Implementing Domain-Driven Design as a pragmatic developer

Halil İbrahim Kalkan

Co-founder, Volosoft

Download the Presentation

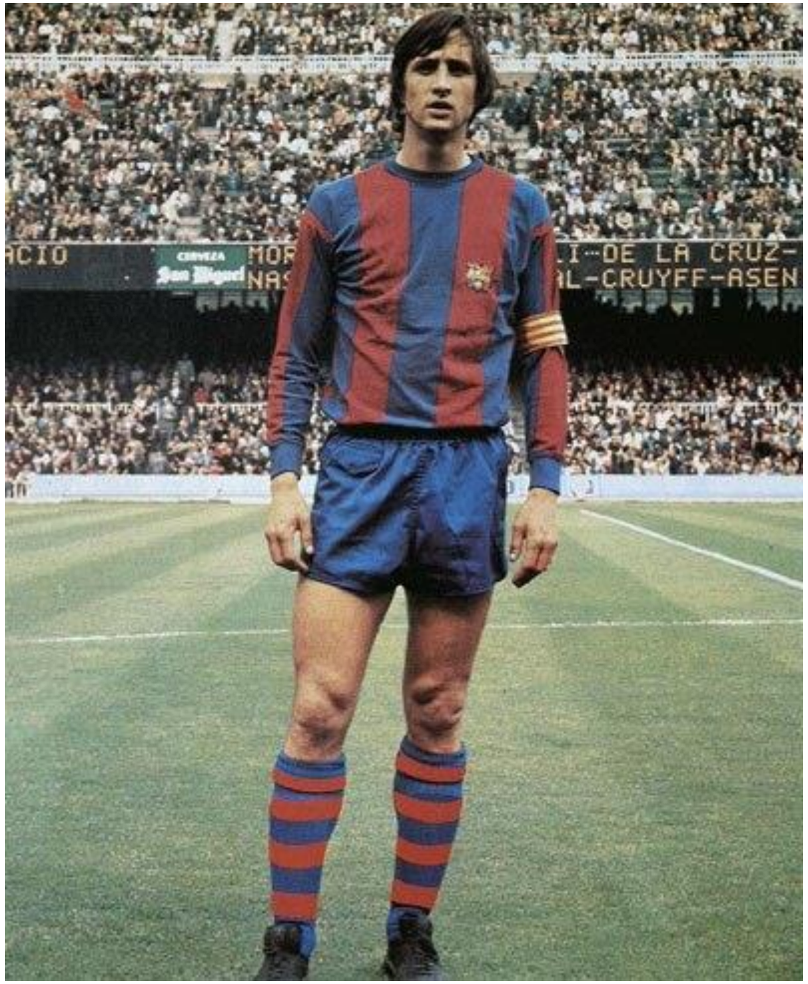
Writing «simple» Code

«**Playing football** is very **simple**, but **playing simple football** is the **hardest thing** there is!»

Johan Cruyff

«**Writing code** is very **simple**, but **writing simple code** is the **hardest thing** there is!»

Halil İbrahim Kalkan



About Me: Halil İbrahim Kalkan

1997, Started programming (at 14 years old, with Turbo Pascal)

2003-2007, Computer Engineering

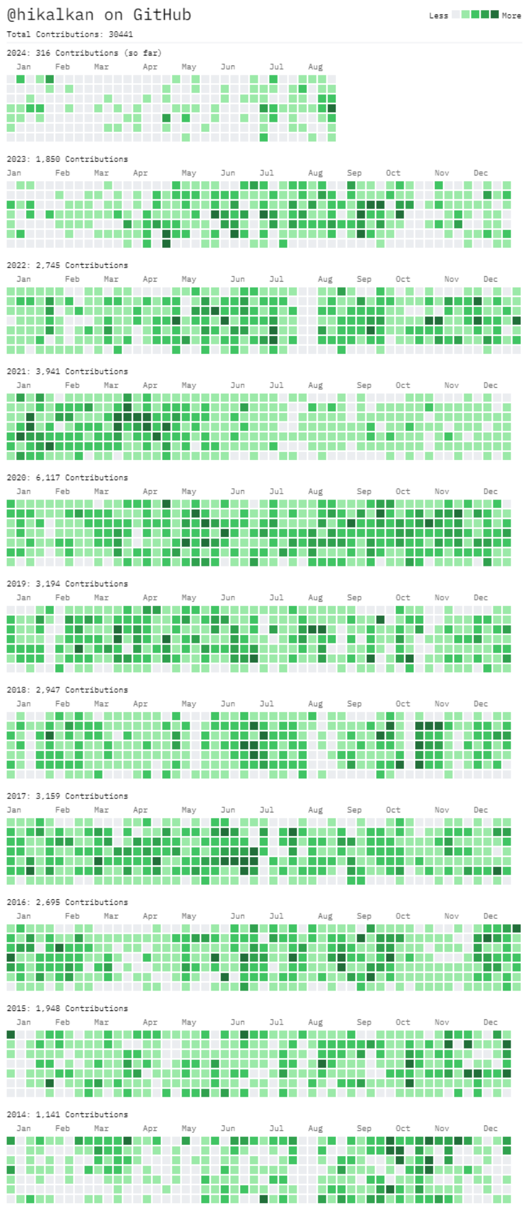
2007 - 2015: Software developer, software architect, team leader

abp 2013 - ∞: Lead developer of the open source ABP Framework

Volosoft 2016 - ∞: Co-founder, software architect

Multi-threading, distributed/microservice systems, OOP, DDD, software architectures.. etc.

Still active coder, open-source contributor
30,000+ total contributions on GitHub



Agenda

- **Part-I: What is DDD?**
  - Architecture & layers
  - Execution flow
  - Building blocks
  - Common principles
- **Part-II: Implementing DDD**
  - Layering a Visual Studio solution
  - Rules & Best Practices
  - Examples

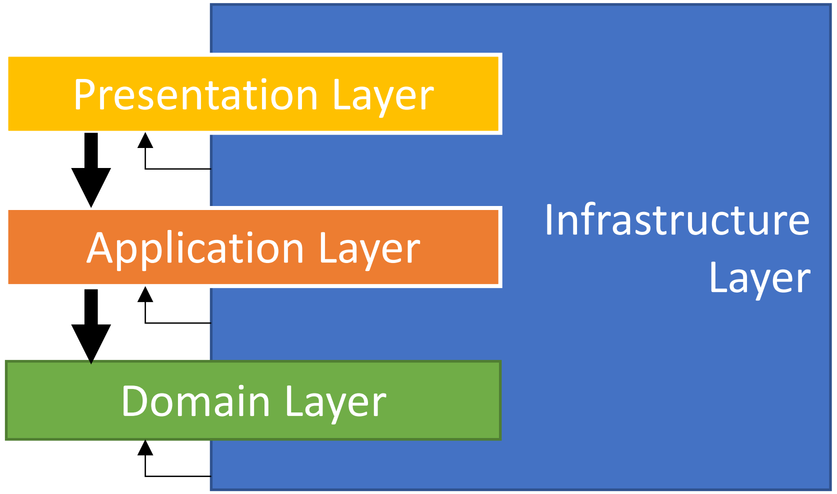
Part-I: What is DDD?

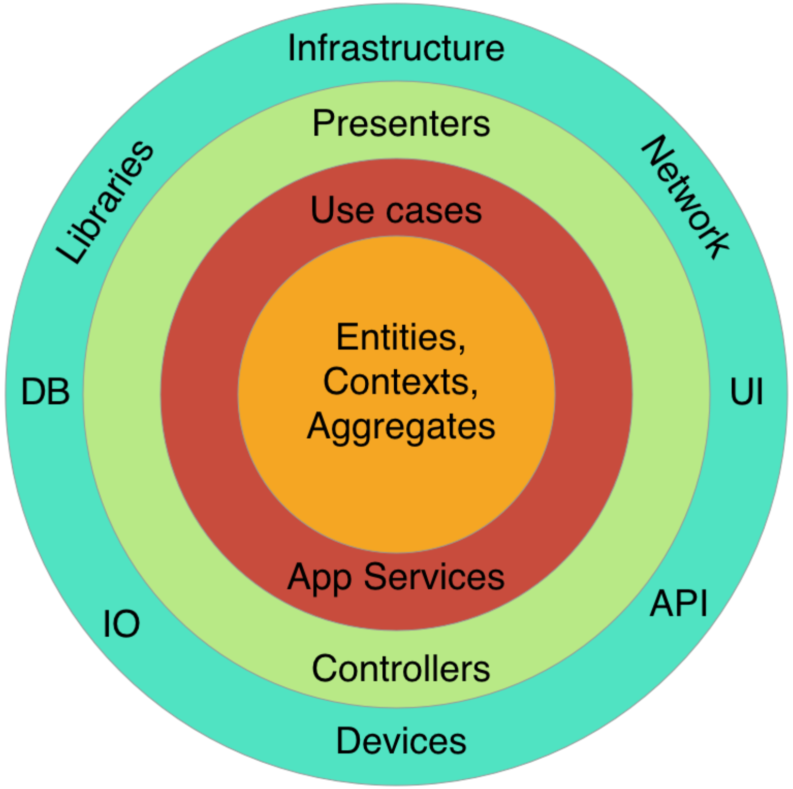
What is DDD?

- Domain-driven design (DDD) is an approach to software development for **complex** needs by connecting the implementation to an **evolving** model
- Focuses on the **core domain logic** rather than the infrastructure.
- Suitable for **complex domains** and **large-scale** applications.
- Helps to build a **flexible**, modular and **maintainable** code base, based on the **Object-Oriented Programming** principles.

Domain Driven Design

Layers & Clean Architecture





Domain Driven Design

Core Building Blocks

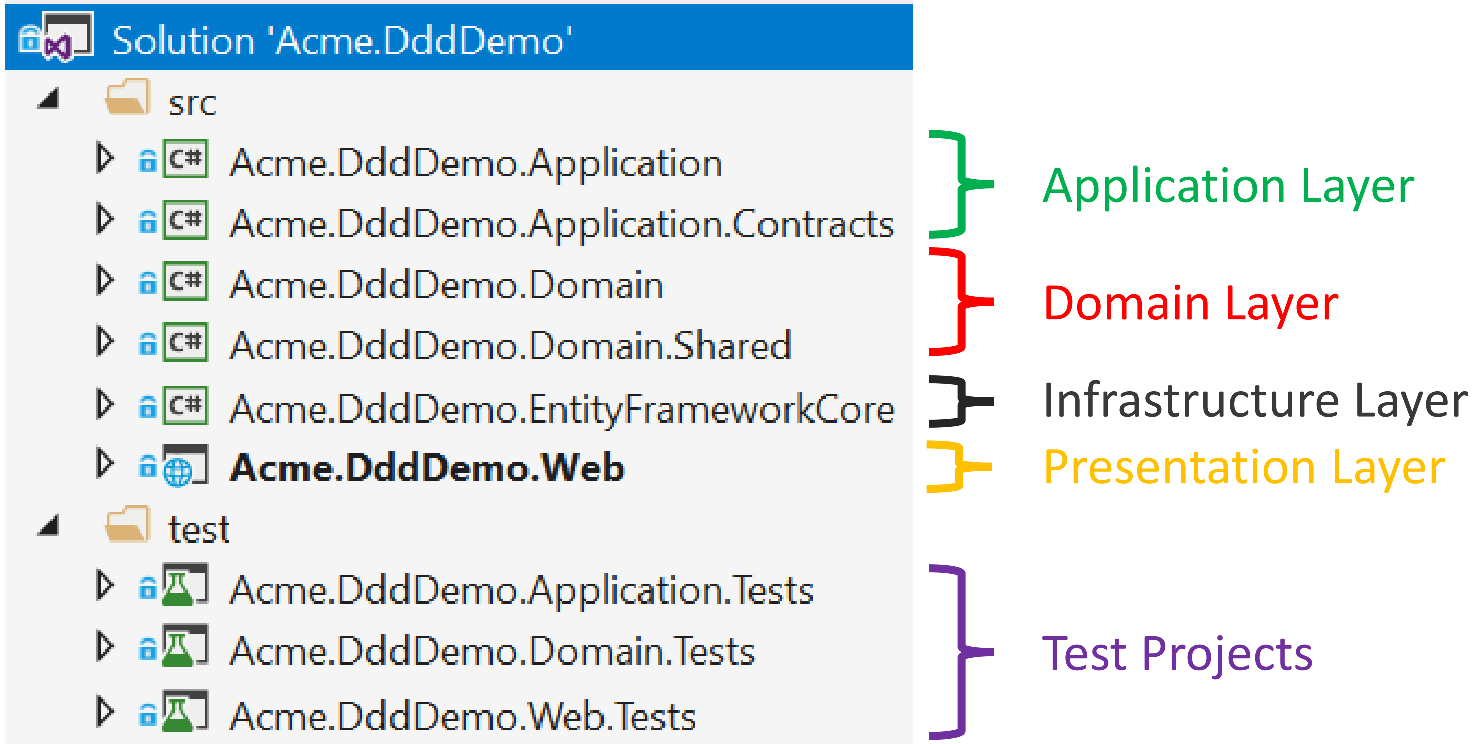
Domain Layer

- Entity
- Value Object
- Aggregate & Aggregate Root
- Repository
- Domain Service
- Specification
- …

Application Layer

- Application Service
- Data Transfer Object (DTO)
- Unit of Work
- …

Domain Driven Design
Layering in Visual Studio

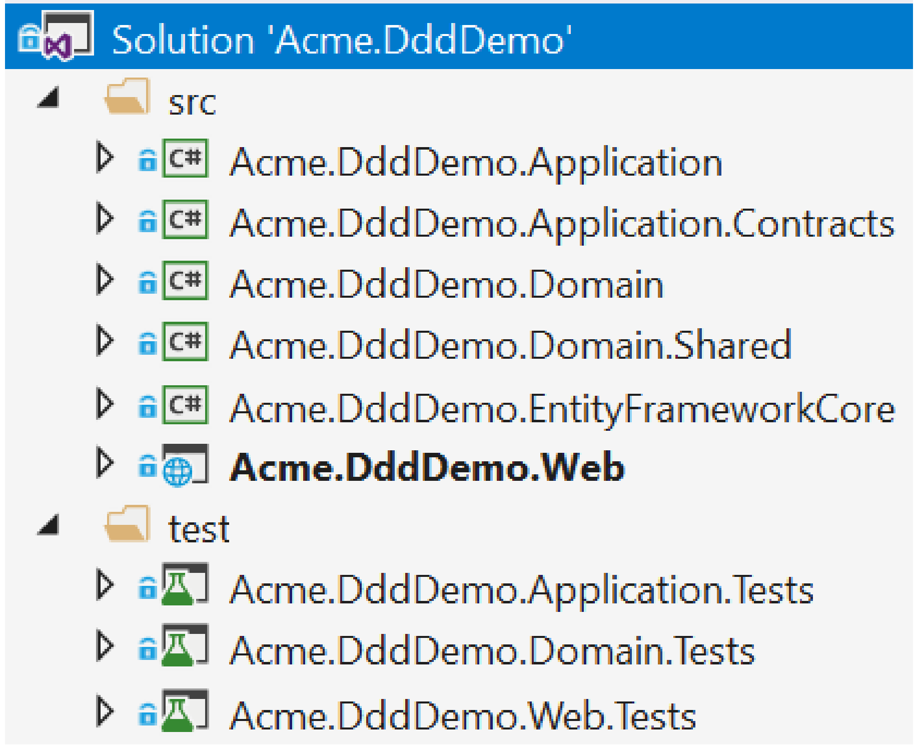


Solution 'Acme.DddDemo'
- src
  - Acme.DddDemo.Application
  - Acme.DddDemo.Application.Contracts
  - Acme.DddDemo.Domain
  - Acme.DddDemo.Domain.Shared
  - Acme.DddDemo.EntityFrameworkCore
  - **Acme.DddDemo.Web**
- test
  - Acme.DddDemo.Application.Tests
  - Acme.DddDemo.Domain.Tests
  - Acme.DddDemo.Web.Tests

- Domain.Shared
  - IssueConsts
  - IssueType (enum)
- Domain
  - Issue
  - IssueManager
  - IIssueRepository
- Application.Contracts
  - IIssueAppService
  - IssueDto, IssueCreationDto
- Application
  - IssueAppService (implementation)
- Infrastructure / EntityFrameworkCore
  - EfCoreIssueRepository
  - MyDbContext
- Web
  - IssueController
  - IssueViewModel
  - Issues.cshtml
  - Issues.js

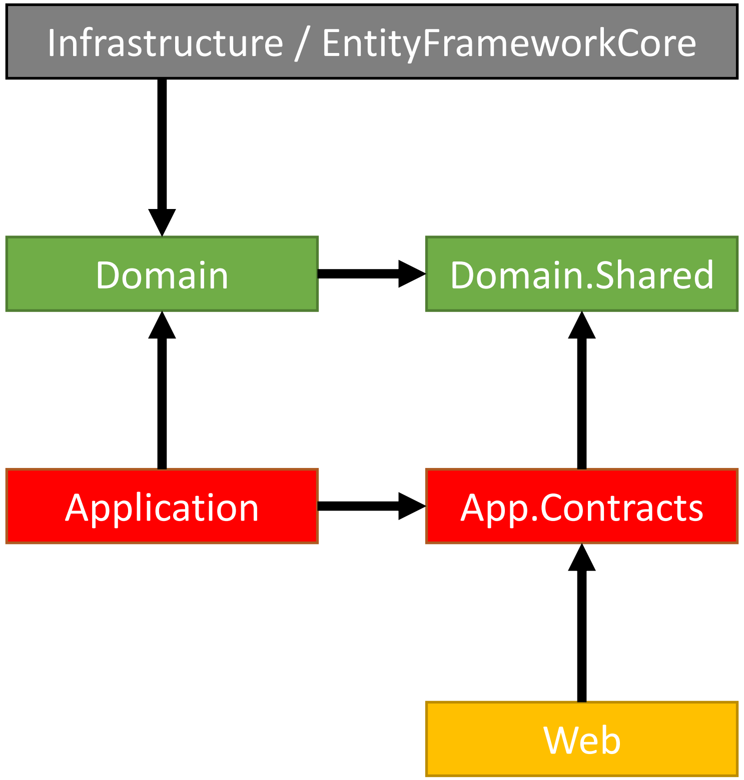
Domain Driven Design

Layering in Visual Studio

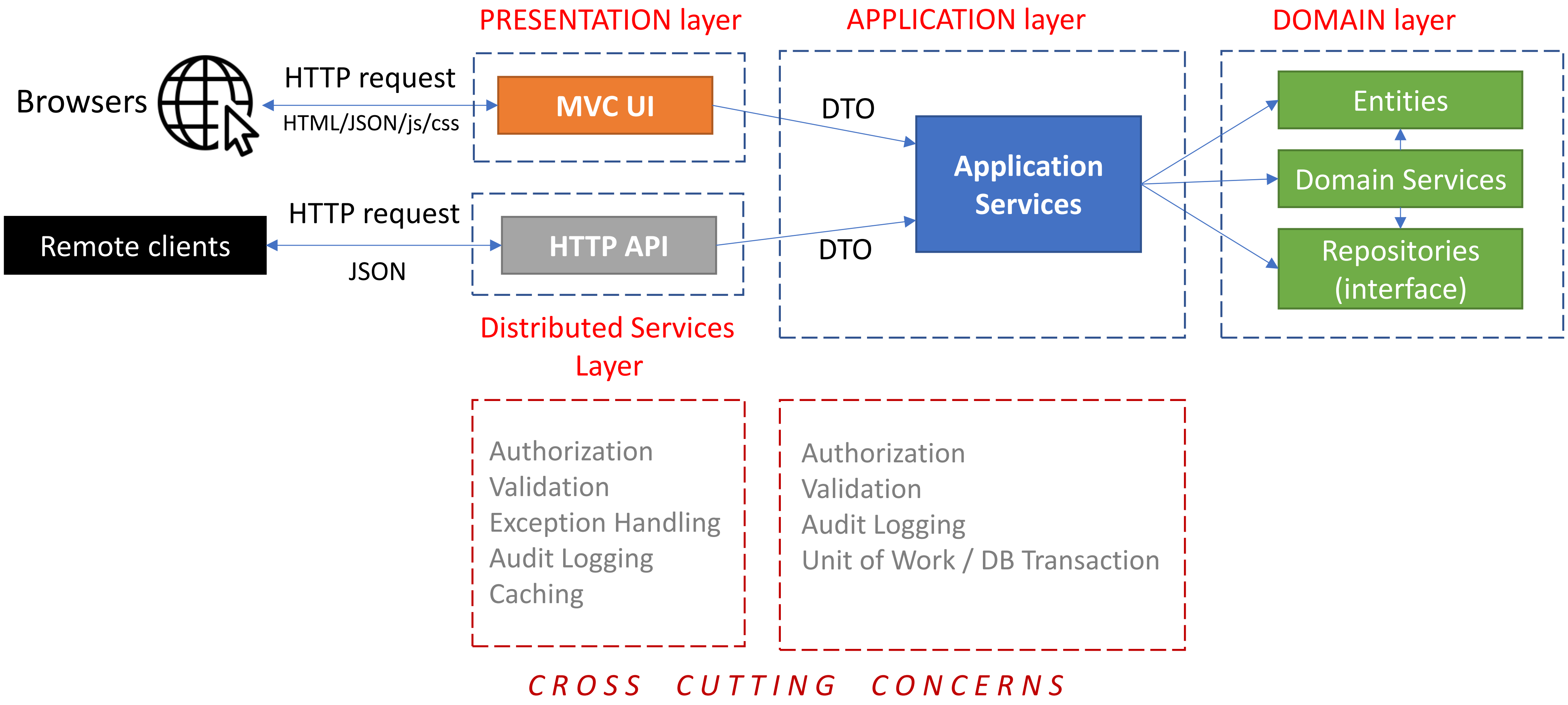


Solution 'Acme.DddDemo'

