Dr. Dobb's Journal

December 2013

Next

Dependency Injection

By injecting methods, it's possible to insert different actions into a function without changing code — the ultimate in loose coupling!

ALSO INSIDE

UML 2.5: Do You Even Care? >>

Lean UX:
Making Sure You're Building the Right Product >>

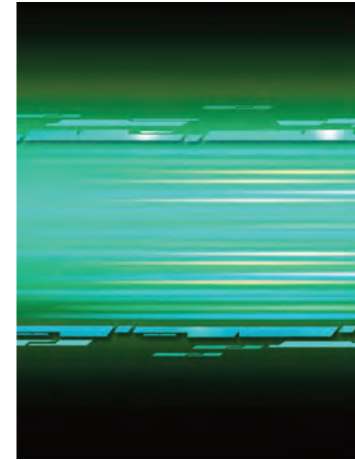
From the Vault:
In Praise of Small Classes >>

www.drdobbs.com

Dr. Dobb's Journal

CONTENTS

December 2013



COVER ARTICLE

7 Dependency Injection

By Mark Seeman

By injecting methods, it's possible to insert different actions into a function without changing code — the ultimate in loose coupling.

FEATURES

13 Lean UX: Making Sure You're Building the Right Product

By Nellie LeMonier

By continuous discovery of users' needs and sharing the research with the entire team, you take the risk out of new projects.

17 From the Vault: In Praise of Small Classes

By Andrew Binstock

Using small classes avoids introducing unnecessary complexity into your code. Now then, how to do it? Start with five not-so-easy steps.

GUEST EDITORIAL

3 UML 2.5: Do You Even Care?

By Scott W. Ambler

Scott argues that while the OMG has been successful in marketing UML, it's been less successful in producing something people find useful.

6 News Briefs

By Adrian Bridgwater

Recent news on tools, platforms, frameworks, and the state of the software development world..

21 Links

Snapshots of interesting items on drdobbs.com including a look at machine learning with Apache Mahout, a discussion about whether Java is dying, and determining the difference between Amdahl's Law and Gustafson-Barsis' Law.

More on DrDobbs.com

The Embarrassing Costs of Not Testing Your Own Code

A recent foul-up at security company Kaspersky shows the high price of letting developers bang out code without testing it.

http://www.drdobbs.com/240162967

In Praise of Haskell

An elegant, reliable, easy-to-maintain, high-level, parallel-friendly, native language. What's not to like?

http://www.drdobbs.com/240163246

Machine Learning with Apache Mahout: Refining the Recommender

Mahout components implement popular algorithms and can be unplugged easily when no longer needed.

http://www.drdobbs.com/240163537

Sorting An Immutable List

The reason this algorithm works is that the recursive part of it ultimately reduces the list to one-element fragments; merging those fragments eventually puts them all in sequence.

http://www.drdobbs.com/240163397

Writing a Particle System on the Raspberry Pi

The Raspberry Pi's graphics processing and HDMI video interface can easily run and display a particle system designed from scratch.

http://www.drdobbs.com/240163294

IN THIS ISSUE
Guest Editorial >>
News >>
Lean UX >>
Dependency Injection >>
Small Classes >>
Links >>
Table of Contents >>

UML 2.5: Do You Even Care?

Scott argues that while the OMG has been successful in marketing UML, it's been less successful in producing something people find useful.

By Scott W. Ambler

Since its beginnings in 1996, the Object Management Group's Unified Modeling Language (UML) has become all but ubiquitous within the IT community. The effect of the UML on our industry has been impressive — there have been dozens of UML books published, thousands of articles and blogs posted, and thousands of training classes delivered — some by me. UML v2.5 is just about to be released, but I'm not sure that I even care. Here's why.

The UML was originally developed by Rational Software, now a division of IBM (my former employer), in 1996 under the leadership of "The Three Amigos": Jim Rumbaugh, Grady Booch, and Ivar Jacobson. The 1.0 version was proposed in January 1997 and officially adopted by the Object Management Group (OMG) later that year. Since then, the UML has undergone many revisions, with UML 2.0 released in 2005 and most recently UML 2.4.1 in August 2011. An "in process version" of UML 2.5 was published in October 2012 and is expected to be officially released soon. My guess is that it will happen at the next OMG Technical Meeting, being held in Santa Clara the second week of December.

Although I have no doubt that everyone involved with the UML 2.5 release has done a great job, I'm struggling to find a reason to be interested. This is particularly strange given my background with the UML. I've written several UML books, including *The Elements of UML 2.0 Style* (2005), *The Object Primer 3rd Edition* (2004), and Jolt Productivity Award winning *Building Object Applications that Work* (1997). From an agile point of view, showing how to create UML diagrams in an agile manner was an important aspect of *Agile Modeling* (2002), which came about from the article "Extreme Modeling" (http://www.drdobbs.com/184414673). My point is that my history shows I'm about as pro-UML as you're going to get.

A Simpler Specification?

The goal of UML 2.5 is to simplify and clarify a specification document so as to reduce implementation problems and promote interoperability between tools. There was a significant pushback against UML v2.0 due to its complexity, so simplification is a step in the right direction. One of UML's complexities is the addition of diagrams that seem to have little value for most practitioners. For example, have you ever seen, let alone used, a composite structure diagram, an interaction overview diagram, or a communication diagram? Do you even know what I'm talking about? The good news is that in UML v2.5, the language itself remains virtually unchanged. However, new diagrams are being added for a total of 19 (up from 16 in UML 2.0). The additions are:

IN THIS ISSUE

Guest Editorial >>
News >>
Lean UX >>
Dependency Injection >>
Small Classes >>
Links >>
Table of Contents >>

- Model Diagram, which is a specialization of a Package Diagram (akin to free-form architecture diagrams)
- Manifestation Diagram, a specialization of either a Deployment Diagram or Component Diagram that shows how components are manifested in the physical solution
- Network Architecture Diagram, which is effectively a high-level Deployment Diagram.

Interestingly, there are still no diagrams specific to user interface or database development. This is simplicity? Sigh.

A second complexity with UML 2.5, at least for tool vendors, is inconsistency in the semantics of the specification itself. The primary focus of the v2.5 release will be cleaning up the specification, which in theory, should lead to better tool interoperability. But to be blunt, I believe that it's time we called BS on modeling tool interoperability. With a few exceptions in the SysML space, tool interoperability has stayed at the "marchitecture" level since the advent of computer-aided software engineering (CASE) tools in the late 1980s; I'll let the marketshare of UML-based Model Driven Architecture (MDA) tools speak for itself. For me, the absolute minimum for tool interoperability means that I can edit a model in tool A, export it to tool B, update it there, and export it back to tool A without loss of information. Furthermore, tool B may not necessarily be a modeling tool. What I really need is this level of interoperability in my entire tool stack, from whatever I use to capture high-level requirements all the way down to running code. Better yet would be tools that truly plug-and-play seamlessly and provide a fully functional solution-delivery stack.

I doubt that we'll ever see meaningful modeling tool operability except in very well-defined spaces, such as where SysML is typically applied. The primary challenges to interoperability aren't technical in nature, they're political. Modeling tool vendors, regardless of their marketing claims, aren't motivated to produce tools that play well with others because that's a great way for them to lose customers. Worse yet, there are even some vendors offering multiple modeling tools that don't interoperate with themselves, let alone another vendor's offerings. The bottom line is that modeling tool interoperability has been "just around the corner" for over a decade and I suspect it will stay there for a long time to come. Follow some of the links provided in the "More UML Reading" section to see my point.

