Strangling Legacy Code

by Mike Thomas

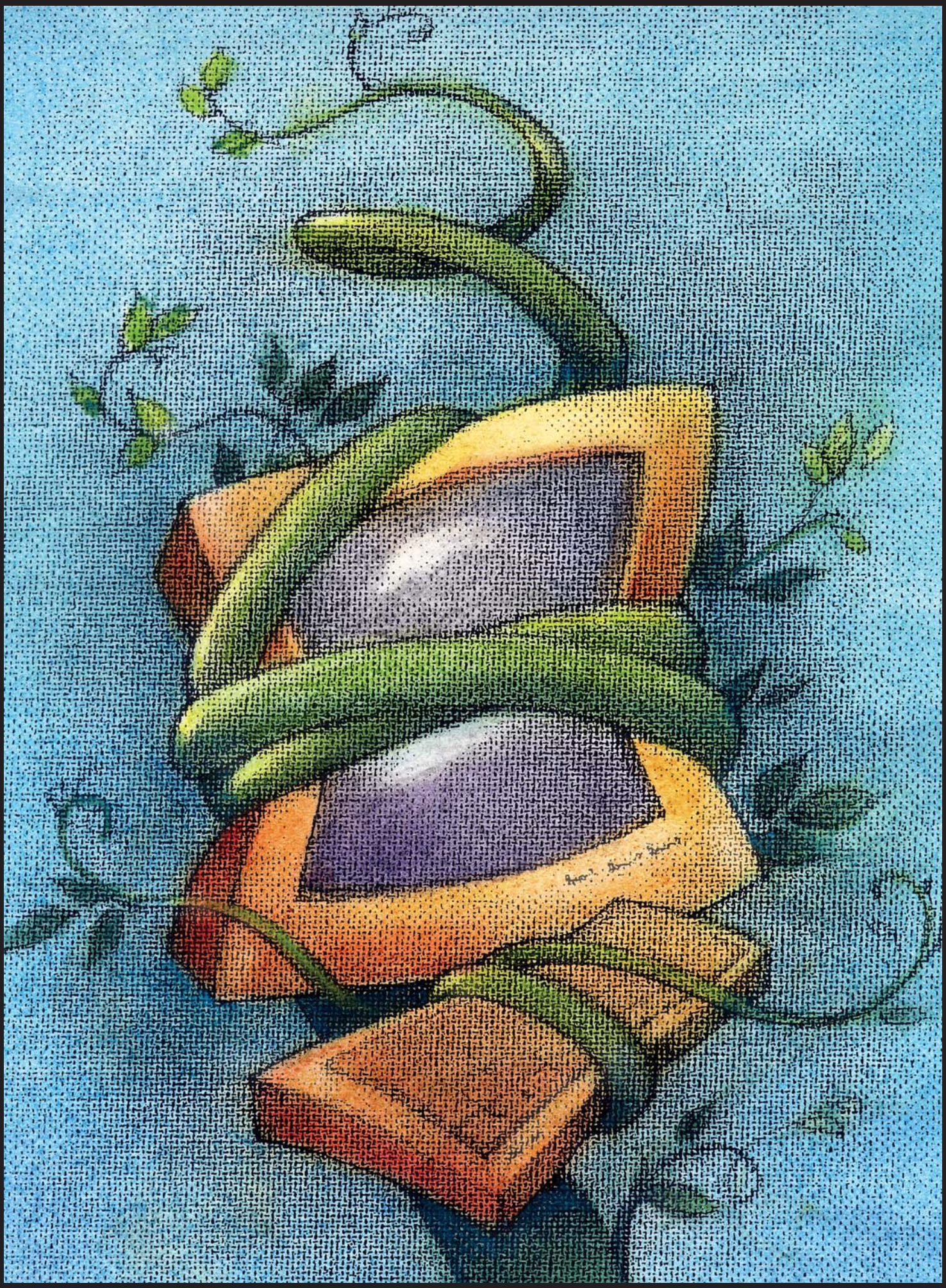
Introduction

The dot com boom generated a lot of Web applications that suffered from poor architecture and design. While these applications may function as required, they are difficult to maintain and extend. Though only a few years old, some already may be considered legacy code by the organizations that maintain them.