- src
  - Acme.DddDemo.Application
  - Acme.DddDemo.Application.Contracts
  - Acme.DddDemo.Domain
  - Acme.DddDemo.Domain.Shared
  - Acme.DddDemo.EntityFrameworkCore
  - **Acme.DddDemo.Web**
- test
  - Acme.DddDemo.Application.Tests
  - Acme.DddDemo.Domain.Tests
  - Acme.DddDemo.Web.Tests



Domain Driven Design
The Execution Flow

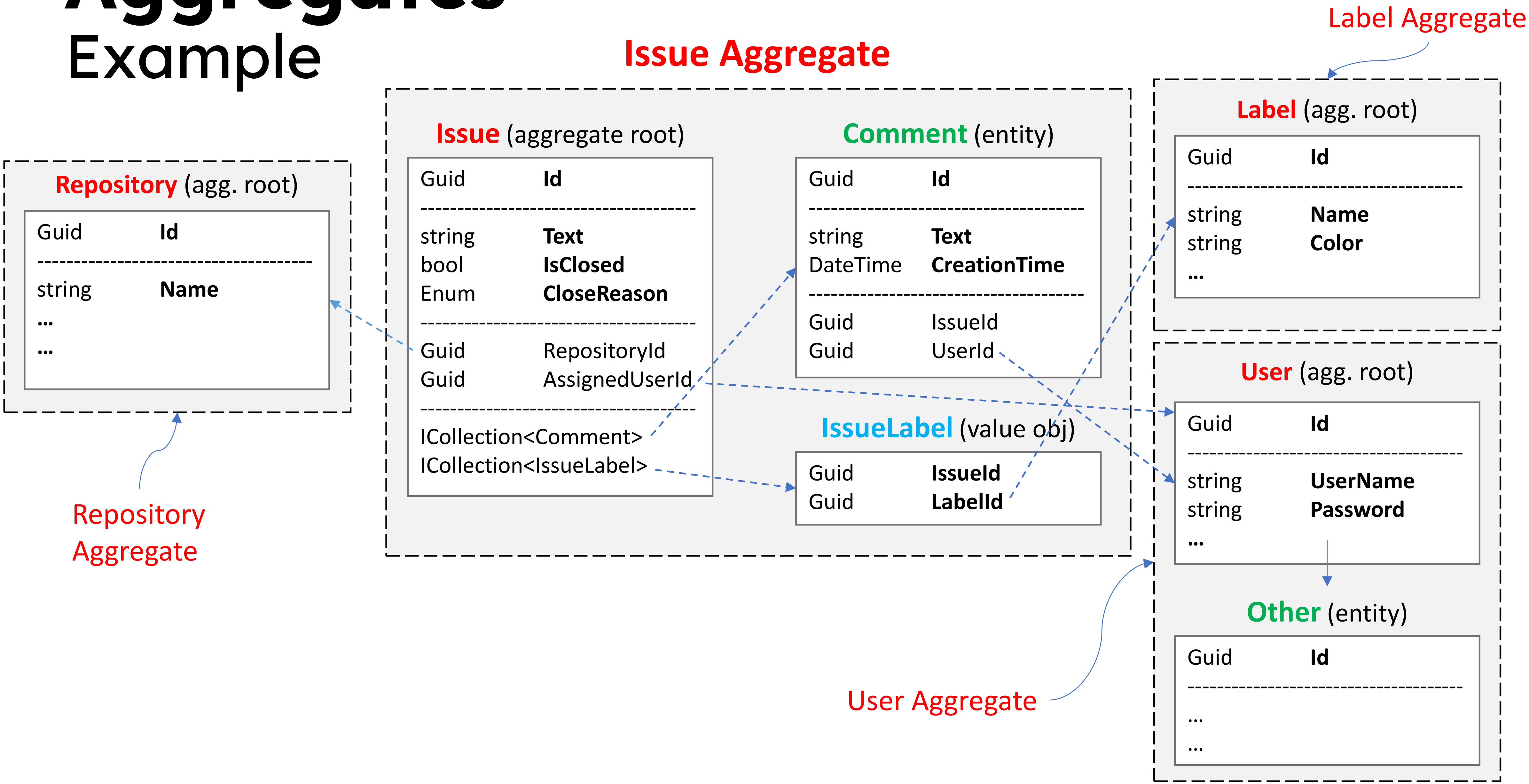


Domain Driven Design
Common Principles

- Database / ORM independence
- Presentation technology agnostic
- Doesn’t care about reporting / mass querying
- Focuses on state changes of domain objects

Part-II: Implementation

Aggregates
Example



Issue Aggregate

Issue (aggregate root)

Guid	**Id**
string	**Text**
bool	**IsClosed**
Enum	**CloseReason**
Guid	RepositoryId
Guid	AssignedUserId
ICollection<Comment>	
ICollection<IssueLabel>	

Comment (entity)

Guid	**Id**
string	**Text**
DateTime	**CreationTime**
Guid	IssueId
Guid	UserId

IssueLabel (value obj)

Guid	**IssueId**
Guid	**LabelId**

Repository Aggregate

Repository (agg. root)

Guid	**Id**
string	**Name**
…	
…	

Label Aggregate

Label (agg. root)

Guid	**Id**
string	**Name**
string	**Color**
…	

User Aggregate

User (agg. root)

Guid	**Id**
string	**UserName**
string	**Password**
…	

Other (entity)

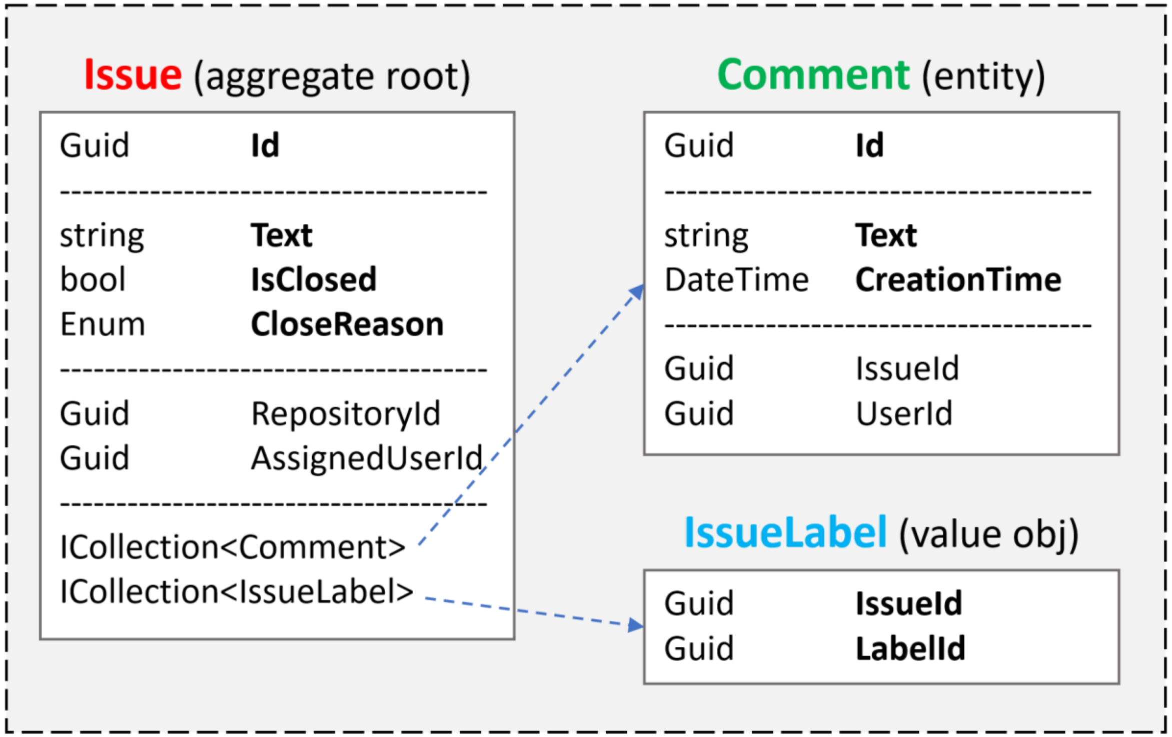
Guid	**Id**
…	
…	

Aggregate Roots

Principles

- Saved & retrieved as a **single unit** (with all sub-collections & all properties)
- Should maintain self **integrity & validity** by implementing **domain rules & constraints**
- Responsible to **manage** sub entities/objects
- An aggregate is generally considered as a **transaction boundary**.
- Should be **serializable** (already required for NoSQL databases)

Issue Aggregate



Issue (aggregate root)

Guid **Id**

string **Text**
bool **IsClosed**
Enum **CloseReason**

Guid RepositoryId
Guid AssignedUserId

ICollection<Comment>
ICollection<IssueLabel>

Comment (entity)

Guid **Id**

string **Text**
DateTime **CreationTime**

Guid IssueId
Guid UserId

IssueLabel (value obj)

Guid **IssueId**
Guid **LabelId**

Aggregate Roots

Rule: Reference Other Aggregates only by Id

```
public class GitRepository : AggregateRoot<Guid>
{
    public string Name { get; set; }

    public int StarCount { get; set; }

    public Collection<Issue> Issues { get; set; } ✗
}

public class Issue : AggregateRoot<Guid>
{
    public string Text { get; set; }

    public GitRepository Repository { get; set; } ✗

    public Guid RepositoryId { get; set; } ✓
}
```

- **Don’t** define **collections** to other aggregates!
- **Don’t** define **navigation property** to other aggregates!
- **Reference** to other aggregate roots **by Id**.

Aggregate Roots

Tip: Keep it small

```
public class Role : AggregateRoot<Guid>
{
    public string Name { get; set; }

    public Collection<UserRole> Users { get; set; } ✗
}

public class User : AggregateRoot<Guid>
{
    public string Name { get; set; }

    public Collection<UserRole> Roles { get; set; } ✓
}
```

```
public class UserRole : ValueObject
{
    public Guid UserId { get; set; }

    public Guid RoleId { get; set; }
}
```

Considerations

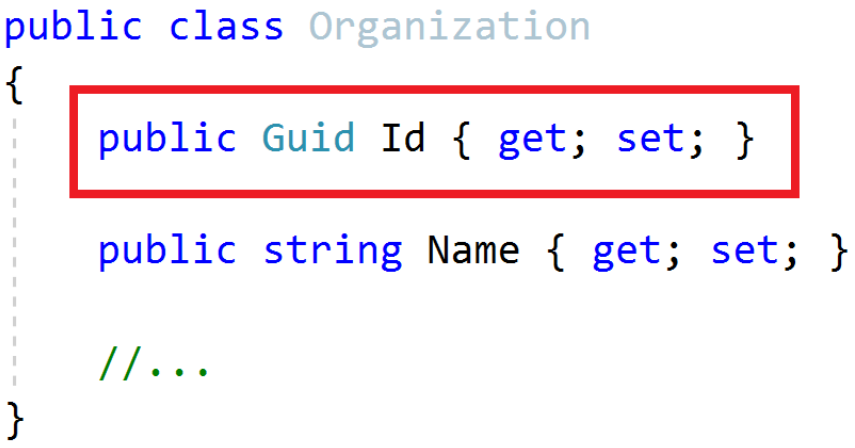
- Objects used together
- Query Performance
- Data Integrity & Validity

Aggregate Roots / Entities

Primary Keys

Aggregate Root

Define a **single** Primary Key (Id)

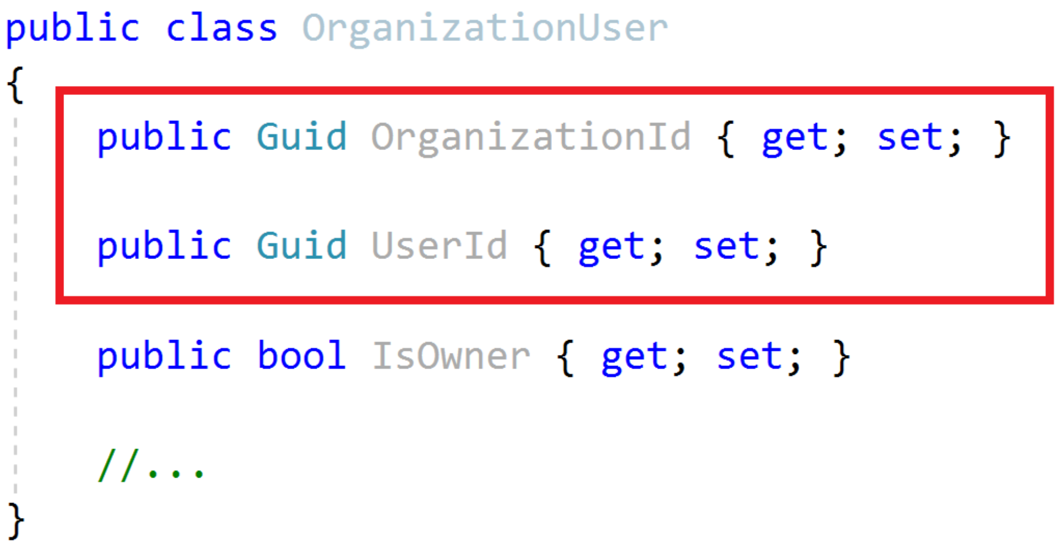


```
public class Organization
{
    public Guid Id { get; set; }

    public string Name { get; set; }

    //...
}
```

*Suggestion: Prefer **GUID** as the PK*

Entity

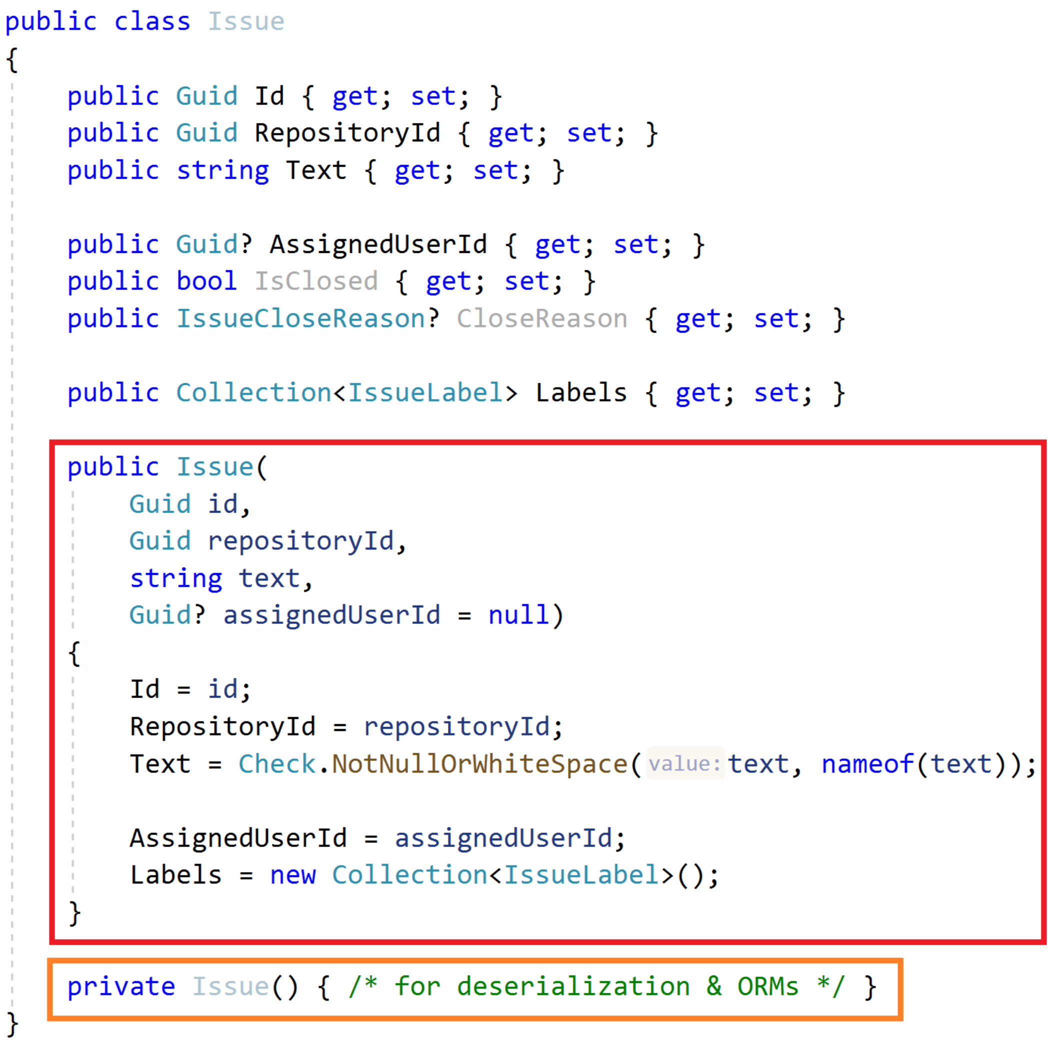
Can define a **composite** Primary Key



```
public class OrganizationUser
{
    public Guid OrganizationId { get; set; }

    public Guid UserId { get; set; }

    public bool IsOwner { get; set; }

    //...
}
```

