

Specification Property Framework

This document touches on the following three trait aspects of the specified software.

Results

= The *results* the user wants to get from the software.

Analysis is concerned with describing the interface of the *results*.

Process

= The *process* of producing the results.

Design is concerned with describing the interface of the *process*.

Platform

= The *environment* in which the software is used.

Architecture is concerned with describing the platform interface of the software.

Implementation

= Coding specifics of the software.

Engineering is concerned with describing the approach to the implementation of the software.

Analysis Property Framework

Property Framework aims to reduce the difficulty of engineering programs on the .Net/C#|VB platform by eliminating a considerable amount of code in business logic.

Goals

- Provide an all-declarative experience for logic

- Provide asynchronous experience without locks

- Keep constructs as compact as possible

- “You don’t use it – you don’t have it”

 - Make it easy to integrate with any other technology / enable a feature

 - Keep none of these technologies / features in the core

- Make it out-of-the-box usable on a wide range of versions of the engineering platform

- Keep the platform-to-platform difference of the codebase below 1%

- Keep the test coverage of the components 100%

- Engineer it for testability of the client code

Present State

Standard C#/VB supports the concept of *property*. We use *properties* in a class to define how the respective private states should be read and set in its instances.

Today, all data binding mechanisms in C#/VB rely on the use of properties in the presentation logic layer. That means that mutable states rather than sequences of immutable values are used for data binding. To support such model of operation in asynchronous scenarios without locks we have to introduce mechanisms for brokering state changes.

Validation and change notification are only enabled for members of objects if their classes implement certain interfaces.

The above facts make designing presentation logic classes a challenging task. The model of one class with n traditional properties is be hard to implement with brokering because there is no single model for brokering change of multiple states. In addition, inheriting support for validation and state change may affect the code of each property. Migrating to another validation platform will require using different events or adapting interfaces of the base classes.

Approach

Assume we have a class that has a single state, brokers the change of this state, notifies when the state is set, does the validation, and implements any other logic of handling the state. Instances of such classes we will call *self-reliant properties*.

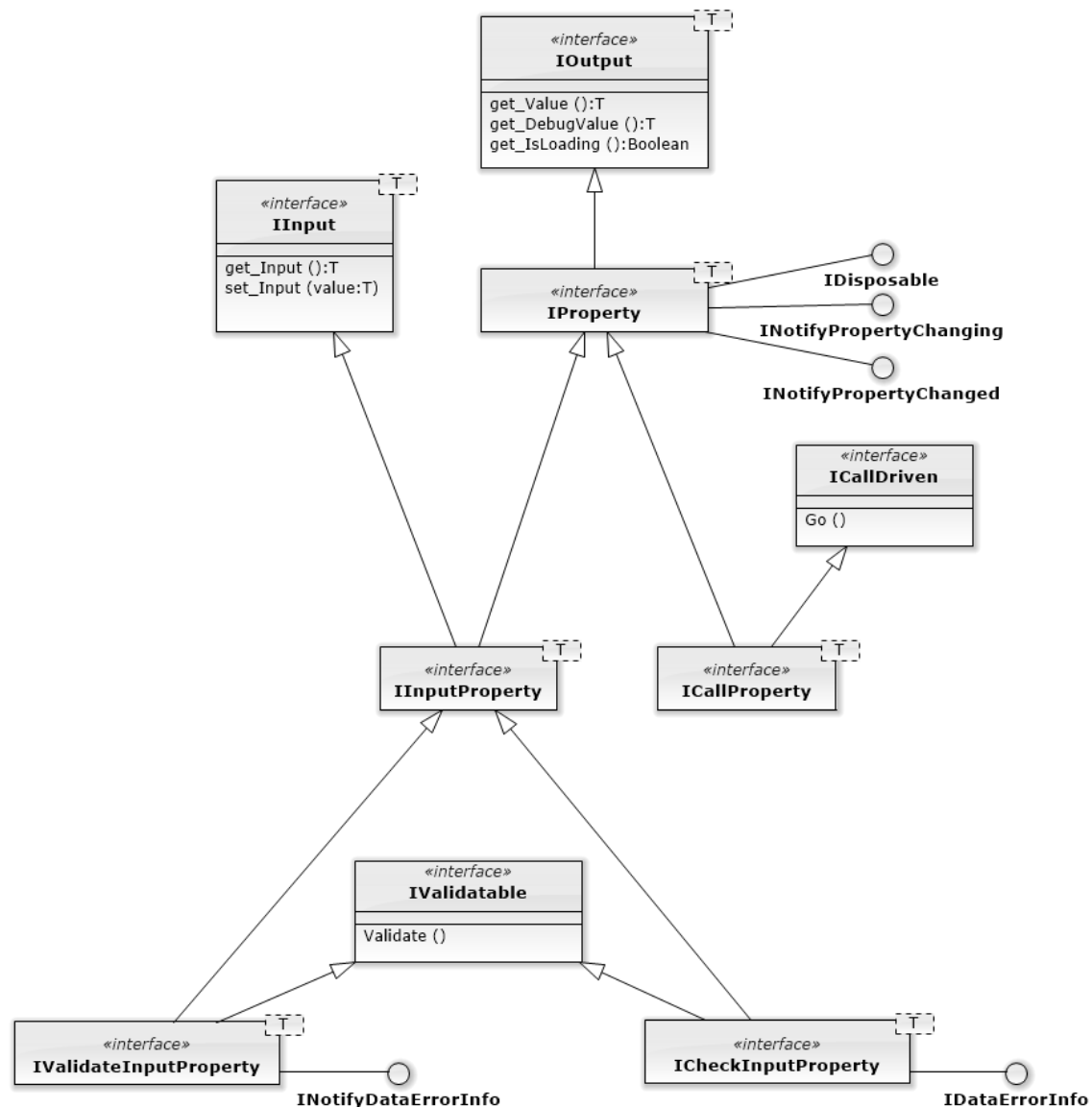
Then, our presentation logic classes will only need to contain n read-only traditional properties each redirecting the binding mechanism to the respective *self-reliant properties*.

$$\text{Class} \xrightarrow{\text{Property.get}} \text{SelfReliantProperty}$$

Property Framework provides means for composition and use of self-reliant properties.