Organizations saddled with legacy Web applications often rewrite the applications from scratch—an expensive and even risky proposition. What if an application could instead be rewritten a bit at a time by the same team that maintains it?

This article is partly a case study of work done on my own team's application and partly a catalog of techniques you can try on your own project. Our application is a financial record-keeping product. End-users interact with it over the Web, communicating with servers running at our site. Note that our application is a Java-based Web application, but these ideas may work for other environments.

ILLUSTRATION/THOM BUTTNER



Strangling Defined

Well-known author Martin Fowler coined the term "strangler application" in a Weblog entry on his site. The term is taken from nature: A "strangler vine" establishes itself on a host tree and over a period of time slowly engulfs the tree until the tree dies and only the vine remains.

In the development world, the analogy is clear: A new application overtakes a legacy application a bit at a time until the old application no longer exists. Depending on your situation, this technique can be much more attractive and practical than a "rewrite from scratch" approach.

The Strangling Attitude

Know Your Goals and Values

Strangling is an opportunity to change your development culture. Therefore, before strangling an application, you should have your goals and team values figured out.

Much as a company uses a vision statement to direct corporate decision making, your goals will provide decision-making guidance as the project progresses. At any crossroads in the process you should consider whether a given decision would bring the project closer to its goals.

For us, the primary goal is "agility, leading to competitive advantage." We need to stay ahead of our competition, and if we are held back by a limiting architecture, it is counter to that goal. From this we derived subsidiary goals, such as "promote testability," "increase flexibility," and "decrease complexity." Thus, a given project decision or approach should promote one of these goals.

Your team's values determine how team members carry out day-to-day activities. We take a "pragmatic, not dogmatic" approach to Agile development, as follows:

- Pairing is encouraged, especially on difficult or key code
- Test-driven development (TDD) is preferred
- Avoid rules; use guidelines and judgment
- Value openness and a willingness to change approaches and refactor code
- Quality before development speed
- Strive for continuous improvement
- Testing is key

While we are not dogmatic about most things, we make an exception in the area of testing. Testability is a core value of the team, and our testing team does a great job of reminding us of that fact! Any piece of strangler code has unit tests, FitNesse tests (FitNesse is a free Web-based acceptance testing tool), and UI tests if applicable. When we touch legacy code, we make an effort to introduce tests although this often can be a challenge. (See the StickyNotes for a list of strangling tools.)

Stay on Course

Our high-level development directive is to develop all new features in strangler code while "pouring" existing functionality from the legacy system into the new architecture.

We always knew this would occur in fits and starts. Our metaphor for this is pouring a thick milkshake from one container to another: Sometimes there will be a smooth flow of smaller changes or new features, along with an occasional large, difficult-to-manage chunk of functionality. We also accept that we have to stay realistic—some changes just naturally have to be performed in the legacy system. Each iteration-planning meeting involves discussion about where and how to implement a given feature or bug fix. (See the StickyNotes for suggested metrics to keep your project on course.)

The team has a strong desire to avoid the sins of the past (although team members have differing opinions on what constitutes a "sin"). I have coined a slightly tongue-in-cheek term for this that also may be of use to your project. I call it The Costanza Principle.

The Costanza Principle

Fans of the TV series *Seinfeld* will remember the character George Costanza with a smile. George was the eternally unemployed, neurotic-but-likeable loser who, in an episode titled "The Opposite," discovered the antidote to his loserhood:

George: Yeah, I should do the opposite, I should.

Jerry: If every instinct you have is wrong, then the opposite would have to be right.

We can think of past application development sins as the instincts to be reversed, thus turning our application into a winner. Keeping this principle in mind while strangling our application produced a number of "opposites" that led to the approaches described in the "The Costanza Principle 'Opposites'" sidebar on page 40.