Aggregate Roots & Entities

Constructor



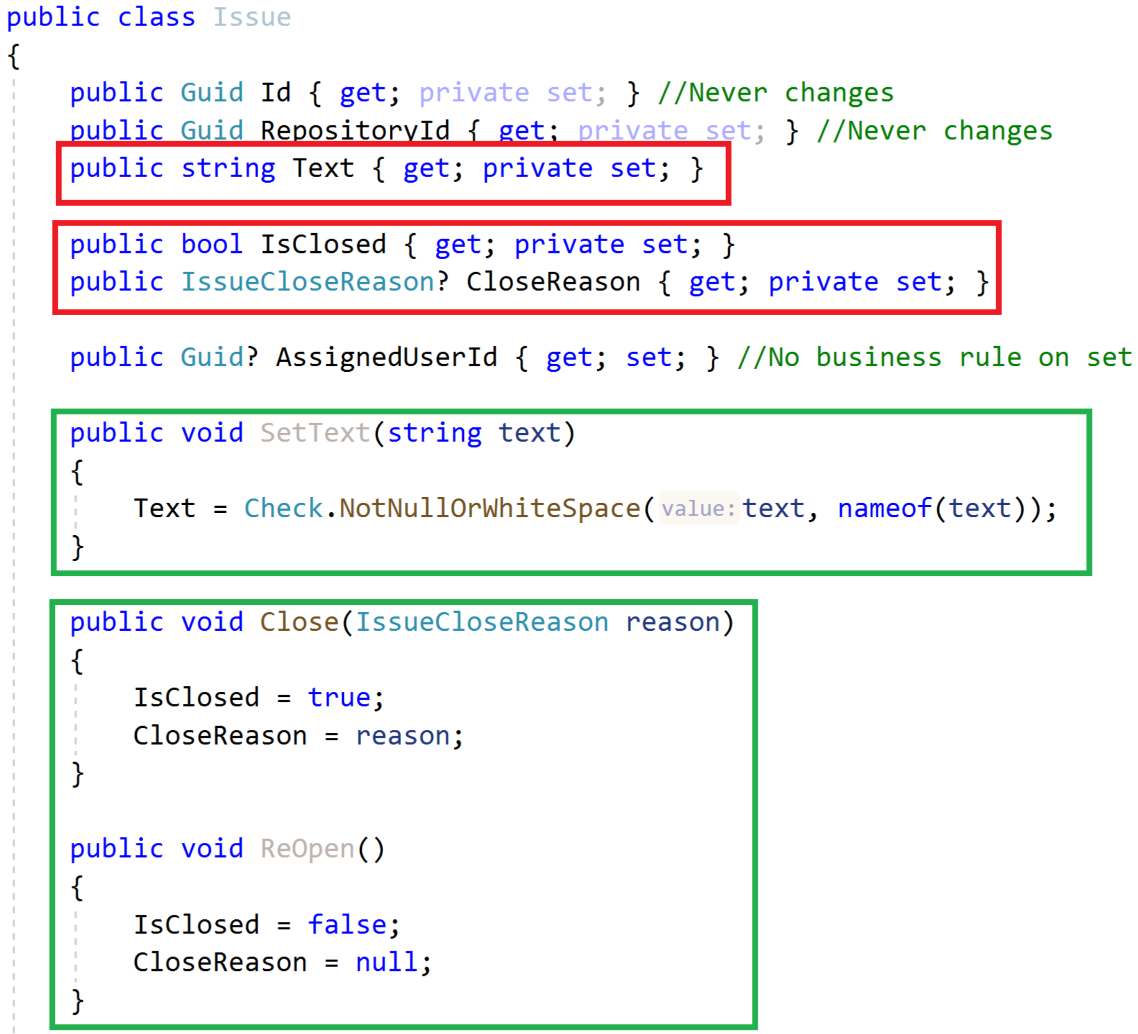
```
public class Issue
{
    public Guid Id { get; set; }
    public Guid RepositoryId { get; set; }
    public string Text { get; set; }

    public Guid? AssignedUserId { get; set; }
    public bool IsClosed { get; set; }
    public IssueCloseReason? CloseReason { get; set; }

    public Collection<IssueLabel> Labels { get; set; }

    public Issue(
        Guid id,
        Guid repositoryId,
        string text,
        Guid? assignedUserId = null)
    {
        Id = id;
        RepositoryId = repositoryId;
        Text = Check.NotNullOrWhiteSpace(value:text, nameof(text));

        AssignedUserId = assignedUserId;
        Labels = new Collection<IssueLabel>();
    }

    private Issue() { /* for deserialization & ORMs */ }
}
```

- Force to create a **VALID** entity
  - Get **minimum** required arguments
  - Check **validity** of inputs
  - Initialize **sub collections**
- Create a **private default constructor** for ORMs & deserialization
- Tip: Get id as an **argument**, don't use Guid.NewGuid() inside the constructor
  - Use a service to create GUIDs

Aggregate Roots & Entities

Property Accessors & Methods

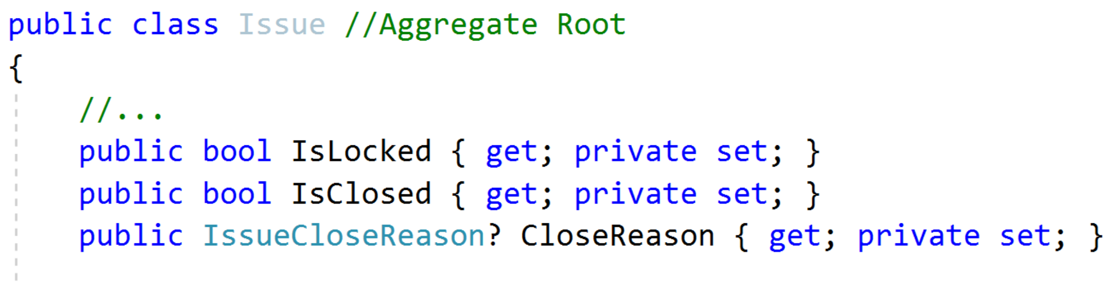
- Maintain object **validity**
- Use **private setters** when needed
- Change properties via **methods**



```
public class Issue
{
    public Guid Id { get; private set; } //Never changes
    public Guid RepositoryId { get; private set; } //Never changes
    public string Text { get; private set; }

    public bool IsClosed { get; private set; }
    public IssueCloseReason? CloseReason { get; private set; }

    public Guid? AssignedUserId { get; set; } //No business rule on set

    public void SetText(string text)
    {
        Text = Check.NotNullOrWhiteSpace(value: text, nameof(text));
    }

    public void Close(IssueCloseReason reason)
    {
        IsClosed = true;
        CloseReason = reason;
    }

    public void ReOpen()
    {
        IsClosed = false;
        CloseReason = null;
    }
```

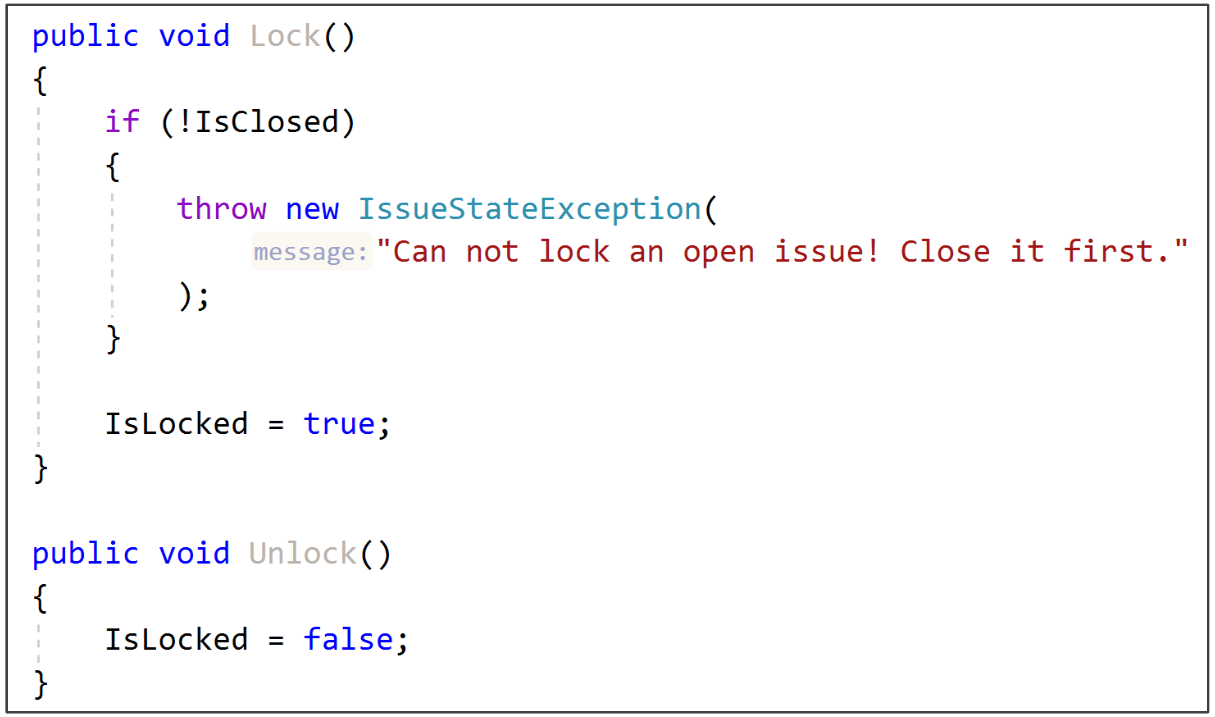
Aggregate Roots & Entities

Business Logic & Exceptions

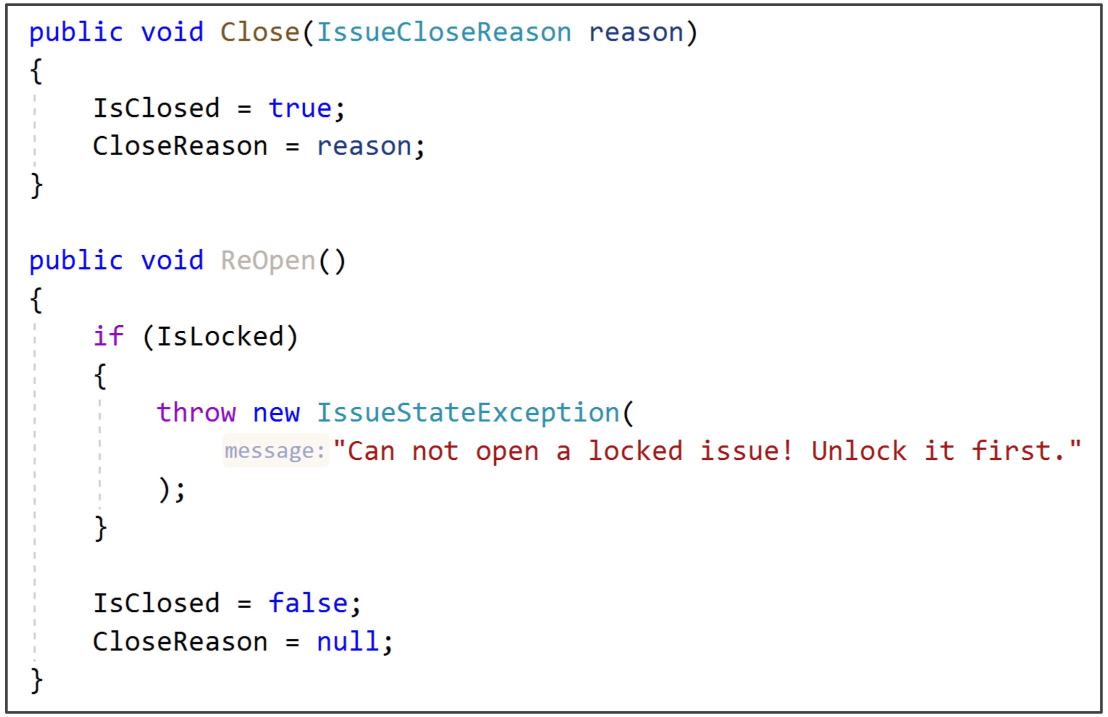


```
public class Issue //Aggregate Root
{
    //...
    public bool IsLocked { get; private set; }
    public bool IsClosed { get; private set; }
    public IssueCloseReason? CloseReason { get; private set; }
```

- Implement **Business Rules**
- Define and throw specialized **exceptions**



```
public void Lock()
{
    if (!IsClosed)
    {
        throw new IssueStateException(
            message: "Can not lock an open issue! Close it first."
        );
    }

    IsLocked = true;
}

public void Unlock()
{
    IsLocked = false;
}
```

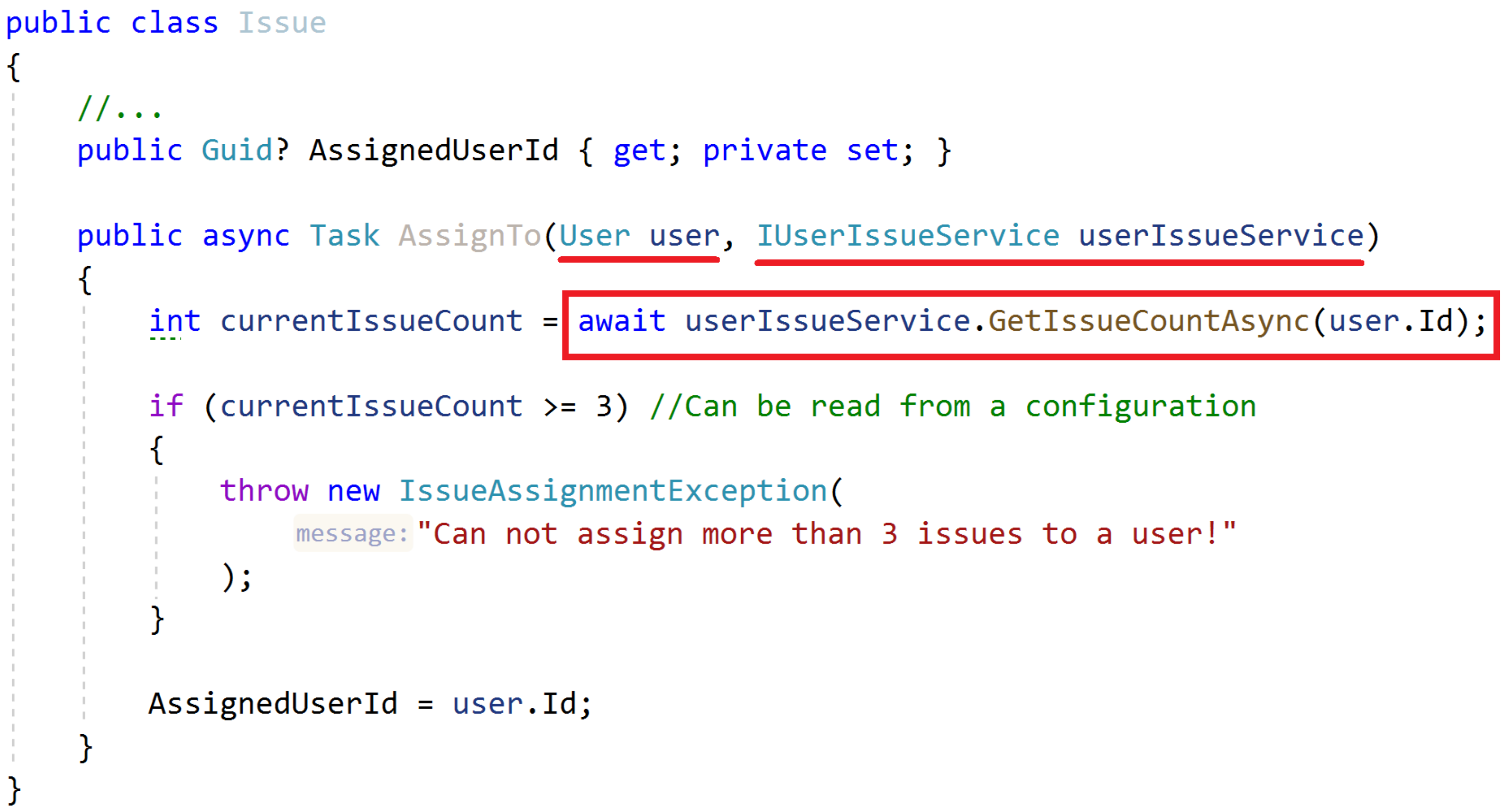


```
public void Close(IssueCloseReason reason)
{
    IsClosed = true;
    CloseReason = reason;
}

public void ReOpen()
{
    if (IsLocked)
    {
        throw new IssueStateException(
            message: "Can not open a locked issue! Unlock it first."
        );
    }

    IsClosed = false;
    CloseReason = null;
}
```

Aggregate Roots & Entities

Business Logic Requires External Services

- How to implement when you need external services?
  - Business Rule: **Can not assign more than 3 issues to a user!**



```
public class Issue
{
    //...
    public Guid? AssignedUserId { get; private set; }

    public async Task AssignTo(User user, IUserIssueService userIssueService)
    {
        int currentIssueCount = await userIssueService.GetIssueCountAsync(user.Id);

        if (currentIssueCount >= 3) //Can be read from a configuration
        {
            throw new IssueAssignmentException(
                message: "Can not assign more than 3 issues to a user!"
            );
        }

        AssignedUserId = user.Id;
    }
}
```

Aggregate Roots & Entities

Business Logic Requires External Services

- How to implement when you need external services?
  - Business Rule: **Can not assign more than 3 issues to a user!**

ALTERNATIVE..?

Create a **Domain Service**!

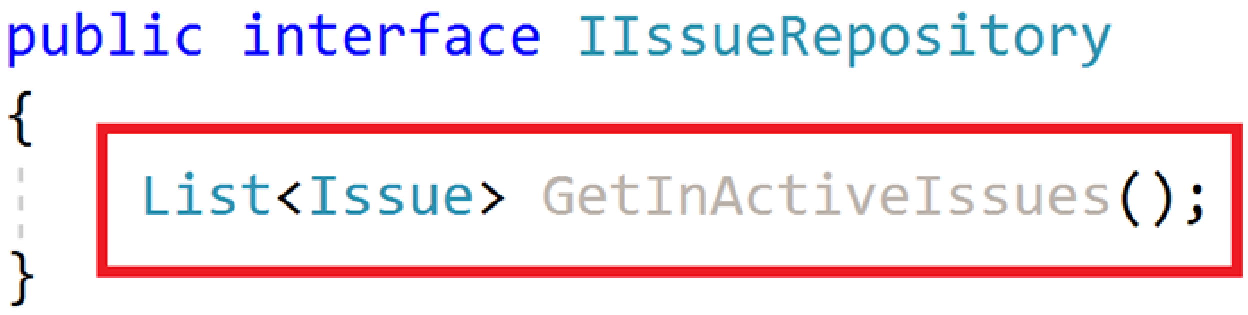
Repositories
Principles

A repository is a **collection-like** interface to interact with the database to **read and write entities**

- Define interface in the **domain layer**, implement in the **infrastructure**
- **Do not include** domain logic
- Repository interface should be **database / ORM independent**
- Create repositories for **aggregate roots**, not all entities

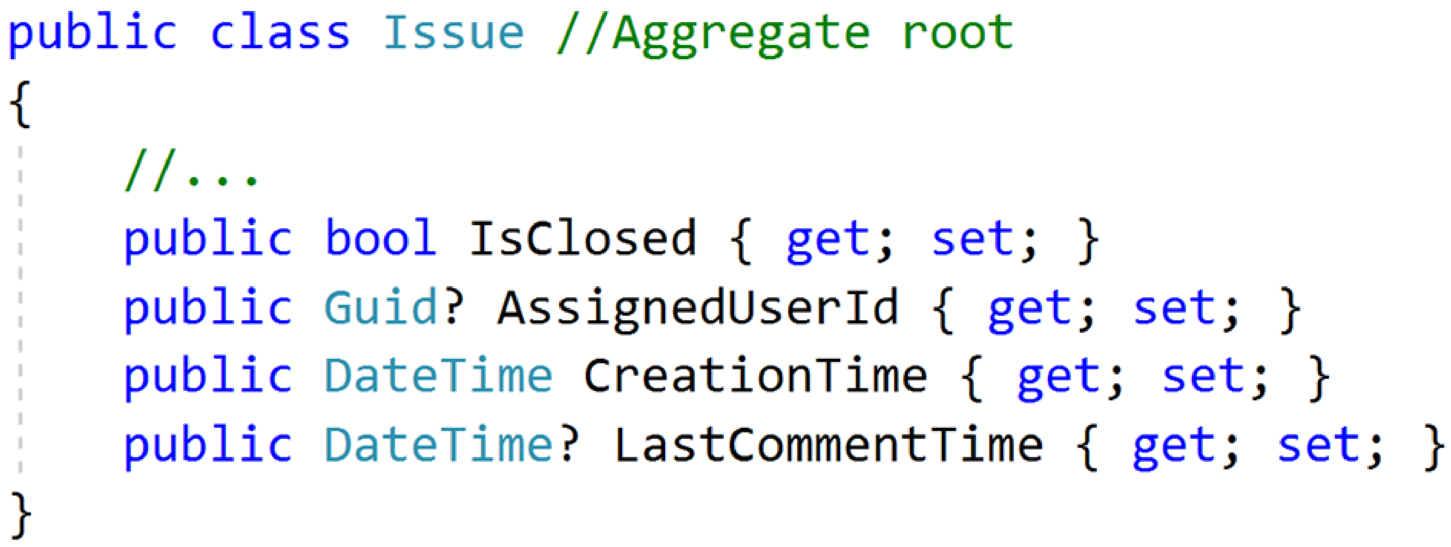
Repositories

Do not Include Domain Logic



```
public interface IIssueRepository
{
    List<Issue> GetInActiveIssues();
}
```

