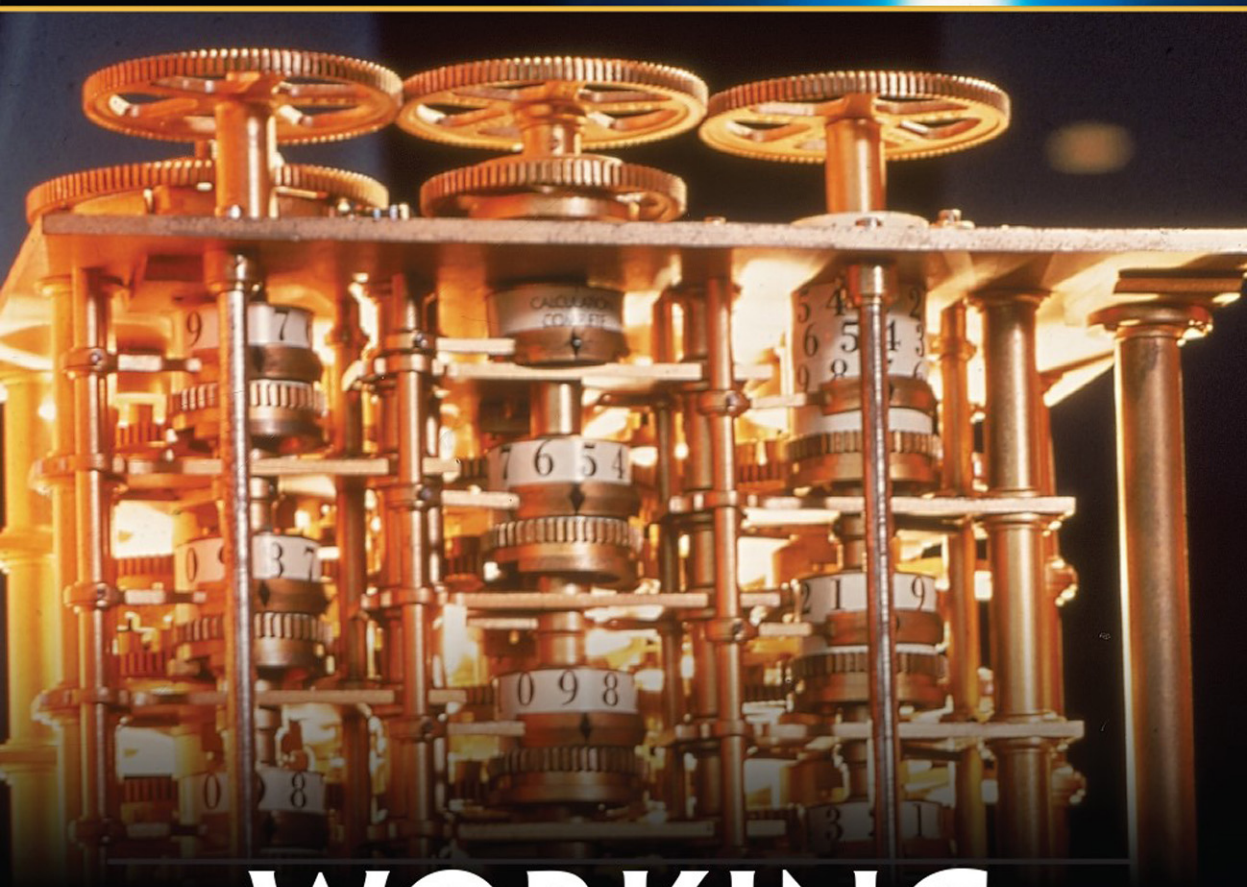
Robert C. Martin Series



WORKING EFFECTIVELY WITH LEGACY CODE

Michael C. Feathers

Working Effectively with Legacy Code

We can create an instance of this class in a test, but it's probably not going to do us much good. First of all, we'll have to link to the mail libraries and configure the mail system to handle registrations. And if we use the `send_message` function in our tests, we'll really be sending mail to people. It will be hard to test that functionality in an automated way unless we set up a special mailbox and connect to it repeatedly, waiting for a mail message to arrive. That could be great as an overall system test, but if all we want to do now is add some new tested functionality to the class, that could be overkill. How can we create a simple object to add some new functionality?

The fundamental problem here is that the dependency on `mail_service` is hidden in the `mailing_list_dispatcher` constructor. If there was some way to replace the `mail_service` object with a fake, we could sense through the fake and get some feedback as we change the class.

One of the techniques we can use is *Parameterize Constructor (379)*. With this technique, we externalize a dependency that we have in a constructor by passing it into the constructor.

