Chapter 2
Separate Use from Construction

Moving from a procedural approach to an object-oriented one, with all its assumed benefits, tends to add an additional issue: instantiation. Whereas a procedural program or script tends to load, run, and then unload, the creation of object instances can be much more involved, and can continue throughout the runtime lifecycle of the software. Given this, developers often feel that solving "the instantiation" problem is job one. It is decidedly not. Also, typically instances will be created by the same code that uses them, often proximate to their use. This would seem to make the code more readable and easier to work with, but it often has a negative effect.

We will examine these issues, among others that arise when one fails to separate the use of an instance from its construction.

An important question to ask

In our technical classes, we usually ask the question – "when working on a system that is 1-2 years old, and you need to add a new function, there is effort to write the function and then effort to integrate it into the system. Where do you spend most of your time? Writing the function or integrating it in?" About 95% of the time, the answer is "integrating it in."[1]

In our classes, we typically speculate that the integration process looks like: "in my code that is using the new functionality, I have to go through it and see – ah, do I have this case? Yes, then I do this, else I do that. Now here, I have to remember in this case I have this data to use, but in that case, this data means something else." Most of the people in the audience are nodding their heads in woeful agreement. A key problem with this approach is that when you add features to your system it requires you to change your client code to manage these features all the while making the code more complex. While it may not be a problem at the start, it can quickly decay into one.

If this seems familiar to do, you may have come to feel this is inevitable. You may have even been taught that this is an example of "entropy", and that there's really nothing you can do about it.

You may also notice that the cause of this is that while writing/changing/managing client code, you have to be paying attention to the specific, concrete types of the objects you're dealing with. One promise of object-orientated programming is that we shouldn't necessarily have to be referring to objects by type, except in an abstract sense. In other words, we want to hide (encapsulate) the implementing object we are using at any given point in time.

[1] We base this on the answers we've gotten from our students over the last several years. Unfortunately, this ration does not seem to be improving. The main exceptions to this are when: 1) the developers are properly using design patterns, 2) a complete set of automated unit tests is present, 3) the function being written is amazingly complex.

People know that hiding implementation is important – it enables us to change implementations without changing the client code. But if you take a moment to reflect on this, you might realize that it is equally valuable to hide type as well. In other words, if you have two objects that conceptually do the same thing but whose type is exposed when they are used, then the client code couples to those particular objects when it should not. If we can hide the objects' concrete types, we'll make the calling code simpler, and the system more maintainable.

Perspectives

A very simple, common bit of code that tends to appear throughout a system looks something like this:

```
public class BusinessObject {
      public void actionMethod() {
            // Other things
            Service myServiceObject = new Service();
            myServiceObject.doService();
            // Other things
      }
}
```

There's nothing too surprising here. The BusinessObject class uses an instance of Service for part of its implementation. Perhaps it is a Service object that is needed in several places in the system, and so it is implemented in an object. BusinessObject builds an instance of Service, and then delegates to it for the needed behavior.

As natural and simple as this may seem, it is a mistake to be avoided. It creates two different relationships between the BusinessObject class and the Service class. BusinessObject is the creator of Service, but it is also the user of Service. There are significant advantages to be found it breaking these relationships apart.

Perspective of Creation

When one object creates an instance of another in languages like Java and C#, it by necessity must use the "new" keyword to do it. In modern languages, especially those which have automatic garbage collection, "new" is usually not something you can override. In other words, when the code in one object contains "new Widget()", the object that is created as a result is precisely that, an instance of Widget, and no other class.

Also in order for "new Widget()" to be a legal statement, Widget must be a concrete type, i.e., not an abstract class or interface. This means that any entity A that instantiates any other entity B using the "new" keyword directly is coupled to:

1. The actual type that is being instantiated, and
2. The fact that the type is concrete.

We say "coupled" because if either of these things were to change, then the entity doing the creation will have to be altered as well. Going back to our code example, if the class Service were to be changed to an interface, with a single implementing class called Service_Impl, then our code would no longer compile until we changed it to this (the change is in boldface, italic type):

```
public class BusinessObject {
```

```
	public void actionMethod() {
		// Other things
		Service myServiceObject = new Service_Impl();

		myServiceObject.doService();
		// Other things
	}
}
```

Note, also, that we get little or no value from upcasting the reference of Service_Impl to Service, since the code mentions the class Service_Impl anyway; the coupling is unavoidable. Let's contrast this with the perspective that one object has when it *uses* another object.

