Creation Patterns

Objectives

- Replace the use of new for more loosely coupled alternatives
  - Factory patterns
  - Builder
  - Prototype

The need to replace new

- Use of new keyword implies tight coupling
- What if
  - You want to test a component that needs to create another component
  - The choice of which type to create is complex
  - The type you need to create hasn't been written yet
    - Extension points, Plugins

```
public void MakeAnimalNoise(string type) {
   Animal animal = null;

   switch (type.ToLower()) {
         case "dog": animal = new Dog(); break;
         case "chicken": animal = new Chicken(); break;
         case "cow": animal = new Cow(); break;
         case "sheep": animal = new Sheep(); break;
     }
    animal.Speak();
}
```


Refactor

- Is this the factory pattern ?

```
public void MakeAnimalNoise(string type) {
   Animal animal = AnimalCreator.Create(type);

   animal.Speak();
}
```

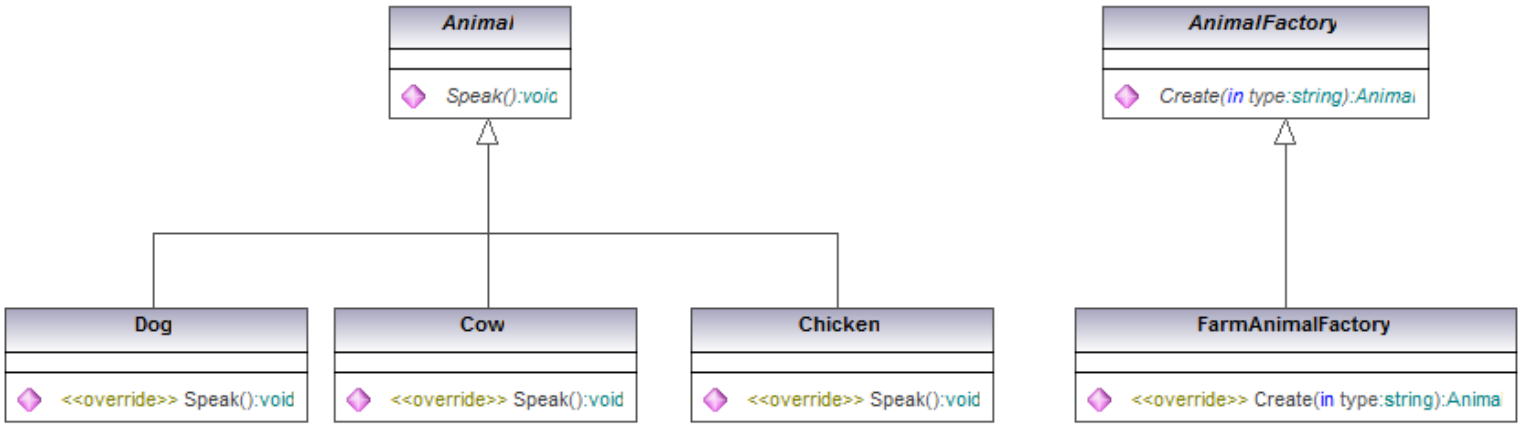
```
public static class AnimalCreator {
   public static Animal Create(string type)  {
      Animal animal = null; ;
      switch (type.ToLower()) {
         case "dog": animal = new Dog(); break;
         case "chicken": animal = new Chicken(); break;
         case "cow": animal = new Cow(); break;
         case "sheep": animal = new Sheep(); break;
     }
     return animal;
}
```

Creation Method

- Simplified the caller, encapsulated creation logic but still
  - Tightly coupled to the AnimalCreator class
  - Hard to test
- Sometimes known as “Creation Method” pattern
- To produce a true factory pattern solution
  - The creation logic needs to be able to vary

Factory Method Pattern

- "Encapsulate what varies and separate from code that's constant"
- Creation abstraction is constant
  - Create(string type)
- Implementation varies
  - Create real Animals or perhaps fake ones for testing