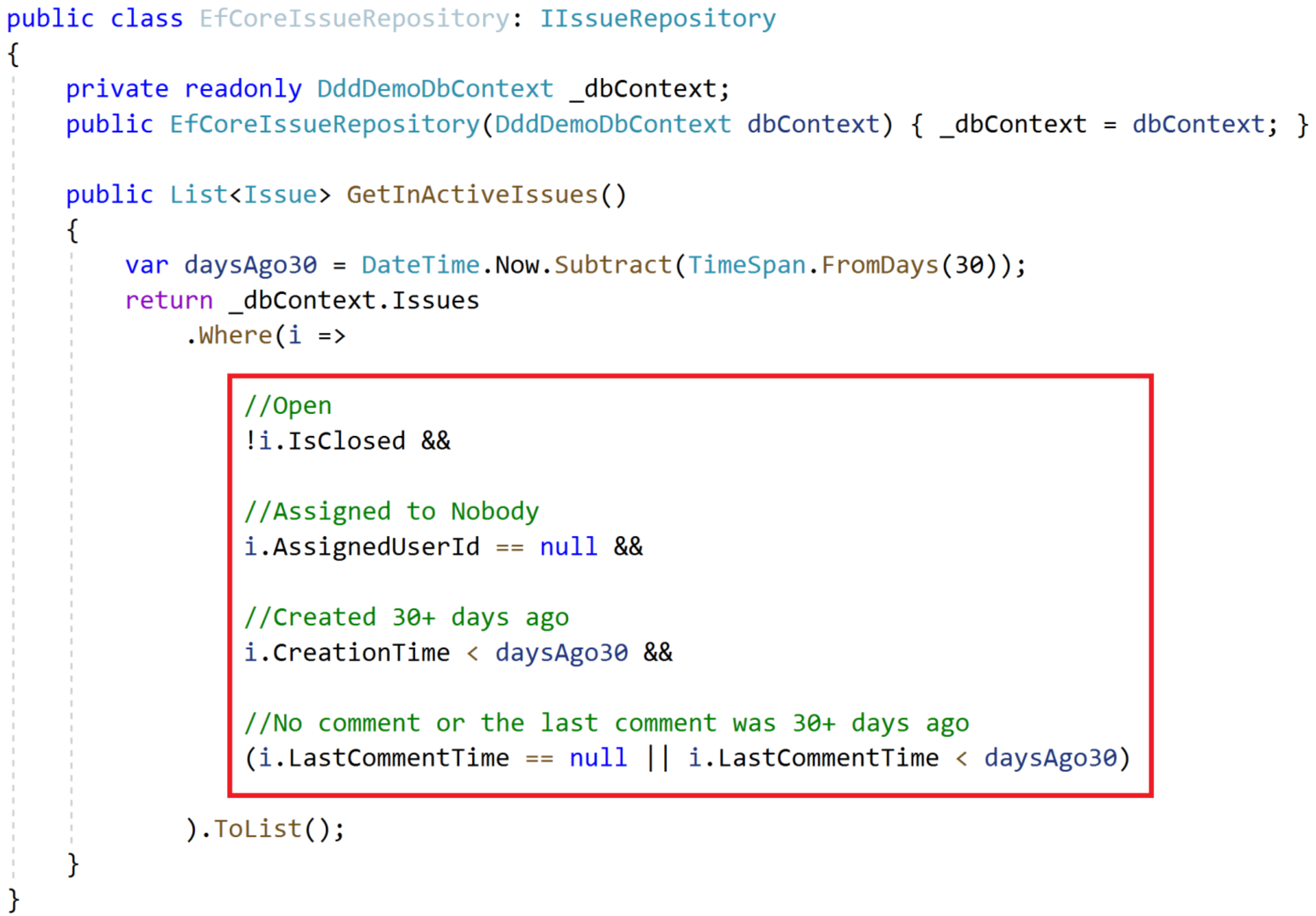
What is an In-Active issue?



```
public class Issue //Aggregate root
{
    //...
    public bool IsClosed { get; set; }
    public Guid? AssignedUserId { get; set; }
    public DateTime CreationTime { get; set; }
    public DateTime? LastCommentTime { get; set; }
}
```

Repositories

Do not Include Domain Logic



```
public class EfCoreIssueRepository: IIssueRepository
{
    private readonly DddDemoDbContext _dbContext;
    public EfCoreIssueRepository(DddDemoDbContext dbContext) { _dbContext = dbContext; }

    public List<Issue> GetInActiveIssues()
    {
        var daysAgo30 = DateTime.Now.Subtract(TimeSpan.FromDays(30));
        return _dbContext.Issues
            .Where(i =>

                //Open
                !i.IsClosed &&

                //Assigned to Nobody
                i.AssignedUserId == null &&

                //Created 30+ days ago
                i.CreationTime < daysAgo30 &&

                //No comment or the last comment was 30+ days ago
                (i.LastCommentTime == null || i.LastCommentTime < daysAgo30)

            ).ToList();
    }
}
```

- **Implicit** definition of a domain rule!
- How to **re-use** this expression?

Repositories

Do not Include Domain Logic

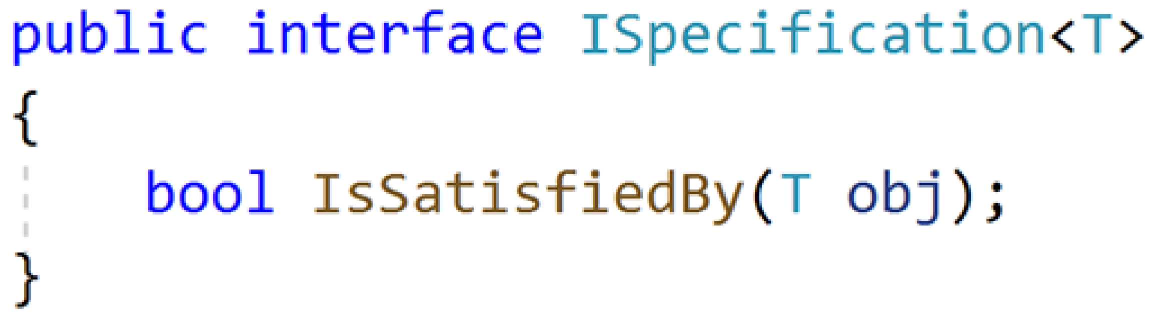
```
public class Issue //Aggregate root
{
    //...
    public bool IsClosed { get; set; }
    public Guid? AssignedUserId { get; set; }
    public DateTime CreationTime { get; set; }
    public DateTime? LastCommentTime { get; set; }

    public bool IsInActive()
    {
        var daysAgo30 = DateTime.Now.Subtract(TimeSpan.FromDays(30));
        return
            //Open
            !IsClosed &&

            //Assigned to Nobody
            AssignedUserId == null &&

            //Created 30+ days ago
            CreationTime < daysAgo30 &&

            //No comment or the last comment was 30+ days ago
            (LastCommentTime == null || LastCommentTime < daysAgo30);
    }
}
```

- **Implicit** definition of a domain rule!
- How to **re-use** this expression?
- **Copy/paste**?
- **Solution: The Specification Pattern!**

Specifications

The Specification Interface

A specification is a **named**, **reusable** & **combinable** **class** to filter objects.

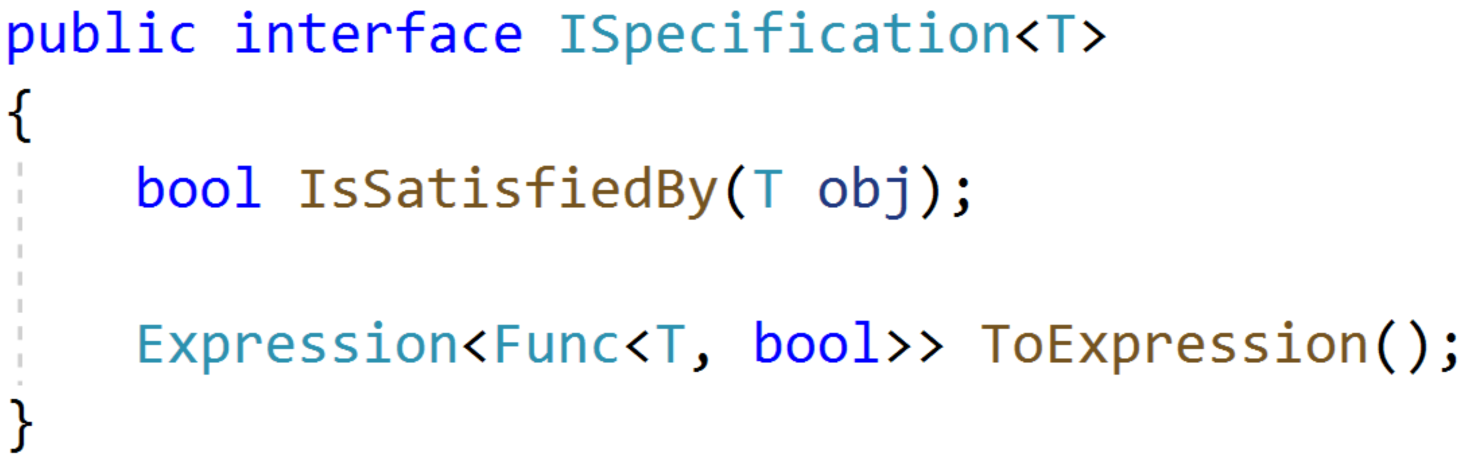


```
public interface ISpecification<T>
{
    bool IsSatisfiedBy(T obj);
}
```

Specifications

Extended Specification Interface

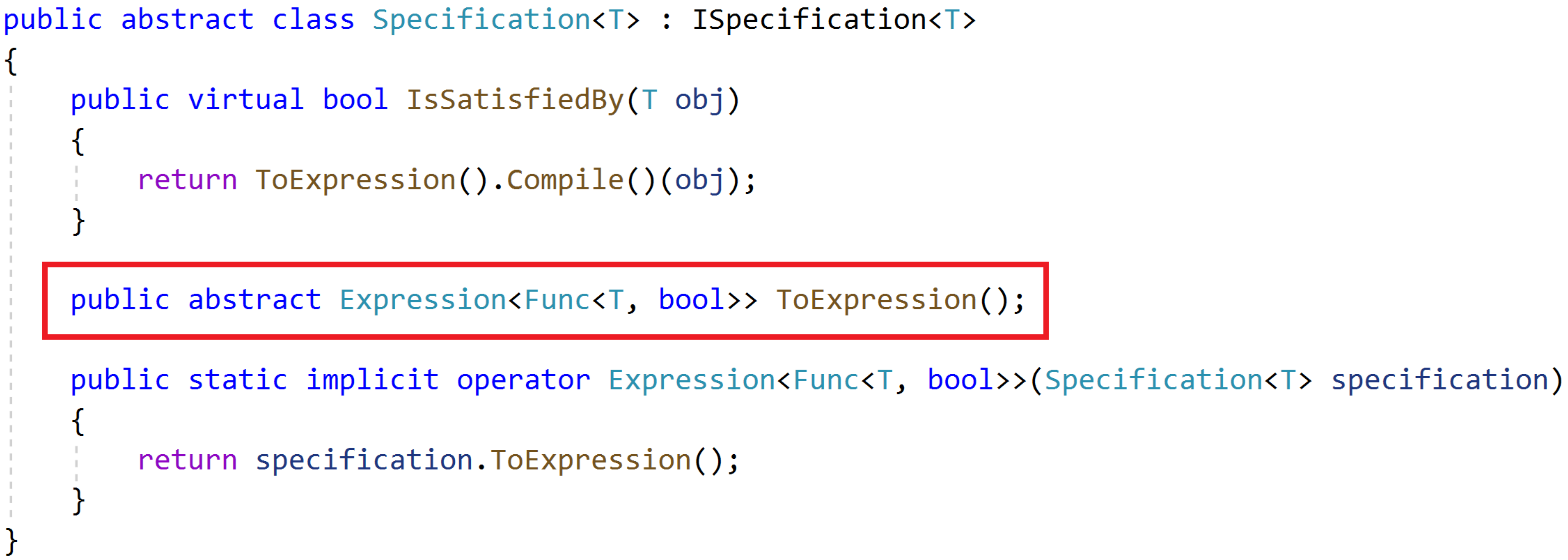
A specification is a **named**, **reusable** & **combinable** **class** to filter objects.



```
public interface ISpecification<T>
{
    bool IsSatisfiedBy(T obj);

    Expression<Func<T, bool>> ToExpression();
}
```

Specifications

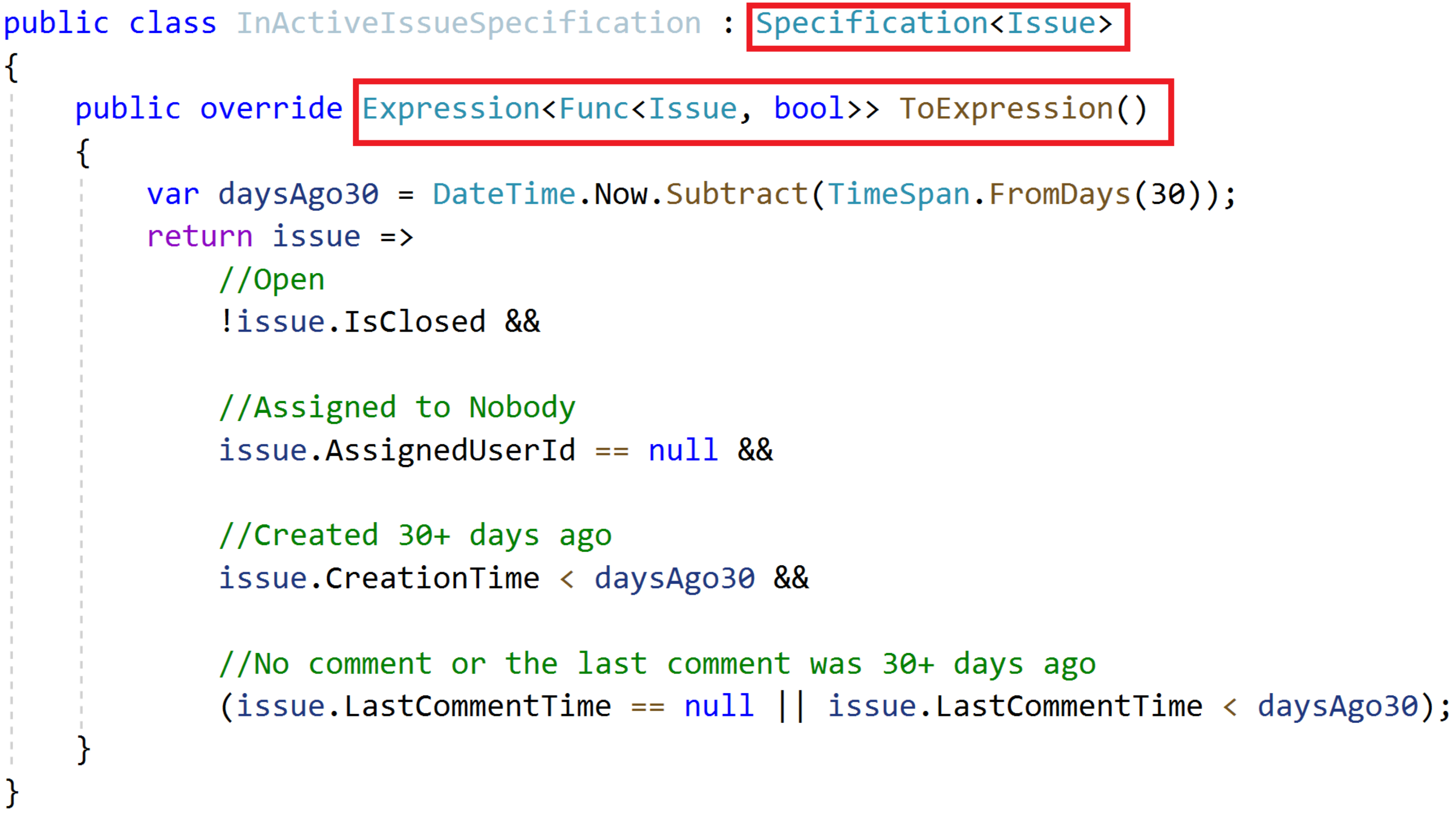
A Base Specification Class



```
public abstract class Specification<T> : ISpecification<T>
{
    public virtual bool IsSatisfiedBy(T obj)
    {
        return ToExpression().Compile()(obj);
    }

    public abstract Expression<Func<T, bool>> ToExpression();

    public static implicit operator Expression<Func<T, bool>>(Specification<T> specification)
    {
        return specification.ToExpression();
    }
}
```

Specifications

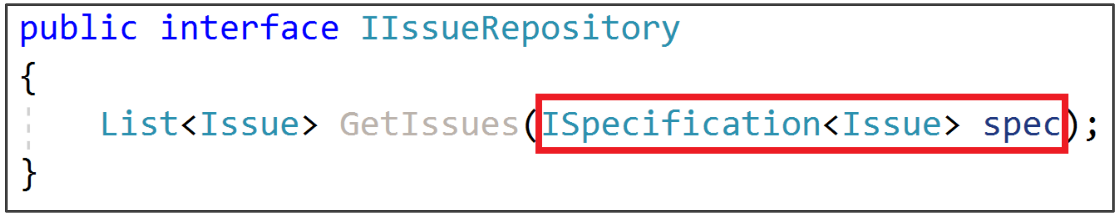
Define a Specification



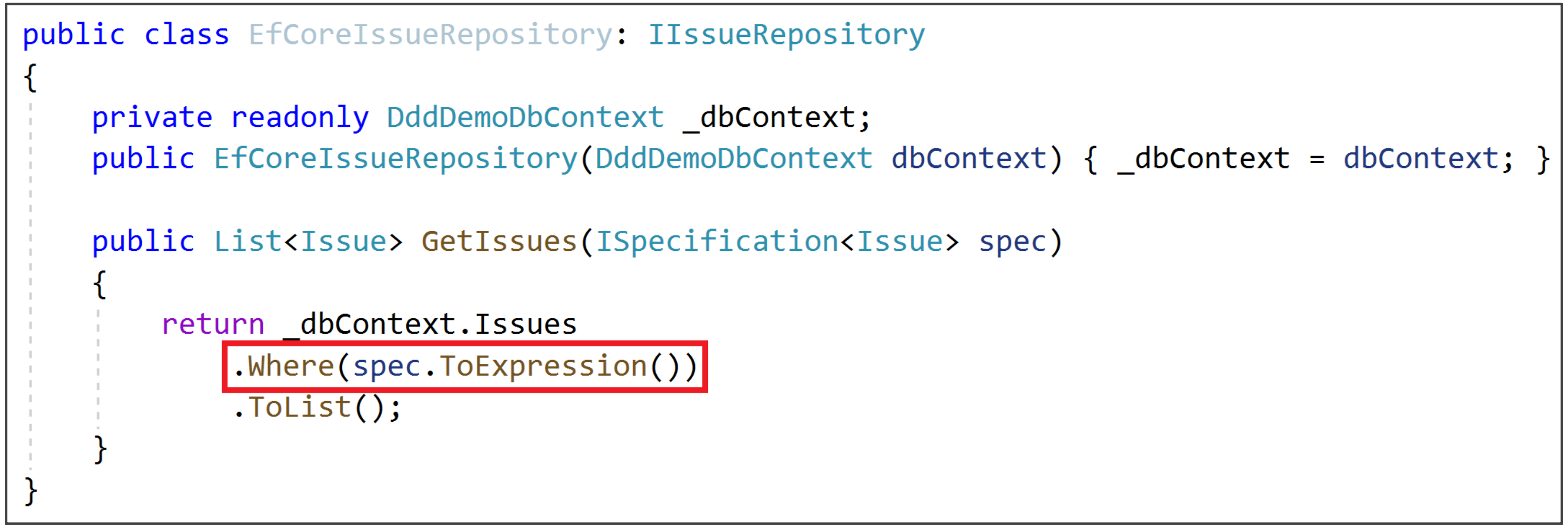
```
public class InActiveIssueSpecification : Specification<Issue>
{
    public override Expression<Func<Issue, bool>> ToExpression()
    {
        var daysAgo30 = DateTime.Now.Subtract(TimeSpan.FromDays(30));
        return issue =>
            //Open
            !issue.IsClosed &&

            //Assigned to Nobody
            issue.AssignedUserId == null &&

            //Created 30+ days ago
            issue.CreationTime < daysAgo30 &&

            //No comment or the last comment was 30+ days ago
            (issue.LastCommentTime == null || issue.LastCommentTime < daysAgo30);
    }
}
```