Perspective of Use

Let's alter our code example slightly. Rather than having BusinessObject build its instance directly, we'll hand it one via its constructor.

You're probably thinking "but wait, *something* had to create it." Just so, but for our purposes here, we'll leave that as an open issue for now, and focus just on this class, and its relationship to Service when we limit it to "use."

```
public class BusinessObject {
	private Service myServiceObject;
	public BusnessObject(Service aService) {
		myServiceObject = aService();
	}

	public void actionMethod() {
		// Other things
		myServiceObject.doService();
		// Other things
	}
}
```

If Service began as a concrete type, and then later we changed it to an abstract type, with a separate implementing class, but no change was made to its interface (the doService() method) what would happen to this code? The answer is: nothing at all. Whatever implementing class was created, it would be implicitly up-cast to Service and used as such.

If, on the other hand, we changed the method "doService()" in some way; changed its name, or what parameters it takes, then this code would again fail to compile until we changed it to call the method in the altered way, whatever that might be.

In other words, if entity A *only* calls methods on entity B, then it is not coupled to which concrete implementation entity B is, nor that it is (or isn't) in fact concrete. It couples to the method signature only.

What You Hide You Can Change

To summarize our points here:

Table 2.1
Coupling by Perspective

Perspective	Coupled to
Creation	The type being created, and the fact that it is a concrete type
Use	The public method signature(s) being called

Creators are coupled to type, while users are coupled to interface. Creators are coupled to what something is, while users are coupled to how something operates. These should be considered separate concerns, because they will often change for different reasons.

The problem with our initial code example is that BusinessObject establishes both of these perspectives relative to Service. It is the creator and the user of the Service, and therefore is coupled to it very tightly. If Service changes in any way other than its internal implementation, BusinessObject will have to change as well.

The more we do this, the more work is required to alter an existing system, and the more likely we are to make mistakes.

“What you hide you can change” is a fairly common mantra among developers who favor encapsulated systems, but this really only makes sense when one defines “can change” correctly. Otherwise, a developer reading such a statement might rightly think “I can change anything I have the source code for.”

When we say “can change” we really mean “can freely change”; can change without having to be hesitant, without having to do extensive investigation of possible side effects, without stress and concern. Another way to say this is “we can change it here, and we don’t have to change it anywhere else.”

Also, we must consider the likely motivations for change.

Why do interfaces change? Ideally, we’d like to create our interfaces from the point of view of those entities that consume their services, rather than from the point of view of any specific implantation code they contain. Thus, an interface will tend to change when the clients of that interface develop a new need.

Why do classes change from concrete to abstract? This usually happens when the design of the solution changes, arguably by adding some form of indirection in order to achieve polymorphism, and/or to break a direct dependency for testing purposes, or for reuse.

These are very different motivations, and would tend to happen at different times in a product’s development lifecycle. Also, if we are circumspect about our interfaces, they will not change very frequently[2]. On the other hand, if we want to avoid over-design in the early stages of development, we will frequently have to accommodate changing/adding/eliminating types later, without creating waste or risk.

Let’s consider another approach:

```
public class BusinessObject {
      private Service myServiceObject;
      public BusnessObject() {
            myServiceObject = ServiceFactory.getService();
      }

      public void actionMethod() {
            // Other things
            myServiceObject.doService();
```

[2] …and even if they do, this likely can be accomplished using an adapter or façade pattern; see our pattern repository as www.netobjectivesrepository.com if you are not familiar with these patterns.

```
            // Other things
      }
}

class ServiceFactory{
      public static Service getService(){
            return new Service();
      }
}
```

If the methods of Service change, this will affect the code in BusinessObject, but not in ServiceFactory (they are not called by the factory). On the other hand, if the Service class itself changes, becomes an abstract class or interface, and if there may even be more than one implementing class, this will change ServiceFactory but not BusinessObject.