Survey Says…

I'm not the only one who is jaded when it comes to the UML. During the fourth week of October 2013, I ran a mini-survey exploring how people were approaching modeling, including the use of both UML and Business Process Model and Notation (BPMN). There were 162 responses and the survey details can be downloaded free of charge at http://is.gd/BOf4Zu. Every respondent had heard of UML (as I mentioned earlier, UML truly is ubiquitous), but 26% had never heard of BPMN. Only 13% found UML to be very useful, and 45% indicated that UML was useful but they could get by without it. An additional 20% indicated that UML was more trouble than it was worth, and 22% indicated that they don't use UML at all (although 10% of that subset indicated that they had looked at a UML diagram within the past month). In short, the OMG has been very successful in marketing the UML, but not so successful in producing something people find useful.

Perhaps one day, UML-based tools will truly provide the long promised higher level of abstraction that results in a leap in software development productivity. But at this stage, I suspect that any such produc-

IN THIS ISSUE
Guest Editorial >>
News >>
Lean UX >>
Dependency Injection >>
Small Classes >>
Links >>
Table of Contents >>

tivity improvements will come from a very different direction. We'll need to wait and see.

More UML Reading

The OMG's UML v2.5 Beta home page (http://is.gd/65dm5x).

In 2010, Ivar Jacobson and Steve Cook described the need for simplification and support for better tool integration in *The Road Ahead for UML*(http://www.drdobbs.com/224701702).

An ALM Perfect Storm? (http://www.drdobbs.com/240159783) by Adrian Bridgwater discusses Atego's Artisan Studio, which leverages both SysML and UML.

Application Lifecycle Management Meets Model-Driven Development (http://www.drdobbs.com/210300020) by John Carrillo and Scott McKorkle describes their 2008 vision of how ALM and MDD tools can be combined to support the full solution delivery lifecycle. It's five years later; where are the tools?

SysML: The Systems Modeling Language (http://www.drdobbs.com/192700757) by Bruce Powel Douglas overviews the relationhip and differences between UML and SysML.

A Road Map to Agile MDA (http://www.drdobbs.com/184415153) is an article I wrote in 2004 about how to make the OMG's MDA work in practice. This article questioned the viability of the tool-driven, UML-centric vision of the MDA and summarized eight concerns that I had about the MDA. These concerns have stood the test of time.

In *MDA: A Motivated Manifesto* (http://www.drdobbs.com/184415169), written in 2004, Grady Booch described seven reasons why the OMG's MDA strategy should provide a reasonable step up from today's popular development techniques. Those seven points were almost the exact opposite of what I wrote in the Agile MDA article a few months earlier.

The Introduction to UML Diagrams page (http://is.gd/sklakv) provides a brief tutorial on the diagrams of UML 2.x.

— Scott Ambler is a long-time contributor to **Dr. Dobb's** *and is the coauthor of Disciplined* **Agile Delivery: A Practitioner's Guide to Agile Software Delivery in the Enterprise** *(http://is.gd/fc8owe).*

Comment

Symantec Code Signing Certificates
Deliver More Downloads While Building Customer Trust
Click here to learn more.

IN THIS ISSUE

Guest Editorial >>
News >>
Lean UX >>
Dependency Injection >>
Small Classes >>
Links >>
Table of Contents >>

News Briefs

By Adrian Bridgwater

Azure SDK 2.2 Supports Visual Studio 2013

Microsoft has announced the release of the new Azure SDK 2.2 with support for Visual Studio 2013 through an informal but technically extensive blog post. Scott Guthrie, who is corporate vice president in the Microsoft Developer Division, explained that this new version of the SDK is intended to help developers target Azure for every function from creating, configuring, debugging, deploying, and diagnosing cloud services. Version 2.2 of the Windows Azure SDK is the first official version of the SDK to support the final RTM release of Visual Studio 2013 — if you installed the 2.1 SDK with the Preview of Visual Studio 2013, Microsoft recommends that you upgrade your projects to SDK 2.2, although SDK 2.2 also works side by side with the SDK 2.0 and SDK 2.1 releases on Visual Studio 2012. Of note is the Azure Remote Debugging capability, which enables a live debugging session with a remote service hosted in Azure itself. This works whether the deployment is a website, a cloud service, or an application hosted in a Virtual Machine.
http://www.drdobbs.com/240163499

An Extra U For Java on Hadoop

Easy to misspell but doing arguably well with its "scale-out" application server for Apache Hadoop, Continuuity (the company with more letter U power than most) has this month announced the availability of the Continuuity Reactor 2.0. release. The latest version of Continuuity includes MapReduce scheduling, Resource Isolation, High Availability, and full support for REST APIs. Continuuity Reactor 2.0 now what its development is calling a "simplified programming framework and scalable runtime" and is capable of getting Java developers to quickly start building applications on Apache Hadoop and HBase. The platform takes advantage of YARN (Apache Hadoop 2.0), which allows developers to run data-rich applications with support for batch processing and real-time streaming capabilities in a single Hadoop cluster. Continuuity Reactor 2.0 enables developers to run an entire end-to-end application in one platform with an integrated web server.
http://www.drdobbs.com/240163371

Progress Pacific PaaS Is A Wider Developer's PaaS

Progress has used its Progress Exchange 2013 exhibition and developer conference to announce new features in the Progress Pacific platform-as-a-service (PaaS) that allow more time and energy to be spent solving business problems with data-driven applications and less time worrying about technology and writing code. This is a case of cloud-centric data-driven software application development supporting workflows that are engineered to Real Time Data (RTD) from disparate sources, other SaaS entities, sensors, and points within the Internet of Things — for developers, these workflows must be functional for mobile, on premise, and hybrid apps where minimal coding is required such that the programmer is isolated to a degree from the complexity of middleware, APIs, and drivers. Easy, on-demand access to all Pacific capabilities in one place maximizes developer productivity, says the company. It also eliminates complexity for faster implementation and easier workflows, business logic, and data integration. Apps with built-in real-time access to disparate SaaS, relational database, Big Data, social, CRM, and ERP systems can be developed with just one connection using the Pacific rapid application development technology and the Pacific data connectivity service.
http://www.drdobbs.com/240162366

IN THIS ISSUE
Guest Editorial >>
News >>
Lean UX >>
Dependency Injection >>
Small Classes >>
Links >>
Table of Contents >>

Dependency Injection

By injecting methods, it's possible to inject different actions into a process without changing code — the ultimate in loose coupling.

By Mark Seeman

There are several principal patterns in dependency injection, the most common of which is constructor injection. A less common but useful pattern is method injection, which I discuss in this article. When a dependency can vary with each method call, you can supply it via a method parameter, which is the basis of method injection.