Specifications

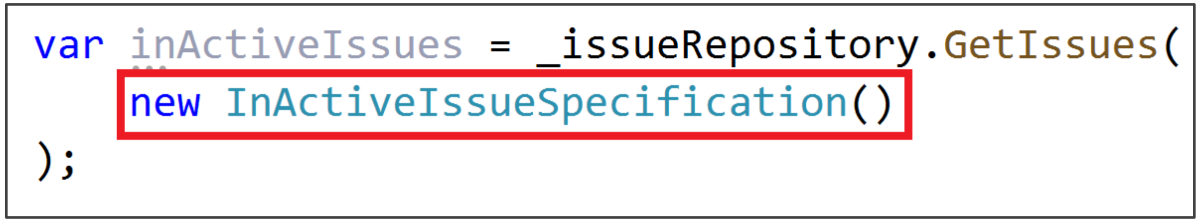
Use the Specification



```
public interface IIssueRepository
{
    List<Issue> GetIssues(ISpecification<Issue> spec);
}
```



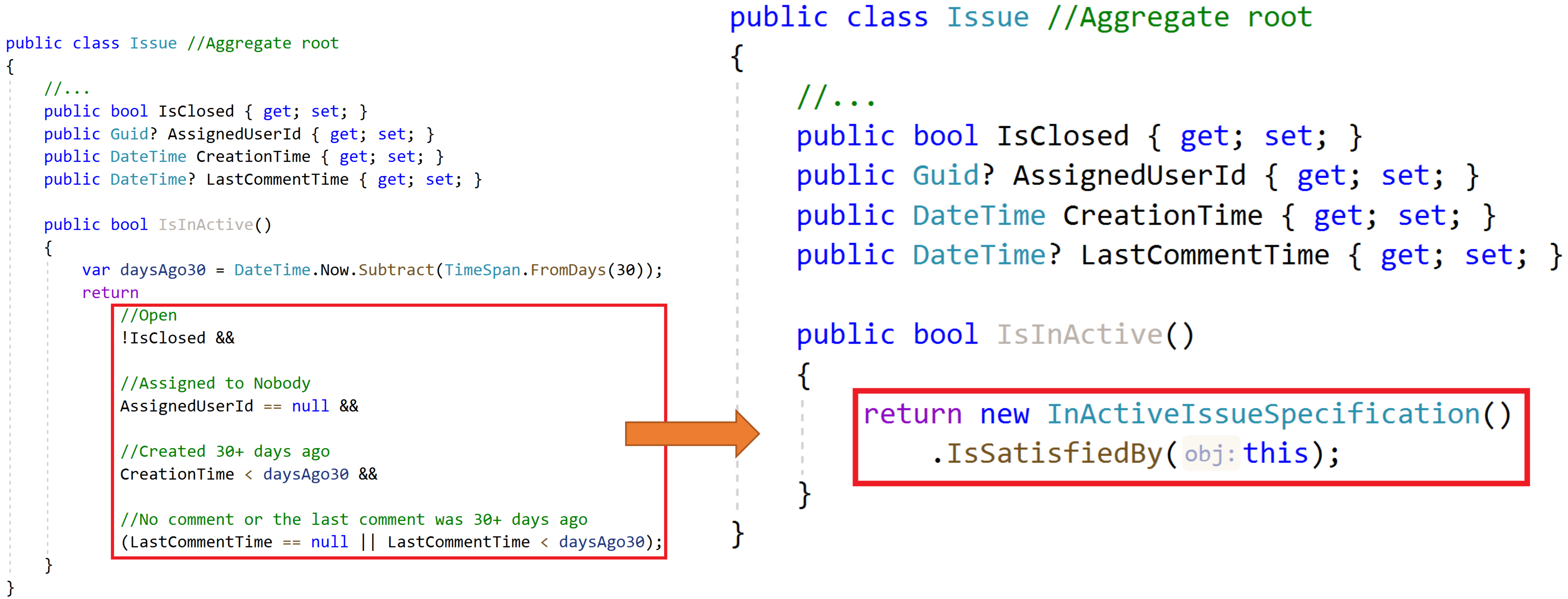
```
public class EfCoreIssueRepository: IIssueRepository
{
    private readonly DddDemoDbContext _dbContext;
    public EfCoreIssueRepository(DddDemoDbContext dbContext) { _dbContext = dbContext; }

    public List<Issue> GetIssues(ISpecification<Issue> spec)
    {
        return _dbContext.Issues
            .Where(spec.ToExpression())
            .ToList();
    }
}
```



```
var inActiveIssues = _issueRepository.GetIssues(
    new InActiveIssueSpecification()
);
```

Specifications

Use the Specification

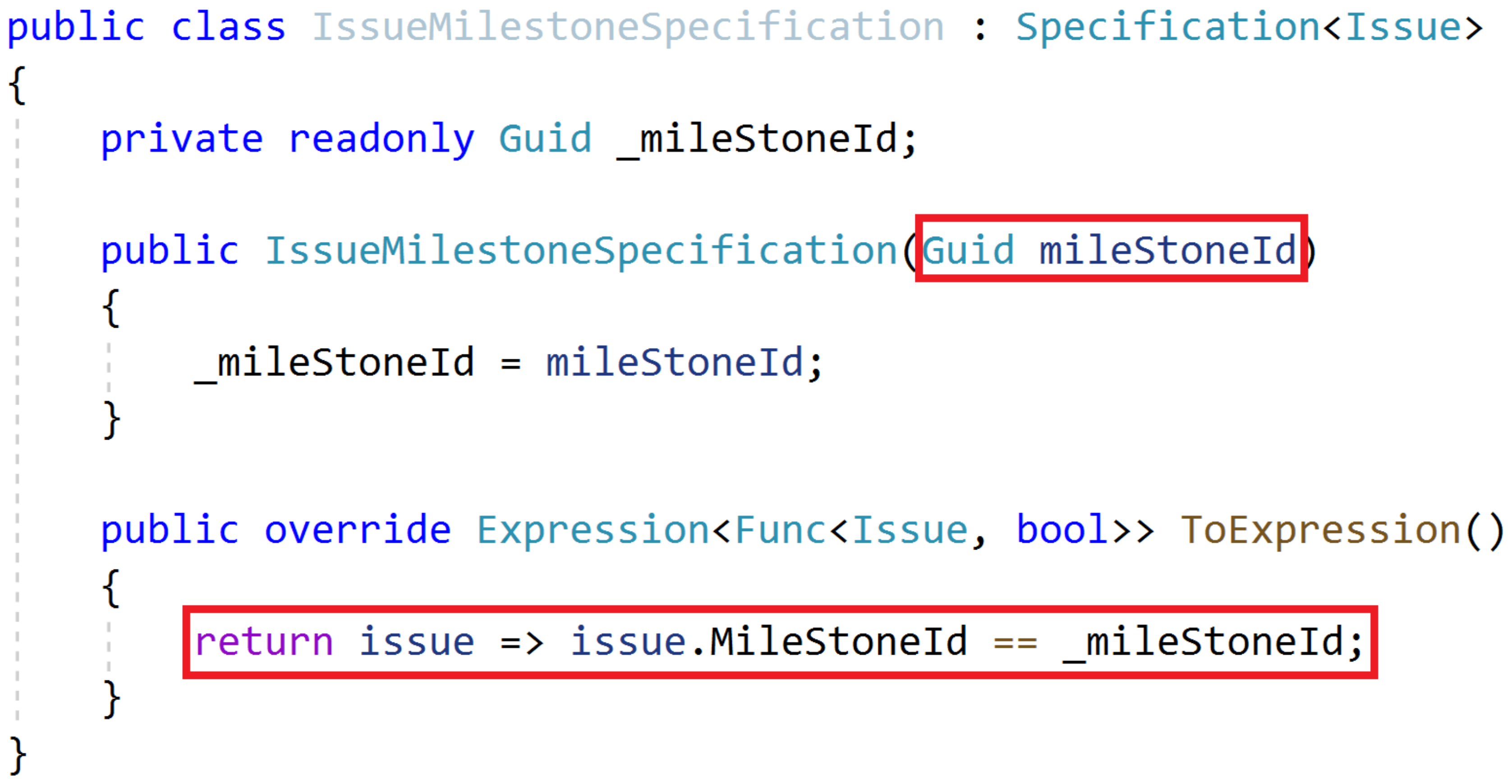


```
public class Issue //Aggregate root
{
    //...
    public bool IsClosed { get; set; }
    public Guid? AssignedUserId { get; set; }
    public DateTime CreationTime { get; set; }
    public DateTime? LastCommentTime { get; set; }

    public bool IsInActive()
    {
        var daysAgo30 = DateTime.Now.Subtract(TimeSpan.FromDays(30));
        return
            //Open
            !IsClosed &&

            //Assigned to Nobody
            AssignedUserId == null &&

            //Created 30+ days ago
            CreationTime < daysAgo30 &&

            //No comment or the last comment was 30+ days ago
            (LastCommentTime == null || LastCommentTime < daysAgo30);
    }
}
```

```
public class Issue //Aggregate root
{
    //...
    public bool IsClosed { get; set; }
    public Guid? AssignedUserId { get; set; }
    public DateTime CreationTime { get; set; }
    public DateTime? LastCommentTime { get; set; }

    public bool IsInActive()
    {
        return new InActiveIssueSpecification()
            .IsSatisfiedBy(obj: this);
    }
}
```

Specifications

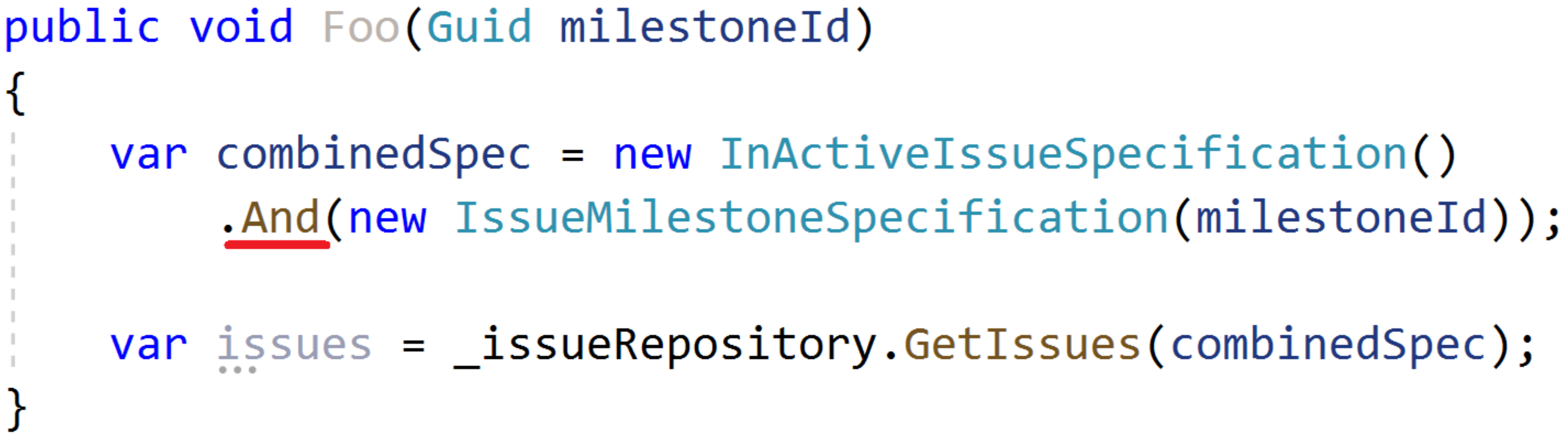
Parameterized Specifications



```
public class IssueMilestoneSpecification : Specification<Issue>
{
    private readonly Guid _mileStoneId;

    public IssueMilestoneSpecification(Guid mileStoneId)
    {
        _mileStoneId = mileStoneId;
    }

    public override Expression<Func<Issue, bool>> ToExpression()
    {
        return issue => issue.MileStoneId == _mileStoneId;
    }
}
```

Specifications

Combining Multiple Specifications



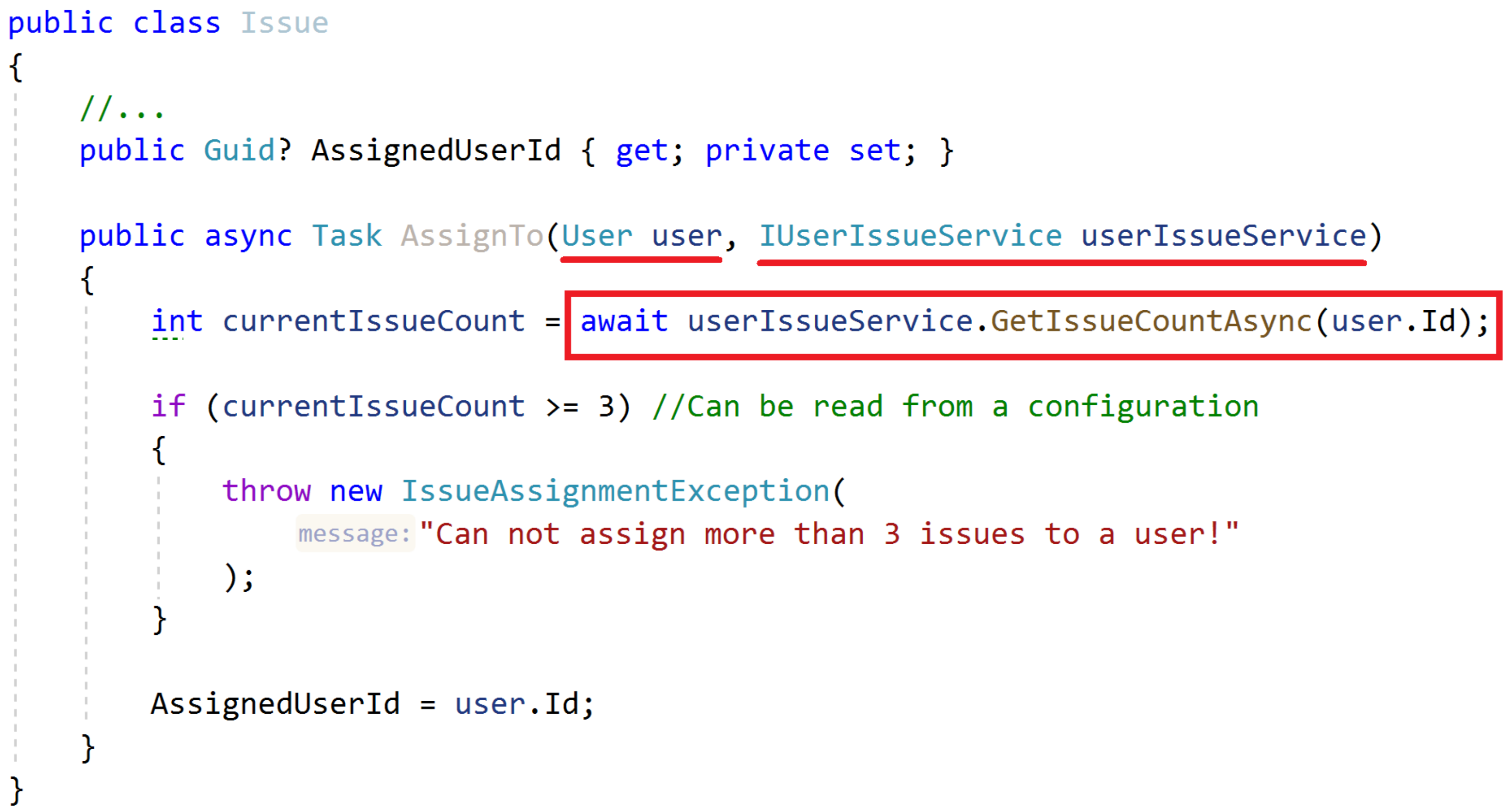
```
public void Foo(Guid milestoneId)
{
    var combinedSpec = new InActiveIssueSpecification()
        .And(new IssueMilestoneSpecification(milestoneId));

    var issues = _issueRepository.GetIssues(combinedSpec);
}
```

Domain Services
Principles

- Implements domain logic that;
  - Depends on **services and repositories**
  - Needs to work with **multiple entities** / entity types
- Works with **domain objects**, not DTOs

Domain Services
Example

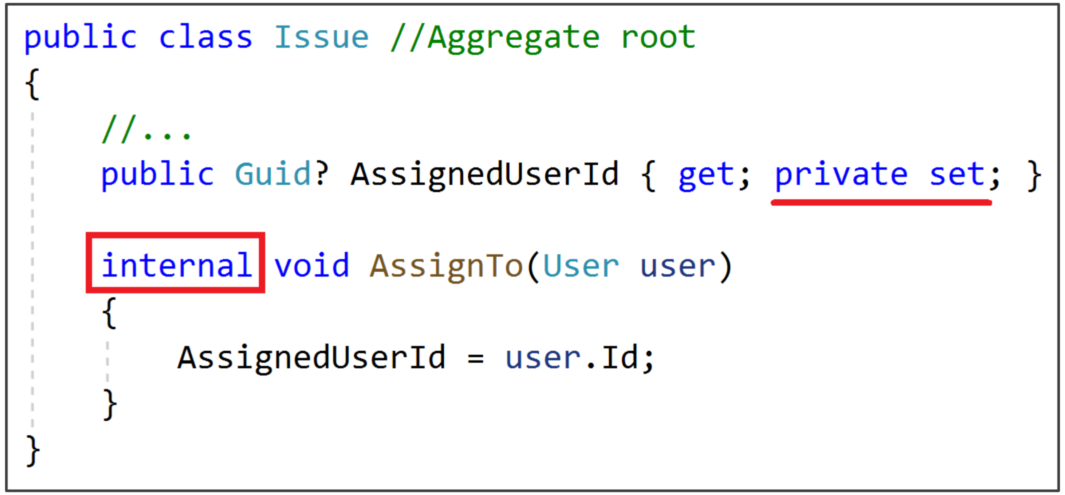
Business Rule: Can not assign more than 3 issues to a user



