How complex software impacts your cognitive abilities

@CorstianBoerman

@corstian

corstian@whaally.com

The plan

- Terminology
- When things break apart
- How to prevent things from breaking apart

Cognition

"The mental action or process of acquiring knowledge and understanding through thought, experience, and the senses."

- https://www.lexico.com/definition/cognition

Cognitive ability

Senses

- Vision
- Hearing
- Feeling
- Smelling
- Tasting

Experiences

- Prime source of bias
- Impacted by life events
- Too broad to do justice

Thoughts

- Conscious / unconscious
- Rational / emotional
- Slow / fast

Cognitive Disabilities?

Inclusivity

- Requires accommodation for the weakest links in the chain
- Shows the privilege of the abled

- To what extent are our discriminatory practices disguised as cognitive demands?

“Everyone has the right to work, to free choice of employment, to just and favourable conditions of work and to protection against unemployment.”

- Article 23, Universal Declaration of Human Rights

“disregard and contempt for human rights have resulted in barbarous acts which have outraged the conscience of mankind”

Preamble, Universal Declaration of Human Rights

Rights & Responsibilities

- Rights cannot exist without responsibilities
- Rights w/o responsibilities result in rights for some
  - Based upon resources, qualities or other forms of privilege

*“People are free from depression when their feeling of **self worth** is **derived from the truth of their own feelings**, and **not on the possession of certain qualities**”*

- Alice Miller, The Drama of the Gifted Child

Resilience

- Cognition as a problem solving system

- Surplus of cognitive capacity improves resilience
- Overexertion of resilience causes cognitive decline

- Proactive response vs. reactive response

Burnout.

Software Complexity

- The cognitive abilities required to work with a given piece of software

- Includes coding
- Includes process
- Includes usage
- …

Hypothesis time!

Is there a relationship between software complexity and cognitive abilities?

a. Complexity positively impacts cognition
b. Complexity does not impact cognition
c. Complexity negatively impacts cognition

What if expectations > abilities

How far can we push it?

Mismatch between abilities & expectations

Deliver more software
In less time
At higher quality

Amazing for learning.

Devastating for wellbeing.

People burning out tend to make themselves obsolete.