Using the factory

- Method is now loosely coupled from implementation of animal creation logic
- Animal creation logic can now therefore vary

```
public static void MakeAnimalNoise(string type,
                                   AnimalFactory factory)
{
 Animal animal = factory.Create(type);

 animal.Speak();
}
```

Implementing the factory

- **Abstract** class provides abstraction for creation
- **Derived** classes provide implementation of creation logic

```
public abstract class AnimalFactory{
  public abstract Animal Create(string type);
}

public class FarmAnimalFactory : AnimalFactory{
 public override Animal Create(string type) {
  Animal animal = null;

  switch (type.ToLower()){
   case "dog": animal = new Dog(); break;
   case "chicken": animal = new Chicken(); break;
   case "cow": animal = new Cow(); break;
   case "sheep": animal = new Sheep(); break;
 }
 return animal;
}
}
```

Implementation enhancement

- Base factory class provides base implementation providing common point of interception
- Delegates to derived class for specific creation logic
- Can layer in caching, singleton behaviour , lazy creation

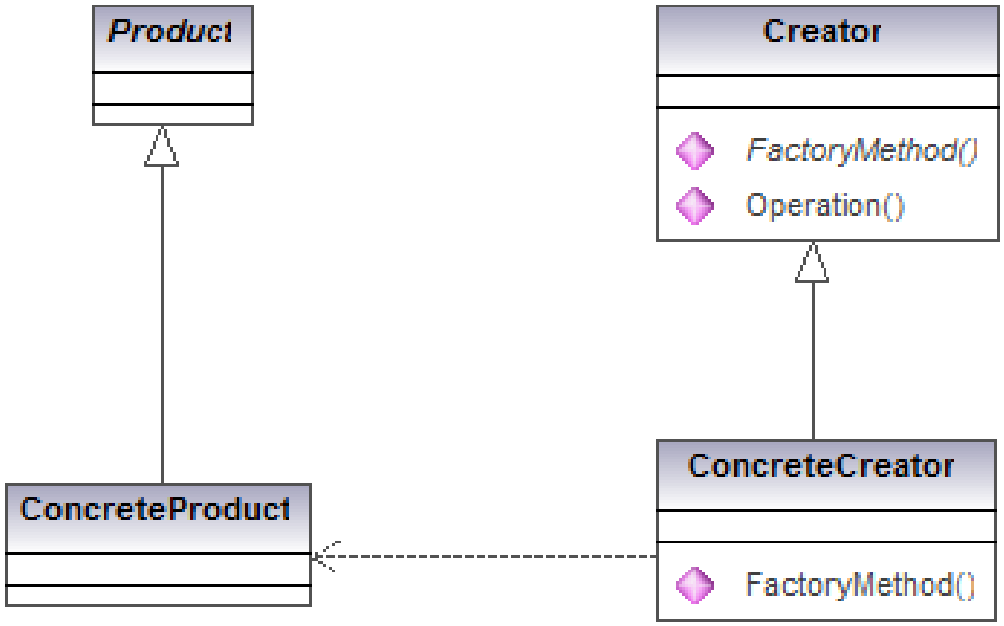
```
public abstract class AnimalFactory{
   public virtual Animal Create(string type) {
    // Do something before creation
    Animal animal = InternalCreate(type);
    // Do something after creation
    return animal;
   }

   protected abstract Animal InternalCreate(string type);
 }

}
```

Factory Method Pattern

- Define an interface for creating an object, but let subclasses decide which class to instantiate.
- Factory Method lets a class defer instantiation to subclasses



The New of the future

- Replacing New offers flexibility
- Utilise reflection to create types unknown at compile time

```
public class AnimalAttribute : Attribute {
  public string Type {get;set;}
}
```

```
[AnimalAttribute(Type="cow")]
public class Cow:Animal {
 }
```

```
public class ReflectionAnimalFactory : AnimalFactory   {
  private readonly Dictionary<string, Type> animals;
  public ReflectionAnimalFactory(){
   animals =
   (from type in this.GetType().Assembly.GetTypes()
     let animalAttrs = type
         .GetCustomAttributes(typeof(AnimalAttribute), false)
         .OfType<AnimalAttribute>()
     where animalAttrs.Count() > 0
     select new { Type = type, Kind = animalAttrs.First().Type }
   ).ToDictionary(e => e.Kind, e => e.Type);
  }
  public override Animal Create(string type){
    return (Animal) Activator.CreateInstance(animals[type.ToLower()]);
  }
}
```