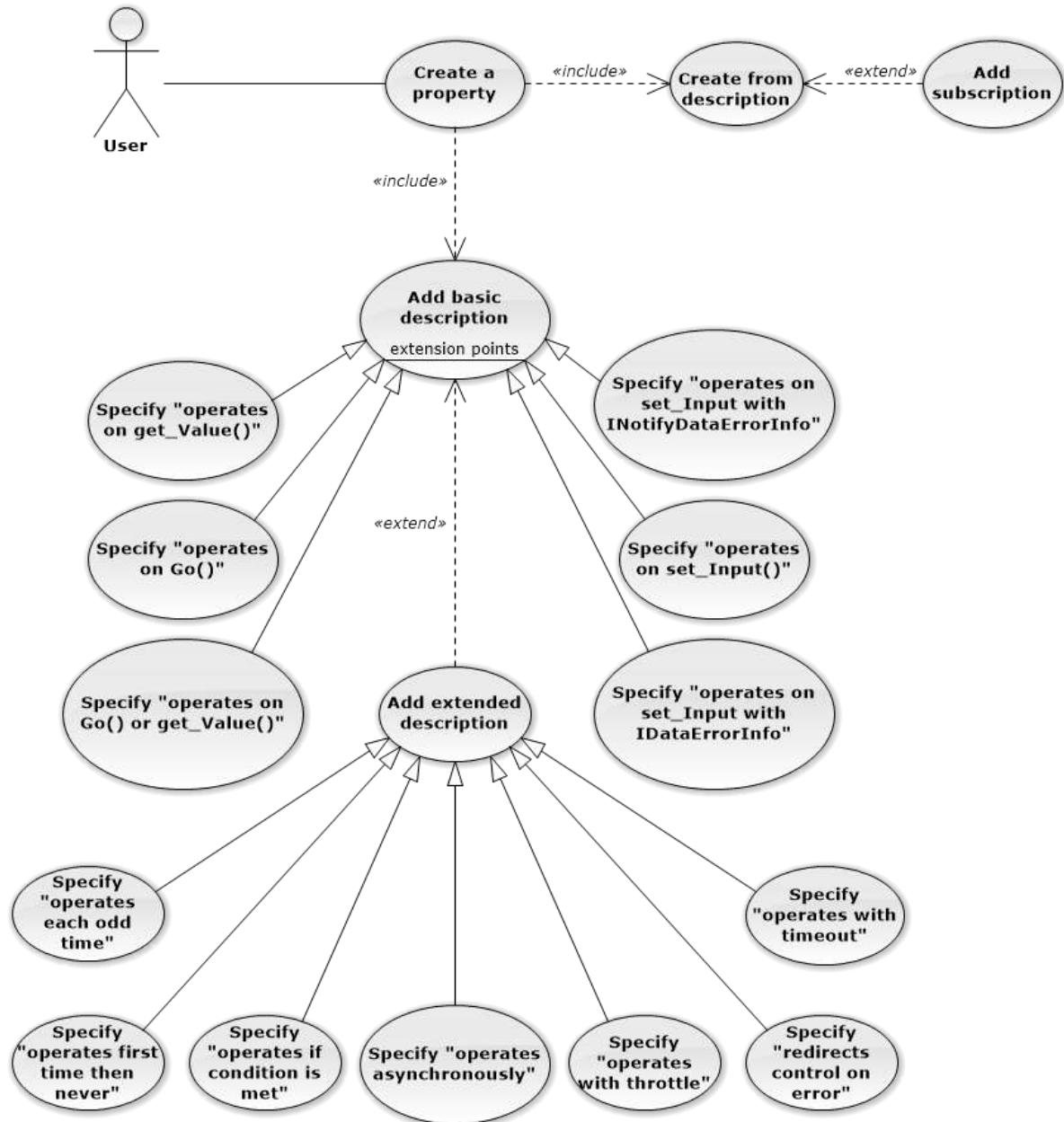
Interface

The following diagram shows the desired interfaces of PF-properties.



Operation

The following diagram shows the requirements for process and operation for PF-properties.



Additional Design Requirements

A PF-property must only change its state using an injected broker object, that we call a *reloader*. Reloaders should be call-driven, i.e. their operation should be triggered using a parameterless *Go()* method only. Property descriptions are composed and provide the only means for instantiating properties.

Design

This section explores certain concepts that allow specifying the design of an application in a concise and relatively generalized way.

Engineering Platform

An *engineering platform* is a set of tools used for programming i.e. creating and running programs.

Applicability

Let us assume an engineering platform supports the concept of a datum or an *instance* or a portion of data usable as a single unit, and it has a concept for *static categorization* of such instances. Then, if the platform supports the relation of equality for instances, we can use the following definitions and concepts to design applications on this platform.

Type

Type T is a set of all *instances t* (instances of type T) that belong to the same *category*.

Type Transition

A *type transition* with *identifier τ* from *type T_s* to *type T_r* is an abstract function $f_{\tau, T_s, T_r} : T_s \rightarrow T_r$ for which the relation of *equality* is defined as follows.

Two *type transitions* are *equal* if and only if their *identifiers* and *source types* match.

$$f_{\tau, T_s, T_r} = f_{\tau', T'_s, T'_r} \Leftrightarrow T_s = T'_s \wedge \tau = \tau'$$

If two type transitions are equal then their *result types* match.

$$f_{\tau, T_s, T_r} = f_{\tau', T'_s, T'_r} \Rightarrow T_r = T'_r$$

Type Transition Notation

If f_{τ, T_s, T_r} denotes a *type transition*, then we say the transition has the *identifier* τ , the *source* type T_s and the *result* type T_r .

To denote a type transition we can use the *transition operator* $\xrightarrow{\tau}$, thus, presenting f_{τ, T_s, T_r} as $T_s \xrightarrow{\tau} T_r$.

To define a composite transition $f_{\tau_2, T_r, T'_r} \circ f_{\tau_1, T_s, T_r}$ (meaning “ f_{τ_2, T_r, T'_r} after f_{τ_1, T_s, T_r} ”) we can conveniently use the semicolon notation $f_{\tau_1, T_s, T_r}; f_{\tau_2, T_r, T'_r}$ which presented in the *transition operator* notation $T_s \xrightarrow{\tau_1} T_r; T_r \xrightarrow{\tau_2} T'_r$ is finally reduced to $T_s \xrightarrow{\tau_1} T_r \xrightarrow{\tau_2} T'_r$ or $T_s \xrightarrow{\tau_1; \tau_2} T'_r$ depending on the required level of detail.

Solution

Solution U is a set of all type transitions specified for the designed system.

Scope

Scope S is a subset of *solution* U .

Two *scopes* are *mutually exclusive* if they intersect.

Language

Language L_U is a set of identifiers of all type transitions in *solution* U . Members of L_U we will call *words*.

$$\exists f_{\tau, T_s, T_r} \in U \Leftrightarrow \tau \in L_U$$

Word

A *word* is a member of the language L_U .

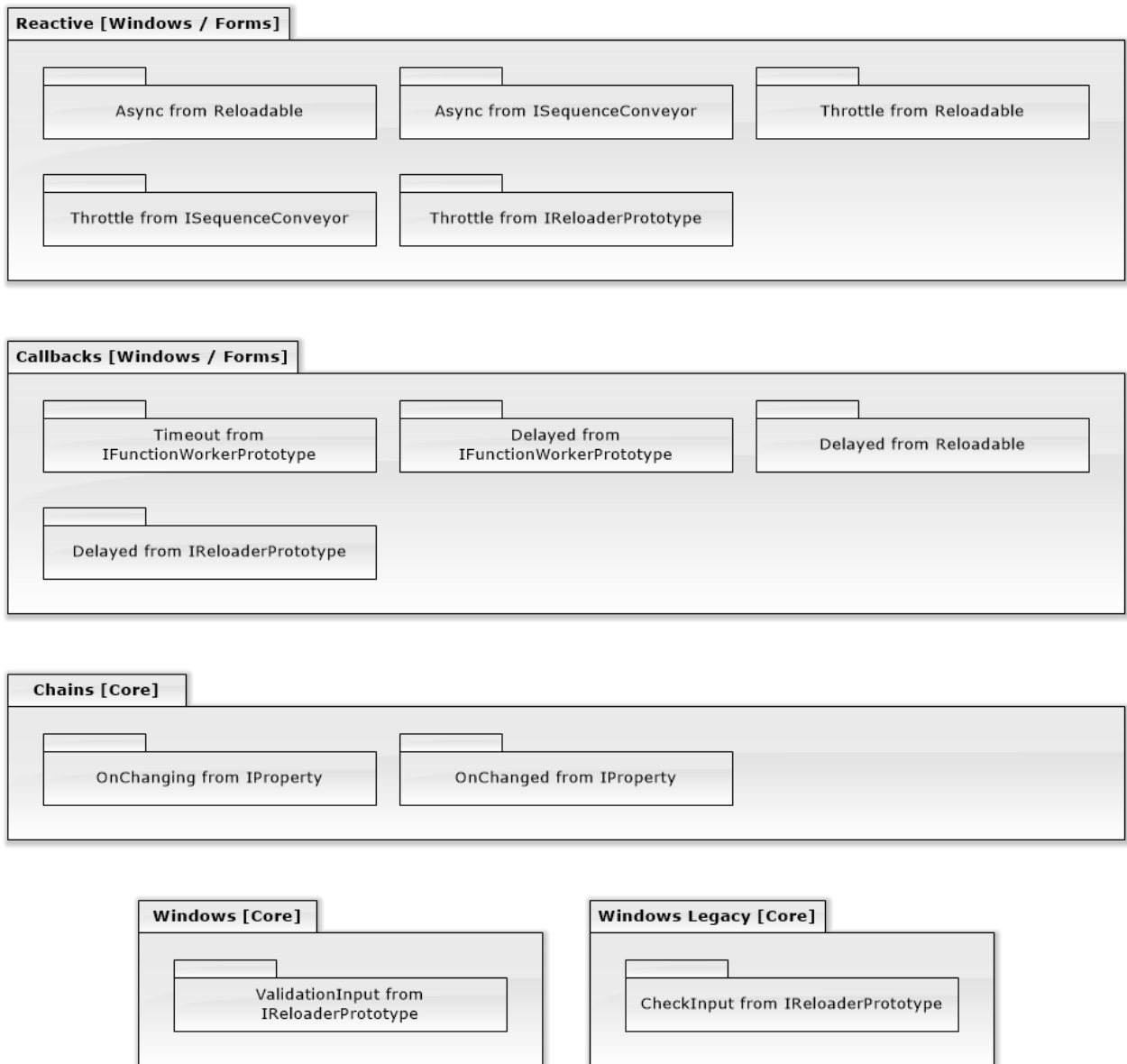
Design Property Framework

In this section, we will define a solution for Property Framework according to the analysis-defined requirements.