Cognition →

SW
Complexity

My mental
state

Time →

Coding
With the mental ability
That of a four year old

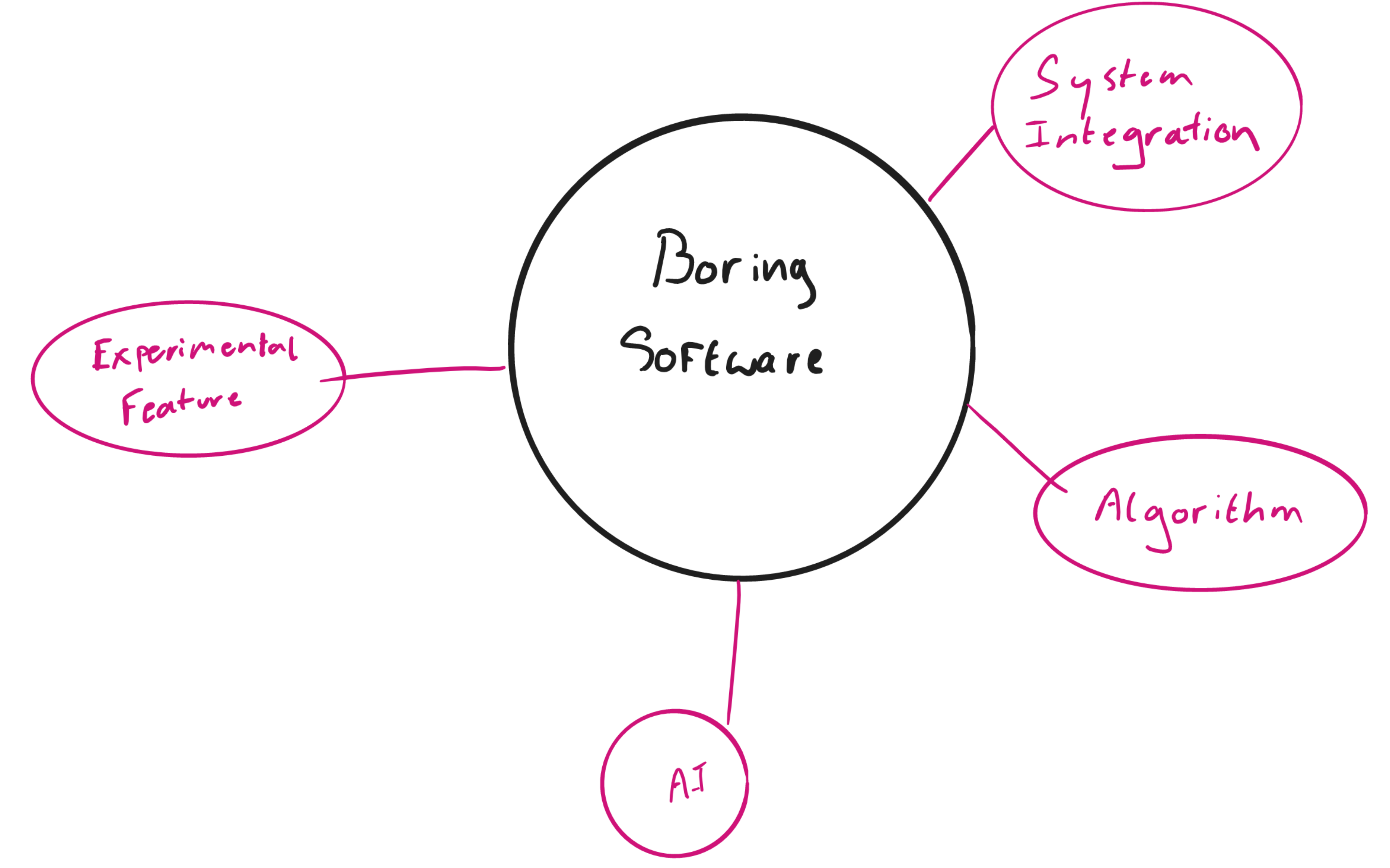
To deal with cognitive demand

- Allow implications:
  - Reduced quality
  - Increased resource requirements
  - Decreased wellbeing
- Temper business demands
- Split up
  - Create artificial boundaries
  - Introduce abstractions

What if expectations < abilities

How to facilitate this?

Boring Software
Exciting Software


System Integration
Boring Software
Experimental Feature
Algorithm
AI

But... why?

Boring Aspects

- Well defined boundaries
- Well defined solutions to:
    - Structure / data organization
    - Concurrency
    - Consistency
    - Scalability
    - Testability
    - Maintainability
- Stability
- Modelling does not involve externalities
- Development to become a routine task