Have Guiding Architectural and Design Principles

If your strangling project is to be a success, you have to know exactly how you're going to avoid the architectural and design problems that led you to strangling in the first place. Therefore, you'll need a strong guiding architecture and effective design guidelines to keep the team in line.

How to Strangle

Strangling an application is a process of pouring existing functionality into a new, better architecture. To promote quality in the strangler application, our team follows architecture and design techniques derived from The Costanza Principle.

Implement a Layered Architecture

One of the main reasons you'd strangle an application is to improve its design. Therefore, it's important to have a guiding architecture. In our case, we developed a layered architecture—a set of "shoeboxes" into which we organized our code.

A layered architecture promotes the principle of "separation of concerns." That is, each layer has one—and only one—architectural responsibility (as opposed to functional responsibility) within the larger structure of the application. Because each layer has only one responsibility, the code is simpler and more direct. A developer looking at the data access objects (DAO) layer knows to expect only database-oriented code there.

In general, a given layer can access only the layers directly below it, which helps to manage dependencies. Good dependency management results in fewer side effects when changes are made. Thus, the system is easier to understand, modify, and test. (See Figure 1.)

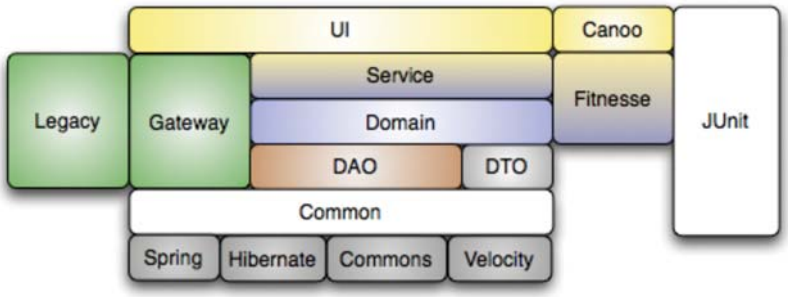


Figure 1: A layered architecture. Layers are generally organized top to bottom—most specific to most general.

Our strangler application is separated into fairly traditional layers. Among these are the user interface, services, and DAO layers. Additionally, we treat our legacy system as an independent layer. With the exception of the legacy layer, many properly architected business applications have layers similar to these.

The service layer implements the business and application logic in a manner transparent to the UI layer. The implementation of the service layer may call the legacy gateway in the process of servicing a request; no strangler code may call legacy code directly.

However, legacy code is free to use strangler code. To paraphrase Dr. Strangelove, this is not only acceptable—it is essential. We often will deprecate legacy code in favor of strangler code. Legacy code that depended upon the functionality provided by the deprecated code must now call strangler code.

Use Careful Physical Design to Control Dependencies

The careful layering we've designed can easily be undermined by circular dependencies. Even though we've drawn a picture—and have made a rule that a given layer can only call the layer beneath it, never the layer above it—Java won't prevent that rule from being broken. To address this problem, physical design (how an application's code is laid out in the filesystem) has to be considered.

Most projects use a single large tree for all Java source in the application. This is convenient because the source can be compiled with one `javac` command. However, this approach does not help to enforce our layering rules. It is better to use a physical design that prevents breaking layer rules. Placing each layer's code into a separate source directory and separately compiling each layer's code achieves this.

Using the layers laid out in Figure 1, we'd have directories for the user interface, services, domain objects, DAOs, common, and legacy code, each of which would be compiled separately. The nature of the dependencies between these layers gives us a strict compilation order. (See the StickyNotes for information on dependencies, inversion of control, and lightweight containers.)

For example, the common package is accessible to all other code (as noted in Figure 2). Therefore, it must be compiled first. If any code is placed in the common package that has dependencies elsewhere, the build will fail, preventing the creation of a disallowed dependency.

Similarly, we build our legacy code last because we've decided to allow legacy code access to all other code. On the other hand, no strangler code is allowed access to legacy code. The build order enforces this rule. (See Figure 2.)