$$\begin{aligned} S = \{ & \text{Reloadable}_T \xrightarrow{\text{Each}} \text{IReloaderPrototype}_T, \text{Reloadable}_T \xrightarrow{\text{Conditional}} \text{IReloaderPrototype}_T, \\ & \text{Reloadable}_T \xrightarrow{\text{First}} \text{IReloaderPrototype}_T, \text{Reloadable}_T \xrightarrow{\text{Odd}} \text{IReloaderPrototype}_T, \\ & \text{Reloadable}_T \xrightarrow{\text{Delayed}} \text{IReloaderPrototype}_T, \text{Reloadable}_T \xrightarrow{\text{Throttle}} \text{IReloaderPrototype}_T, \\ & \text{Reloadable}_T \xrightarrow{\text{Filtered}} \text{IReloaderPrototype}_T, \text{Reloadable}_T \xrightarrow{\text{Reactive}} \text{ISequenceConveyor}_T, \\ & \text{Reloadable}_T \xrightarrow{\text{Async}} \text{ISequenceConveyor}_T, \text{Reloadable}_T \xrightarrow{\text{Worker}} \text{IFunctionWorkerPrototype}_T, \\ & \text{IFunctionWorkerPrototype}_T \xrightarrow{\text{Catch}} \text{IFunctionWorkerPrototype}_T, \\ & \text{IFunctionWorkerPrototype}_T \xrightarrow{\text{Retry}} \text{IFunctionWorkerPrototype}_T, \\ & \text{IFunctionWorkerPrototype}_T \xrightarrow{\text{Timeout}} \text{IFunctionWorkerPrototype}_T, \\ & \text{IFunctionWorkerPrototype}_T \xrightarrow{\text{Each}} \text{IReloaderPrototype}_T, \\ & \text{IFunctionWorkerPrototype}_T \xrightarrow{\text{First}} \text{IReloaderPrototype}_T, \\ & \text{IFunctionWorkerPrototype}_T \xrightarrow{\text{Odd}} \text{IReloaderPrototype}_T, \\ & \text{IFunctionWorkerPrototype}_T \xrightarrow{\text{Conditional}} \text{IReloaderPrototype}_T, \\ & \text{IFunctionWorkerPrototype}_T \xrightarrow{\text{Delayed}} \text{IReloaderPrototype}_T, \\ & \text{ISequenceConveyor}_T \xrightarrow{\text{Modify}} \text{ISequenceConveyor}_T, \text{ISequenceConveyor}_T \xrightarrow{\text{Catch}} \text{ISequenceConveyor}_T, \\ & \text{ISequenceConveyor}_T \xrightarrow{\text{Retry}} \text{ISequenceConveyor}_T, \text{ISequenceConveyor}_T \xrightarrow{\text{Timeout}} \text{ISequenceConveyor}_T, \\ & \text{ISequenceConveyor}_T \xrightarrow{\text{Async}} \text{ISequenceConveyor}_T, \text{ISequenceConveyor}_T \xrightarrow{\text{Each}} \text{IReloaderPrototype}_T, \\ & \text{ISequenceConveyor}_T \xrightarrow{\text{First}} \text{IReloaderPrototype}_T, \text{ISequenceConveyor}_T \xrightarrow{\text{Odd}} \text{IReloaderPrototype}_T, \\ & \text{ISequenceConveyor}_T \xrightarrow{\text{Throttle}} \text{IReloaderPrototype}_T, \text{ISequenceConveyor}_T \xrightarrow{\text{Filtered}} \text{IReloaderPrototype}_T, \\ & \text{ISequenceConveyor}_T \xrightarrow{\text{Conditional}} \text{IReloaderPrototype}_T, \\ & \text{IReloaderPrototype}_T \xrightarrow{\text{First}} \text{IReloaderPrototype}_T, \text{IReloaderPrototype}_T \xrightarrow{\text{Odd}} \text{IReloaderPrototype}_T, \\ & \text{IReloaderPrototype}_T \xrightarrow{\text{Conditional}} \text{IReloaderPrototype}_T, \\ & \text{IReloaderPrototype}_T \xrightarrow{\text{Filtered}} \text{IReloaderPrototype}_T, \\ & \text{IReloaderPrototype}_T \xrightarrow{\text{Delayed}} \text{IReloaderPrototype}_T, \text{IReloaderPrototype}_T \xrightarrow{\text{Throttle}} \text{IReloaderPrototype}_T, \\ & \text{IReloaderPrototype}_T \xrightarrow{\text{Input}} \text{IInputPropertyPrototype}_T, \text{IReloaderPrototype}_T \xrightarrow{\text{Get}} \text{IPropertyPrototype}_T, \\ & \text{IReloaderPrototype}_T \xrightarrow{\text{Call}} \text{ICallPropertyPrototype}_T, \\ & \text{IReloaderPrototype}_T \xrightarrow{\text{CallGet}} \text{ICallPropertyPrototype}_T, \\ & \text{IReloaderPrototype}_T \xrightarrow{\text{CheckInput}} \text{ICheckInputPropertyPrototype}_T, \end{aligned}$$

$$\begin{aligned}
I\text{ReloaderPrototype}_T &\xrightarrow{\text{ValidationInput}} I\text{ValidationInputPropertyPrototype}_T, \\
I\text{FunctionWorkerPrototype}_T &\xrightarrow{\text{Create}} I\text{FunctionWorker}_T, I\text{SequenceConveyor}_T \xrightarrow{\text{Create}} I\text{Observable}_T, \\
I\text{ReloaderPrototype}_T &\xrightarrow{\text{Create}} I\text{Reloader}_T, I\text{InputPropertyPrototype}_T \xrightarrow{\text{Create}} I\text{InputProperty}_T, \\
I\text{ValidationInputPropertyPrototype}_T &\xrightarrow{\text{Create}} I\text{ValidationInputProperty}_T, \\
I\text{CheckInputPropertyPrototype}_T &\xrightarrow{\text{Create}} I\text{CheckInputProperty}_T, \\
I\text{PropertyPrototype}_T &\xrightarrow{\text{Create}} I\text{Property}_T, I\text{CallPropertyPrototype}_T \xrightarrow{\text{Create}} I\text{CallProperty}_T\}
\end{aligned}$$

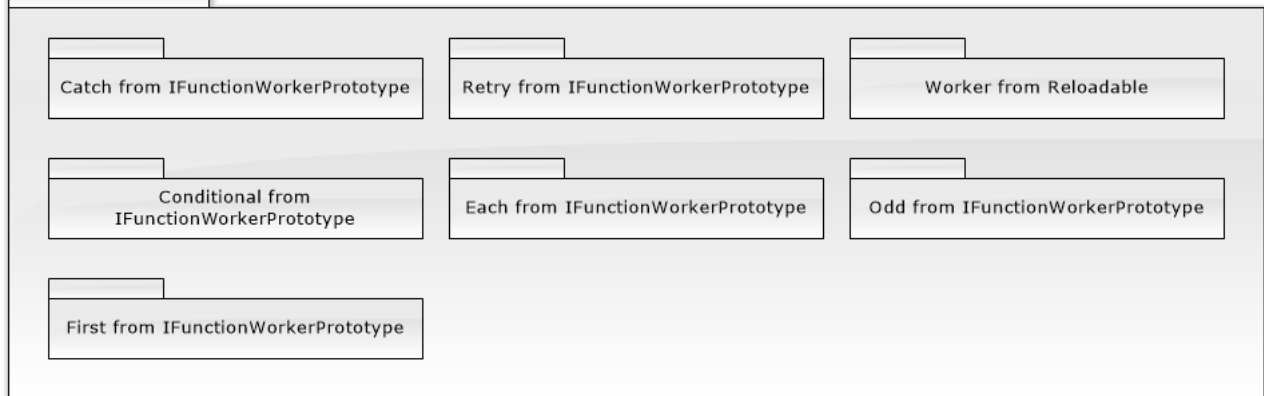
The following package diagram shows the scopes defined for type transitions. The $\xrightarrow{\text{Create}}$ transitions are not shown on the diagrams and they are expected to be available wherever the respective source prototype types are.