A design change is now possible, without changing BusinessObject at all:

```
public class BusinessObject {
      private Service myServiceObject;
      public BusnessObject() {
            myServiceObject = ServiceFactory.getService();
      }

      public void actionMethod() {
            // Other things
            myServiceObject.doService();
            // Other things
      }
}

class ServiceFactory{
      public static Service getService(){
            if (someCondition) {
                  return new Service_Impl1();
            } else {
                  return new Service_Impl2()
            }
      }
}

interface Service {
      void doService();
}

Service_Impl1 : Service {
      void doService() { // one implemementation }
}

Service_Impl2 : Service
      void doService() { // different implemementation }
}
```

What’s the advantage? Are we simply not moving a problem from one place to another? Yes, but consider:

Today, software is becoming increasingly service-oriented (in fact, the term "Service-Oriented Architecture" represents a major movement in most large organizations). Thus, a service may develop many clients over time, creating economies of scale through reuse. In fact, one could reasonably say that the more clients a service serves the more valuable it has proven itself to be.

However, is is far less common for us to create more than one factory for a given service. What we have done is moved our one and only change of existing code into a single, encapsulated place. Also, if the factory does nothing else except create instances, this will be a place with typically less complexity that other objects will tend to have.

Realistic Approach

The point here is that the entity that uses a service should not *also* create it, with the clear implication that "something else" will do so. What that something else is will vary.

In the previous example, we used a separate factory. Sometimes that is warranted. There are other ways:

1. Objects can be created by a tool. Object-Relational mappers, or store-retrieve persistence tools, etc… separate the creation of the object(s) from their consumer(s)
2. Objects can be created in one place, serialized, and then de-serialized in another place by different entities. Again, the two operations are separated.
3. Objects can be created outside the class and passed in via constructors, or using setter() methods. This is often termed "dependency injection."

…and so on. The work required to use these methods are not always justified, however. If we are to do minimal designs, adding complexity only where it is actually warranted, we cannot decide to do any of these things with the anticipation that a class *may* change in the future. After all, anything could change.

Overdesign often comes from the desire to reduce risk in an environment where change is difficult to predict, and where accommodating change by altering a design is perceived as dangerous and leading to decay. However, we want to reduce these concerns about change, while adding the least complexity necessary.

What is needed is a bare-minimum practice that one can always engage in, when there is no justification for anything more complex. Here is what we[3] recommend:

```
public class BusinessObject {
      public void actionMethod() {
            // Other things
            Service myServiceObject = Service.getInstance();
            myServiceObject.doService();
            // Other things
      }
}

class Service {
      private Service(){
            //any needed construction behavior
      }
```

[3] The origin of this idea is hard to determine, but we learned it here: Bloch, Joshua. Effective Java Programming Language Guide. Indianapolis, Indiana: Prentice Hall PTR, 2001

```
	public static Service getInstance() {
		return new Service();
	}
	public void doService() {
		//implementation here
	}
}
```

We have added a few lines of code in Service, and simply changed the “new Service()” that used to be in BusinessObject to “Service.getInstance()”.

This is simple, essentially no additional work, but allows a clean transition to the following design, only when and if it ever becomes needed, and even under that circumstance where many other clients besides BusinessObject are also using Service in this same way:

```
public class BusinessObject {
	public void actionMethod() {
		// Other things

		//No Change!
		Servicec myServiceObject = Service.getInstance();
		myServiceObject.doService();
		// Other things
	}
}

abstract class Service {
	private Service(){ //any needed construction behavior }
	public static Service getInstance() {
		return ServiceFactory.getService()
	}
	abstract void doService();
}

class ServiceFactory{
	public static Service getService(){
		if (someCondition) {
			return new Service_Impl1();
		} else {
			return new Service_Impl2()
		}
	}
}

Service_Impl1 : Service {
	void doService() { // one implemementation }
}

Service_Impl2 : Service
	void doService() { // different implemementation
}
```

Other Practical Considerations

- One cannot always afford to make a constructor literally private. We have done so here in our examples to make it clear that "new" is not to appear in any of the client objects. If you remote objects, or serialize them, etc… you cannot make your constructors private, and thus need to rely on the convention of using only getInstance() in client objects as a best practice.
- Our factory uses a static method to build the object. We did this to keep the code brief, and because we wished to make a single, simple point. However, static methods are not recommended (except for getInstance()), and in practice we tend to make such factories Singletons instead. See http://www.netobjectivesrepository.com/TheSingletonPattern for information on this pattern.
- Sometimes a separate factory is overkill for a simple conditional, and the object creation implementation can be left directly in the getInstance() method. However, this does weaken the object's cohesion.
- Sometimes the creation of the service object is a complex issue in and of itself. In this case, a separate factory is almost certainly warranted, and may in fact be in implementation of a Design Pattern in and of itself.

