Behead Your Legacy Beast

Refactor and Restructure relentlessly with

The Mikado Method

Daniel Brolund
Ola Ellnestam

Foreword by Tom Poppendieck

The Mikado Method: Kata in Java

Daniel Brolund, Ola Ellnestam

Friday 9th May, 2014

Contents

Chapter 1

An example in code

Jake:	Melinda, I have this problematic code and I need to add new functionality for a client. Can you help me?
Melinda:	I would love to. Do you use version control for your code?
Jake:	Yes?
Melinda:	Great, we'll use that. Have you ever done small, atomic, refactorings, like the "Move method" or "Extract interface"?
Jake:	Yes, a little.
Melinda:	Good, we will probably use those as well.
Jake:	So, where do we begin?
Melinda:	Well, what is your goal?
Jake:	Goal?
Melinda:	Let me rephrase that. How do you know when you're done?
Jake:	When I have a deliverable for the new client.
Melinda:	Sweet, that will be our Mikado Goal and we'll work from there.
Jake:	Just like that?
Melinda:	Just like that.

1.1 New business for Pasta Software.

Pasta Software is a small family-owned company. By good luck and coincidence, their pride and joy MasterCrüpt (TM), has been sold to a new customer.

But now, problems arise. The very, very secret obfuscation algorithm in the application cannot be exposed between the old customer Gargantua Inc and the new customer, Stranger Eons Ltd.

Without really knowing it, Pasta Software has just put themselves into technical debt by changing their business model, i.e getting a customer with slightly different needs. They did not see this coming and the code isn't flexible enough to offer an easy way out. The design has to change so that confidential information doesn't leak between their customers. That would be the end of their business.

> *It is virtually impossible to tell how a system will evolve in advance. Trying to anticipate all future changes to a system and make the code flexible enough to handle them only makes the codebase bloated. In fact, that kind of over-design makes the code more complex and will likely become a problem in itself.*

1.1.1 The codebase

The code at Pasta Software is written in Java. This is an overview of the classes in the system:

```
package mastercrupt;

import static org.junit.Assert.assertEquals;
import mastercrupt.UI;

import org.junit.Test;
public class AcceptanceTest {
  @Test
  public void testLeeting() throws Exception {
    UI ui = new UI();
    assertEquals("Leeted:␣S3cr3t", ui.leetMessage("Secret"));
  }
}
```

Listing 1.1: The AcceptanceTest class

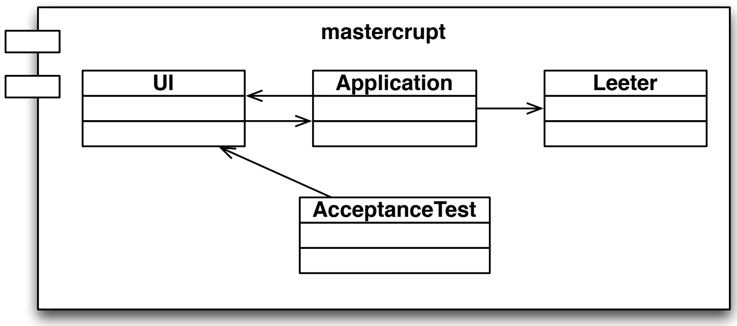


Figure 1.1: Class diagram of the existing system

Lets start with the test case, as in *Listing 1.1(p.3)*.

This is the acceptance test for the application as of today. It simply creates a `UI` instance and uses that to obfuscate a message. The algorithm used for obfuscation is a *leetspeak* algorithm.

> ***Leetspeak** is the way hackers and crackers avoided text filters on Bulletin Board Systems (BBS) in the eighties by re-*

```
package mastercrupt;

public class UI {
    Application application = new Application();
    private String leeted;

    public String leetMessage(String unLeeted) {
        application.leet(unLeeted, this);
        return "Leeted:␣" + leeted;
    }

    public void setLeeted(String leeted) {
        this.leeted = leeted;
    }
}
```

Listing 1.2: The UI class

> *placing alphabetic characters with non-alphabetic, but resembling, characters. For instance, in one leet dialect 'leet' is translated to 'l33t', where '3' is the mirrored capital 'E'.*

The `UI` class in *Listing 1.2(p.4)* creates an `Application` instance.

The `Application` instance is called with the string to leet and a reference to this `UI` class. The `Application` instance, see *Listing 1.3(p.5)*, is supposed to call back to `setLeeted(String leeted)` to set the leeted `String` value.

The `Application` class uses the `Leeter` class, see *Listing 1.4(p.5)*, to leet the given string and calls back to the provided `UI` instance with the leeted value. The `Application` class also contains the main method for the application.

The `Leeter` simply performs the leeting of the incoming message by substituting the character 'e' with the character '3'.

That is all the code there is in this application. As we can see, it only takes a few classes to create a mess. For instance there's a circular dependency between `UI` and `Application`, not to mention the mysterious callbacks.

