Readable({objectMode: true, read() {}});
    stream.push(players[data.playerId]);
    messageBus.subscribe(playerId, stream.push.bind(stream));
    return stream;
  }
}
```

```
class PlayerReadModelProjection {
  handleEvent({type, data: {playerId}}) {
    const player = playerReadModels[data.playerId];
    if (type == 'PlayerWentOnline') player.isOnline = true;
    if (type == 'PlayerWentOffline') player.isOnline = false;
    if (type == 'PlayerStartedPlaying')
      player.currentGame = await fetchGameTitle(data.gameId);
    if (type == 'PlayerStoppedPlaying')
      player.currentGame = null;
    messageBus.publish(playerId, player);
  }
}
```

Example: Data events (2)

```
const eventSource = new EventSource(`/player/${playerId}`);
eventSource.addEventListener('message', (message) => {
  const {isOnline, currentGame} = JSON.parse(message.data);
  status.innerHTML = isOnline ? 'online' : 'offline';
  playing.innerHTML =
    currentGame ? `Plays ${currentGame}` : '';
});
```

Change events

- initially transmit full Read Model
- again, server maintains projection + data
- updates trigger change description notifications
  - custom format
  - standards (e.g. JSON Patch)
- ✔️ avoids network overhead
- ✔️ does not expose internal events
- ⚠ moderate complexity, concerns are spread

Example: Change events (1)

```
class PlayerQueryHandlers {
  async handleStreamPlayerData(data) {
    const stream = new Readable({objectMode: true, read() {}});
    stream.push(players[data.playerId]);
    messageBus.subscribe(playerId, stream.push.bind(stream));
    return stream;
  }
}
```

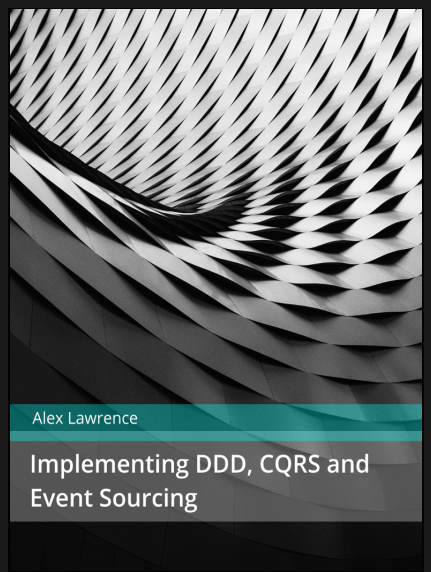
```
class PlayerReadModelProjection {
  handleEvent({type: t, data: {playerId}}) {
    const change = {};
    if (t == 'PlayerWentOnline') change.isOnline = true;
    if (t == 'PlayerWentOffline') change.isOnline = false;
    if (t == 'PlayerStartedPlaying')
      change.currentGame = await fetchGameTitle(data.gameId);
    if (t == 'PlayerStoppedPlaying') change.currentGame = null;
    Object.assign(player, change);
    messageBus.publish(playerId, change);
  }
}
```

Example: Change events (2)

```
const eventSource = new EventSource(`/player/${playerId}`);
eventSource.addEventListener('message', async (message) => {
  const {isOnline, currentGame} = JSON.parse(message.data);
  if (isOnline !== undefined)
    status.innerHTML = isOnline ? 'online' : 'offline';
  if (currentGame !== undefined)
    playing.innerHTML = currentGame ?
      `Plays ${await getTitle(currentGame)}` : '';
  }
});
```

treat full data and changes the same way

Book: Implementing DDD, CQRS and Event Sourcing



- explains concepts in theory
- many standalone examples
- full Sample Application
- uses JavaScript & Node.js
- available on Leanpub

Q & A