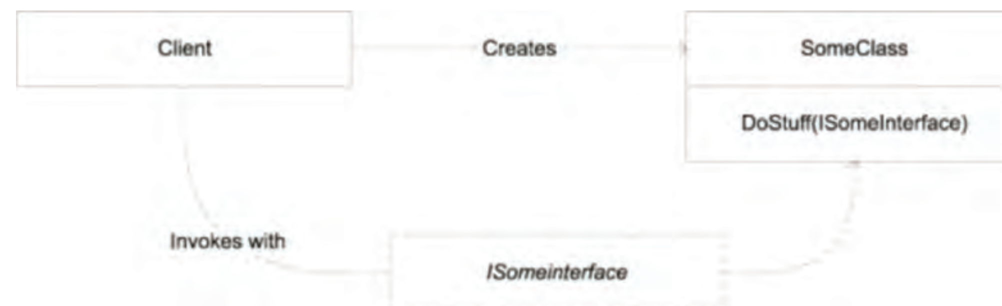


Figure 1: A Client creates an instance of SomeClass, but first injects an instance of the dependency ISomeInterface with each method call.

How It Works

The caller supplies the dependency as a method parameter in each method call. It can be as simple as this method signature:

```
public void DoStuff(ISomeInterface dependency)
```

Often, the dependency will represent some sort of context for an operation that's supplied alongside a "proper" value:

```
public string DoStuff(SomeValue value, ISomeContext context)
```

In this case, the `value` parameter represents the value on which the method is supposed to operate, whereas the context contains information about the current context of the operation. The caller supplies the dependency to the method, and the method uses or ignores the dependency as it best suits it.

IN THIS ISSUE
Guest Editorial >>
News >>
Lean UX >>
Dependency Injection >>
Small Classes >>
Links >>
Table of Contents >>

If the service uses the dependency, it should be sure to test for null references first, as shown in the following code:

```
public string DoStuff(SomeValue value, ISomeContext context)
{
    if (context == null)
    {
        throw new ArgumentNullException("context");
    }
    return context.Name;
}
```

The guard clause guarantees that the context is available to the rest of the method body. In this example, the method uses the context's name to return a value..

If a method doesn't use the supplied dependency, it doesn't need to contain a guard clause. This sounds like a strange situation, but you may need to keep it if the method is part of an interface implementation.

When To Use Method Injection

Method injection is best used when the dependency can vary with each method call. This can be the case when the dependency itself represents a value, but is often seen when the caller wishes to provide the consumer with information about the context in which the operation is being invoked.

This is often the case in add-in scenarios where an add-in is provided with information about the runtime context via a method parameter. In such cases, the add-in is required to implement an interface that defines the injecting methods.

Imagine an add-in interface with this structure:

```
public interface IAddIn
{
    string DoStuff(SomeValue value, ISomeContext context);
}
```

Any class implementing this interface can be used as an add-in. Some classes may not care about the context at all, whereas other implementations will. A client may use a list of add-ins by calling each with a value and a context to return an aggregated result:

```
public SomeValue DoStuff(SomeValue value)
{
    if (value == null)
    {
        throw new ArgumentNullException("value");
    }

    var returnValue = new SomeValue();
    returnValue.Message = value.Message;

    foreach (var addIn in this.addIns)
    {
        returnValue.Message =
        addIn.DoStuff(returnValue, this.context);
    }

    return returnValue;
}
```

The private `addIns` field is a list of `IAddIn` instances, which allows the client to loop through the list to invoke each add-in's `DoStuff` method. Each time the `DoStuff` method is invoked on an add-in, the operation's context represented by the context field is passed as a method parameter.

IN THIS ISSUE
Guest Editorial >>
News >>
Lean UX >>
Dependency Injection >>
Small Classes >>
Links >>
Table of Contents >>

Method injection is closely related to the use of Abstract Factories (http://is.gd/ZSyBQP). Any Abstract Factory that takes an abstraction as input can be viewed as a variation of method injection.

At times, the value and the operational context are encapsulated in a single abstraction that works as a combination of both. Method injection is different from other types of dependency-injection patterns in that the injection doesn't happen in a composition root, but rather, dynamically at invocation time. This allows the caller to provide operation-specific context, which is a common extensibility mechanism used in the .NET BCL.

Known Use

The .NET BCL provides many examples of method injection, particularly in the `System.ComponentModel` namespace. `System.ComponentModel.Design.IDesigner` is used for implementing custom design-time functionality for components. It has an `Initialize` method that takes an `IComponent` instance so that it knows which component it's currently helping to design. Designers are created by `IDesignerHost` implementations that also take `IComponent` instances as parameters to create designers:

```
IDesigner GetDesigner(IComponent component);
```

This is a good example of a scenario where the parameter itself carries information: The component may carry information about which `IDesigner` to create, but at the same time, it's also the component upon which the designer must subsequently operate.

Another example in the `System.ComponentModel` namespace is the `TypeConverter` class. Several of its methods take an instance of `ITypeDescriptorContext` that, as the name says, conveys information about the context of the current operation. Because there are many such methods, I won't list them all, but here is a representative example:

```
public virtual object ConvertTo(ITypeDescriptorContext context,
                                CultureInfo culture,
                                object value,
                                Type destinationType)
```

In this method, the context of the operation is communicated explicitly by the context parameter, while the value to be converted and the destination type are sent as separate parameters. Implementers can use or ignore the context parameter as they see fit.

ASP.NET MVC also contains several examples of method injection. The `IModelBinder` interface can be used to convert HTTP `GET` or `POST` data into strongly typed objects. Its only method is:

```
object BindModel(ControllerContext controllerContext,
    ModelBindingContext bindingContext);
```

In the `BindModel` method, the `controllerContext` parameter contains information about the operation's context (among other things, the `HttpContext`), whereas the `bindingContext` carries more explicit information about the values received from the browser.

If you're building a framework, method injection can often be useful, because it allows you to pass information about the context to add-ins to the framework. That's one reason why we see method injection used so prolifically in the BCL.

Example: Converting Currency

In a sample e-commerce application, a `BasketController`, which

IN THIS ISSUE
Guest Editorial >>
News >>
Lean UX >>
Dependency Injection >>
Small Classes >>
Links >>
Table of Contents >>

manages the user's shopping basked, retrieves the user's preferred currency. I'll show the currency conversion example by converting a `Basket` to the user's currency. `Currency` is an abstraction that models a currency.

```
public abstract class Currency
{
    public abstract string Code { get; }
    public abstract decimal GetExchangeRateFor(
         string currencyCode);
}
```

The `Code` property returns the currency code for the `Currency` instance. `Currency` codes are expected to be international currency codes. For example, the currency code for Danish Kroner is `DKK`, whereas it's `USD` for US Dollars.

The `GetExchangeRateFor` method returns the exchange rate between the `Currency` instance and some other currency. Notice that this is an abstract method, which means that I'm making no assumptions about how that exchange rate is going to be found by the implementer.

In the next section, I'll examine how `Currency` instances are used to convert prices, and how this abstraction can be implemented and wired up so that you can convert some prices into such exotic currencies as US Dollars or Euros.

I'll use the `Currency` abstraction as an information-carrying dependency to perform currency conversions of `Baskets`, so I'll add a `ConvertTo` method to the `Basket` class:

```
public Basket ConvertTo(Currency currency)
```

This will loop through all the items in the basket and convert their calculated prices to the provided currency, returning a new `Basket` instance with the converted items.

Through a series of delegated method calls, the implementation is provided by the `Money` class:

```
public Money ConvertTo(Currency currency)
{
    if (currency == null)
    {
        throw new ArgumentNullException("currency");
    }
    var exchangeRate =
        currency.GetExchangeRateFor(this.CurrencyCode);
    return new Money(this.Amount * exchangeRate, currency.Code);
}
```

The `Currency` is injected into the `ConvertTo` method via the currency parameter (line 1) and checked by the ubiquitous guard clause that guarantees the currency instance is available to the rest of the method body.