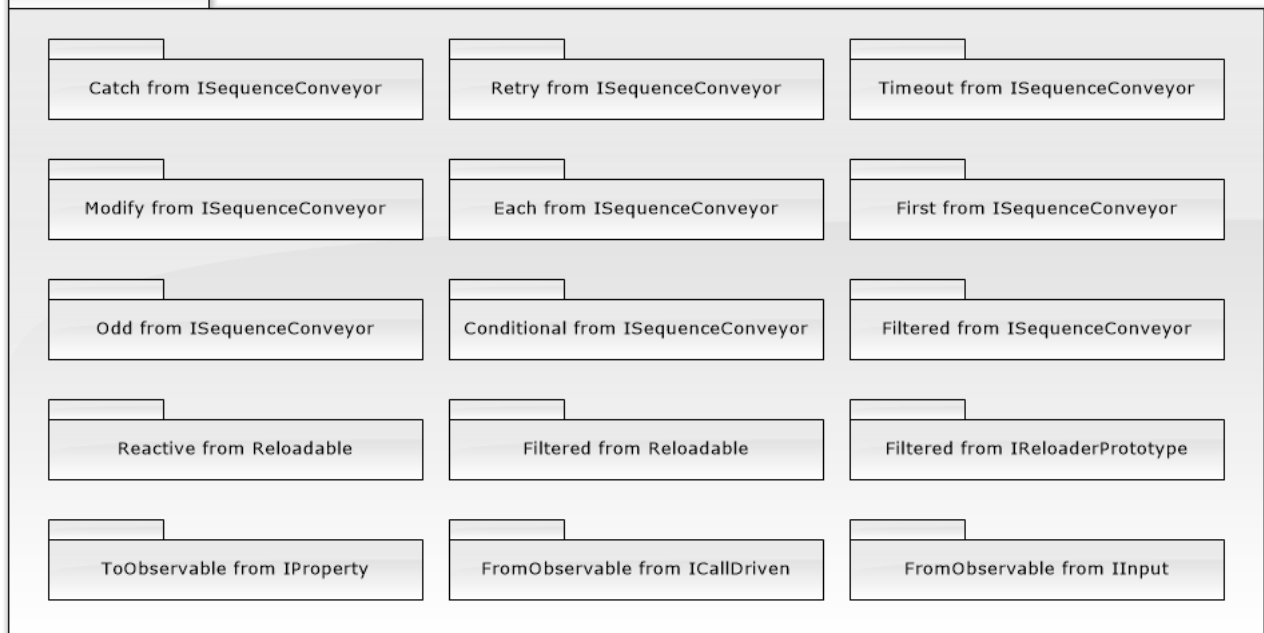
(Continued on the next page)

Scopes (continued)

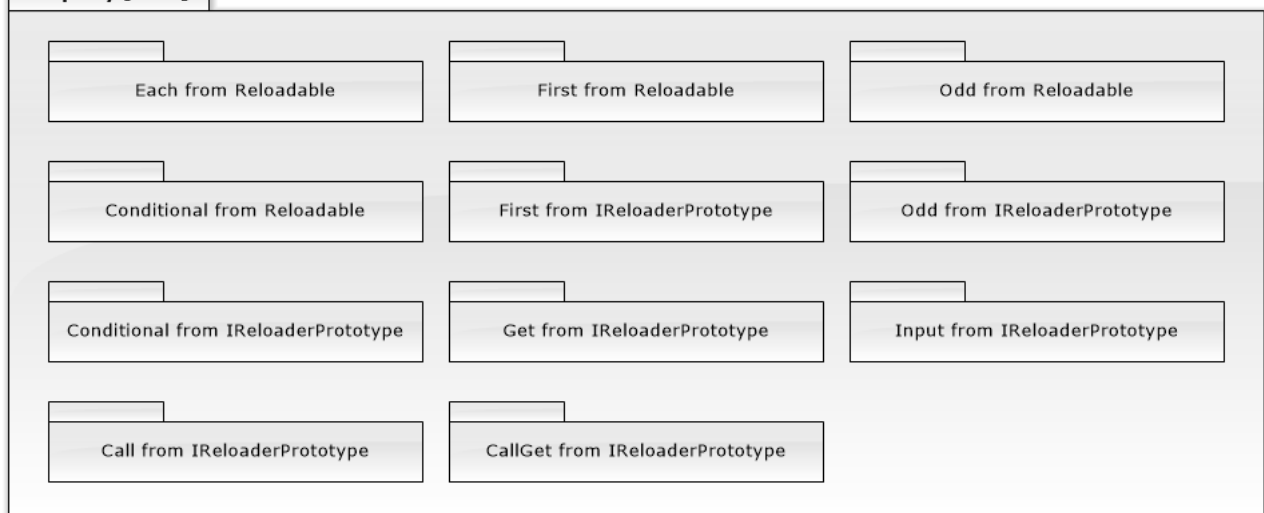
Callbacks [Core]



Reactive [Core]

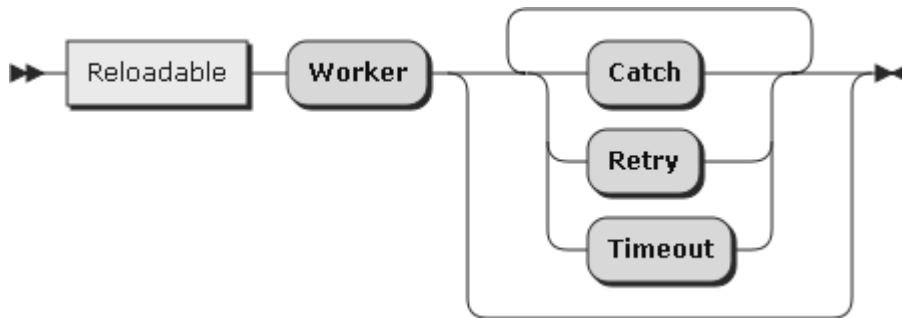


Property [Core]

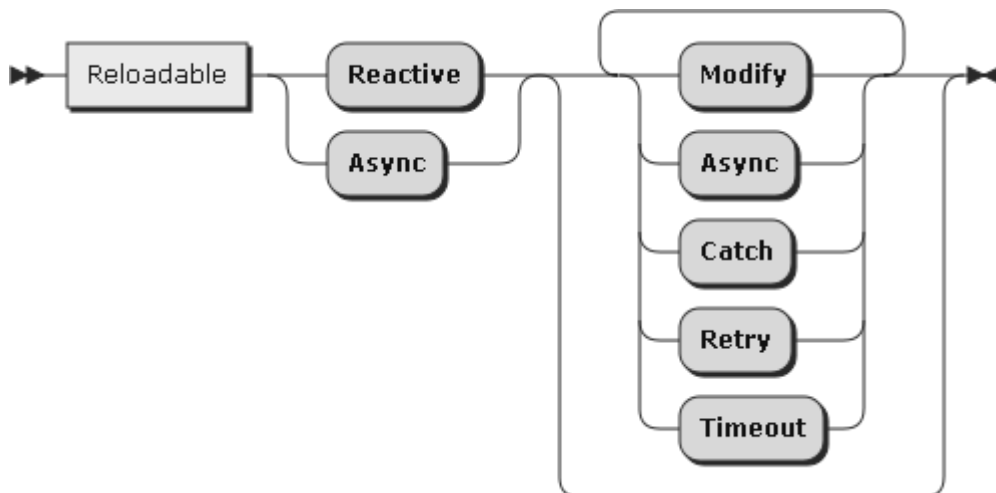


The solution S has a language and describes the grammar of this language unambiguously. The grammar we describe is actually the main purpose of the design we provide here. Types are only means to this end. To illustrate the desired grammar we will use the railroad diagrams and EBNF expressions.