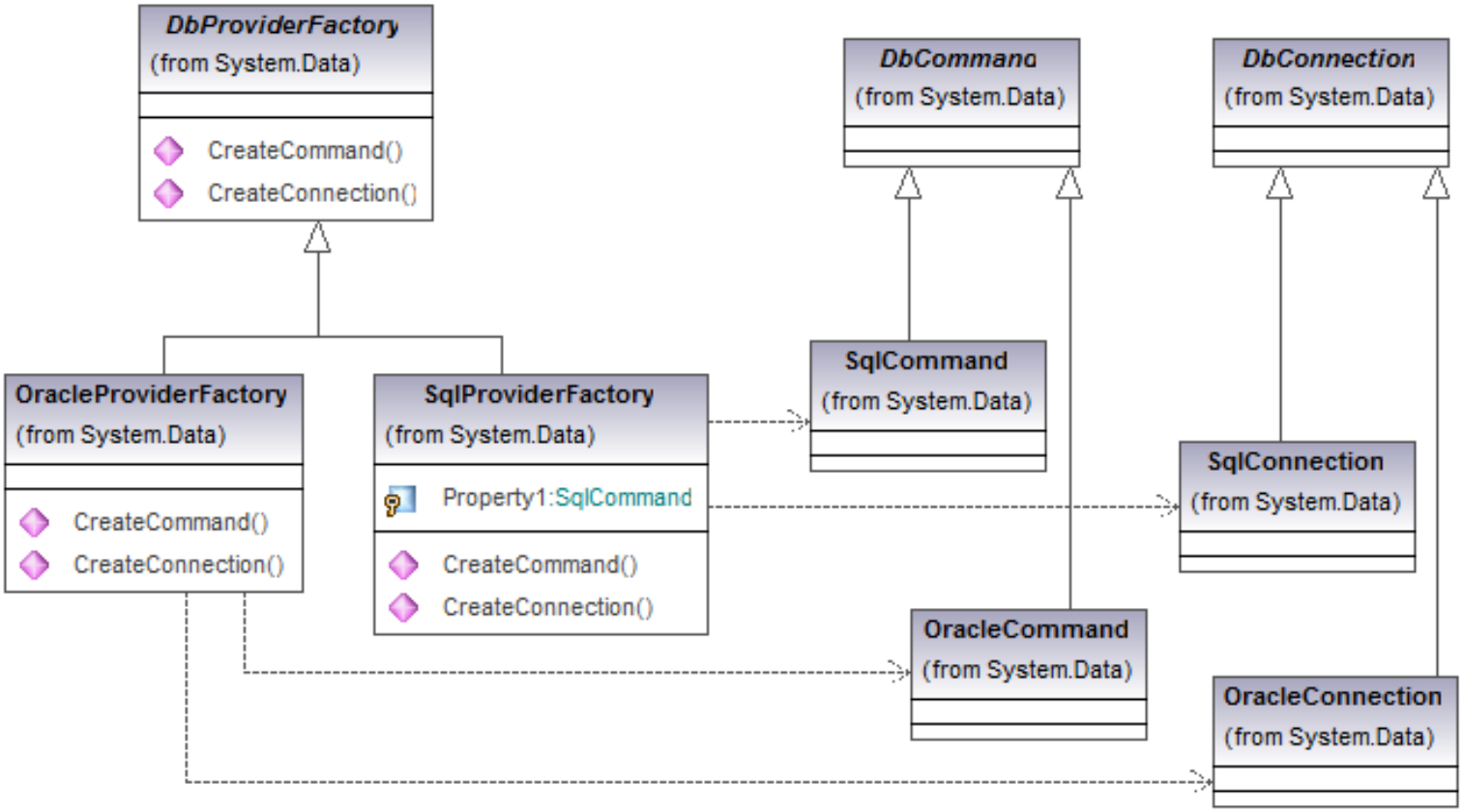
Creating families of objects

- Suppose we want to create a number of related objects
  - Database connection, command, parameter, permission
  - Would want a single factory to create all products
  - One 'create' method would not be enough
- Factory type contains many create methods
- Objects from same factory are compatible

```
public abstract class DbProviderFactory  {