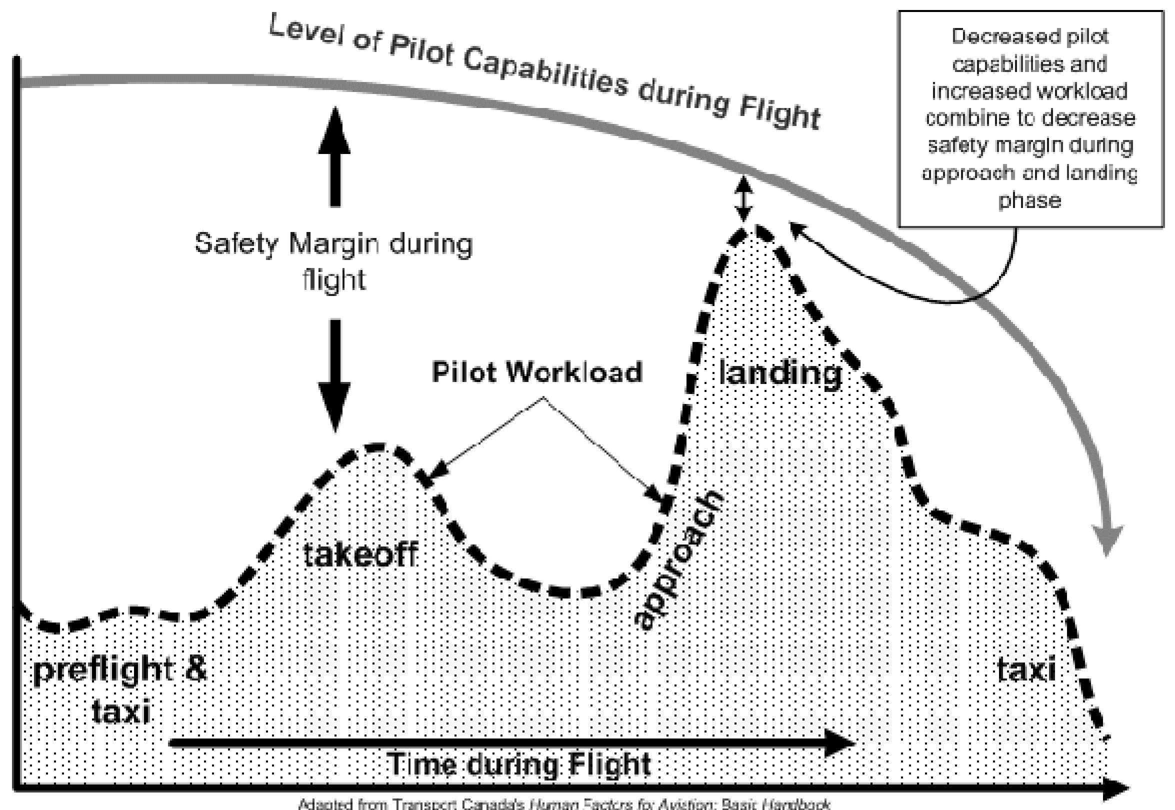
Exciting Aspects

- Contextually decoupled
- Isolated from other systems
- Clear expectation of behaviour
- Independently scalable
- Independently testable
- Development to be a creative process

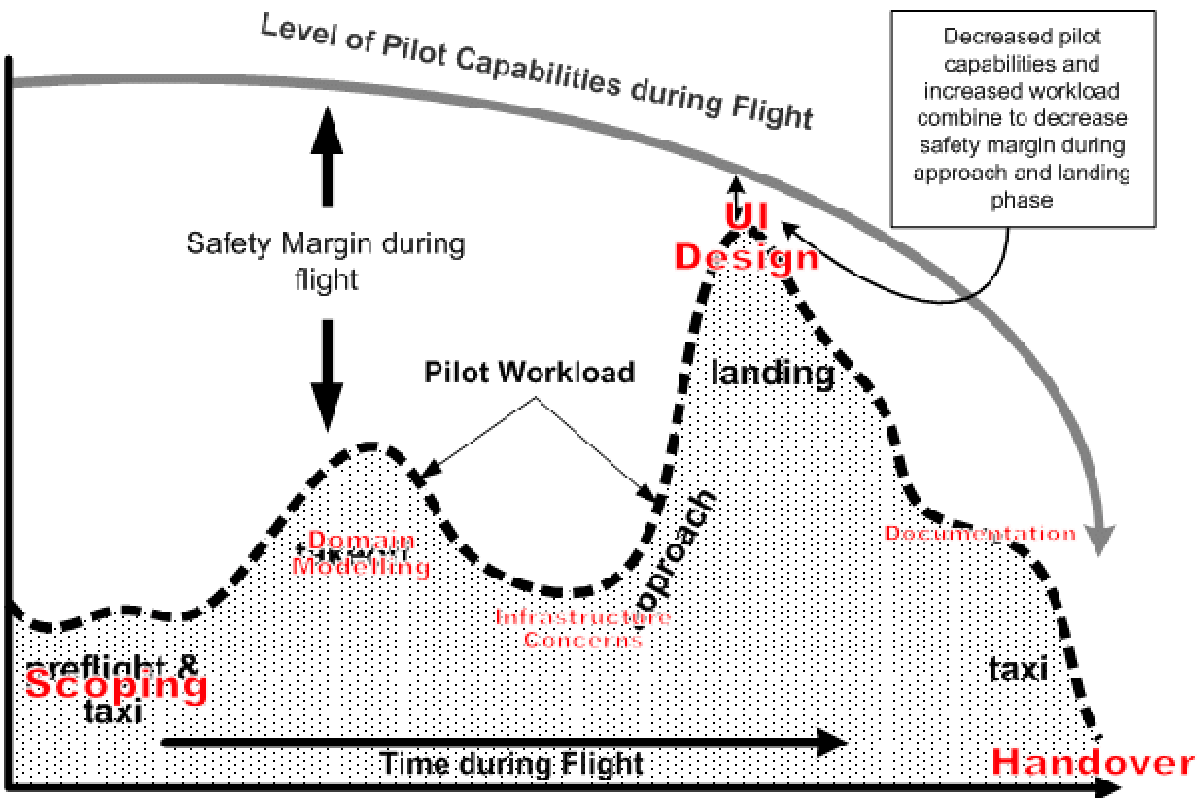
Impact on development process

- Move greatest uncertainties up front *(uncertainty == unquantified risk)*
  - Start development work on the domain
  - Tests as proof of correct behaviour