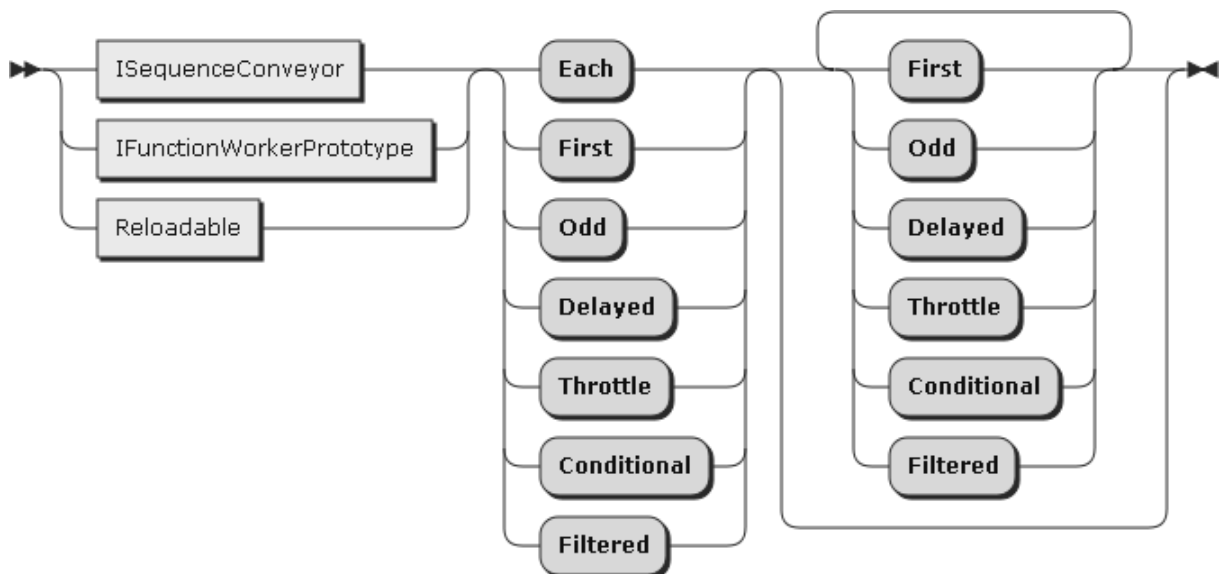
IFunctionWorkerPrototype ::= Reloadable 'Worker' ('Catch' | 'Retry' | 'Timeout')*



ISequenceConveyor ::= Reloadable ('Reactive' | 'Async') ('Modify' | 'Async' | 'Catch' | 'Retry' | 'Timeout')*



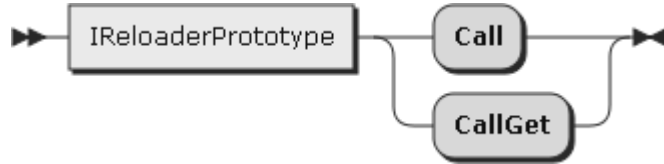
IReloaderPrototype ::= (ISequenceConveyor | IFunctionWorkerPrototype | Reloadable) ('Each' | 'First' | 'Odd' | 'Delayed' | 'Throttle' | 'Conditional' | 'Filtered') ('First' | 'Odd' | 'Delayed' | 'Throttle' | 'Conditional' | 'Filtered')*



IPropertyPrototype ::= IReloaderPrototype 'Get'



ICallPropertyPrototype ::= IReloaderPrototype ('Call' | 'CallGet')



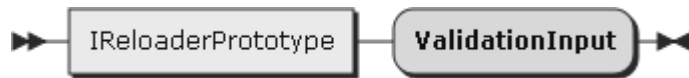
IInputPropertyPrototype ::= IReloaderPrototype 'Input'



ICheckInputPropertyPrototype ::= IReloaderPrototype 'CheckInput'



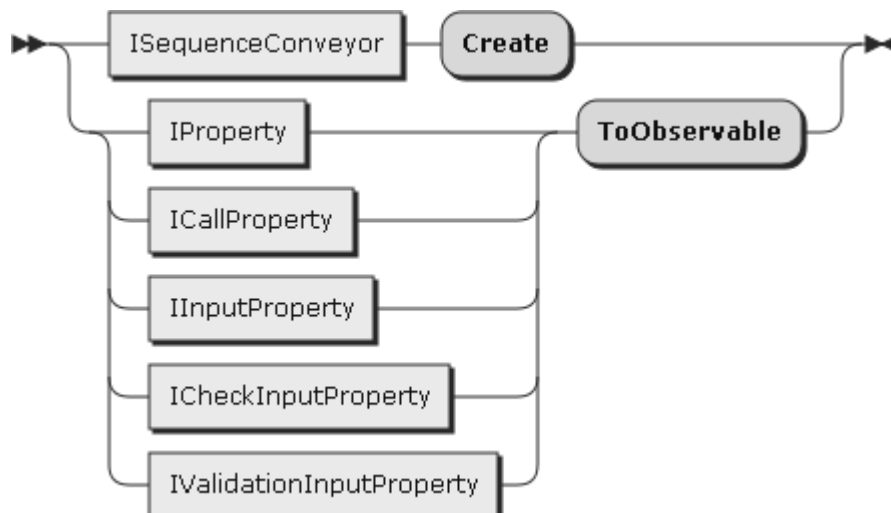
IValidationInputPropertyPrototype ::= IReloaderPrototype 'ValidationInput'



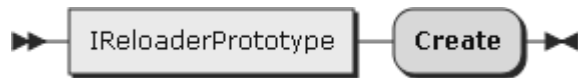
IFunctionWorker ::= IFunctionWorkerPrototype 'Create'



IObservable ::= ISequenceConveyor 'Create' | (IProperty | ICallProperty | IInputProperty | ICheckInputProperty | IValidationInputProperty) 'ToObservable'



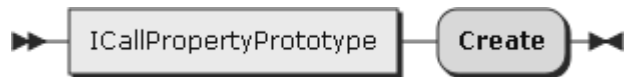
IReloader ::= IReloaderPrototype 'Create'



IProperty ::= IPropertyPrototype 'Create'



ICallProperty ::= ICallPropertyPrototype 'Create'



IInputProperty ::= IInputPropertyPrototype 'Create'



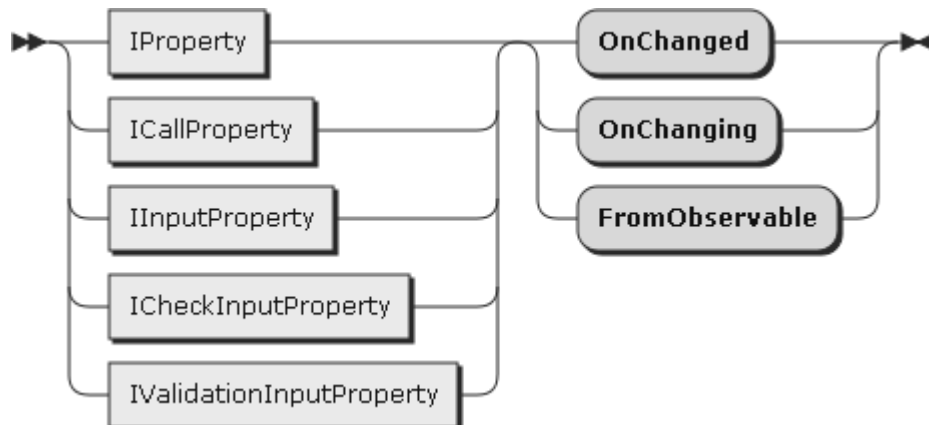
ICheckInputProperty ::= ICheckInputPropertyPrototype 'Create'



IValidationInputProperty ::= IValidationInputPropertyPrototype 'Create'



IDisposable ::= (IProperty | ICallProperty | IInputProperty | ICheckInputProperty | IValidationInputProperty) ('OnChanged' | 'OnChanging' | 'FromObservable')



Engineering C#

This section explores an approach to the engineering of solutions in C#. It illustrates how the design translates into C# engineering in the form of *engineering patterns* that reflect the respective patterns discoverable in the type transitions we use for design.

Lazy Creation

$$IFlySwimBirdPrototype \xrightarrow{\text{Create}} IFlySwimBird$$

`Duck` is the class the instances of which we seek to produce. We want to return the instances as `IFlySwimBird`, however, we also want this class to be usable with other interfaces not convertible to `IFlySwimBird`.

```
public interface IFlySwimBird : IFly, ISwim {}

public class Duck : IFly, ISwim
{
    protected Duck(int initialPosition)
    {
        Position = initialPosition;
    }

    public virtual int Position { get; protected set; }
    public virtual void Fly() { Position+=100; }
    public virtual void Swim() { Position+=1; }
}
```

We want `Duck` to be only used as a part of a product hierarchy. The product classes like `Duck` cannot be instantiated directly, however, in this case we did not use the modifier `abstract` on `Duck` thus showing that the direct subclasses of the class would only need an empty public constructor and no other code to be constructible. The `abstract` we will use to signify that the class will use both means of creation and some functionality of its subclasses. We will not use public constructors in products or their subclasses outside prototypes.