   public virtual DbCommand CreateCommand();
   public virtual DbConnection CreateConnection();
  . . .
}
```

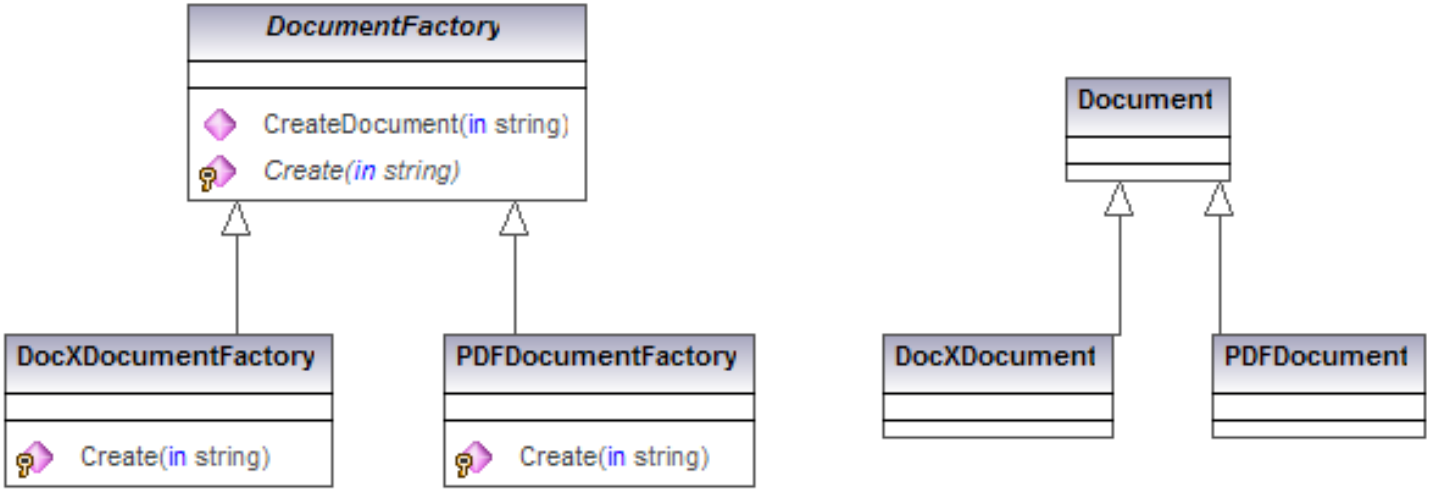
```
DbProviderFactory factory =  DbProviderFactories
                                  .GetFactory("System.Data.SqlClient");
DbConnection connection = factory.CreateConnection();
DbCommand cmd = factory.CreateCommand();
cmd.Connection = connection;
```