Depend on Interfaces, not Classes

Using Java interfaces is an important way to reduce coupling between Java classes. Reduced coupling leads to more flexible and testable systems.

A great place to employ interfaces is in our layered architecture, at layer

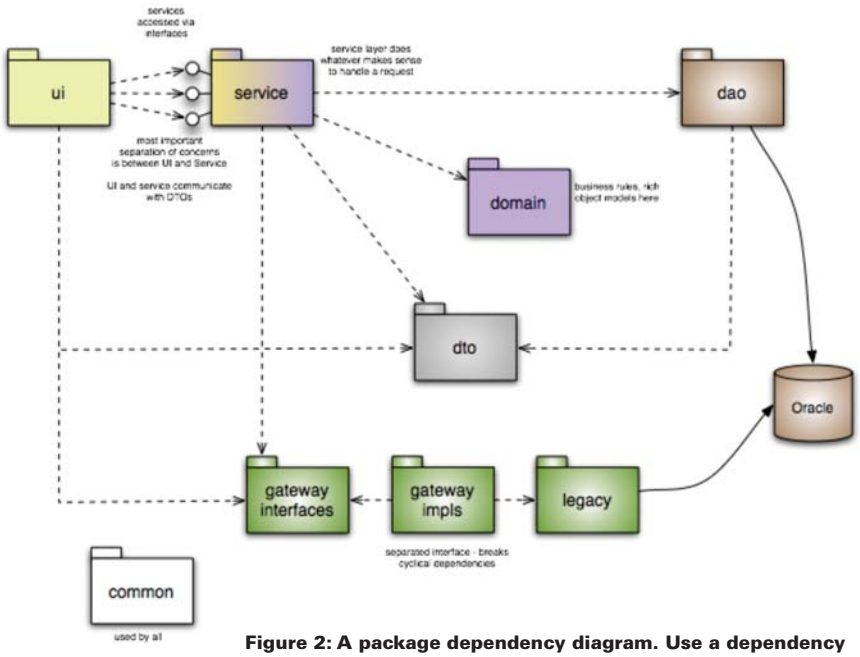


Figure 2: A package dependency diagram. Use a dependency diagram to analyze dependencies. Follow the arrows to determine what must be compiled first.

the common package

is accessible to all other code. Therefore, it must be compiled first.

boundaries. The layers expose only a set of interfaces describing the services that the layer provides. The client of the layer knows only about the interface and therefore is not coupled to any specific implementation of that interface.

Here is a simple Java interface defining a part of a "plan service."

```
public interface PlanService {
  void createPlan(Plan plan);
  // ...
}
```

Note that the interface defines only the operations and not the implementations of those operations.

The production of this interface can be implemented as follows:

```
public class PlanServiceImpl implements PlanService {
  private EmailService emailService;
  private PlanDao planDao;

  public void createPlan(Plan plan) {

    // ... perhaps manipulate the plan in some way ...

    planDao.create(plan);
    emailService.send(new PlanEstablishmentMessage(plan));
  }

  // ...
}
```

Here, the plan is stored and an email message is queued.

The EmailService and PlanDao are interfaces, which allow easy substitution of components. The PlanDao represents a component that can persist a `Plan` object. There are many Java persistence mechanisms. Because PlanServiceImpl does not depend on any of them—only on the common PlanDao interface—you can pick a new mechanism at any time, without having to change client code like `createPlan()`. Likewise, the EmailService interface describes any component that can accept an email message. For testing purposes, you can easily substitute a "mock" implementation of EmailService that doesn't really send the email. That makes testing `createPlan()` much easier and faster.

Leverage URL Integration

It's easy to forget that a great feature of the Web is location transparency. We take it for granted when browsing from site to site. However, a "site" can easily be composed of many bits of content served up from diverse servers—and even different domains. You can leverage this transparency in your own application to have legacy code interact with strangler code (and vice versa) via URL integration.

Suppose you have a feature implemented in legacy code and you wish to replace it with strangler code. Rewrite the feature in the strangler, and then ensure that all links and redirects are aimed at the strangler.