The exchange rate to the current currency (represented by `this.CurrencyCode`) is retrieved from the supplied currency and used to calculate and return the new `Money` instance.

With the implementation of the `ConvertTo` methods, I can implement the `Index` method on the `BasketController`:

```
BasketController:
public ViewResult Index()
{
    var currencyCode =
        this.CurrencyProfileService.GetCurrencyCode();
    var currency =
        this.currencyProvider.GetCurrency(currencyCode);
    var basket = this.basketService
        .GetBasketFor(this.User)
        .ConvertTo(currency);
```

IN THIS ISSUE
Guest Editorial >>
News >>
Lean UX >>
Dependency Injection >>
Small Classes >>
Links >>
Table of Contents >>

```
    if (basket.Contents.Count == 0)
    {
        return this.View("Empty");
    }
    var vm = new BasketViewModel(basket);
    return this.View(vm);
}
```

The `BasketController` uses an `IBasketService` instance to retrieve the user's `Basket`. Once you have the `Basket` instance, you can convert it to the desired currency by using the `ConvertTo` method, passing in the currency instance (line 9).

In this case, you're using method injection because the `Currency` abstraction is information-carrying, but will vary by context (depending on the user's selection).

You could have implemented the `Currency` type as a concrete class, but that would have constrained your ability to define how exchange rates are retrieved. Now that we've seen how the `Currency` class is used, it's time to change our viewpoint and examine how it might be implemented.

Implementing Currency

The `Currency` instance is served by a `CurrencyProvider` instance that was injected into the `BasketController` class at class creation by constructor injection. Let's look at this.

```
private readonly IBasketService basketService;
private readonly CurrencyProvider currencyProvider;

public BasketController(IBasketService basketService,
                        CurrencyProvider currencyProvider)
{
    if (basketService == null)
    {
        throw new
            ArgumentNullException("basketService");
    }

    if (currencyProvider == null)
    {
        throw new
            ArgumentNullException("currencyProvider");
    }

    this.basketService = basketService;
    this.currencyProvider = currencyProvider;
}
```

The dependencies (saved as read-only fields in lines 1 and 2) are injected into the controller by the call to the constructor (lines 4-5).

To keep the example simple, let's look at how you might implement `CurrencyProvider` and `Currency` using a SQL Server database and LINQ to Entities. This assumes that the database has a table with exchange rates that has been populated in advance by some external mechanism. You could also have used a Web service to request exchange rates from an external source.

The `CurrencyProvider` implementation passes a connection string on to the `Currency` implementation that uses this information to create an `ObjectContext`. The heart of the matter is the implementation of the `GetExchangeRateFor` method:

```
public override decimal GetExchangeRateFor(string currency-
Code)
{
    var rates = (from r in this.context.ExchangeRates
        where r.CurrencyCode == currencyCode
        || r.CurrencyCode == this.code
            select r)
        .ToDictionary(r => r.CurrencyCode);
    return rates[currencyCode].Rate / rates[this.code].Rate;
}
```

IN THIS ISSUE
Guest Editorial >>
News >>
Lean UX >>
Dependency Injection >>
Small Classes >>
Links >>
Table of Contents >>

The first thing to do is get the rates from the database. The table contains rates as defined against a single, common currency (`DKK`), so you need both rates to be able to perform a proper conversion between two arbitrary currencies. You will index the retrieved currencies by currency code so that you can easily look them up in the final step of the calculation.

This implementation potentially performs a lot of out-of-process communication with the database. The `ConvertTo` method of `Basket` eventually calls this method in a tight loop, and hitting the database for each call is likely to be detrimental to performance.

Related Patterns

Method injection is mainly used when we already have an instance of the dependency we want to pass on to collaborators, but where we don't know the concrete types of the collaborators at design time (as is the case with add-ins).

Note that with method injection, we're on the other side of the fence compared with other dependency injection patterns: We don't consume the dependency, we supply it.

— This article was adapted from Dependency Injection in .NET *(http://is.gd/Ze405H). The book won the Jolt Productivity Award this year (http://is.gd/YkuZwD).*

Comment

IN THIS ISSUE
Guest Editorial >>
News >>
Lean UX >>
Dependency Injection >>
Small Classes >>
Links >>
Table of Contents >>

Lean UX: Making Sure You're Building the Right Product

By continuous discovery of users' needs and sharing the research with the entire team, you take the risk out of new projects.

By Nellie LeMonier

If you're in the business of commercial software development like I am, then I presume you make software because you hope someone will try it, buy it, and find it useful. I further presume that after many users click that "buy" button on your product, you'll start feeling the product is a success. What does it take to get users to click that button? Answering this question is actually more important than building an incredibly sophisticated, intuitive (insert 20 glorious characteristics for) software product. With all the best intentions of their makers, many products will still fail, simply because their makers didn't fully grasp for whom they were making the product, what it was needed for, and what motivated the users.

Successful software products can result from following the design philosophy of Lean UX. This approach requires teamwork with users in collaboration about deliverables, processes, and rules.

Lean UX Is for UX What Agile Is for Developers

At its core, Lean UX is a mind shift in design philosophy that advocates shared understanding between organizations. A key principle of Lean UX is continuous discovery, which encourages customer engagement during the design and development phases — visiting users and learning what they plan to do with your product.

This research into your users' behavior helps create personas, which provide deep insights into user needs. And it validates the product strategy and design.

Unfortunately, many companies using agile methods don't invest in this kind of user research or in a UX strategy because they don't think they have the time to do so. Often, they believe they already know enough about their users. This is rarely the case.

IN THIS ISSUE
Guest Editorial >>
News >>
Lean UX >>
Dependency Injection >>
Small Classes >>
Links >>
Table of Contents >>

Although the quality of the research improves with the frequency of meetings, the number of team members involved, and the breadth of the roles involved, any amount of research tends to uncover fresh revelations.