ADO.NET Abstract Factories



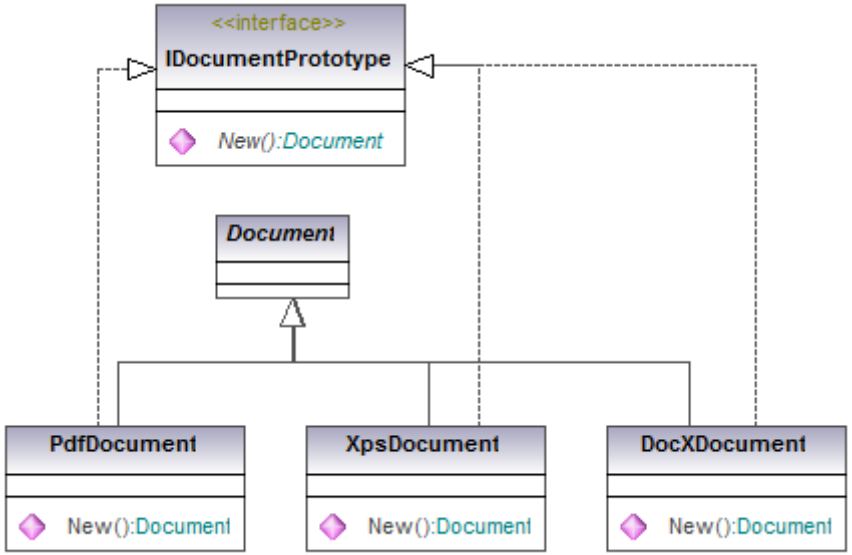
Too many factories

- Often as the number of concrete objects increases the number of factories increases too
  - As we add new document types we may need to add additional factories
  - XPSDocument, XPSDocumentFactory
- What if the concrete objects had knowledge of cloning themselves, we could do away with factories