- Further implementation becomes a formality


Level of Pilot Capabilities during Flight
Decreased pilot capabilities and increased workload combine to decrease safety margin during approach and landing phase
Safety Margin during flight
Pilot Workload
landing
takeoff
approach
preflight & taxi
taxi
Time during Flight
Adapted from Transport Canada's Human Factors for Aviation: Basic Handbook

Whaally


Level of Pilot Capabilities during Flight
Decreased pilot capabilities and increased workload combine to decrease safety margin during approach and landing phase
Safety Margin during flight
Pilot Workload
UI Design
landing
approach
Domain Modelling
Infrastructure Concerns
Documentation
Scoping
preflight & taxi
taxi
Handover
Time during Flight
Adapted from Transport Canada's Human Factors for Aviation: Basic Handbook

Boring complexity

Deconstruct the onion from the inside out

APIs

Actors

Infrastructure

Domain

Behaviour

Domain Driven <u>Development</u>

- The domain doing domain stuff
  - Value objects
  - Aggregates (& Entities)
  - Services
  - Repositories

- Decisions made in the domain are reflected **<u>everywhere</u>**!

DDD Meta: Communication approaches

Functions:

- Easier to get started with
- Feels more connected
- Requires more boilerplate for integration between layers
- Requires conscious effort to maintain consistency

Messages:

- Easier to generalize
- Cause less overhead when integrating layers
- Provide more consistency

```csharp
public class User {
    public string FirstName { get; private set; }
    public string LastName { get; private set; }

    public void Rename(string firstName, string lastName) {
        if (string.IsNullOrWhiteSpace(firstName) || string.IsNullOrWhiteSpace(lastName))
            throw new Exception("Name is required");

        FirstName = firstName;
        LastName = lastName;
    }
}
```

State

```
public class User {
    public string FirstName { get; private set; }
    public string LastName { get; private set; }

    public void Rename(string firstName, string lastName) {
        if (string.IsNullOrWhiteSpace(firstName) || string.IsNullOrWhiteSpace(lastName))
             throw new Exception("Name is required");

        FirstName = firstName;
        LastName = lastName;
    }
}
```

```
public class User {
    public string FirstName { get; private set; }
    public string LastName { get; private set; }

    public void Rename(string firstName, string lastName) {
        if (string.IsNullOrWhiteSpace(firstName) || string.IsNullOrWhiteSpace(lastName))
            throw new Exception("Name is required");

        FirstName = firstName;
        LastName = lastName;
    }
}
```

State

Command

```
public class User {
    public string FirstName { get; private set; }
    public string LastName { get; private set; }

    public void Rename(string firstName, string lastName) {
        if (string.IsNullOrWhiteSpace(firstName) || string.IsNullOrWhiteSpace(lastName))
            throw new Exception("Name is required");

        FirstName = firstName;
        LastName = lastName;
    }
}
```

State

Command

Validation

```
public class User {
    public string FirstName { get; private set; }
    public string LastName { get; private set; }

    public void Rename(string firstName, string lastName) {
        if (string.IsNullOrWhiteSpace(firstName) || string.IsNullOrWhiteSpace(lastName))
            throw new Exception("Name is required");

        FirstName = firstName;
        LastName = lastName;
    }
}
```

State

Command

Validation

State Change

Aggregate state, coupling behaviour

does <u>not</u> scale

We'll cause a cognitive overload.

```csharp
public interface IAggregate
{

}

public interface IAggregate<TAggregate> : IAggregate
    where TAggregate : IAggregate<TAggregate>
{
    // Command handling
    Task<IResult<IEnumerable<IEvent<TAggregate>>>>
        Evaluate(ICommand<TAggregate> command);

    // Event application
    Task Apply(IEvent<TAggregate> @event);
    Task Apply(params IEvent<TAggregate>[] events);

    // Snapshotting
    Task<TModel> GetSnapshot<TModel>()
        where TModel : ISnapshot<TAggregate>, new();
}
```