This approach can be taken quite far. For example, in our system we needed to rewrite a piece of functionality that had a "wizard" interface. Since we didn't have time to rewrite every page in the wizard, we used URL integration to jump back and forth between legacy and strangler pages and controllers. (See Figure 3.) We used session variables to track state between the two "sides" of the feature. Due to the magic of the Web, the weaving of new with old code was transparent to the user.

Build a Legacy Gateway

So, we're replacing legacy code with strangler code. But in many cases the strangler code is not complete—there are times that it needs access to features that haven't yet been strangled. In these cases, the strangler needs access to legacy code. However, it would be a mistake to give the strangler direct access; we don't want our strangler to become too dependent on its host. To accomplish this, we insert an architectural layer called the gateway.

The gateway provides access only to those pieces of legacy functionality that the strangler needs—and no more. By not coupling directly to legacy classes, the strangler application stays dependent only upon legacy features—and not at all on legacy implementations.

An example from our system: The legacy system provides a way to send form email to users of the system (for example, when a user's password is changed). When we strangled the password-reset feature of the system, we weren't ready to rewrite the email functionality as well. So, we built a gateway to the legacy system's email functionality. The UserService calls the EmailGateway, which in turn calls the EmailMessage legacy code.

The production implementation of the gateway talks to the legacy code, whereas mock implementations are used in unit testing. Someday, when we strangle the legacy email code, the production gateway implementation can be replaced with code that talks to strangler code instead of legacy code. Alternatively, if the

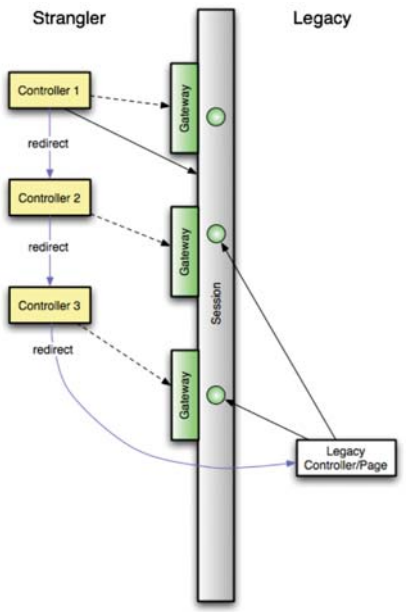


Figure 3: URL integration. The green balls represent shared state. Strangler controllers manage shared state only through the gateway. Local strangler state can be managed directly (as Controller 1 is doing, indicated by the solid arrow from it to the state).

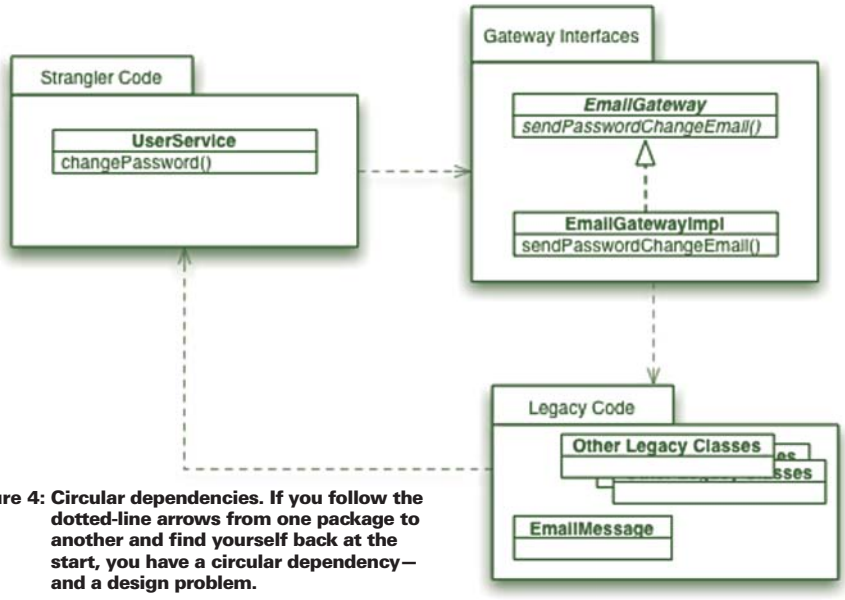


Figure 4: Circular dependencies. If you follow the dotted-line arrows from one package to another and find yourself back at the start, you have a circular dependency—and a design problem.

gateway is called from only a few places, those places can just be recoded to use the strangler's functionality directly.