Abstract classes and inheritance we will use to reduce code duplication in subclasses.

```
public abstract class FlySwimBird : IFly, ISwim
{
    protected FlySwimBird(int initialPosition)
    {
        Position = initialPosition;
    }

    public virtual int Position { get; protected set; }
    public abstract void Fly();
    public abstract void Swim();
}

public class Duck : FlySwimBird
{
    protected Duck(int initialPosition) : base(initialPosition)
    { }

    public override void Fly() { Position += 100; }
    public override void Swim() { Position += 1; }
}

public class Goose : FlySwimBird
{
    protected Goose(int initialPosition) : base(initialPosition)
    { }

    public override void Fly() { Position += 200; }
    public override void Swim() { Position += 2; }
}
```

Each prototype we make is made up of two parts, the first one being home for a private constructible subclass of a product class.

```
public partial class DuckPrototype
{
    private class BirdInstance : Duck, IFlySwimBird
    {
        public BirdInstance(int initialPosition) : base(initialPosition)
        { }
    }
}
```

The second part of the prototype is directly responsible for the type transition it is made for.

```
public partial class DuckPrototype : IFlySwimBirdPrototype
{
    public IFlySwimBird Create(int initialPosition)
    {
        return new BirdInstance(initialPosition);
    }
}
```

The method `Create(...)` here is the engineering representation of the type transition $IFlySwimBirdPrototype \xrightarrow{Create} IFlySwimBird$. Using prototypes to create instances of classes, we call *lazy creation* of instances.

Decoration

$IFlySwimBirdPrototype \xrightarrow{Tame} IFlySwimBirdPrototype$

Product classes can accept prototypes as constructor parameters. If the interface of the product class and the interface of instances the prototype produces match we call such product class a *decorator*.

```
public class TameFlySwimBirdDecorator : IFlySwimBird
{
    protected TameFlySwimBirdDecorator(
        IFlySwimBirdPrototype compositionBasePrototype,
        int initialPosition, Func<bool> allow)
    {
        _CompositionBase = compositionBasePrototype.Create(initialPosition);
        _Allow = allow;
    }

    private readonly IFlySwimBird _CompositionBase;
    private readonly Func<bool> _Allow;

    public override void Fly(){if(_Allow())_CompositionBase.Fly();}
    public override void Swim(){if(_Allow())_CompositionBase.Swim();}
}
```

It is worth noting that constructor parameters should always be tested in classes where they are referenced. Guard clauses in constructors serve this purpose. No need to test an object where it is only passed along and not used.

```
... protected TameFlySwimBirdDecorator(
                                IFlySwimBirdPrototype compositionBasePrototype,
                                int initialPosition, Func<bool> allow)
{
    if (compositionBasePrototype == null)
        throw new ArgumentNullException("compositionBasePrototype");
    if (allow == null) throw new ArgumentNullException("allow");

    _CompositionBase = compositionBasePrototype.Create(initialPosition);

    if (_CompositionBase == null) throw new NotSupportedException();

    _Allow = allow;
} ...
```

Prototypes and not readymade products are used as constructor parameters in decorators because sometimes disposing an object and instantiating another one is the only way to manage it (as it is with BackgroundWorker) and we do not want to change the code if such a requirement arises. Besides, this approach allows keeping all the guard clauses in one place removing virtually all the functionality from the prototype.

```
public partial class TameFlySwimBirdDecoratorPrototype : IFlySwimBirdPrototype
{
    public TameFlySwimBirdDecoratorPrototype(
        IFlySwimBirdPrototype compositionBasePrototype, Func<bool> allow)
    {
        _CompositionBasePrototype = compositionBasePrototype;
        _Allow = allow;
    }

    private readonly IFlySwimBirdPrototype _CompositionBasePrototype;
    private readonly Func<bool> _Allow;

    public IFlySwimBird Create(int initialPosition)
    {
        return new DecoratorInstance(_CompositionBasePrototype,
                                    initialPosition, _Allow);
    }
}
```



```

public partial class TameFlySwimBirdDecoratorPrototype
{
    private class DecoratorInstance : TameFlySwimBirdDecorator
    {
        public DecoratorInstance(
            IFlySwimBirdPrototype compositionBasePrototype,
            int initialPosition, Func<bool> allow)
            : base(compositionBasePrototype, initialPosition, allow)
        { }
    }
}

```

The type transition $IFlySwimBirdPrototype \xrightarrow{Tame} IFlySwimBirdPrototype$ is represented by a specifically designed extension method `Tame (...)`.

```

public static class FlySwimBirdPrototypeExtensions
{
    public static IFlySwimBirdPrototype Tame(
        this IFlySwimBirdPrototype source, Func<bool> allow)
    {
        return new TameFlySwimBirdDecoratorPrototype(source, allow);
    }
}

```

Composition

$IFlySwimBirdPrototype \xrightarrow{Group} IFlockPrototype$

We build our products on top of other products. Product classes that accept prototypes as constructor parameters we will call *composites*. Decorators are among special cases of the *composite*.

In the following example, `Flock` contains a collection of birds. It also has the crucial information needed to produce them and control as a single unit.

```

public interface IFlock : IFly {}

public interface IFlockPrototype
{
    IFlock Create(int initialPosition, int count)
}

```

The engineering of all composites is based on the same principle of injection.

```
public class Flock : IFly
{
    protected Flock (IFlySwimBirdPrototype birdPrototype,
                     int initialPosition, int count)
    {
        if(count < 0)throw new ArgumentOutOfRangeException("count");

        _Birds = Enumerable.Repeat(initialPosition, count)
                           .Select(o => birdPrototype.Create(o));
    }

    private readonly IEnumerable<IFlySwimBird> _Birds;

    public void Fly(){_Birds.ToList().ForEach(o => o.Fly());}
}

public partial class FlockPrototype : IFlockPrototype
{
    public FlockPrototype(IFlySwimBirdPrototype birdPrototype)
    {
        _BirdPrototype = birdPrototype;
    }

    private readonly IFlySwimBirdPrototype _BirdPrototype;

    public IFlock Create(int initialPosition, int count)
    {
        return new FlockInstance(_BirdPrototype, initialPosition, count);
    }
}

public partial class FlockPrototype
{
    private class FlockInstance: Flock, IFlock
    {
        public FlockInstance ( IFlySwimBirdPrototype birdPrototype,
                               int initialPosition, int count)
            : base(birdPrototype, initialPosition, count)
        { }
    }
}
```

The type transition $IFlySwimBirdPrototype \xrightarrow{Group} IFlockPrototype$ from one prototype to always engineered as an extension method. Please notice how we group the extension methods.

```
public static class FlySwimBirdPrototypeExtensions
{
    public static IFlySwimBirdPrototype Tame(
        this IFlySwimBirdPrototype source, Func<bool> allow)
    {
        return new TameFlySwimBirdDecoratorPrototype(source, allow);
    }

    public static IFlockPrototype Group(
        this IFlySwimBirdPrototype source)
    {
        return new FlockPrototype(source);
    }
}
```

Singularities

Marker

Description $\xrightarrow{Duck} IFlySwimBirdPrototype$

A class or interface that participates in type transitions but is never referenced we will call a *marker*. Marker classes/interfaces serve as roots for expressions. Below is an example of a marker class.

```
public abstract class Description
{
    private class Instance : Description { }

    public static Description Start()
    {
        return new Instance();
    }
}
```

What is the best way to create an instance of `IFlySwimBirdPrototype` from scratch? We could allow using constructors of some implementers of this interface, or we could make a class with static factory methods one for each bird. However, a much more beneficial alternative to the above would be using a marker class and defining extension methods to this class one method for each bird.

```
public static class DescriptionExtensions
{
    public static IFlySwimBirdPrototype Duck(
        this Description source)
    {
        return new DuckPrototype();
    }

    public static IFlySwimBirdPrototype Goose(
        this Description source)
    {
        return new GoosePrototype();
    }
}
```

Extension methods are miles ahead of any other C# feature in terms of extensibility. With extension methods we do not have to care about where (in which class or even assembly) they are defined. We know that they can all be used as a single language.

```
var flock = Description.Start().Duck().Group().Create(1, 10);
```

Singularities

Conveyor

$$Description \xrightarrow{BirdOfPrey} IBirdOfPreyConveyor$$

A class that acts like a prototype but is ignorant of the concrete product it creates we will call *conveyor*.