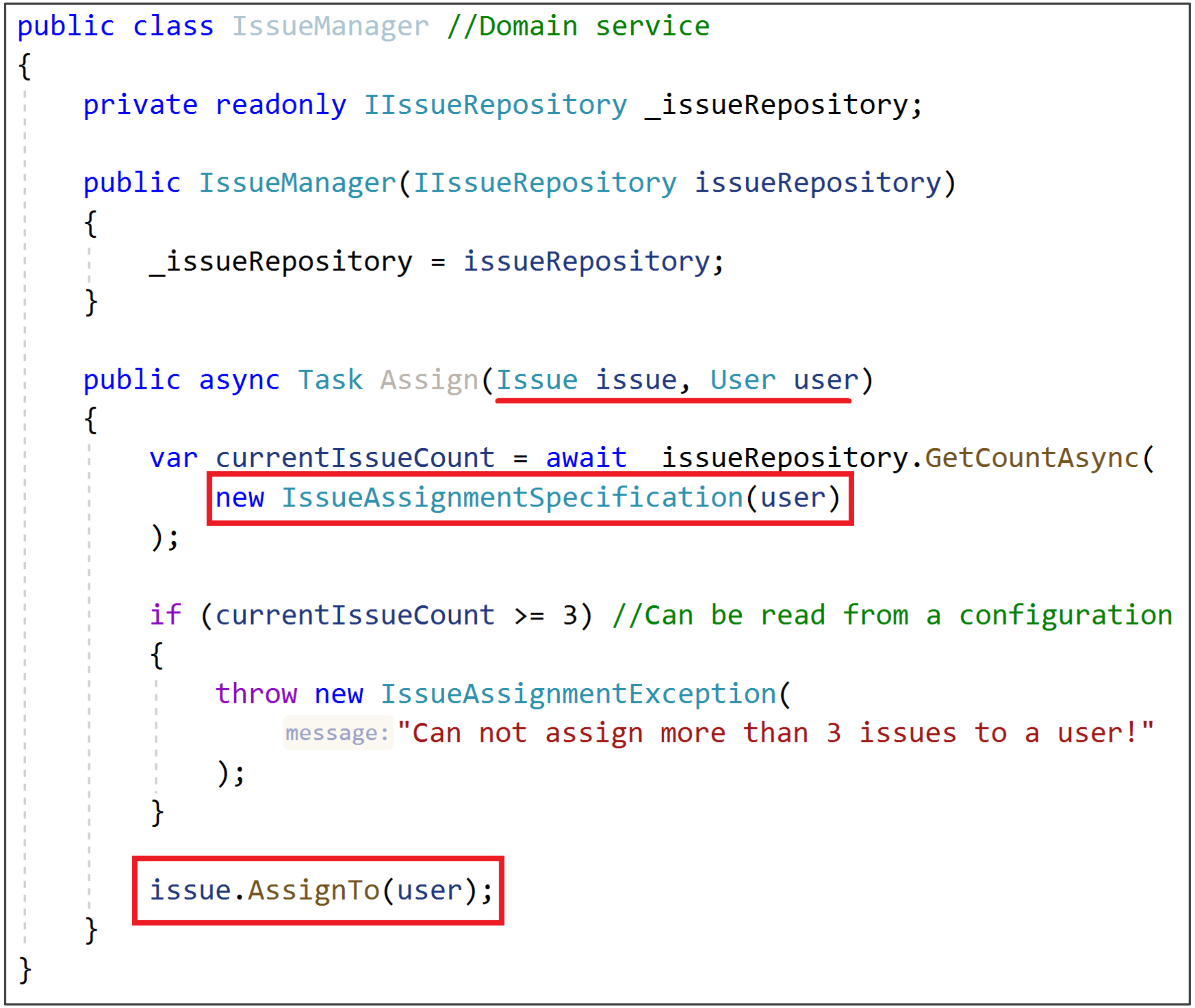
```
public class Issue
{
    //...
    public Guid? AssignedUserId { get; private set; }

    public async Task AssignTo(User user, IUserIssueService userIssueService)
    {
        int currentIssueCount = await userIssueService.GetIssueCountAsync(user.Id);

        if (currentIssueCount >= 3) //Can be read from a configuration
        {
            throw new IssueAssignmentException(
                message: "Can not assign more than 3 issues to a user!"
            );
        }

        AssignedUserId = user.Id;
    }
}
```

Domain Services

Example



```
public class Issue //Aggregate root
{
    //...
    public Guid? AssignedUserId { get; private set; }

    internal void AssignTo(User user)
    {
        AssignedUserId = user.Id;
    }
}
```



```
public class IssueManager //Domain service
{
    private readonly IIssueRepository _issueRepository;

    public IssueManager(IIssueRepository issueRepository)
    {
        _issueRepository = issueRepository;
    }

    public async Task Assign(Issue issue, User user)
    {
        var currentIssueCount = await _issueRepository.GetCountAsync(
            new IssueAssignmentSpecification(user)
        );

        if (currentIssueCount >= 3) //Can be read from a configuration
        {
            throw new IssueAssignmentException(
                message: "Can not assign more than 3 issues to a user!"
            );
        }

        issue.AssignTo(user);
    }
}
```

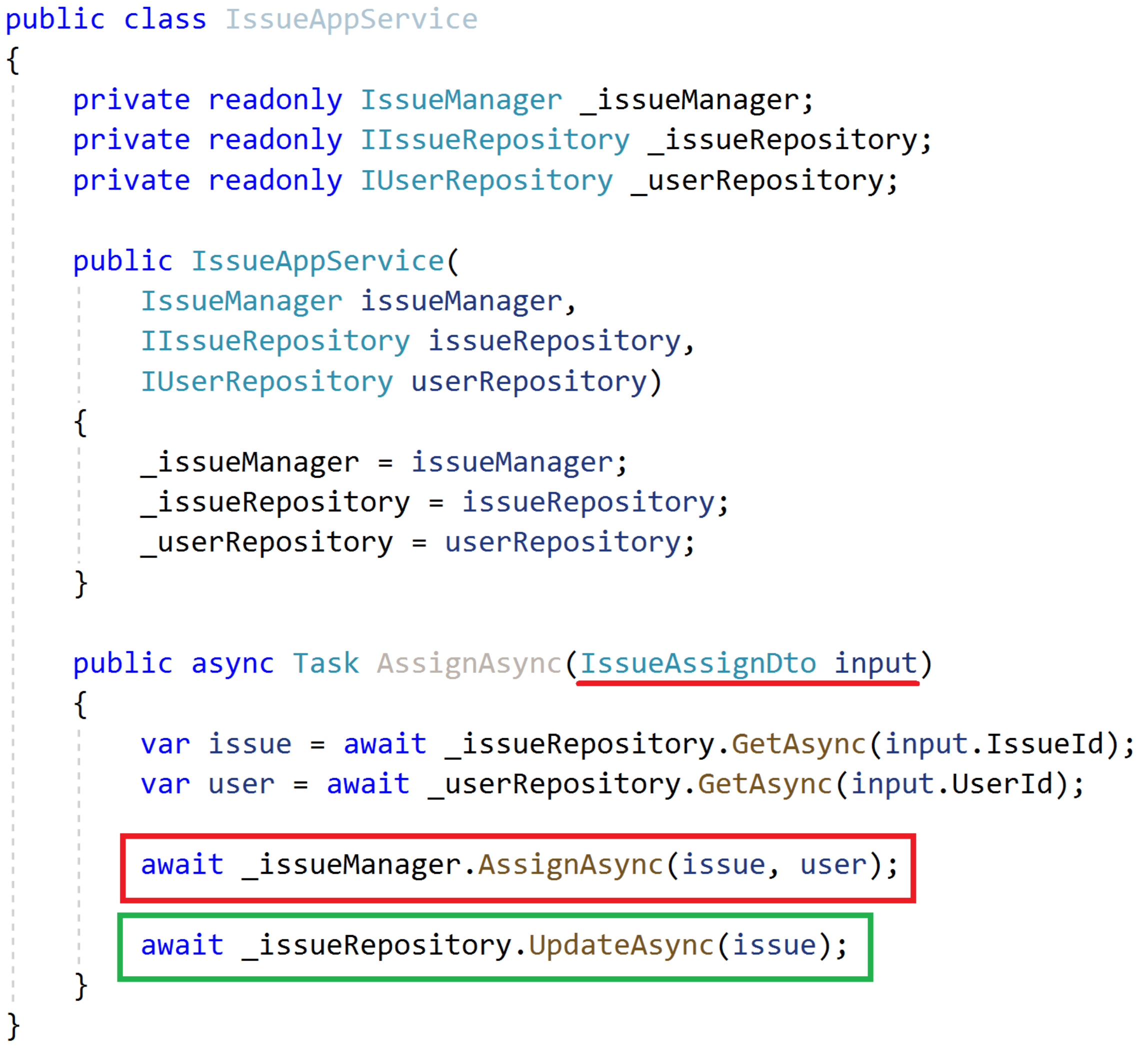
Application Services

Principles

- Implement **use cases** of the application (application logic)
- Do not implement core domain logic
- Get & return **Data Transfer Objects**, not entities
- Use domain services, entities, repositories and other domain objects inside

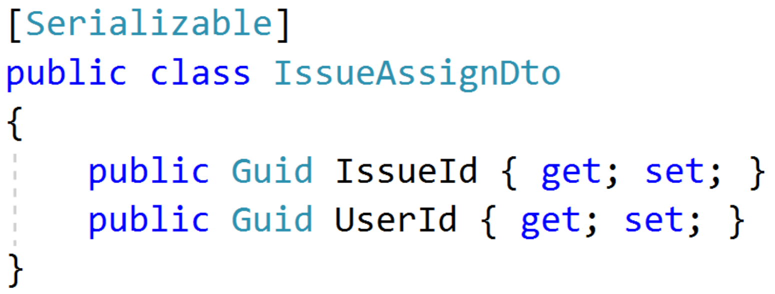
Application Services

Example



```
public class IssueAppService
{
    private readonly IssueManager _issueManager;
    private readonly IIssueRepository _issueRepository;
    private readonly IUserRepository _userRepository;

    public IssueAppService(
        IssueManager issueManager,
        IIssueRepository issueRepository,
        IUserRepository userRepository)
    {
        _issueManager = issueManager;
        _issueRepository = issueRepository;
        _userRepository = userRepository;
    }

    public async Task AssignAsync(IssueAssignDto input)
    {
        var issue = await _issueRepository.GetAsync(input.IssueId);
        var user = await _userRepository.GetAsync(input.UserId);

        await _issueManager.AssignAsync(issue, user);

        await _issueRepository.UpdateAsync(issue);
    }
}
```

- Inject domain services & repositories
- Get DTO as argument



```
[Serializable]
public class IssueAssignDto
{
    public Guid IssueId { get; set; }
    public Guid UserId { get; set; }
}
```

- Get aggregate roots from repositories
- Use domain service to perform the domain logic
- Always update the entity explicitly (don’t assume the change tracking)

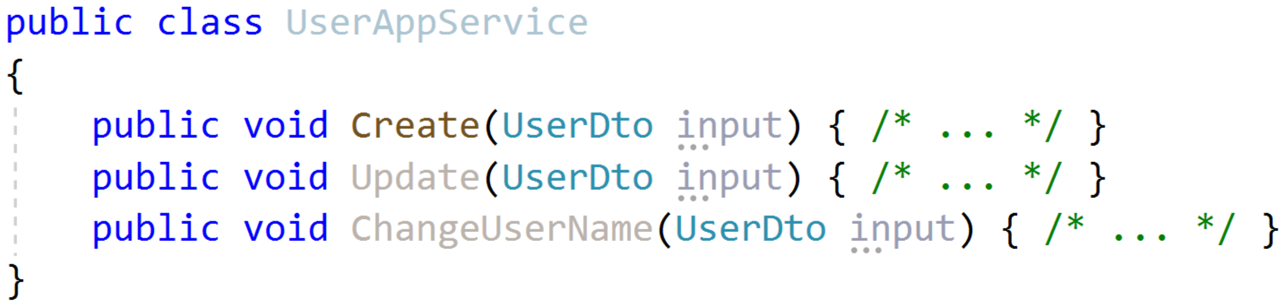
Application Services

Common DTO Principles Best Practices

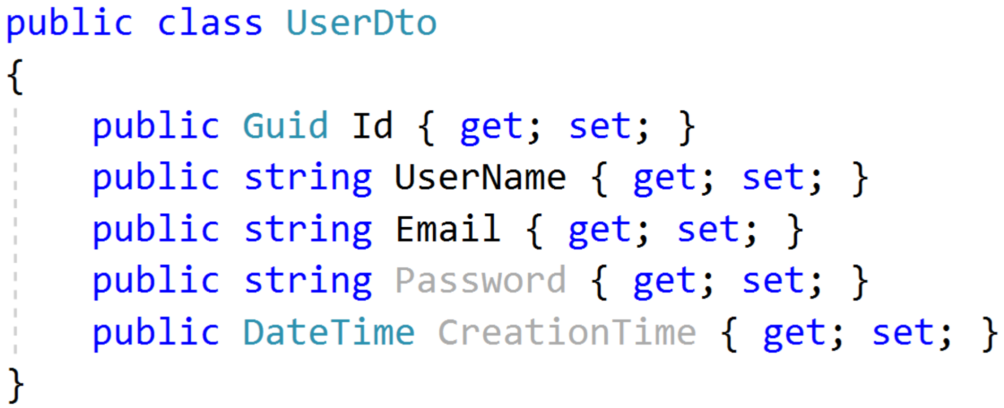
- Should be **serializable**
  - Have a **parameterless** (default) **constructor**
- Should not contain any **business logic**
- Never inherit from **entities**! Never reference to **entities**!

Application Services
Input DTO Best Practices

- Define only the **properties needed** for the use case
- **Do not reuse** same input DTO for multiple use cases (service methods)



```
public class UserAppService
{
    public void Create(UserDto input) { /* ... */ }
    public void Update(UserDto input) { /* ... */ }
    public void ChangeUserName(UserDto input) { /* ... */ }
}
```



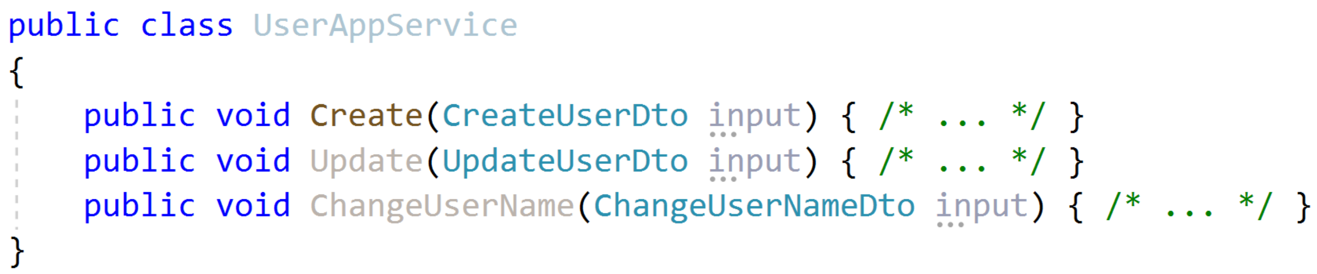
```
public class UserDto
{
    public Guid Id { get; set; }
    public string UserName { get; set; }
    public string Email { get; set; }
    public string Password { get; set; }
    public DateTime CreationTime { get; set; }
}
```

- Id is not used in create! Do not share same DTO for create & update!
- Password is not used in *Update* and *ChangeUserName*!
- CreationTime should not be sent by the client!

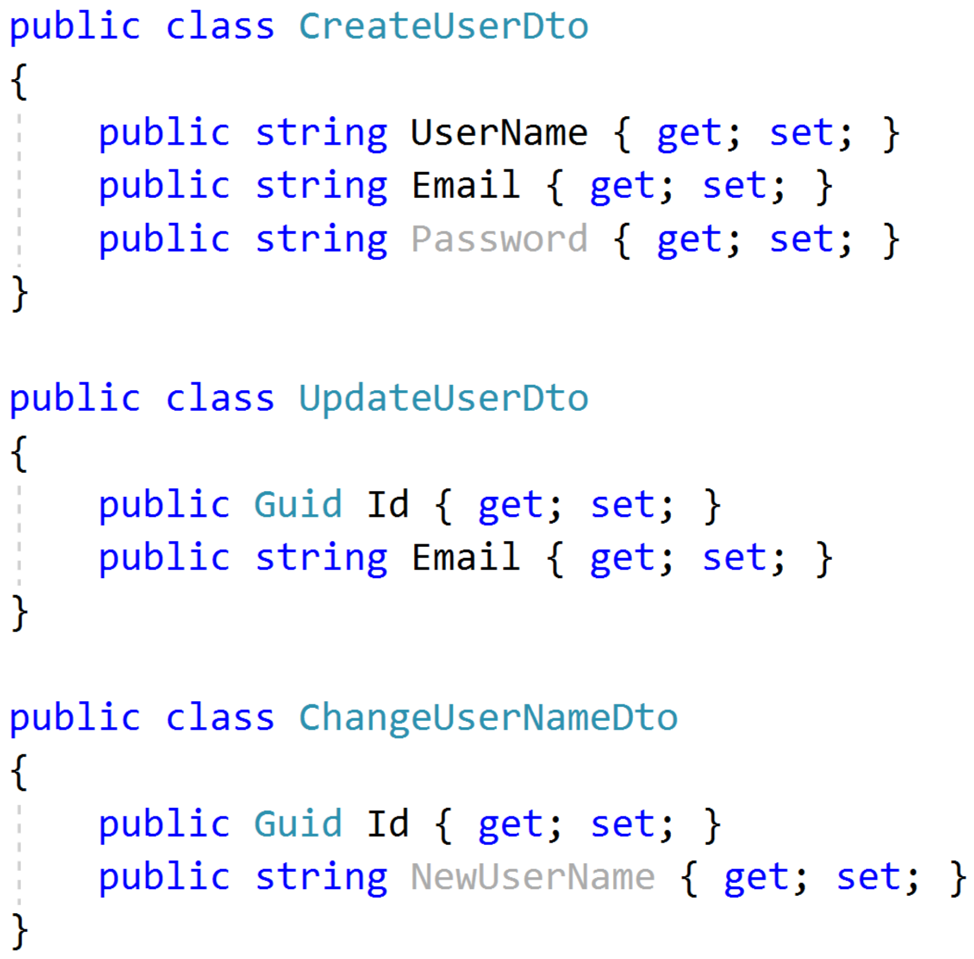
Application Services

Input DTO Best Practices

- Define only the **properties needed** for the use case
- **Do not reuse** same input DTO for multiple use cases (service methods)



```
public class UserAppService
{
    public void Create(CreateUserDto input) { /* ... */ }
    public void Update(UpdateUserDto input) { /* ... */ }
    public void ChangeUserName(ChangeUserNameDto input) { /* ... */ }
}
```

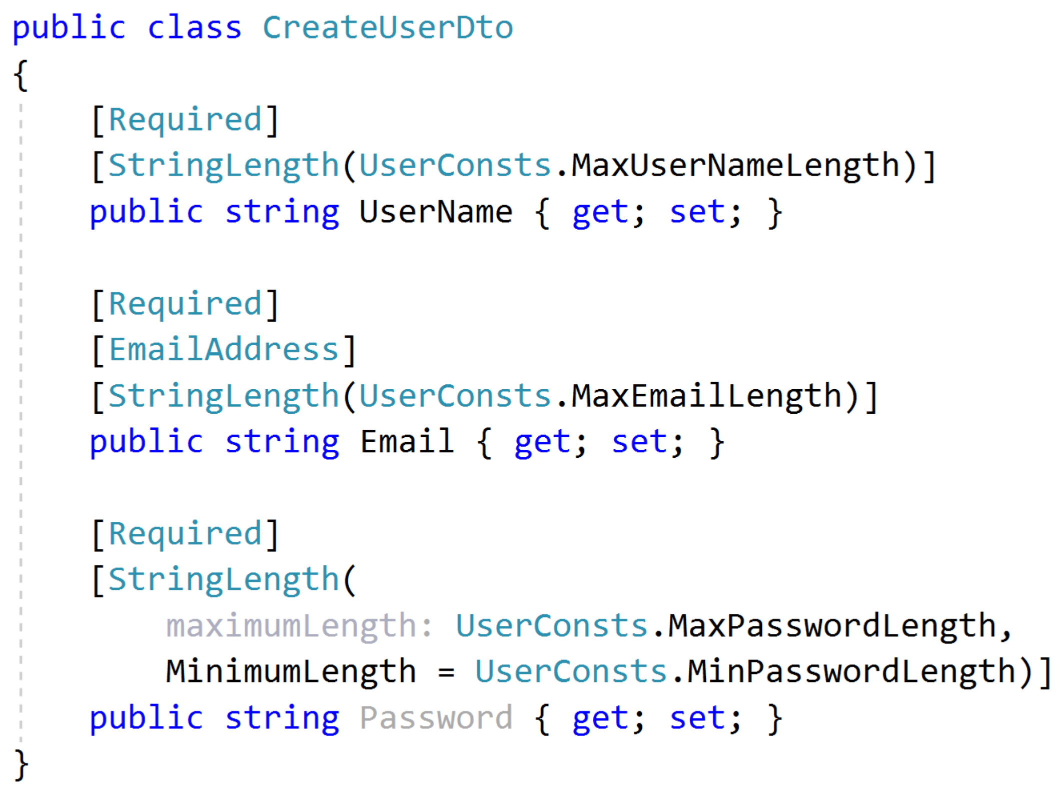


```
public class CreateUserDto
{
    public string UserName { get; set; }
    public string Email { get; set; }
    public string Password { get; set; }
}

public class UpdateUserDto
{
    public Guid Id { get; set; }
    public string Email { get; set; }
}

public class ChangeUserNameDto
{
    public Guid Id { get; set; }
    public string NewUserName { get; set; }
}
```