The gateway presents a possible problem. If any legacy code can call the strangler code directly, and the strangler code can call the legacy code via the gateway, it's easy to end up with a circularity. (See Figure 4.) Packages at the heads of arrows must be compiled before those at the tails. So which package gets compiled first?

The solution is the Separated Interface pattern from Fowler's *Patterns of Enterprise Architecture*. It's shown in Figure 5. The strangler code depends on the gateway interface—not on the implementation. The implementation depends both on the legacy code and the gateway interface.

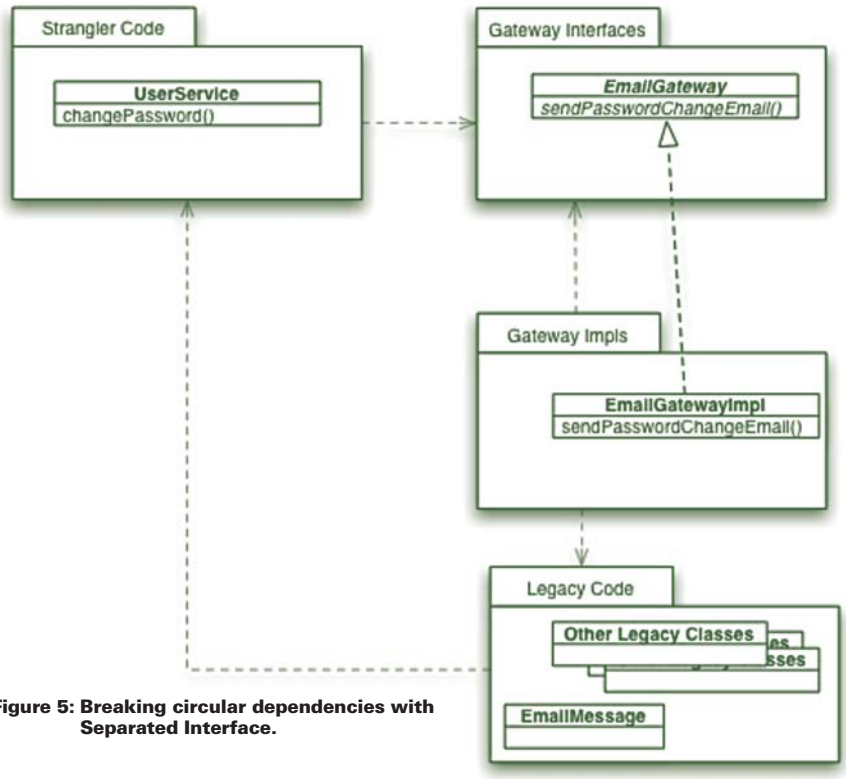


Figure 5: Breaking circular dependencies with Separated Interface.

The circular dependency is now broken, and we can compile in this order: first the gateway interfaces, then the strangler code, then the legacy code, then the gateway implementations.

Leverage Strangler Features from Legacy Code

According to our strangling directive, we want to develop all new features in strangler code. But sometimes a legacy feature needs to be extended rather than replaced, perhaps due to scheduling pressure or other outside forces. Extending the legacy code directly in the legacy code base has the undesirable side effects of adding to the legacy code base and creating future work to move the new functionality to the strangler application.

In most cases all is not lost. Recall that we allow legacy code to access strangler code. If we design our extensions carefully, we can extend the legacy code only enough to delegate to strangler code.