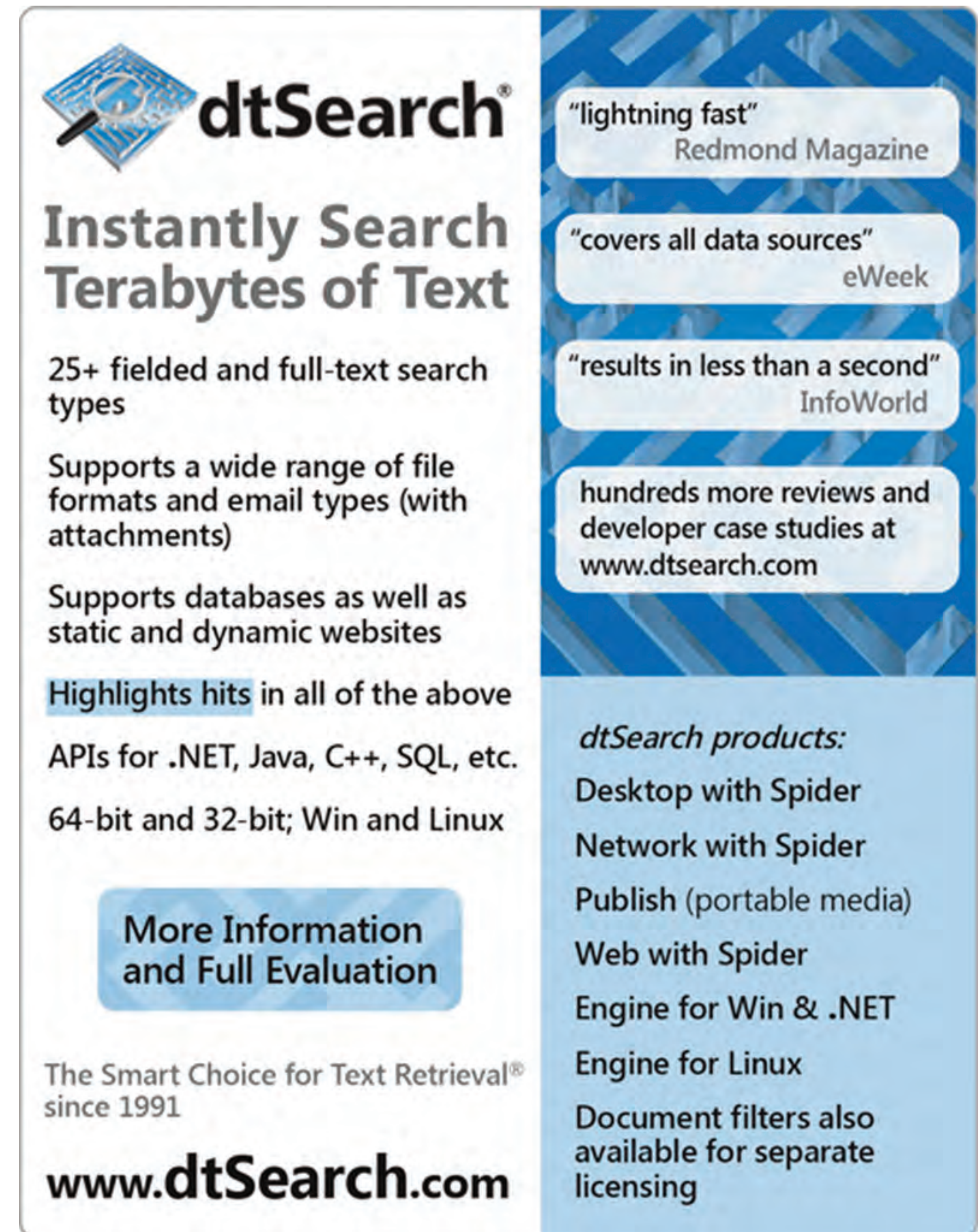
Because developers typically spend months, sometimes years, developing a software product, investing in research to understand customers is not only worthwhile, but essential. Such research doesn't have to take years or months. It can be done in as little as two weeks, during sprint 0.

Make Software for the "U" in UX

More important than, "Why do we make software?" is the question, "Whom do we make software for?" Unless we know who the users are, we're flying blind. The sooner we start user research to determine their needs and goals, the closer we'll be to understanding what they want.

User research has two key steps:

- **Creating personas:** Personas are profiles representing the needs, behaviors, goals, skills, attitudes, and personal characteristics of a user or group of users. Techniques for building personas vary from quick guerilla research to elaborate studies of individual users.
- **Sharing the results:** Everyone on the software development team should have a common understanding of what the end users care about and what they need. Customer personas should be disseminated to the whole team and discussed in depth to ensure cohesion.



IN THIS ISSUE
Guest Editorial >>
News >>
Lean UX >>
Dependency Injection >>
Small Classes >>
Links >>
Table of Contents >>

In *Lean UX: Applying Lean Principles to Improve User Experience* (http://is.gd/cqxJ08), Jeff Gothelf and Josh Seiden sum up the situation elegantly:

"Why do it? Ultimately, the success or failure of your product isn't the team's decision — it's the customers'. They will have to click that 'Buy Now' button you designed. The sooner you give them a voice, the sooner you'll learn whether you've got an idea that's ready to be built."

To ensure your software project doesn't meet the fate of past failed products, consider deploying a UX strategy. You may be surprised to hear what the end users are really looking for and how interesting your product is to them.

Need Help with UX Research?

In practice, there are many ways to do this research. From my own experience, I recommend hiring a graduate or undergraduate student working on a degree in human-computer interaction (HCI), computer science, or user experience design. It's a great way to get short-term help to execute the research plan if your company isn't ready to invest in a full-time researcher. You get assistance in answering questions that need to be addressed, and the interns gain real-world experience in the industry. The interns I've mentored and worked with provided real value by helping define personas and sharing their findings in a meaningful way with the rest of the team.

Lean UX in Practice

The latest project where I was the lead UX designer is a Perforce product called Swarm, which is a flexible code-collaboration platform. On Swarm, we practiced Lean UX from product inception through delivery. Before the project was even a gleam in our eyes, we had already conducted significant customer research and had learned that they were yearning for a new tool from us. They wanted a product to:

- help them speed up the tempo of their development,
- extend the functionality of our existing product,
- bring in social elements that have become standard in software applications.

"UX is truly a team sport and most rewarding when the whole group participates. At this early stage, the feedback helped us adjust priorities."

I created and executed a Lean UX research plan so that we could continuously discover what iterations our product required. This research plan included customer meetings, usability studies, a beta program, and customer surveys. The team gave customers a preview of our product, which included describing primary user scenarios and a demo of the working prototype. All feedback was recorded and shared with the entire team. UX is truly a team sport and most rewarding when the whole group participates. At this early stage, the feedback helped us adjust priorities.

We conducted brief usability studies with internal users whenever new features were added, and the release of our beta program included an extensive usability study. This research helped us identify problem areas with fresh users who had not previously experienced

IN THIS ISSUE
Guest Editorial >>
News >>
Lean UX >>
Dependency Injection >>
Small Classes >>
Links >>
Table of Contents >>

the product. Their feedback carried a lot of weight and helped us learn about usability problems we hadn't encountered.

The beta program was designed to encourage customers to use Swarm in their own environments and provide feedback. Because we