Application Services

Input DTO Best Practices

- Implement only the formal validation (can use data annotation attributes)
- Don’t include domain validation logic (ex: unique username constraint)

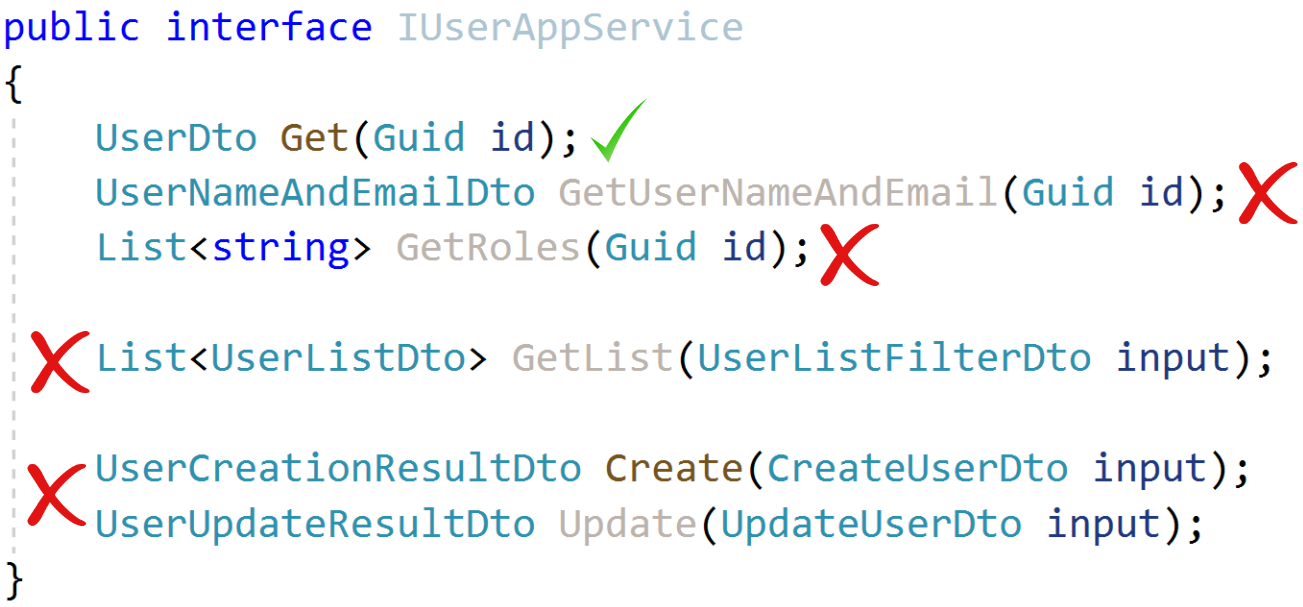


```
public class CreateUserDto
{
    [Required]
    [StringLength(UserConsts.MaxUserNameLength)]
    public string UserName { get; set; }

    [Required]
    [EmailAddress]
    [StringLength(UserConsts.MaxEmailLength)]
    public string Email { get; set; }

    [Required]
    [StringLength(
        maximumLength: UserConsts.MaxPasswordLength,
        MinimumLength = UserConsts.MinPasswordLength)]
    public string Password { get; set; }
}
```

Application Services

Output DTO suggestions

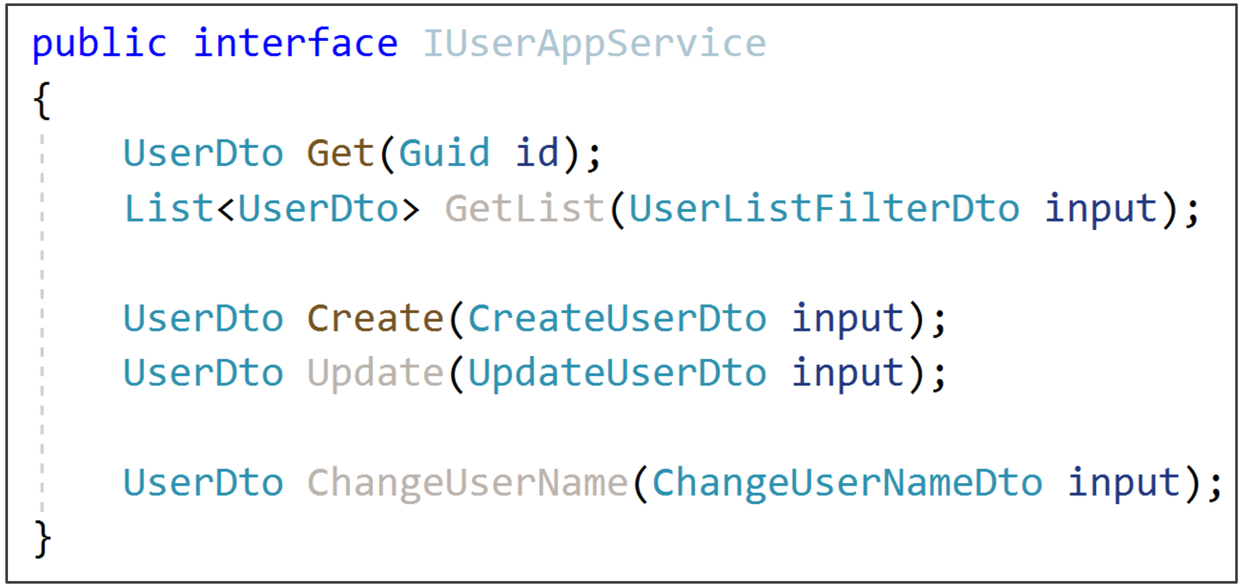
- Keep output **DTO count minimum**. **Reuse** where possible (except input DTOs as output DTO).
- Can contain **more properties** than client needs
- Return the entity DTO from **Create & update** methods.
- Exception: Where **performance** is critical, especially for large result sets.



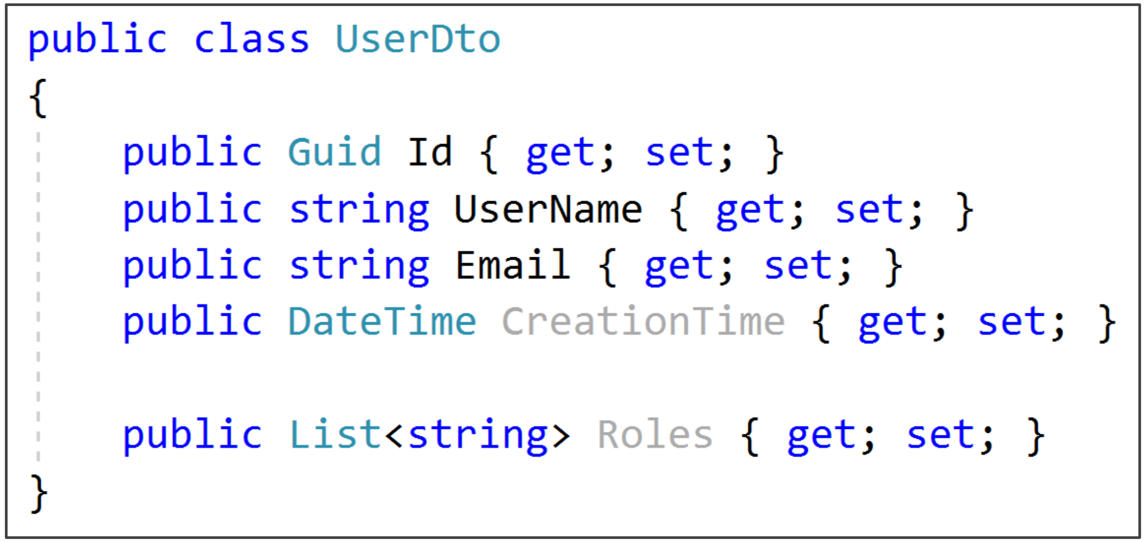
```
public interface IUserAppService
{
    UserDto Get(Guid id); ✓
    UserNameAndEmailDto GetUserNameAndEmail(Guid id); ✗
    List<string> GetRoles(Guid id); ✗

  ✗ List<UserListDto> GetList(UserListFilterDto input);

  ✗ UserCreationResultDto Create(CreateUserDto input);
    UserUpdateResultDto Update(UpdateUserDto input);
}
```

Application Services

Output DTO suggestions



```
public interface IUserAppService
{
    UserDto Get(Guid id);
    List<UserDto> GetList(UserListFilterDto input);

    UserDto Create(CreateUserDto input);
    UserDto Update(UpdateUserDto input);

    UserDto ChangeUserName(ChangeUserNameDto input);
}
```



```
public class UserDto
{
    public Guid Id { get; set; }
    public string UserName { get; set; }
    public string Email { get; set; }
    public DateTime CreationTime { get; set; }

    public List<string> Roles { get; set; }
}
```

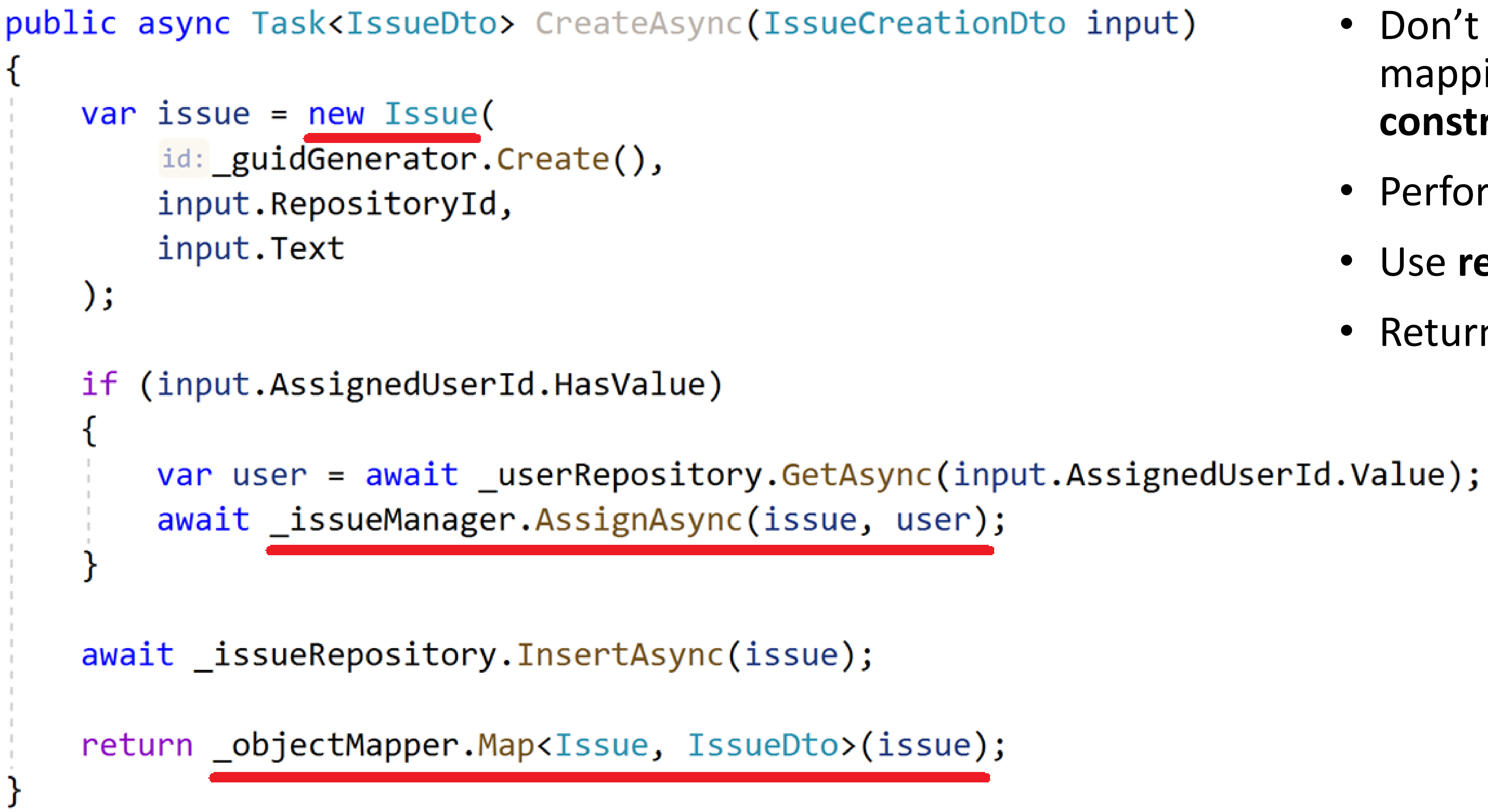
Application Services

Object to Object Mapping

- Use **auto object mapping** libraries (but, carefully – enable configuration validation)
- **Do not map** input DTOs to entities.
- **Map** entities to output DTOs

Application Services

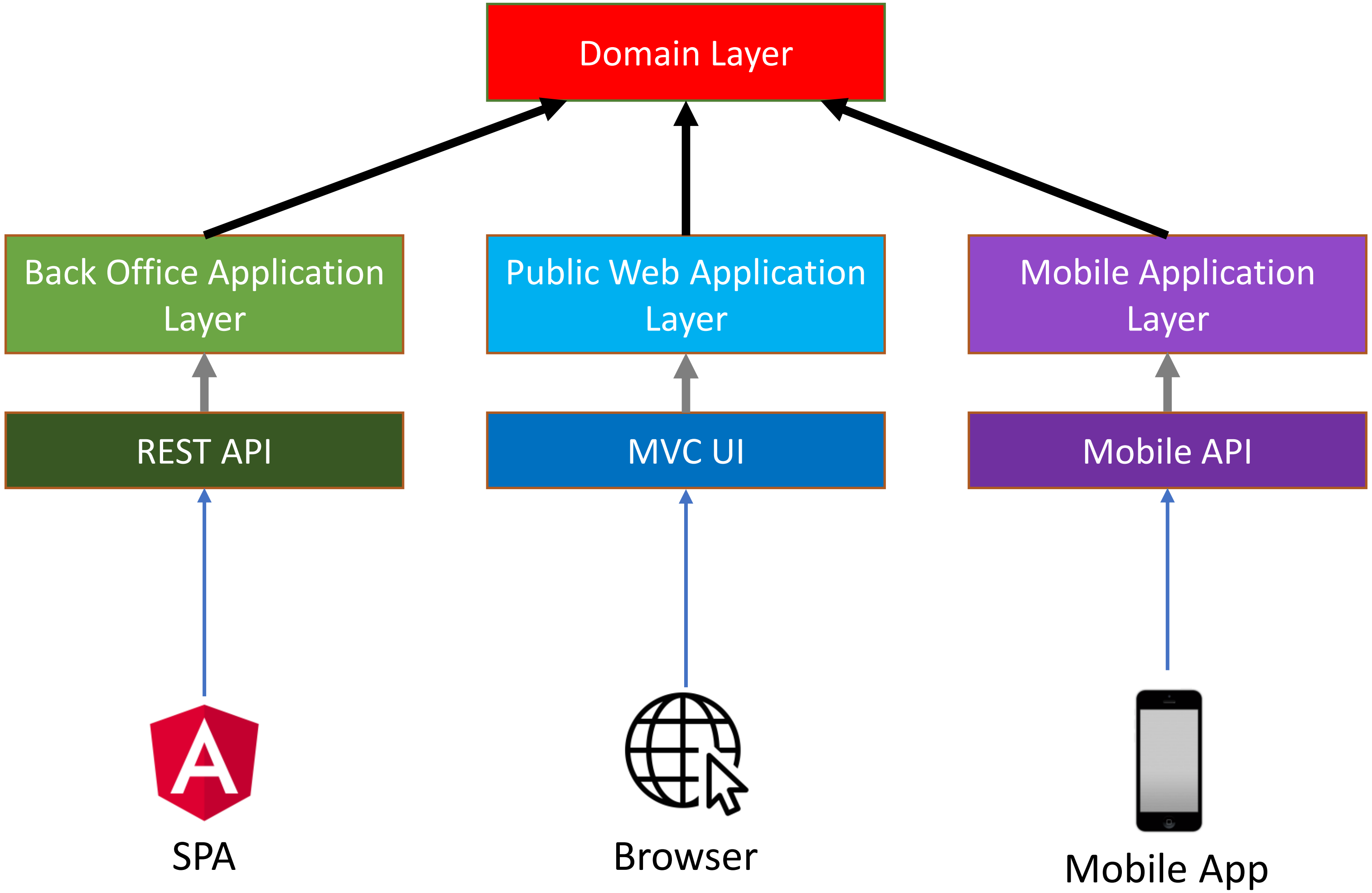
Example: Entity Creation & DTO Mapping



```
public async Task<IssueDto> CreateAsync(IssueCreationDto input)
{
    var issue = new Issue(
        id: _guidGenerator.Create(),
        input.RepositoryId,
        input.Text
    );

    if (input.AssignedUserId.HasValue)
    {
        var user = await _userRepository.GetAsync(input.AssignedUserId.Value);
        await _issueManager.AssignAsync(issue, user);
    }

    await _issueRepository.InsertAsync(issue);

    return _objectMapper.Map<Issue, IssueDto>(issue);
}
```

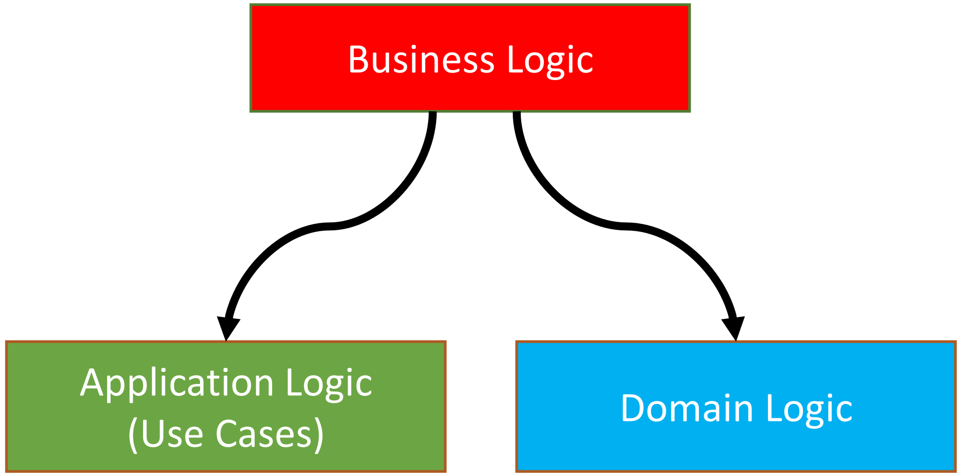
- Don’t use DTO to entity auto-mapping, use the entity **constructor**.
- Perform additional **domain actions**.
- Use **repository** to insert the entity.
- Return DTO using **auto-mapping**.