Define responsibilities of aggregate

```
public interface IAggregate
{

}

public interface IAggregate<TAggregate> : IAggregate
    where TAggregate : IAggregate<TAggregate>
{
    // Command handling
    Task<IResult<IEnumerable<IEvent<TAggregate>>>>
        Evaluate(ICommand<TAggregate> command);

    // Event application
    Task Apply(IEvent<TAggregate> @event);
    Task Apply(params IEvent<TAggregate>[] events);

    // Snapshotting
    Task<TModel> GetSnapshot<TModel>()
        where TModel : ISnapshot<TAggregate>, new();
}
```

@CorstianBoerman

Define responsibilities of aggregate

Step 1: Evaluate command

```
public interface IAggregate
{

}

public interface IAggregate<TAggregate> : IAggregate
    where TAggregate : IAggregate<TAggregate>
{
    // Command handling
    Task<IResult<IEnumerable<IEvent<TAggregate>>>>
        Evaluate(ICommand<TAggregate> command);

    // Event application
    Task Apply(IEvent<TAggregate> @event);
    Task Apply(params IEvent<TAggregate>[] events);

    // Snapshotting
    Task<TModel> GetSnapshot<TModel>()
        where TModel : ISnapshot<TAggregate>, new();
}
```

Define responsibilities of aggregate

Step 1: Evaluate command

Step 2: Apply results

```
public interface IAggregate
{

}

public interface IAggregate<TAggregate> : IAggregate
    where TAggregate : IAggregate<TAggregate>
{
    // Command handling
    Task<IResult<IEnumerable<IEvent<TAggregate>>>>
        Evaluate(ICommand<TAggregate> command);

    // Event application
    Task Apply(IEvent<TAggregate> @event);
    Task Apply(params IEvent<TAggregate>[] events);

    // Snapshotting
    Task<TModel> GetSnapshot<TModel>()
        where TModel : ISnapshot<TAggregate>, new();
}
```

@CorstianBoerman

Define responsibilities of aggregate

```
public interface IAggregate
{

}

public interface IAggregate<TAggregate> : IAggregate
    where TAggregate : IAggregate<TAggregate>
{
    // Command handling
    Task<IResult<IEnumerable<IEvent<TAggregate>>>>
        Evaluate(ICommand<TAggregate> command);

    // Event application
    Task Apply(IEvent<TAggregate> @event);
    Task Apply(params IEvent<TAggregate>[] events);

    // Snapshotting
    Task<TModel> GetSnapshot<TModel>()
        where TModel : ISnapshot<TAggregate>, new();
}
```

Step 1: Evaluate command

Step 2: Apply results

Whenever: check state

@CorstianBoerman

Behaviour is offloaded to commands & events

Proxies may implement aggregate contract

Event sourcing as 1st class citizen

- Is not about recreating state from events. Though possible…
  Instead provides sensible logical boundaries.
- Clearly defined process
  - Command creation
  - Command evaluation
  - Result handling
  - Event application
- Forces chunking up behaviour → good for cognitive load
- Abstractions force messaging → less code → reduces cognitive load
  - Alternatively requires custom behaviour on each layer → increases cognitive load

```
public class User : Aggregate<User>
{
    public string Name { get; internal set; }
    public string Email { get; internal set; }
}
```

```
public class User : Aggregate<User>
{
    public string Name { get; internal set; }
    public string Email { get; internal set; }
}
```

State

Generalized command / event handling behaviour

State

```
public class User : Aggregate<User>
{
    public string Name { get; internal set; }
    public string Email { get; internal set; }
}
```

```csharp
public class Rename : ICommand<User, Renamed>
{
    public string Name { get; init; }

    IResult<Renamed> ICommand<User, Renamed>.Evaluate(User state)
    {
        if (string.IsNullOrWhiteSpace(Name))
            return DomainResult.Fail<Renamed>($"{nameof(Name)} should not be empty");

        return DomainResult.Ok(new Renamed
        {
            Name = Name
        });
    }
}
```

```
public class Rename : ICommand<User, Renamed>
{
    public string Name { get; init; }

    IResult<Renamed> ICommand<User, Renamed>.Evaluate(User state)
    {
        if (string.IsNullOrWhiteSpace(Name))
            return DomainResult.Fail<Renamed>($"{nameof(Name)} should not be empty");

        return DomainResult.Ok(new Renamed
        {
            Name = Name
        });
    }
}
```