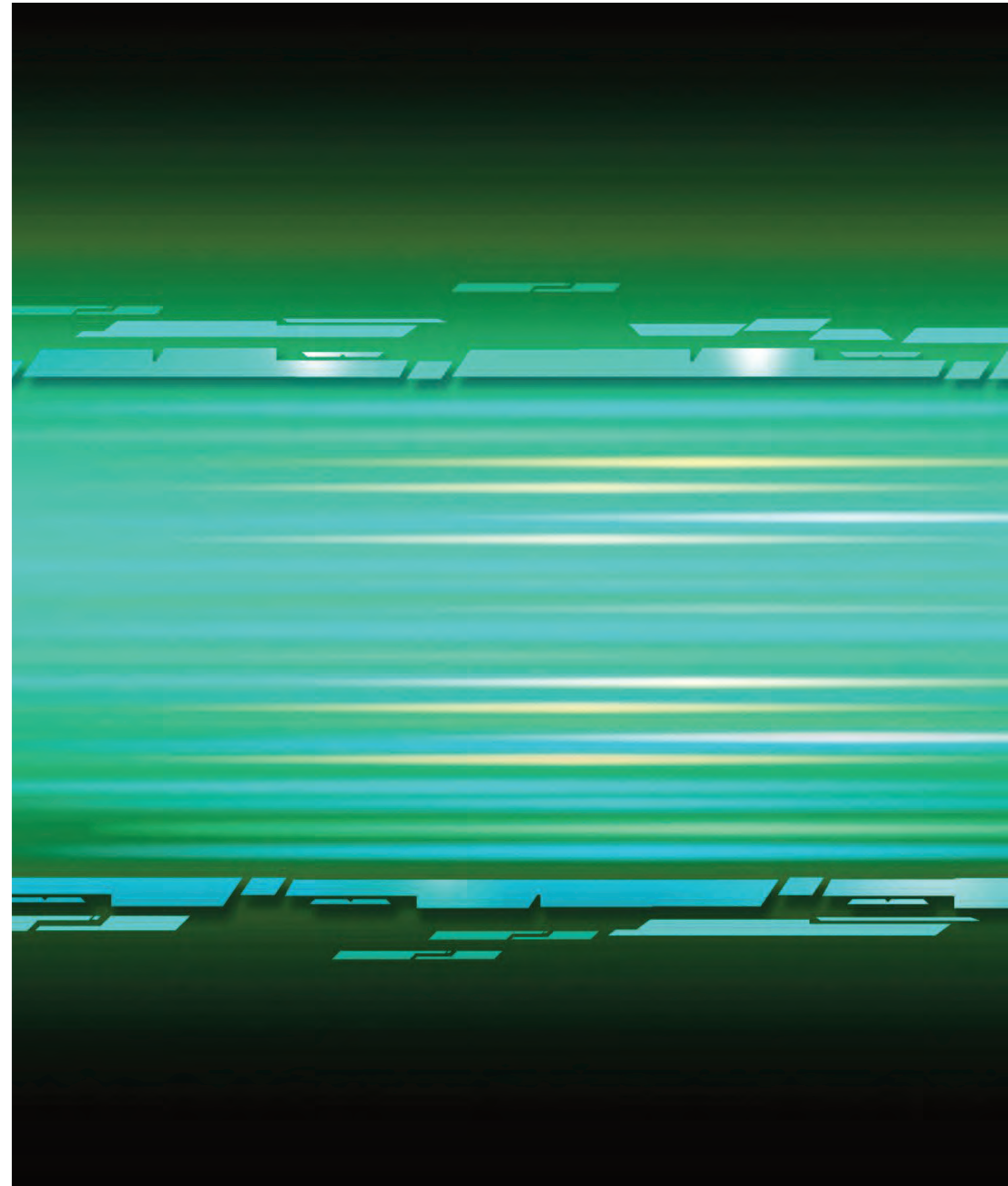
"As we practice Lean UX, we continue to evolve our own internal relationships among our sales, marketing, engineering, and UX teams."

had fostered customer relationships through our customer calls, a high percentage of these customers were interested in the early version of the product. We learned more from the use of our product in their environments than we learned from using it on our own. Finally, the surveys we conducted helped us gain quantitative insights into how to prioritize our features.

As we practice Lean UX, we continue to evolve our own internal relationships among our sales, marketing, engineering, and UX teams. By successfully sharing and increasing our collective knowledge about our users, we make our products more successful because we know it helps people do what they need, which is ultimately what provides value to our company.

— Nellie LeMonier (http://is.gd/XWhcdg) is a user experience researcher and designer

Comment



IN THIS ISSUE
Guest Editorial >>
News >>
Lean UX >>
Dependency Injection >>
Small Classes >>
Links >>
Table of Contents >>

From the Vault

In Praise of Small Classes

Using small classes avoids introducing unnecessary complexity into your code. Now then, how to do it? Start with five not-so-easy steps.

By Andrew Binstock

If you've been doing OO programming for a while, you've surely run into the seemingly endless essays on testability. The issue they debate focuses on how to write code to make it more amenable to automated testing. It's a vein that is particularly intriguing to exponents of test-driven development (TDD), who argue that if you write tests first, as in the orthodox approach to TDD, your code will be inherently testable.

In real life, however, this is not always how it happens. TDD developers frequently shift to the standard code-before-tests approach when hacking away at a complex problem or one in which testability is not easily attained. They then write tests after the fact to exercise the code; then modify the code to increase code coverage. There are good reasons why code can be hard to test, even for the most disciplined developers. A simple example is testing private methods; a more complex one is handling singletons. These are issues at the unit testing level. At higher levels, such as UAT, a host of tools help provide testability. Those products, however, tend to focus principally on the GUI aspect (and startlingly few of those handle GUIs on mobile devices). In other areas, such as document creation, there is no software that provides automated UAT-level validation because parsing and recreating the content of a report or output document is often an insuperable task.

IN THIS ISSUE

Guest Editorial >>
News >>
Lean UX >>
Dependency Injection >>
Small Classes >>
Links >>
Table of Contents >>

I don't want to get off my main point, however, which is that what makes code untestable is frequently not anything I've touched on so far, but rather excessive complexity. High levels of complexity, generally measured with the suboptimal cyclomatic complexity measure (CCR), is what the agile folks correctly term a "code smell." Intricate code doesn't smell right. According to numerous studies, it generally contains a higher number of defects and it's hard — sometimes impossible — to maintain. Fortunately, there are many techniques available to the modern programmer to reduce complexity. One could argue that Martin Fowler's masterpiece, *Refactoring* (http://is.gd/DT6xM3), is almost entirely dedicated to this topic. (Michael Feathers' *Working Effectively With Legacy Code* (http://is.gd/jyApsU) is the equivalent tome for the poor schlemiels who are handed a high-CCR codebase and told to fix it.)

My question, though, is how to avoid creating complexity in the first place? This topic too has been richly mined by agile trainers, who offer the same basic advice: Follow the Open-Closed principle, obey the Hollywood principle, use the full panoply of design patterns, and so on. All of this is good advice; but ultimately, it doesn't cut it. When you're deep into a problem such as parsing text or writing business logic for a process that calls on many feeder processes, you don't think about Liskov Substitution or the Open-Closed principle. Typically, you write the code that works and you change it minimally once it passes the essential tests. In other words, as you're writing the code there is little to tell you, "Whoa! You're doing it wrong."

For that, you need another measure, one which I've found to be extraordinarily effective in reducing initial complexity and greatly expanding testability: class size. Small classes are much easier to understand and to test.

If small size is an objective, then the immediate next question is, "How small?" Jeff Bay, who contributed a brilliant essay entitled "Object Calisthenics" (in the book *The Thoughtworks Anthology*; http://is.gd/mFClsL), which touches on this topic, suggests the number should be in the 50-60 line range. Essentially, what fits on one screen.

Most developers, endowed as we are with the belief that our craft does not and should not be constrained to hard numerical limits, will scoff at this number (or at any number of lines) and will surely conjure up an example that is irreducible to such a small size. Let them enjoy their big classes. But I suspect they are wrong about the irreducibility.

I have lately been doing a complete rewrite of some major packages in a project I contribute to. These are packages that were written in part by a contributor whose style I never got the hang of. Now that he's moved on, I want to understand what he wrote and convert it to a development style that looks familiar to me and is more or less consistent with the rest of the project. Since I was dealing with lots of large classes, I decided this would be a good time to hew closely to Bay's guideline. At first, predictably, it felt like a silly straitjacket. But I persevered, and things began to change under my feet. Here is what was different:

Big classes became collections of small classes. I began to group these classes in a natural way at the package level. My packages became a lot "bushier." I also found that I spent more time in managing the package tree, but this grouping feels more natural. Previously, packages were broken up at a coarse-grained level that dealt with major program components and they were rarely more than two or three levels deep. Now, their structure is deeper and wider and is a useful roadmap to the project.