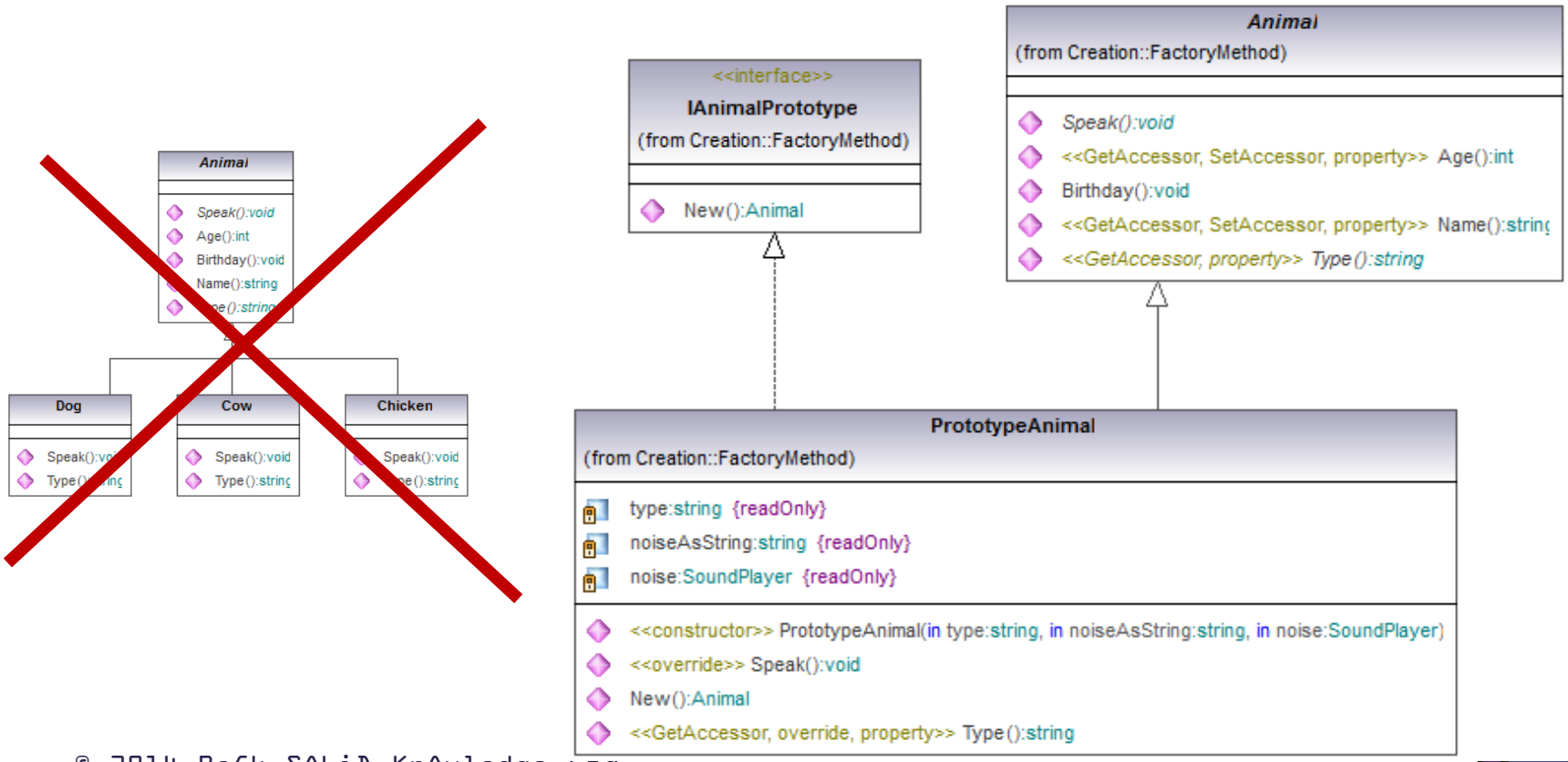
Prototype pattern

- Objects know how to clone themselves
- Application creates prototype object
  - Configures object for initial settings
- Client utilises New method rather than relying on a factory



Prototype pattern (cont)

- Prototype objects can sometimes be used instead of creating additional classes
- Creates the illusion of new types at runtime
  - Instantiate from prototype rather than by class



Multi step construction

- Factories are great at single step construction
- What if constructing complex object that requires many parts
  - Factory methods start to grow
  - Providing every permutation not practical

```
public abstract class Mail {
 public abstract void Send();
}

public abstract class MailFactory {
 public abstract Mail Create(string from, string to, string subject,
                             string body);
 public abstract Mail Create(string from, string to, string subject,
                             string body, string[] attachments);
 public abstract Mail Create(string from , string to , string cc,
                             string subject , string body )
 . . .
}
```

Builder pattern

- Builders allow the construction of a complex object over a series of steps
  - Builder hides the complexity of the construction
  - Complex object representation is independent of method of construction
- Construction methods typically return self to allow natural method chaining
- Build method returns complex object

```
public abstract class MailBuilder  {
        public abstract MailBuilder SetFrom(string name);
        public abstract MailBuilder SetBody(string body);
        public abstract MailBuilder SetSubject(string subject);
        public abstract MailBuilder AddTo(string name);
        public abstract MailBuilder AddCC(string name);
        public abstract MailBuilder AddAttachment(string filename);

        public abstract Mail Build();
}
```

Builder in action

- Client
  - creates new builder for each construction
  - Calls construction methods
  - When construction complete, calls Build to return complex object

```
MailBuilder builder = new SmtpMailBuilder();

builder
    .SetFrom("aclymer@develop.com")
    .SetSubject("Patterns are cool")
    .SetBody("...")
    .AddTo("rich@develop.com")
    .AddCC("kev@develop.com")
    .AddAttachment("uml.png")
    .Build()
    .Send();
```


Builders in the framework

- StringBuilder
- SqlConnectionStringBuilder
- UriBuilder
  - Define Scheme,Host,Port,Path,Query variables and it builds Uri

```
var connectionStringBuilder = new SqlConnectionStringBuilder()
 {
    DataSource = @".\SQL2008",
    InitialCatalog = "pubs",
    IntegratedSecurity = true
 };

Console.WriteLine(connectionStringBuilder.ConnectionString);
```


Summary

- Replacing the direct use of new creates flexible designs
- Creation Method
  - Commonly used to encapsulate creation logic, often wrongly thought of as factory pattern
- Factory Method
  - Replaces the new keyword with a virtual method for creation, allowing creation logic to vary
- Abstract Factory
  - Creates families of related products
- Prototype
  - Used to prevent factory and product type explosion
- Builder
  - Used instead of factories for complex object creation
  - Separates complex object representation from construction