Command arguments

```csharp
public class Rename : ICommand<User, Renamed>
{
    public string Name { get; init; }

    IResult<Renamed> ICommand<User, Renamed>.Evaluate(User state)
    {
        if (string.IsNullOrWhiteSpace(Name))
            return DomainResult.Fail<Renamed>($"{nameof(Name)} should not be empty");

        return DomainResult.Ok(new Renamed
        {
            Name = Name
        });
    }
}
```

Command arguments

Evaluation

@CorstianBoerman

```csharp
public class Rename : ICommand<User, Renamed>
{
    public string Name { get; init; }

    IResult<Renamed> ICommand<User, Renamed>.Evaluate(User state)
    {
        if (string.IsNullOrWhiteSpace(Name))
            return DomainResult.Fail<Renamed>($"{nameof(Name)} should not be empty");

        return DomainResult.Ok(new Renamed
        {
            Name = Name
        });
    }
}
```

Command arguments

Evaluation

Intended change

```csharp
public class Renamed : IEvent<User>
{
    internal Renamed() { }

    public string Name { get; init; }

    void IEvent<User>.Apply(User state)
    {
        state.Name = Name;
    }
}
```

```csharp
public class Renamed : IEvent<User>
{
    internal Renamed() { }

    public string Name { get; init; }

    void IEvent<User>.Apply(User state)
    {
        state.Name = Name;
    }
}
```

Description of intended change

```csharp
public class Renamed : IEvent<User>
{
    internal Renamed() { }

    public string Name { get; init; }

    void IEvent<User>.Apply(User state)
    {
        state.Name = Name;
    }
}
```

Description of change intent

State change

```csharp
IAggregate<User> user = new User();

var command = new Rename
{
    Name = "John Doe"
};

var result = await user.Evaluate(command);

if (result.IsSuccess)
    await user.Apply(result.Value);
```

```
Domain.Example
  Dependencies
  Aggregates
    GroupAggregate
    OrderAggregate
    PaymentAggregate
    UserAggregate
      Commands
        AddGroup.cs
        ChangeEmail.cs
        ChangeInfo.cs
        ChangePassword.cs
        RemoveGroup.cs
        Rename.cs
        ValidatePassword.cs
      Events
        EmailChanged.cs
        GroupAdded.cs
        GroupRemoved.cs
        PasswordChanged.cs
        PasswordValidationCompleted.cs
        Renamed.cs
      Snapshots
        PublicUserInfo.cs
      User.cs
  ProcessManagers
  Sagas
  Services
```

Notes

- Coupling between commands and events is one to many
- Which is why to prefer small events over large ones
- Events and commands can independently change
  - Distinctively different from transducers

- *Services are built upon the aforementioned concepts*

Infrastructure

- Just infrastructure things
- Interface implementations
- Probably the only place I want to encounter GoF patterns
- Whatever goes. Infrastructure is a mess anyway.

An actor

- One unit having:
  - Isolated storage
  - Computational capabilities
  - Networking capabilities (can call other actors)

- Provides a granular abstraction from hardware concerns

Actor System

- Still infrastructure
- An abstraction over hardware concerns
- Has nice scalability & availability characteristics
- Encapsulates the domain
    - 1 aggregate instance => 1 actor
    - Services are stateless actors
- Still limited by single point failures (DBs 😠)

Actor System

- Is a **proxy to domain behaviour**
  - Actor implements domain interfaces
- Provides generalized infrastructure related behaviour to the domain
  - Logging
  - 2PC/3PC atomicity
- Provides scalable properties virtually free of charge!

A plain example of such proxy:

```
public interface IAggregateGrain<T> : IAggregate<T>, IGrainWithGuidKey
    where T : IAggregate<T> { }


public class AggregateGrain<T> : JournaledGrain<T, IEvent<T>>, IAggregateGrain<T>
    where T : class, IAggregate<T>, new()
{
    public async Task Apply(IEvent<T> @event)
        => RaiseEvent(@event);

    public async Task Apply(params IEvent<T>[] events)
        => RaiseEvents(events);

    public async Task<IResult<IEnumerable<IEvent<T>>>> Evaluate(ICommand<T> command)
        => await State.Evaluate(command);

    public async Task<TModel> GetSnapshot<TModel>() where TModel : ISnapshot<T>, new()
        => await State.GetSnapshot<TModel>();
}
```