This is what the constructor code looks like after *Parameterize Constructor (379)*:

```
mailing_list_dispatcher::mailing_list_dispatcher(mail_service *service)
: status(MAIL_OKAY)
{
    const int client_type = 12;
    service->connect();
    if (service->get_status() == MS_AVAILABLE) {
        service->register(this, client_type, MARK_MESSAGES_OFF);
        service->set_param(client_type, ML_NOBOUNCE | ML_REPEATOFF);
    }
    else
        status = MAIL_OFFLINE;
    ...
}
```


The only difference, really, is that the `mail_service` object is created outside the class and passed in. That might not seem like much of an improvement, but it does give us incredible leverage. We can use *Extract Interface (362)* to make an interface for `mail_service`. One implementer of the interface can be the production class that really sends mail. Another can be a fake class that senses the things that we do to it under test and lets us make sure that they happened.

Parameterize Constructor (379) is a very convenient way to externalize constructor dependencies, but people don't think of it very often. One of the stumbling blocks is that people often assume that all clients of the class will have to be changed to pass the new parameter, but that isn't true. We can handle it like this. First we extract the body of the constructor into a new

method that we can call initialize. Unlike most method extractions, this one is pretty safe to attempt without tests because we can *Preserve Signatures (312)* as we do it.

```
void mailing_list_dispatcher::initialize(mail_service *service)
{
    status = MAIL_OKAY;
    const int client_type = 12;
    service.connect();
    if (service->get_status() == MS_AVAILABLE) {
        service->register(this, client_type, MARK_MESSAGES_OFF);
        service->set_param(client_type, ML_NOBOUNCE | ML_REPEATOFF);
    }
    else
        status = MAIL_OFFLINE;
    ...
}

mailing_list_dispatcher::mailing_list_dispatcher(mail_service *service)
{
    initialize(service);
}
```

Now we can supply a constructor that has the original signature. Tests can call the constructor parameterized by mail_service, and clients can call this one. They don't need to know that anything has changed.

```
mailing_list_dispatcher::mailing_list_dispatcher()
{
    initialize(new mail_service);
}
```

This refactoring is even easier in languages such as C# and Java because we can call constructors from other constructors in those languages.

For instance, if we were doing something similar in C#, the resultant code would look like this:

```
public class MailingListDispatcher
{
    public MailingListDispatcher()
    : this(new MailService())
    {}

    public MailingListDispatcher(MailService service) {
        ...
    }
}
```

Dependencies hidden in constructors can be tackled with many techniques. Often we can use *Extract and Override Getter (352)*, *Extract and Override*

Factory Method (350), and *Supersede Instance Variable (404)*, but I like to use *Parameterize Constructor (379)* as often as I can. When an object is created in a constructor and it doesn't have any construction dependencies itself, *Parameterize Constructor* is a very easy technique to apply.

The Case of the Construction Blob

Parameterize Constructor (379) is one of the easiest techniques that we can use to break hidden dependencies in a constructor, and it is the one that I often turn to first. Unfortunately, it isn't always the best choice. If a constructor constructs a large number of objects internally or accesses a large number of globals, we could end up with a very large parameter list. In worse situations, a constructor creates a few objects and then uses them to create other objects, like this:

```
class WatercolorPane
{
public:
    WatercolorPane(Form *border, WashBrush *brush, Pattern *backdrop)
    {
        ...
        anteriorPanel  = new Panel(border);
        anteriorPanel->setBorderColor(brush->getForeColor());
        backgroundPanel = new Panel(border, backdrop);

        cursor = new FocusWidget(brush, backgroundPanel);
        ...
    }
    ...
}
```


If we want to sense through the cursor, we are in trouble. The cursor object is embedded in a blob of object creation. We can try to move all of the code used to create the cursor outside of the class. Then a client can create the cursor and pass it as an argument. But that isn't very safe if we don't have tests in place, and it could be a big burden on clients on this class.

If we have a refactoring tool that safely extracts methods, we can use *Extract and Override Factory Method (350)* on code in a constructor, but that doesn't work in all languages. In Java and C#, we can do it, but C++ doesn't allow calls to virtual functions in constructors to resolve to virtual functions defined in derived classes. And in general, it isn't a good idea. Functions in derived classes often assume that they can use variables from their base class. Until the constructor of the base class is completely finished, there is a chance that an overridden function that it calls can access an uninitialized variable.