IN THIS ISSUE
Guest Editorial >>
News >>
Lean UX >>
Dependency Injection >>
Small Classes >>
Links >>
Table of Contents >>

Testability jumped dramatically. By breaking down complex classes into their organic parts and then reducing those parts to the minimum number of lines, each class did one small thing that I could test. The top level class, which replaced its complex forebear, became a sort of main line that simply coordinated the actions of multiple subordinate classes. This top class generally was best tested at the UAT level, rather than with unit tests.

The single-responsibility principle (SRP), which states that each class should do only one thing, became the natural result of the new code, rather than a maxim that needed to apply consciously.

And finally, I have enjoyed an advantage foretold by Bay in his essay: I can see the entire class in the IDE without having to scroll. Dropping in to look at something is now quick. If I use the IDE to search, the correct hit is easy to locate, because the package structure leads me directly to the right class. In sum, everything is more readable; and on a conceptual level, everything is more manageable.

Making Large Classes Small (In 5 Not-So-Easy Steps)

I've discussed the benefits and other effects on code bases of using small classes, which I defined using a limit of 50-60 lines. Note that I'm not discussing a single function, but rather an entire class, which implies multiple functions in most cases. Coding classes as diminutive as 60 lines struck other developers as simply too much of a constraint and not worth the effort.

But it's precisely the discipline that this number of lines imposes that creates the very clarity that's so desirable in the resulting code. The belief that this discipline cannot be consistently maintained suggests that the standard techniques for keeping classes small are not as widely known as I would have expected. (Given that this article was inspired by an extended effort to clean up a project that contains much of my own code, I say this with all due humility.)

Let's go over the principal techniques. I presume in this discussion that design has been done and it's now just a matter of writing the code. Or in the less attractive case, of maintaining code.

Diminish the workload. The first technique to apply is the single responsibility principle (SRP), which states that classes should do only one thing. How big that one thing is will determine in large part how big your classes are going to be. Reduce the work of each class; then, use other classes to marshal these smaller classes correctly.

Avoid primitive obsession. This obsession refers to the temptation to use collections in their raw form. This is definitely a code smell. If you have a linked list of objects, that linked list should be in its own class, with a descriptive name. Expose only the methods that the other classes need. This prevents other classes from performing operations without your knowledge on an object they don't own. The purpose of the list is also supremely clear and this encapsulation enables you to change easily to a different data structure if the need should arise later on.

Reduce the number of class and instance variables. A profusion of instance variables is a code smell. It strongly suggests that the class is doing more than one thing. It also makes it very difficult for subsequent developers to figure out what the class does. Very often, some subset of the variables form a natural grouping. Group them into a new class. And move the operations that manipulate them directly into that class.

Subclass special-case logic. If you have a class that includes rarely used logic, determine whether that logic can be moved to a subclass or even to another class entirely. The classic example of

IN THIS ISSUE
Guest Editorial >>
News >>
Lean UX >>
Dependency Injection >>
Small Classes >>
Links >>
Table of Contents >>

the benefits of object orientation is polymorphism. Use it to handle special variants.

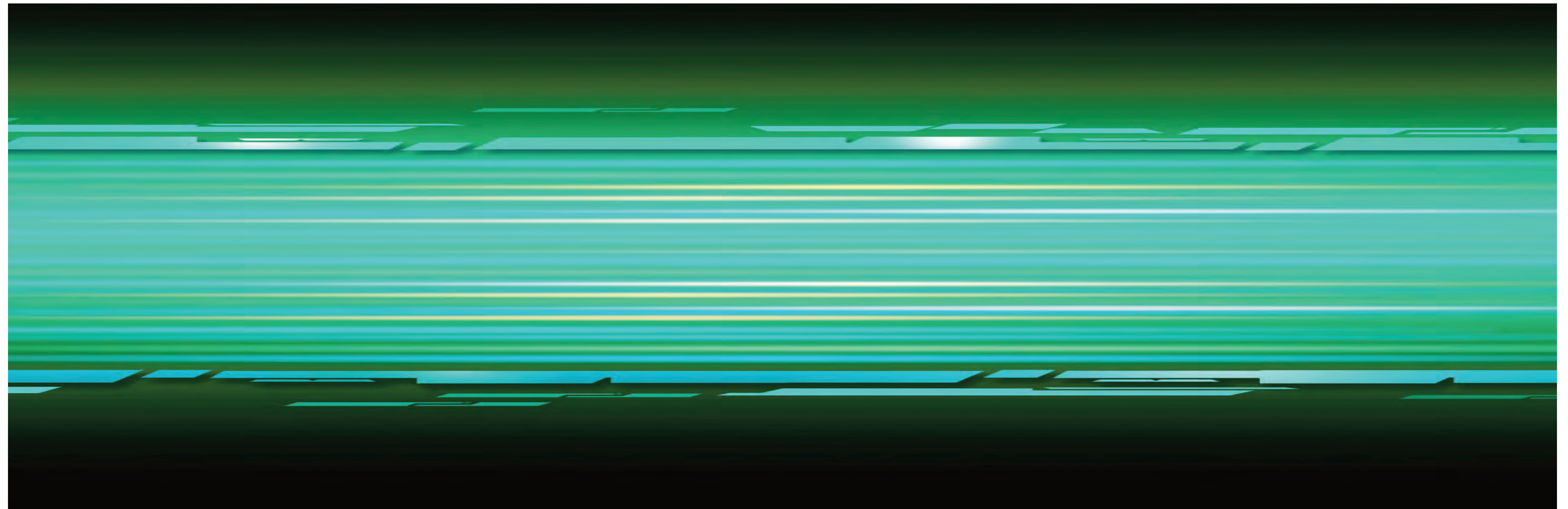
Don't repeat yourself (DRY). This suggestion appears pointlessly obvious. However, even coders who are attentive to this rule will repeat code in two methods that differ only in a single detail. In addition, they can overlook the introduction of duplicate code during maintenance. More than the other guidelines here, which are all techniques, DRY is a discipline within a discipline.

Taken together, these tools get you most of the way to small classes. To see how they are implemented in real life, I once again suggest Fowler's *Refactoring* (http://is.gd/DT6xM3), which is essentially a cookbook of techniques for cleaning up code.

Returning back to my own experience, I am finding that as I insist on this particular discipline in my code rework, my brain is slowly developing a "muscle memory" and is beginning to think automatically about class size prior to class development — and certainly during the cleanup of existing code. Cheers!

— Andrew Binstock
Editor in Chief
alb@drdobbs.com
Twitter: platypusguy

Comment



IN THIS ISSUE
Guest Editorial >>
News >>
Lean UX >>
Dependency Injection >>
Small Classes >>
Links >>
Table of Contents >>

This Month on DrDobbs.com

Items of special interest posted on www.drdobbs.com over the past month that you may have missed

AMDAHL'S LAW VS. GUSTAFSON-BARSIS' LAW

The advantage of parallel programming over serial computing is increased computing performance. Parallel performance improvements can be achieved by way of reducing latency, increasing throughput, and reducing CPU power consumption. Because these three factors are often interrelated, a developer must balance all three to ensure that the efficiency of the whole is maximized.
http://www.drdobbs.com/240162980

HOW DO YOU DECIDE ON INTERMEDIATE STATES?