Still the same behavioural contract

```
public interface IAggregateGrain<T> : IAggregate<T>, IGrainWithGuidKey
    where T : IAggregate<T> { }


public class AggregateGrain<T> : JournaledGrain<T, IEvent<T>>, IAggregateGrain<T>
    where T : class, IAggregate<T>, new()
{
    public async Task Apply(IEvent<T> @event)
        => RaiseEvent(@event);

    public async Task Apply(params IEvent<T>[] events)
        => RaiseEvents(events);

    public async Task<IResult<IEnumerable<IEvent<T>>>> Evaluate(ICommand<T> command)
        => await State.Evaluate(command);

    public async Task<TModel> GetSnapshot<TModel>() where TModel : ISnapshot<T>, new()
        => await State.GetSnapshot<TModel>();
}
```

Still the same behavioural contract

```
public interface IAggregateGrain<T> : IAggregate<T>, IGrainWithGuidKey
    where T : IAggregate<T> { }
```

Therefore proxying domain behaviour

```
public class AggregateGrain<T> : JournaledGrain<T, IEvent<T>>, IAggregateGrain<T>
    where T : class, IAggregate<T>, new()
{
    public async Task Apply(IEvent<T> @event)
        => RaiseEvent(@event);

    public async Task Apply(params IEvent<T>[] events)
        => RaiseEvents(events);

    public async Task<IResult<IEnumerable<IEvent<T>>>> Evaluate(ICommand<T> command)
        => await State.Evaluate(command);

    public async Task<TModel> GetSnapshot<TModel>() where TModel : ISnapshot<T>, new()
        => await State.GetSnapshot<TModel>();
}
```

Still the same behavioural contract

```
public interface IAggregateGrain<T> : IAggregate<T>, IGrainWithGuidKey
    where T : IAggregate<T> { }
```

Therefore proxying domain behaviour

```
public class AggregateGrain<T> : JournaledGrain<T, IEvent<T>>, IAggregateGrain<T>
    where T : class, IAggregate<T>, new()
{
    public async Task Apply(IEvent<T> @event)
        => RaiseEvent(@event);

    public async Task Apply(params IEvent<T>[] events)
        => RaiseEvents(events);

    public async Task<IResult<IEnumerable<IEvent<T>>>> Evaluate(ICommand<T> command)
        => await State.Evaluate(command);

    public async Task<TModel> GetSnapshot<TModel>() where TModel : ISnapshot<T>, new()
        => await State.GetSnapshot<TModel>();
}
```

Behaviour deferred to the domain

Interaction with domain or actor system

Virtually equivalent

Therefore preserving valuable conceptual principles.

APIs become thin mappers

- May return events (represents an operation)
- May return snapshots (current state)

And we're out of the onion

Into the frontend

Ps. this approach results in frontend work becoming much easier.

The benefits of transparency

- Allows reuse of the same conceptual (domain) model on every architectural layer
  - Domain
  - Actor (“just a proxy”)
  - API (“just an argument mapper”)
  - Frontend / Integration (“just delegating behaviour”)

- Fewer moving parts means less bugs

Complex software is an occupational health hazard!

~~Complex software~~ is an occupational health hazard!

Cognitive depletion is an occupational health hazard!

As soon as the business’ risk of complexity is offloaded to employees,

we are in danger.

What can we do?

- Reduce code complexity → Frees up cognitive abilities
- Reduce cognitive demands → Improves resilience

Why do we want to do it?

- Reduces development costs
- Improves product quality
- Improves developer wellbeing

Mental

Wellbeing.

Is the first requirement.

Solve problems.

Like the climate crisis.

HappyPorch

giki zero.

Whaally

“Our house is on fire…
I am here to say, our house is on fire!”

- Greta Thunberg

THIS IS FINE.

We are on track to +3°C

We are on track to +3°C

Systemic change is required.

How can we live
more sustainably,

If our self worth
is defined by possessions?

“Extinction is the rule.
Survival is the exception.”

- Carl Sagan

- https://corstianboeman.com
- https://github.com/corstian/domain-components

- Slides: https://www.corstianboerman.com/blog/2022-06-24/how-complex-software-impacts-your-cognitive-abilities
- Join the discussion: https://github.com/corstian/domain-components/discussions/1