Another option is *Supersede Instance Variable (404)*. We write a setter on the class that allows us to swap in another instance after we construct the object.

```
class WatercolorPane
{
public:
    WatercolorPane(Form *border, WashBrush *brush, Pattern *backdrop)
    {
        ...
        anteriorPanel  = new Panel(border);
        anteriorPanel->setBorderColor(brush->getForeColor());
        backgroundPanel = new Panel(border, backdrop);

        cursor = new FocusWidget(brush, backgroundPanel);
        ...
    }

    void supersedeCursor(FocusWidget *newCursor)
    {
        delete cursor;
        cursor = newCursor;
    }
}
```

In C++, we have to be very careful with this refactoring. When we replace an object, we have to get rid of the old one. Often that means that we have to use the delete operator to call its destructor and destroy its memory. When we do that, we have to understand what the destructor does and whether it destroys anything that is passed to the object's constructor. If we are not careful about how we clean up memory, we can introduce some subtle bugs.

In most other languages, *Supersede Instance Variable (404)* is pretty straightforward. Here is the result recoded in Java. We don't have to do anything special to get rid of the object that `cursor` was referring to; the garbage collector will get rid of it eventually. But we should be very careful not to use the superseding method in production code. If the objects that we are superseding manage other resources, we can cause some serious resource problems.


```
void supersedeCursor(FocusWidget newCursor) {
    cursor = newCursor;
}
```

Now that we have a superseding method, we can attempt to create a `FocusWidget` outside the class and then pass it into the object after construction. Because we need to sense, we can use *Extract Interface (362)* or *Extract Implementer (356)* on the `FocusWidget` class and create a fake object to pass in. It will certainly be easier to create than the `FocusWidget` that is created in the constructor.

```
TEST(renderBorder, WatercolorPane)
{
    ...
    TestingFocusWidget *widget = new TestingFocusWidget;
    WatercolorPane pane(form, border, backdrop);

    pane.supersedeCursor(widget);

    LONGS_EQUAL(0, pane.getComponentCount());
}
```

I don't like to use *Supersede Instance Variable (404)* unless I can't avoid it. The potential for resource-management problems is too great. However, I do use it in C++ at times. Often I'd like to use *Extract and Override Factory Method (350)*, and we can't do that in C++ constructors. For that reason, I use *Supersede Instance Variable (404)* occasionally.

The Case of the Irritating Global Dependency

For years in the software industry, people have bemoaned the fact that there aren't more reusable components on the market. It's getting better over time; there are plenty of commercial and open-source frameworks, but in general, many of them are not really things that we use; they are things that use our code. Frameworks often manage the lifecycle of an application, and we write code to fill in the holes. We can see this in all sorts of frameworks, from ASP.NET to Java Struts. Even the xUnit frameworks behave this way. We write test classes; xUnit calls them and displays their results.

Frameworks solve many problems, and they do give us a boost when we start projects, but this isn't the kind of reuse that people really expected early on in software development. Old-style reuse happens when we find some class or set of classes that we want to use in our application and we just do it. We just add them to a project and use them. It would be nice to be able to do this routinely, but frankly, I think we are kidding ourselves even thinking about that sort of reuse if we can't pull a random class out of an average application and compile it independently in a test harness without doing a lot of work (grumble, grumble).

Many different kinds of dependency can make it hard to create and use classes in a testing framework, but one of the hardest to deal with is global variable usage. In simple cases, we can use *Parameterize Constructor (379)*, *Parameterize Method (383)*, and *Extract and Override Call (348)* to get past these dependencies, but sometimes dependencies on globals are so extensive that it is

easier to deal with the problem at the source. We run into this situation in this next example, a class in a Java application that records building permits for a governmental agency. Here is one of the primary classes:

```
public class Facility
{
    private Permit basePermit;

    public Facility(int facilityCode, String owner, PermitNotice notice)
                throws PermitViolation {

        Permit associatedPermit =
            PermitRepository.getInstance().findAssociatedPermit(notice);

        if (associatedPermit.isValid() && !notice.isValid()) {
            basePermit = associatedPermit;
        }
        else if (!notice.isValid()) {
            Permit permit = new Permit(notice);
            permit.validate();
            basePermit = permit;
        }
        else
            throw new PermitViolation(permit);
    }
    ...
}
```

We want to create a `Facility` in a test harness, so we start by trying to create an object in the test harness:

```
public void testCreate() {
    PermitNotice notice = new PermitNotice(0, "a");
    Facility facility = new Facility(Facility.RESIDENCE, "b", notice);
}
```

The test compiles okay, but when we start to write additional tests, we notice a problem. The constructor uses a class named `PermitRepository`, and it needs to be initialized with a particular set of permits to set up our tests properly. Sneaky, sneaky. Here is the offending statement in the constructor:


```
Permit associatedPermit =
            PermitRepository.getInstance().findAssociatedPermit(notice);
```

We could get past this by parameterizing the constructor, but in this application, this isn't an isolated case. There are 10 other classes that have roughly the same line of code. It sits in constructors, regular methods, and static methods. We can imagine spending a lot of time confronting this problem in the code base.