Conveyors are usually used to construct interfaces or classes for which a convenient extension method language has already been developed, and thus their instantiation can be easily and safely implemented right in our own extension methods.

```

public class BirdOfPreyConveyor : IBirdOfPreyConveyor
{
    public BirdOfPreyConveyor(Func<int,IBirdOfPrey> getBird)
    {
        _GetBird = o == null ? null : getBird(o);
    }

    private readonly Func<int,IBirdOfPrey> _GetBird;

    public IBirdOfPrey Create(int initialPosition)
    {
        return _GetBird(initialPosition);
    }
}

public static class DescriptionExtensions
{
    public static IBirdOfPreyConveyor BirdOfPrey(
        this Description source)
    {
        return new BirdOfPreyConveyor(o => Birds.OfPrey.Create(1,o));
    }
}

```

Sometimes, we also need to decorate a conveyor. The only reason for that is keeping our grammar simple and cohesive. Otherwise, we could allow the user to specify everything when using the source conveyor.

```

public class BirdOfPreyConveyorDecorator : IBirdOfPreyConveyor
{
    public BirdOfPreyConveyorDecorator(IBirdOfPreyConveyor compositionBase,
        Func<IBirdOfPrey,IBirdOfPrey> modification)
    {
        _CompositionBase = compositionBase;
        _Modification = modification;
    }

    private readonly IBirdOfPreyConveyor _CompositionBase;
    private readonly Func<IBirdOfPrey,IBirdOfPrey> _Modification;

    public IBirdOfPrey Create(int initialPosition)
    {
        return _Modification(_CompositionBase.Create(initialPosition));
    }
}

```

We are now able to define the type transition

$IBirdOfPreyConveyor \xrightarrow{SoHigh} IBirdOfPreyConveyor.$

```
public static class BirdOfPreyConveyorExtensions
{
    public static IBirdOfPreyConveyor SoHigh(
        this IBirdOfPreyConveyor source)
    {
        return new BirdOfPreyConveyorDecorator
            (source, o => o.SoHigh());
    }
}
```

Engineering C# Style

The patterns described above are enough to build a library like Property Framework. However, to improve the perception of code and stimulate the use of accepted practices the following minimalist coding style can be used.

Related – keep together

Unrelated – keep apart

PascalCaseState	private mutable fields;
_PascalCase	private immutable fields;

TPascalCase	type parameters;
camelCase	method parameters;

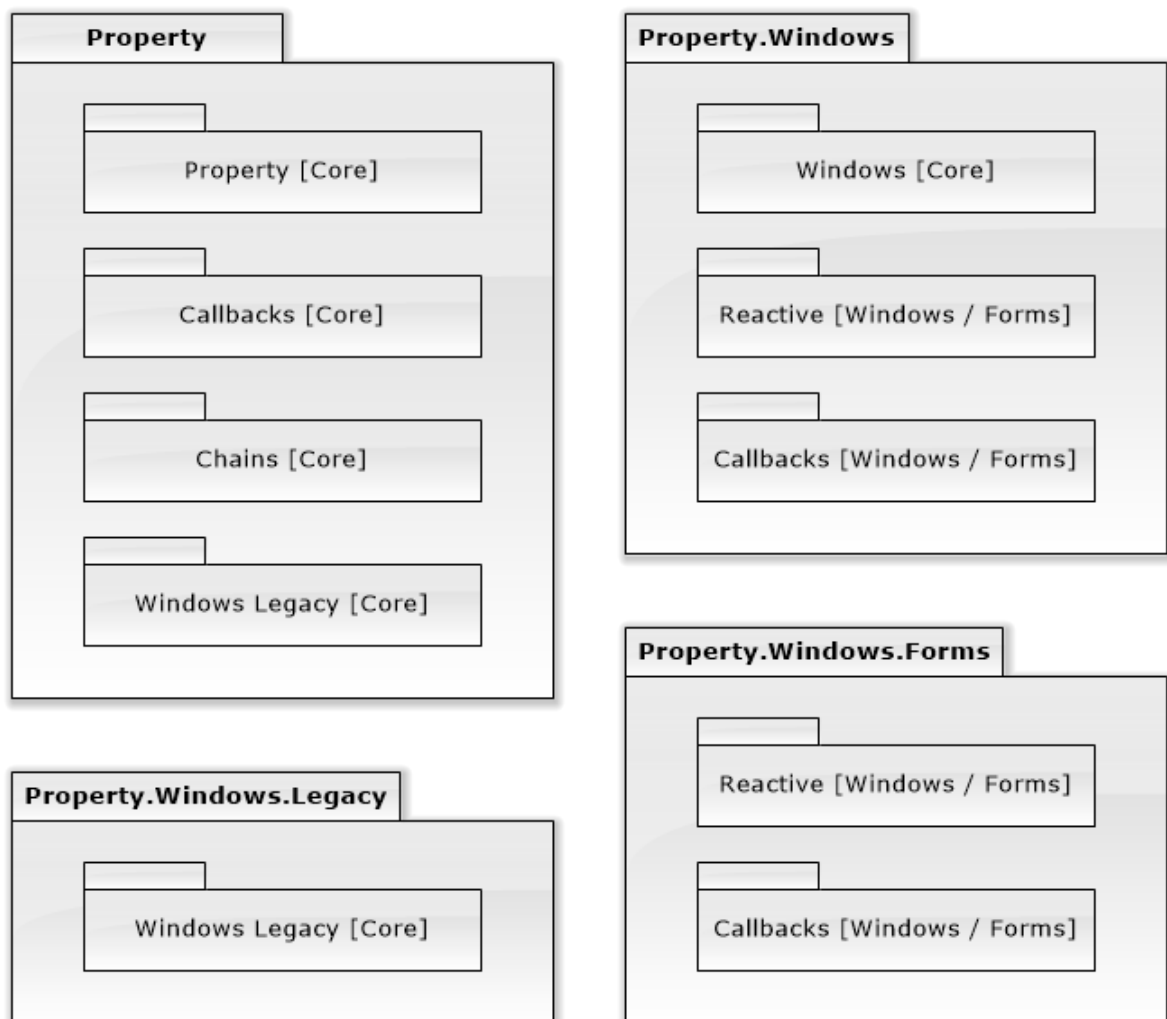
IPascalCase	interface declaration;
ALL_CAPITAL	constants;
PascalCase	everything else;

//=//	‘grass’ is used after constructors to attract attention;
/// <Comments/>	comments are used on whatever users are supposed to use.

Engineering Property Framework

Property Framework will be based on the engineering patterns we described in the previous section. In this section, we will provide some more details on the C# specific implementation details of PF.