Timing your decisions

Even when we separate the use and construction issues, each is still a design consideration to be determined. We have to determine the proper objects and relationships for the desired behavior of the system, and how to best get our instances created and "wired together". In each case, we are making design decisions that proceed from our analysis of the problem to be solved.

One question that often arises is: which decisions should we make first: our systemic design (for behavior), or how we will solve the instantiation problems?

It is very important that you hold off on committing to any particular instantiation scheme, until you have a good idea of the design of your system. Determining your objects and their relationships must always come first, and not until you are fairly far along in making these determinations should the topic of object creation be addressed in any way.

Why? There are many different ways to solve the creational problems in OO, and in fact there are a number of well-established patterns, such as the Abstract Factory, Prototype, Singleton, Builder, and so forth. How do we pick the right one?

As in all pattern-oriented design, we must proceed from and be informed by the context of the problem we are solving, and in the case of instantiation, the objects and relationships in the design define this context. You cannot know which creational pattern is best before you know the nature of what is to be built.

Worse, if you do happen to select a pattern creation, perhaps capriciously ("I like Builders"), then almost certainly your system design will be whatever that creational pattern happens to build well, and not the design that is uniquely appropriate to your problem domain.

Those of us who began as procedural programmers often have a hard time accepting this, and adhering to it. We have a hard time resisting the instantiation issue, because it is an issue that did not really exist in the more straightforward load-and-run world of procedural programming.

Nevertheless, we need to be disciplined on this issue, and here is where colleagues can really help. In initial discussions about system design, we must always remind each other not to leap too quickly to instantiation/creation issues.

Using getInstance() in favor of the direct use of the constructor can help. Once this method is in place, it can delegate (in the future) to any sort of factory we might decide upon, with little or no change to the system.

Overloading and C++

If you are wondering what to do when you have to have overloaded constructors, the answer is simple – create overloaded getInstance methods.

Those of you who are C++ programmes may notice the serious memory leaks here (if you are not a C++ programmer, please skip the rest of this paragraph). To solve this, you must add a corresponding static releaseInstance method that you call where you would have called delete on the object. The releaseInstance object must take a pointer to the object being released and it calls delete on the object. Note, however, that by shielding the creation and destruction of the objects, you may find you can get performance improvements by having the getInstance and releaseInstance methods control when things are constructed and deleted. For example, you can convert a regular object to a singleton without the using objects ever noticing (you'd change the getInstance method to as needed for a singleton and you'd have the releaseInstance not do anything).

Validating This for Yourself

In our classes, we often ask pose the following situation and corresponding question to our students. "Imagine we divide this class so those on the left will decide what objects are needed and how they will be used. But, when using them, they don't have to worry about which particular objects they will use in a particular situation. Those people on the right side of the room will write factories that will give those people on the left the proper objects when asked for." That is, those on the left will have the perspective of use while those on the right will have the perspective of creation. "Which side of the room has the easier problem?"

Virtually everyone agrees that merely figuring out the rules for creation is much easier than figuring out which objects are needed, figuring out how to use them and then implementing them. However, now consider how much work the side on the left has to do compared to what they would have to do without this division. It's almost certainly well less than half the work they had before. In other words, if you don't have to consider the particular objects you are using (the side on the right is doing that) your problem becomes significantly easier. Also, notice what happens when you get new functionality – you only have to write it and have the right side change the creating entities. Our biggest problem has just disappeared (integration of new code).

You should notice that this process – just considering the abstract type you have and not the specific implementation – encourages the thinking that "while I am implementing this now, I have other types that may be implemented later." This distinction between an implementation and the concept that it is an implementation of is very useful.

Conclusions

As consultants, we are often called in to help teams that are in trouble. In fact, if a team is "doing great" there would seem little reason to ask for help. Because of this, we see a lot of software systems that have significant challenges.

In your first day onsite as a consultant, you have to quickly find a way to make some kind of positive impact, but you are probably the least knowledgeable person about the system in question at that point, having just been introduced to it. To deliver value quickly, you need to know certain things to look for that a team in trouble will likely have failed to do.

One of the most common mistakes we see is the failure to separate the use of objects from their creation. This happens not because the development team is lazy or foolish, most likely the fact is that this is simply an issue they have not considered, and once they do they can begin to introduce this practice and gain immense benefit from this single change.