For both iterative and recursive functions, a key programming strategy is to ask a single question: What information do we need to store while a computation is in progress?
http://www.drdobbs.com/240163033

FLIP FLOP HOLDING

Any of the flip flops that require a clock have two key parameters that are crucial to understand if you are going to do any sort of digital logic design: setup time and hold time.
http://www.drdobbs.com/240163311

MACHINE LEARNING WITH APACHE MAHOUT: THE LAY OF THE LAND

Mahout greatly simplifies extracting recommendations and relationships from input datasets. Here we look at setting up Mahuout and running its recommender on a small data sample.
http://www.drdobbs.com/240163272

BUILD AND DEPLOY OPENCL KERNELS IN PYTHON

Advanced features in PyOpenCL reduce the code required to build kernels for many parallel algorithms.
http://www.drdobbs.com/240162981

SCRATCHING THE ARDUINO ITCH

Even if a young person never actually becomes a practicing programmer, it is valuable to get the logic and problem-solving discipline that a programming project provides.
http://www.drdobbs.com/240162910

1000 RESPONSES TO JAVA IS NOT DYING

When 1000 comments are posted on an editorial, it's worth considering what is being said.
http://www.drdobbs.com/240162680

Previous Next

IN THIS ISSUE
Guest Editorial >>
News >>
Lean UX >>
Dependency Injection >>
Small Classes >>
Links >>
Table of Contents >>

Dr. Dobb's

Andrew Binstock Editor in Chief, Dr. Dobb's
andrew.binstock@ubm.com

Deirdre Blake Managing Editor, Dr. Dobb's
deirdre.blake@ubm.com

Amy Stephens Copyeditor, Dr. Dobb's
amy.stephens@ubm.com

Jon Erickson Editor in Chief Emeritus, Dr. Dobb's

CONTRIBUTING EDITORS
Scott Ambler
Mike Riley
Herb Sutter

DR. DOBB'S EDITORIAL
751 Laurel Street #614
San Carlos, CA
94070
USA

UBM TECH
303 Second Street,
Suite 900, South Tower
San Francisco, CA 94107
1-415-947-6000

UBM TECH
Paul Miller, CEO
Robert Faletra, CEO, Channel
Kelley Damore, Chief Community Officer
Marco Pardi, President, Business Technology Events
Adrian Barrick, Chief Content Officer
David Michael, Chief Information Officer
Sandra Wallach CFO
Simon Carless, EVP, Game & App Development and Black Hat
Lenny Heymann, EVP, New Markets
Angela Scalpello, SVP, People & Culture
Andy Crow, Interim Chief of Staff

UBM TECH EVENT COMMUNITIES
4G World
App Developers Conference
ARM TechCon
Big Data Conference
Black Hat
Cloud Connect
DESIGN
DesignCon
E2
Enterprise Connect
ESC
Ethernet Expo
GDC
GDC China
GDC Europe
GDC Next
GTEC
HDI Conference
Independent Games Festival
Interop
Mobile Commerce World
Online Marketing Summit
Telco Vision
Tower & Cell Summit

INFORMATIONWEEK
Rob Preston VP and Editor In Chief, InformationWeek
rob.preston@ubm.com 516-562-5692

Chris Murphy Editor, InformationWeek
chris.murphy@ubm.com 414-906-5331

Lorna Garey Content Director, Reports, InformationWeek
lorna.garey@ubm.com 978-694-1681

Brian Gillooly, VP and Editor In Chief, Events
brian.gillooly@ubm.com

INFORMATIONWEEK.COM
Laurianne McLaughlin Editor
laurianne.mclaughlin@ubm.com 516-562-5336

Roma Nowak Senior Director,
Online Operations and Production
roma.nowak@ubm.com 516-562-5274

Joy Culbertson Web Producer
joy.culbertson@ubm.com

Atif Malik Director,
Web Development
atif.malik@ubm.com

MEDIA KITS
http://createmarketingservices.com/

UBM TECH
AUDIENCE DEVELOPMENT
Director, Karen McAleer
(516) 562-7833, karen.mcaleer@ubm.com

SALES CONTACTS—WEST
Western U.S. (Pacific and Mountain states) and Western Canada (British Columbia, Alberta)

Sales Director, Michele Hurabiell
(415) 378-3540, michele.hurabiell@ubm.com

Strategic Accounts

Account Director, Sandra Kupiec
(415) 947-6922, sandra.kupiec@ubm.com

Account Manager, Vesna Beso
(415) 947-6104, vesna.beso@ubm.com

Account Executive, Matthew Cohen-Meyer
(415) 947-6214, matthew.meyer@ubm.com

MARKETING
VP, Marketing, Winnie Ng-Schuchman
(631) 406-6507, winnie.ng@ubm.com

Marketing Director, Angela Lee-Moll
(516) 562-5803, angele.leemoll@ubm.com

Marketing Manager, Monique Luttrell
(949) 223-3609, monique.luttrell@ubm.com

Program Manager, Nicole Schwartz
516-562-7684, nicole.schwartz@ubm.com

SALES CONTACTS—EAST
Midwest, South, Northeast U.S. and Eastern Canada (Saskatchewan, Ontario, Quebec, New Brunswick)

District Manager, Steven Sorhaindo
(212) 600-3092, steven.sorhaindo@ubm.com

Strategic Accounts

District Manager, Mary Hyland
(516) 562-5120, mary.hyland@ubm.com

Account Manager, Tara Bradeen
(212) 600-3387, tara.bradeen@ubm.com

Account Manager, Jennifer Gambino
(516) 562-5651, jennifer.gambino@ubm.com

Account Manager, Elyse Cowen
(212) 600-3051, elyse.cowen@ubm.com

Sales Assistant, Kathleen Jurina
(212) 600-3170, kathleen.jurina@ubm.com

BUSINESS OFFICE
General Manager,
Marian Dujmovits
United Business Media LLC
600 Community Drive
Manhasset, N.Y. 11030
(516) 562-5000

Copyright 2013.
All rights reserved.

UNITED BUSINESS MEDIA LLC
Pat Nohilly Sr. VP, Strategic Development and Business Administration
Marie Myers Sr. VP, Manufacturing

UBM TECH ONLINE COMMUNITIES
Bank Systems & Tech
Dark Reading
DataSheets.com
Designlines
Dr. Dobb's
EBN
EDN
EE Times
EE Times University
Embedded
Gamasutra
GAO
Heavy Reading
InformationWeek
IW Education
IW Government
IW Healthcare
Insurance & Technology
Light Reading
Network Computing
Planet Analog
Pyramid Research
TechOnline
Wall Street & Tech

http://createmarketingservices.com/

Entire contents Copyright © 2013, UBM Tech/United Business Media LLC, except where otherwise noted. No portion of this publication may be reproduced, stored, transmitted in any form, including computer retrieval, without written permission from the publisher. All Rights Reserved. Articles express the opinion of the author and are not necessarily the opinion of the publisher. Published by UBM Tech/United Business Media, 303 Second Street, Suite 900 South Tower, San Francisco, CA 94107 USA 415-947-6000.