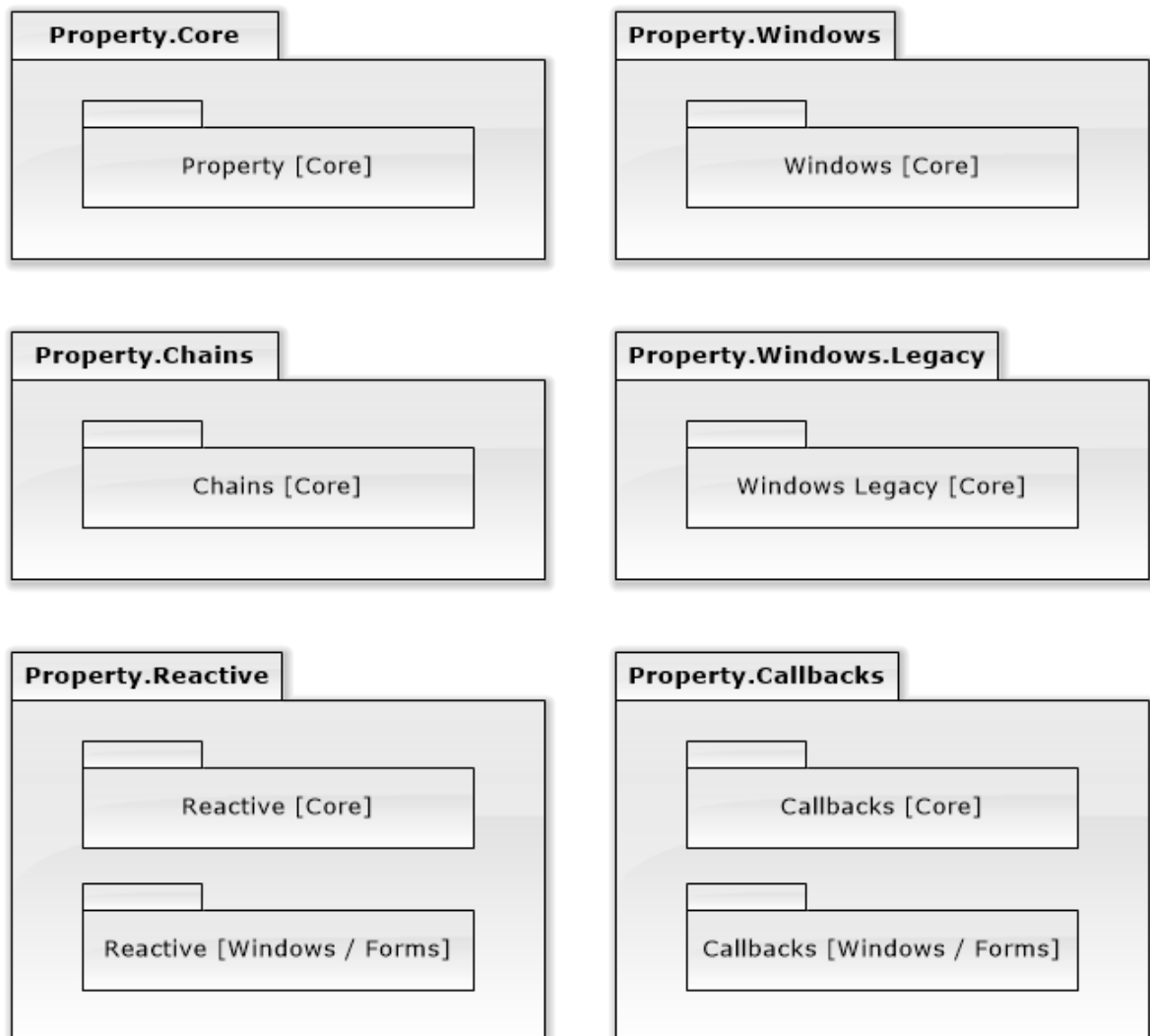
In C# scoping is achieved through the use of namespaces. In the design section, we have defined a number of scopes that determine the availability of type transitions. When implementing the type transitions with extension methods we should define namespaces that would provide the implementation context for the type transitions. The following diagram shows the namespaces used for extension methods in PF. Other classes should be placed in namespaces that have minimal interference with the following ones.



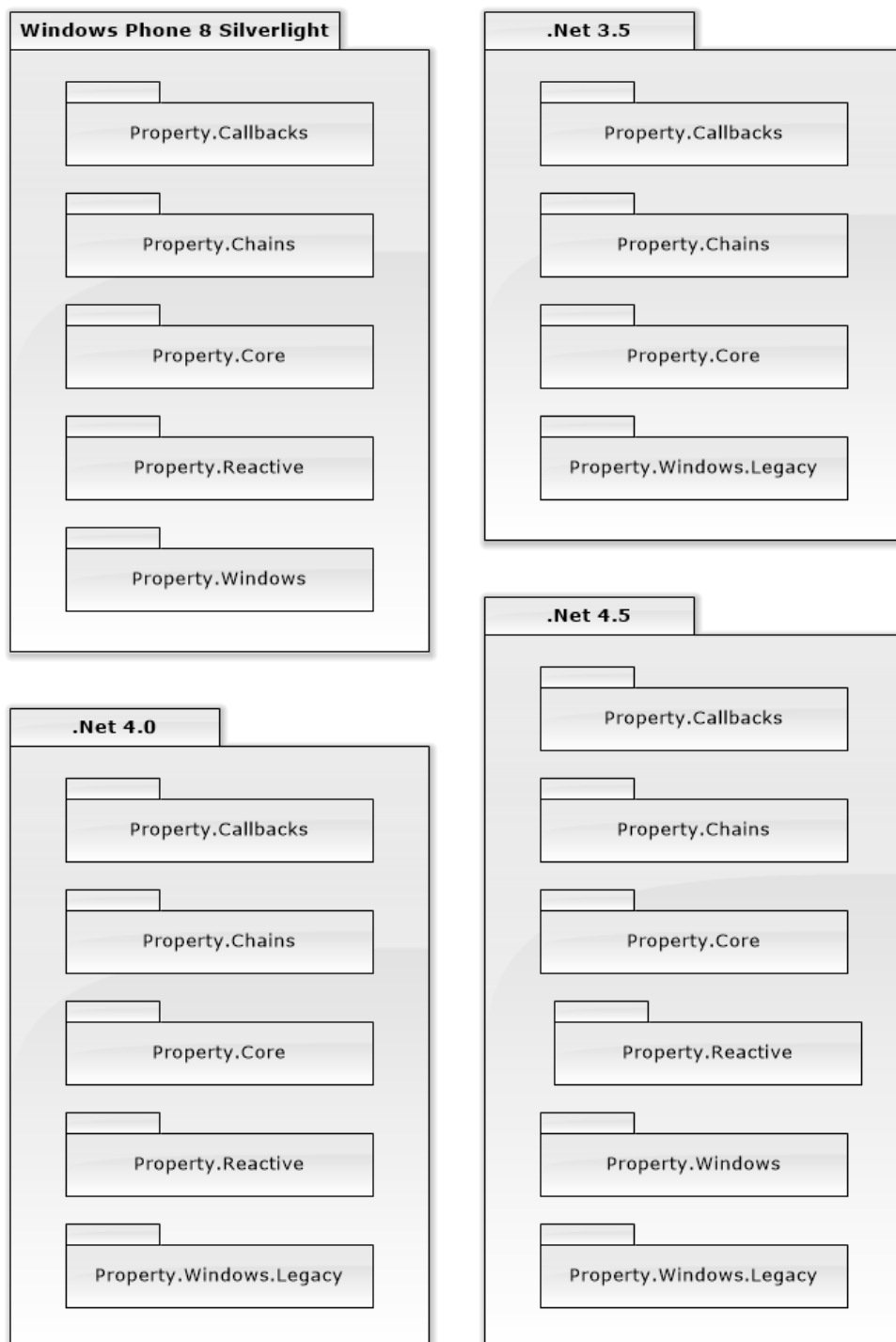
Architecture Property Framework

This section provides information on the packaging of Property Framework and its platform specifics.

Property Framework is shipped as a set of DLL assemblies. Each assembly enables a certain subset of the PF functionality and thus serves as another scoping instrument. The main reason for dividing the functionality is to provide the user with only the features that they want to use and avoid the distraction that comes from seeing unneeded extension methods on the list. The following diagram shows the assemblies that make up Property Framework.



The other reason for using multiple assemblies is the need for scalability and portability of Property Framework. Property Framework is supported on Windows Phone 8, .Net Framework versions 3.5, 4.0, and 4.5. Each of these platforms has its own limitations that we can address by including or excluding assemblies. Windows Phone 8 applications will hardly need legacy validation techniques, .Net 3.5 has not support for Reactive Extensions and like .Net 4.0 does not allow us to use `INotifyDataErrorInfo`. We will package the platform-specific versions of PF according to the following diagram.



Dependencies

Some parts of the PF source code / compiled assemblies may require third-party components. Below is the list of such dependencies.

Test projects - Moq

All the test projects that come with PF source code require Moq. Moq assemblies are included into the source code package under \Lib directory. Moq license is shipped together with the assemblies.

Property.Reactive - Rx

To use the features of the Property.Reactive assembly you will need Reactive Extensions assemblies installed. The Rx package is obtainable from the Microsoft download site at <http://www.microsoft.com/en-us/download/details.aspx?id=30708>.

Showcase.ScrollAsYouLoad for .NET 3.5 - Blend 3.0 SDK

To run the Showcase.ScrollAsYouLoad demonstration app for .Net 3.5 you will need Blend SDK installed on your system. The SDK is available for download at: <http://www.microsoft.com/en-us/download/details.aspx?id=22829>.