There is a lesson in this. Value often flows from aligning ourselves with the essential nature, or truth of something. The fact is that the relationship "I use X" is very different from "I make X", and it is actually more natural to handle them in different places, and in different ways.

I never make soup while bathing the baby, or I'm likely to end up with onions in the shampoo…

Business-Driven Software Development™

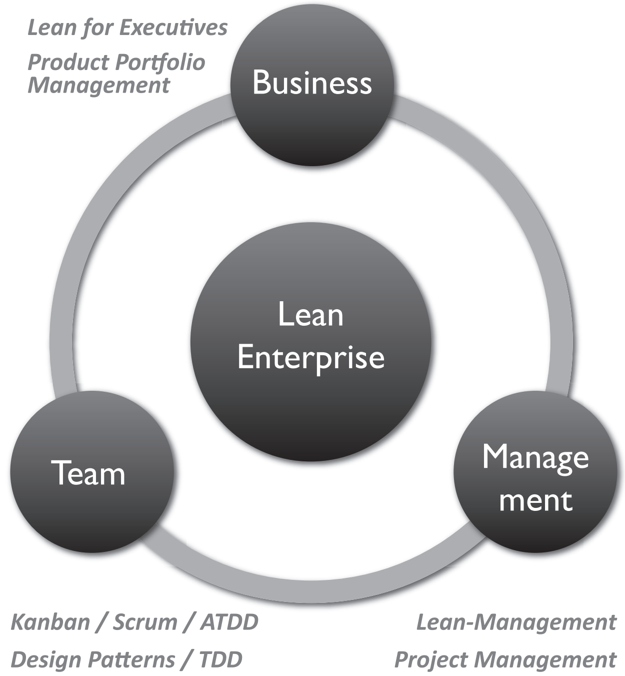
Assessments, Consulting, Training, and Coaching

Business-Driven Software Development (BDSD) is Net Objectives' proprietary integration of Lean-Thinking with Agile methods across the business, management and development teams to maximize the value delivered from a software development organization. BDSD has built a reputation and track record of delivering higher quality products faster and with lower cost than other methods

BDSD goes beyond the first generation of Agile methods such as Scrum and XP by viewing the entire value stream of development. Lean-Thinking enables product portfolio management, release planning and critical metrics to create a top-down vision while still promoting a bottom-up implementation.

BDSD integrates business, management and teams. Popular Agile methods, such as Scrum, tend to isolate teams from the business side and seem to have forgotten management's role altogether. These are critical aspects of all successful organizations. In BDSD:

- **Business** provides the vision and direction; properly selecting, sizing and prioritizing those products and enhancements that will maximize your investment
- **Teams** self-organize and do the work; consistently delivering value quickly while reducing the risk of developing what is not needed
- **Management** bridges the two; providing the right environment for successful development by creating an organizational structure that removes impediments to the production of value. This increases productivity, lowers cost and improves quality



Become a Lean-Agile Enterprise

All levels of your organization will experience impacts and require change management. We help prepare executive, mid-management and the front-line with the competencies required to successfully change the culture to a Lean-Agile enterprise.

Prioritization is only half the problem. Learn how to both prioritize and size your initiatives to enable your teams to implement them quickly.

Learn to come from business need not just system capability. There is a disconnect between the business side and development side in many organizations. Learn how BDSD can bridge this gap by providing the practices for managing the flow of work.

Why Net Objectives

While many organizations are having success with Agile methods, many more are not. Much of this is due to organizations either starting in the wrong place (e.g., the team when that is not the main problem) or using the wrong method (e.g., Scrum, just because it is popular). Net Objectives is experienced in all of the Agile team methods (Scrum, XP, Kanban, Scrumban) and integrates business, management and teams. This lets us help you select the right method for you.

Assessments

See where you are, where you want to go, and how to get there.

Business and Management Training

Lean Software Development

Product Portfolio Management

Enterprise Release Planning

Productive Lean-Agile Team Training

Team training in Kanban, Scrum

Technical Training in ATDD, TDD, Design Patterns

Roles Training

Lean-Agile Project Manager

Product Owner

For more information about Business Driven Software Development Services
Contact Mike Shalloway at Net Objectives: mike.shalloway@netobjectives.com / 404-593-8375
NetObjectives.com / 1-888-LEAN-244 (888-532-6244)