The techniques used to delegate to strangler code obviously vary by context. I'll give a couple of examples from our system.

Replace Legacy Implementation

Probably the simplest way to leverage strangler features from legacy code is to "hollow out" a given legacy class, keeping its external interface the same, and replacing the old implementation with delegation to a strangler implementation.

We have used this approach in a few of our legacy "row gateway" classes. These classes have an instance variable for each column in the related database table and have methods corresponding to database operations (create, read, update, delete). In a number of cases we have replaced the legacy instance variables with a reference to a strangler domain object and delegated the database operation methods to strangler DAO code. This ensures that the data and operations are implemented in only one place—the strangler.

Introduce Delegating Subclass

Our application has an algorithm to generate financial transactions. The algorithm operates on a list of "processing instructions." These instructions are subtyped into several flavors of instruction that the transaction generation

algorithm calls polymorphically.

We recently had to introduce a new type of instruction. The easiest thing would have been to implement a new instruction subtype in the legacy code, using the already existing instruction framework. However, this would only increase our legacy code base. On the other hand, we couldn't possibly move the entire transaction generation algorithm to the strangler application.

We solved the dilemma by extending the hierarchy of legacy processing instructions to include the new instruction subtype. However, the implementation of this new subtype didn't call into legacy code the way all the other subtypes did. Rather, this new instruction subtype delegated to a new, minimal transaction generation framework that we implemented in the strangler code.

Over time we'll be able to use Replace Legacy Implementation to move legacy instruction types to use the strangler transaction generation framework. Eventually, when the strangler handles the majority of the instruction subtypes, we'll move the remainder of the transaction generation feature to the strangler.

Some Lessons

While our strangling effort is moving along well, it hasn't all been easy. My team has learned some lessons along the way. Here are a couple worth relating:

Sometimes it's better to bite off more—rather than less—when strangling. Sometimes trying to integrate small pieces involves handling details that could be ignored when integrating large pieces. For example, while our URL integration approach worked out well in the "wizard weaving" implementation, once we were finished we felt as though it might have been better to just replace the whole wizard. Of course, this was hindsight, but it's still worth keeping in mind.

Be ever vigilant in protecting your values while working on the strangler application. It's trite—but old habits really do die hard. We still wrestle occasionally with short-term convenience versus long-term maintainability and quality. **{end}**

The Costanza Principle "Opposites"

Processes

Status Quo	The Opposite
Little contact with stakeholders	*"High touch" environment where stakeholders are continuously involved*
Lengthy iterations, up to six months	*Short, two-week iterations, based on SCRUM process*
Manual builds	*Continuous and nightly builds*
Little developer testing, no automated tests	*Lots of developer testing, tending toward test-driven development (TDD). Many automated tests, both JUnit and FitNesse*

Architecture and Design

Status Quo	The Opposite
UI and business rules mixed	*Layered architecture* *Employ Model View Controller pattern*
High coupling	*Code to interfaces* *Layered architecture* *Inversion of control*
Overuse of singletons and session variables to get access to collaborating classes	*Use an* inversion of control *framework (the Spring Framework, in our case) to resolve collaborators transparently*
Unmanaged dependencies	*Layered architecture* *Careful physical design*
String and integer type codes	*Typesafe enumerations* *Replace type code with inheritance*
No developer tests	*Lots of developer tests (JUnit and FitNesse)*

Mike Thomas has worked in IT since 1986. He has been employed both as an employee and as a consultant on a variety of computing platforms—from mainframes to micros—for companies varying in size from eleven to sixty thousand employees. Since 1999 Mike has concentrated on building enterprise systems in Java. Mike maintains an extensive personal Web site at http://www.samoht.com and can be reached at mike_thomas@yahoo.com.

Sticky Notes

For more on the following topics, go to www.StickyMinds.com/bettersoftware

- Strangling tools
- Metrics
- Information on dependencies, inversion of control, and lightweight containers

Find out more about strangling legacy code. Log on to www.StickyMinds.com to join Mike Thomas's RoundTable discussion.