Multiple Application Layers



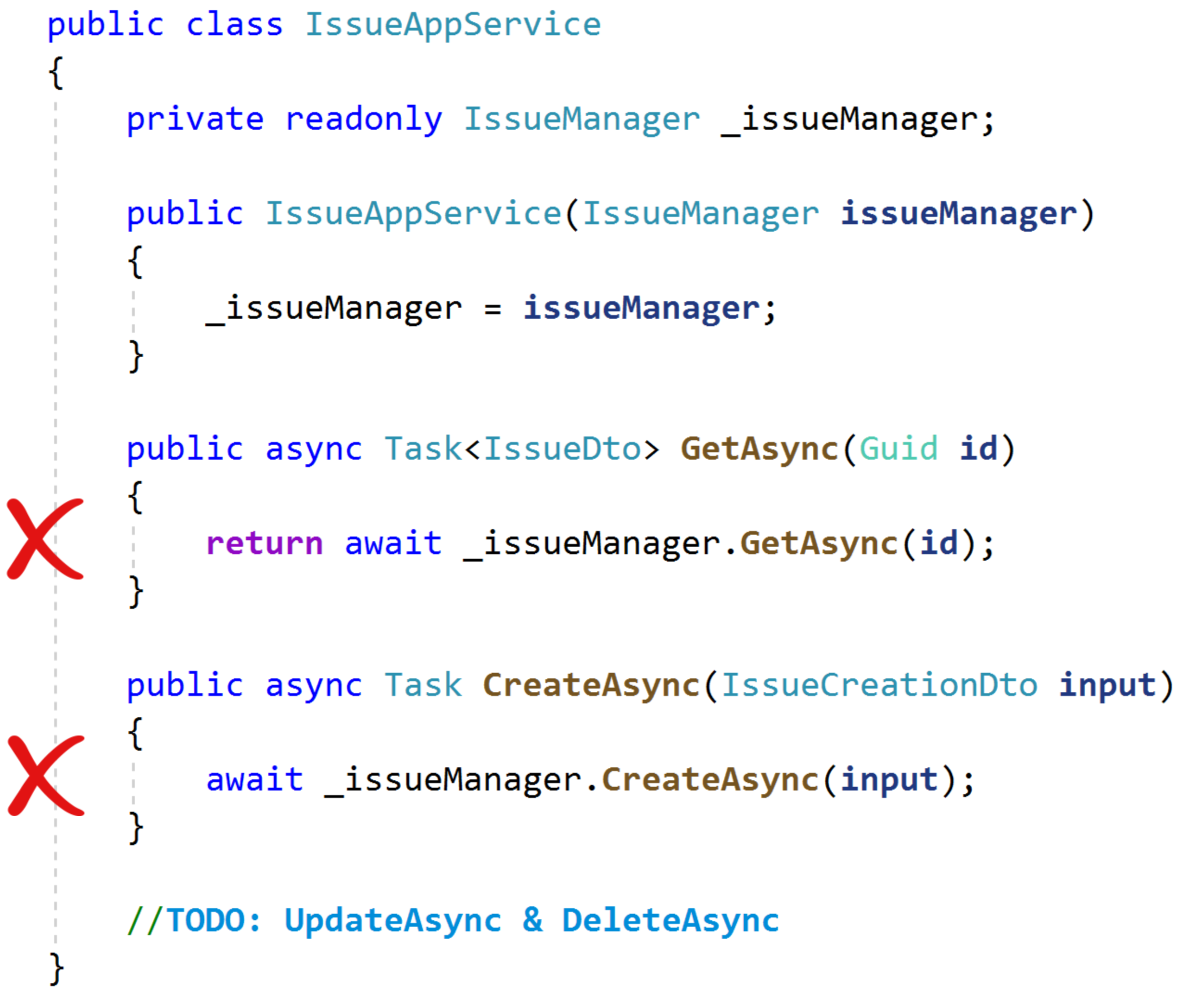
- Create **separate application layers** for each application type.
- Use a **single domain layer** to share the core domain logic.

Application Logic vs Domain Logic



Examples..?

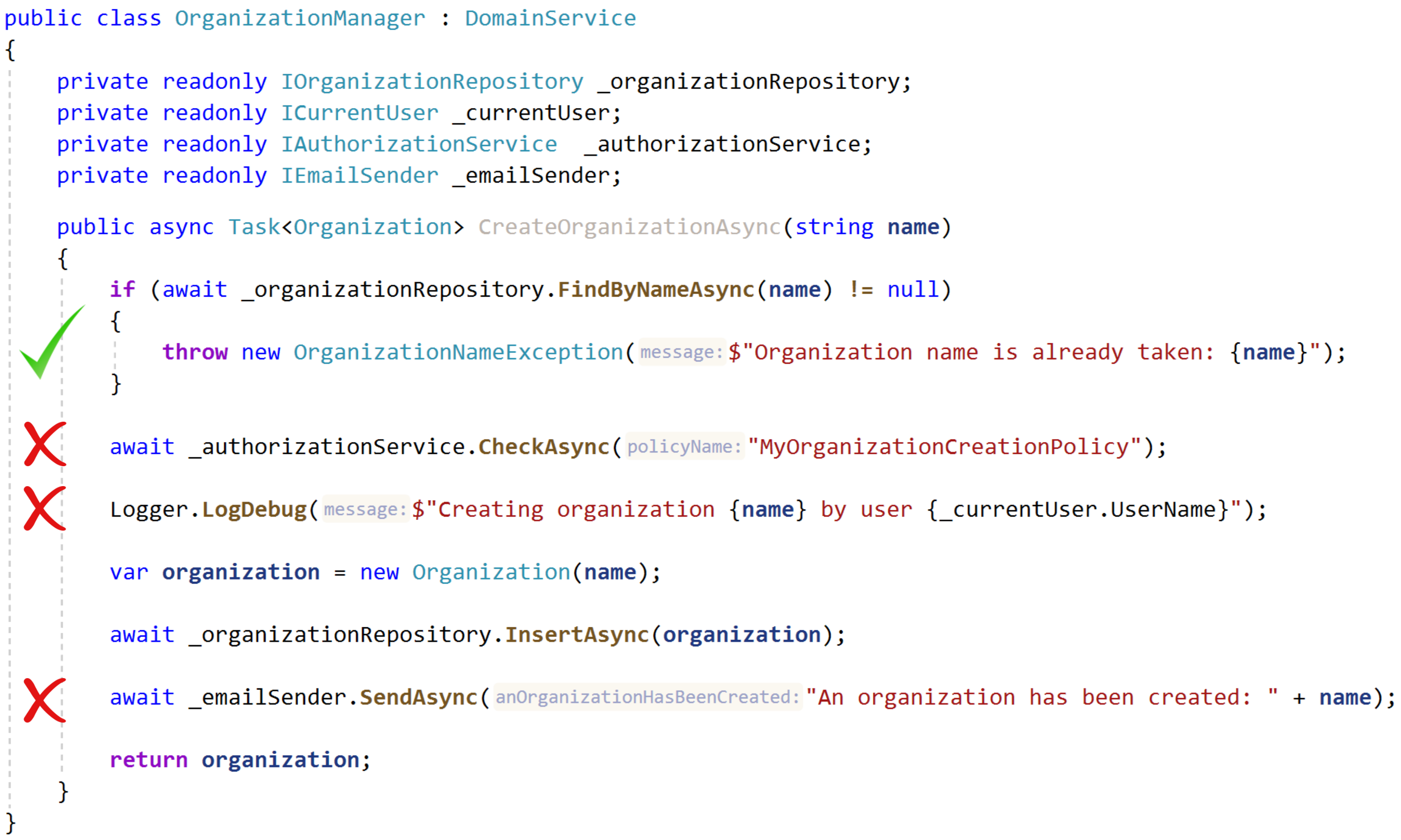
Application Logic vs Domain Logic



```
public class IssueAppService
{
    private readonly IssueManager _issueManager;

    public IssueAppService(IssueManager issueManager)
    {
        _issueManager = issueManager;
    }

    public async Task<IssueDto> GetAsync(Guid id)
    {
        return await _issueManager.GetAsync(id);
    }

    public async Task CreateAsync(IssueCreationDto input)
    {
        await _issueManager.CreateAsync(input);
    }

    //TODO: UpdateAsync & DeleteAsync
}
```

- **Don’t** create domain services to perform simple **CRUD** operations!
    - Use **Repositories** in the application services.
- **Never** pass **DTOs** to or return **DTOs** from domain services!
    - DTOs should be in the **application layer**.

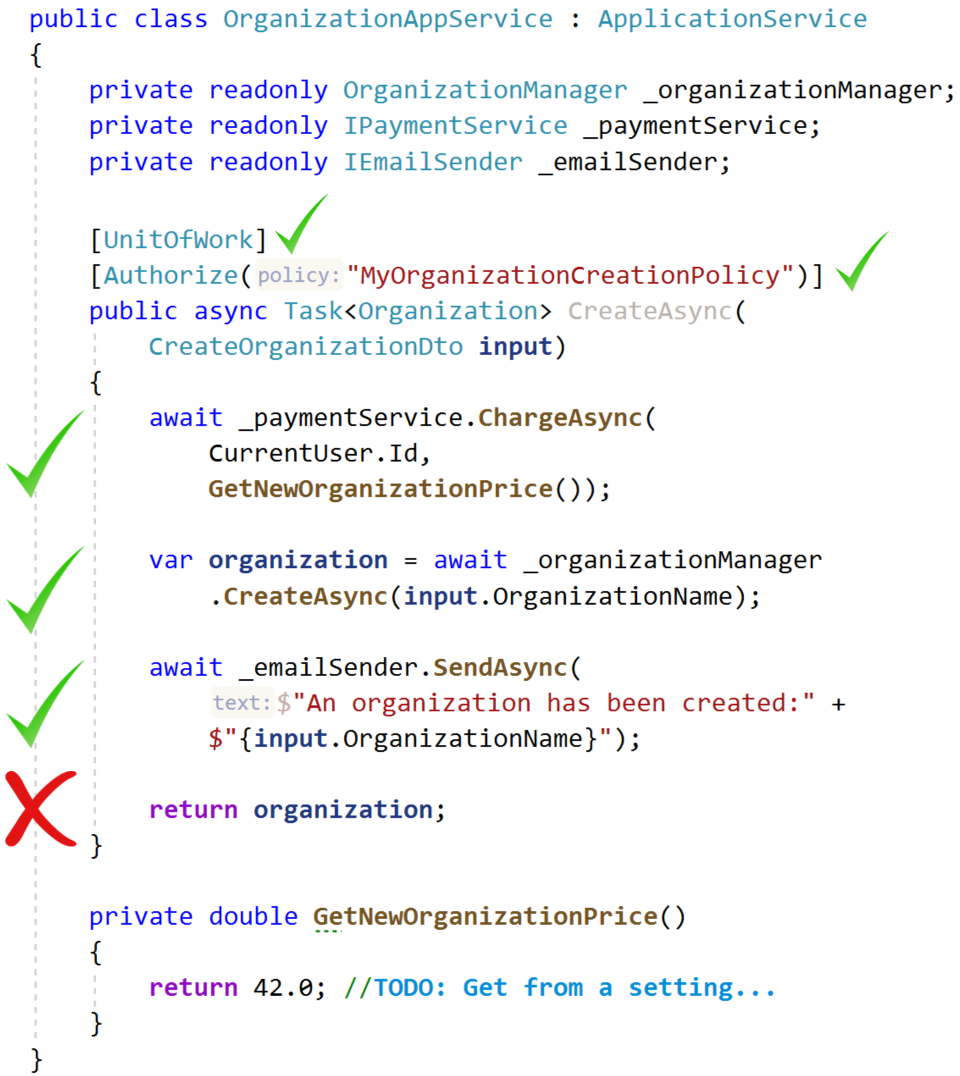
Application Logic vs Domain Logic



```
public class OrganizationManager : DomainService
{
    private readonly IOrganizationRepository _organizationRepository;
    private readonly ICurrentUser _currentUser;
    private readonly IAuthorizationService  _authorizationService;
    private readonly IEmailSender _emailSender;

    public async Task<Organization> CreateOrganizationAsync(string name)
    {
        if (await _organizationRepository.FindByNameAsync(name) != null)
        {
 ✓          throw new OrganizationNameException(message: $"Organization name is already taken: {name}");
        }

 ✗      await _authorizationService.CheckAsync(policyName: "MyOrganizationCreationPolicy");

 ✗      Logger.LogDebug(message: $"Creating organization {name} by user {_currentUser.UserName}");

        var organization = new Organization(name);

        await _organizationRepository.InsertAsync(organization);

 ✗      await _emailSender.SendAsync(anOrganizationHasBeenCreated: "An organization has been created: " + name);

        return organization;
    }
}
```

- Domain service **should check** duplicate organization name.
- Domain service **doesn't perform authorization**!
  - Do in the **application layer**!
- Domain service must **not depend** on the **current user**!
  - Do in the **application/UI/API layer**!
- Domain service **must not** send **email** that is not related to the actual business!
  - Do in the application layer or implement via domain events.

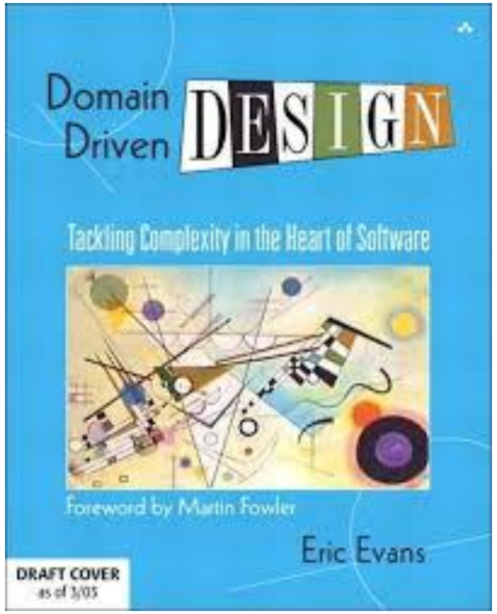
Application Logic vs Domain Logic



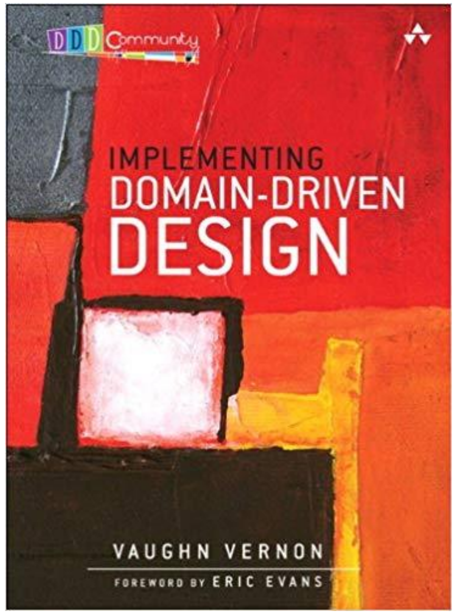
```
public class OrganizationAppService : ApplicationService
{
    private readonly OrganizationManager _organizationManager;
    private readonly IPaymentService _paymentService;
    private readonly IEmailSender _emailSender;

    [UnitOfWork] ✓
    [Authorize(policy: "MyOrganizationCreationPolicy")] ✓
    public async Task<Organization> CreateAsync(
        CreateOrganizationDto input)
    {
        await _paymentService.ChargeAsync( ✓
            CurrentUser.Id,
            GetNewOrganizationPrice());

        var organization = await _organizationManager ✓
            .CreateAsync(input.OrganizationName);

        await _emailSender.SendAsync( ✓
            text: $"An organization has been created:" +
            $"{input.OrganizationName}");

        return organization; ✗
    }

    private double GetNewOrganizationPrice()
    {
        return 42.0; //TODO: Get from a setting...
    }
}
```

- Application service methods **should be** a **unit of work** (transactional) if contains more than one database operations.
- **Authorization is done** in the application layer.
- **Payment** (infrastructure service) and **Organization creation** should not be combined in the domain layer. **Should be orchestrated** by the application layer.
- **Email sending can be** done in the application service.
- **Do not** return **entities** from application services!
- **Why not** moving **payment** logic inside the **domain service**?

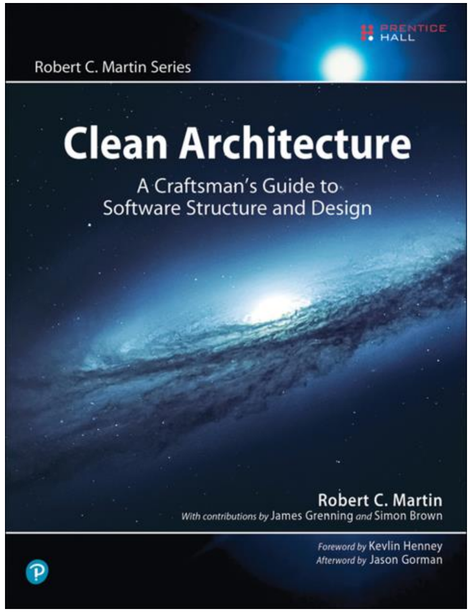
Recommended Books


Domain Driven DESIGN
Tackling Complexity in the Heart of Software
Foreword by Martin Fowler
Eric Evans
DRAFT COVER
as of 3/03

Domain Driven Design
Eric Evans

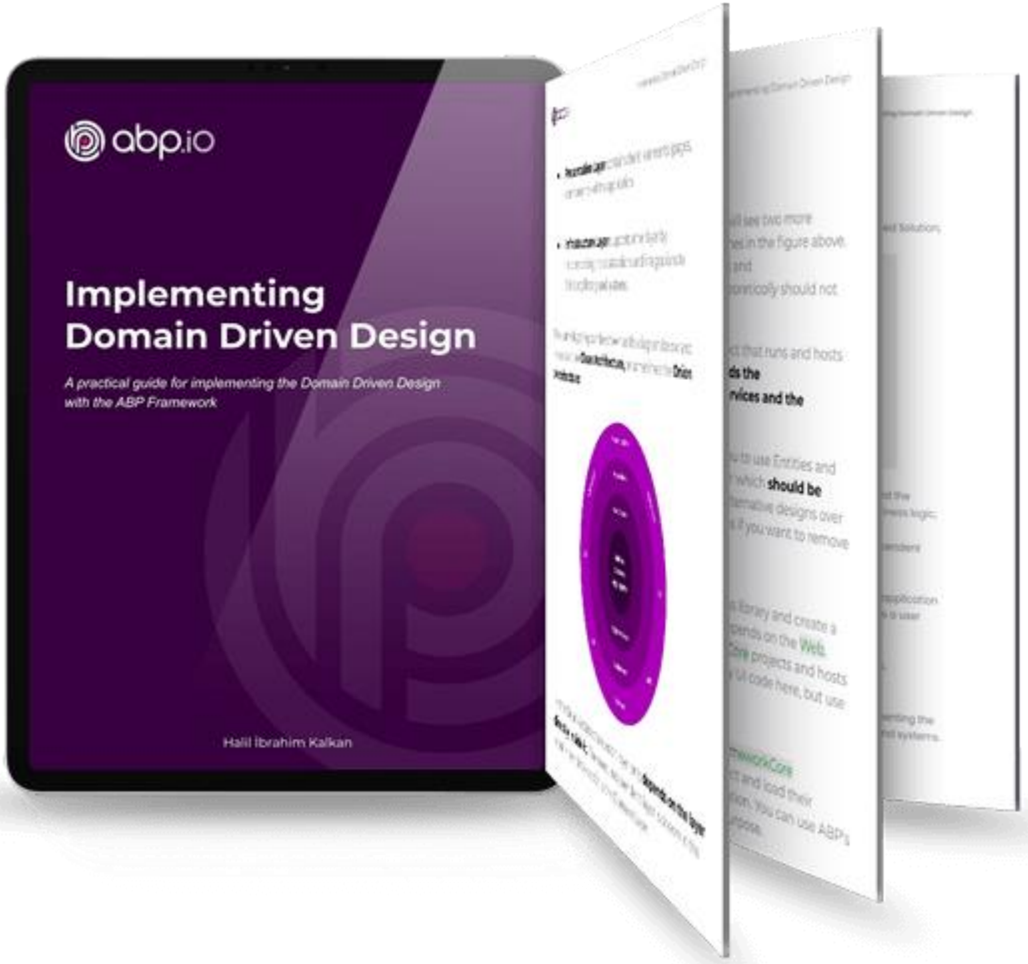


Implementing
Domain Driven Design
Vaughn Vernon


PRENTICE HALL
Robert C. Martin Series
Clean Architecture
A Craftsman's Guide to
Software Structure and Design
Robert C. Martin
With contributions by James Grenning and Simon Brown
Foreword by Kevlin Henney
Afterword by Jason Gorman

Clean Architecture
Robert C. Martin

The **Reference** Book!


abp.io
Implementing Domain Driven Design
A practical guide for implementing the Domain Driven Design with the ABP Framework
Halil İbrahim Kalkan

DOWNLOAD THE E-BOOK FOR FREE

THANKS FOR WATCHING

Download the Presentation

Questions..?