In reality, codebases are a lot bigger and more complex by nature, so the ways of creating a mess are almost unlimited. The principles for the

```java
package mastercrupt;

public class Application {
    public void leet(String string, UI ui) {
        ui.setLeeted(Leeter.leet(string));
    }
    public static void main(String[] args) {
        UI ui = new UI();
        System.out.println(ui.leetMessage(args[0]));
    }
}
```

Listing 1.3: The Application class

```java
package mastercrupt;

public class Leeter {
        public static String leet(String message) {
                return message.replace('e', '3');
        }
}
```

Listing 1.4: The Leeter class

Mikado Method can still be applied though, no matter how big or small the codebase is. The method serves as a guide and helps to identify the critical change path in order to be able to deliver. Lets see what we need to do with this particular piece of code in order to create a new deliverable for Stranger Eons Ltd.

1.1.2 Starting out

Before making any changes to the code, we make sure that everything works, we *Back out broken code (p.**??**)*. The code compiles, all the existing tests run and there are no checked out files nor uncommitted changes. We start from a clean state. We could also label/tag the current state in the Version Control System (VCS) with *"Before the new client"*, or something similar, to be able to return to it later if we need to.

Now, our goal is to create a system that can be delivered to Stranger Eons Ltd. We will use the term 'New deliverable for Stranger Eons Ltd', and we *Write down the goals (p.**??**)* as in *Fig.* 1.2 *(p.*6*)*.

Mikado Goal: Mikado Goal: new deliverable for Stranger Eons Ltd

New deliverable for
Stranger Eons Ltd

Figure 1.2: New deliverable for Stranger Eons Ltd

In the Java world, that means we will probably have a separate project source root from which we then create a JAR-file for Stranger Eons Ltd. We add this as a prerequisite to our business goal, the root of the Mikado Graph, as in *Fig.* 1.3 *(p.*7*)*.

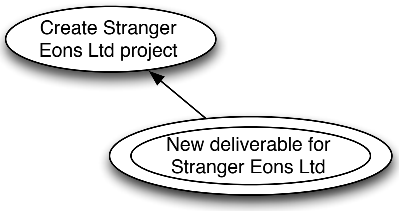

Create Stranger Eons Ltd project
New deliverable for Stranger Eons Ltd

Figure 1.3: Create new project

Indeed, we analyze the situation to find out what we need to do, which is in contrast to the Naïve Approach. This is OK, because we *Seek things to try (p.**??**)*, where actual consequences of our changes tell us what steps we need to take next. Hence, we strive to make changes that will give us that feedback, as soon as possible.

Step: Create a project for Stranger Eons Ltd

Now we *Fix the leaves first (p.**??**)* and the Naive Approach will come in handy. In the naïve spirit, we just create the new project to see what happens. Luckily, we can do that without any problem.

In *Ch.**??** Kick start your improvements (p.**??**)*, the rule says 'we check in to the main development branch if everything works as it should *and the changes make sense*'. In this case, the newly created project makes little sense to check in. In other cases, a created project might make a lot of sense.

So, we are back at the Mikado Goal. What next? Once again, we want to *Seek things to try (p.**??**)*. Driving development using Acceptance Tests or Customer Tests is something we prefer to do, and tests often give us real feedback about the next step. Therefore, we choose to create a test for Stranger Eons Ltd in the new project, a test which will look much like the one we have already for Gargantua Inc.

Step: Create test case

If we need to roll back the changes, we want to remember what we need to do so we continue to *Write down the goals (p.**??**)*. This step is also

```
...
    @Test
    public void testLeeting() throws Exception {
        UI ui = new UI();
        assertEquals("Leeted:␣5ecret", ui.leetMessage("Secret"));
    }
...
```

Listing 1.5: The testcase for Stranger Eons Ltd.

added to the Mikado Graph, as in *Fig. 1.4 (p.8)*.

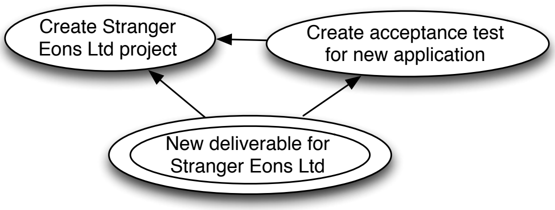

Create Stranger Eons Ltd project
Create acceptance test for new application
New deliverable for Stranger Eons Ltd

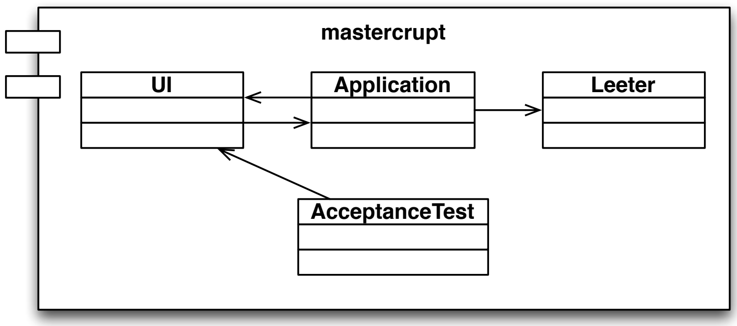
Figure 1.4: Create acceptance test case for new app

Still in the name of naïvety and since we have the project prerequisite in place, we create a new test whose only real difference from the one in the Gargantua Inc project is that this one expects a different return value, as in *Listing 1.5(p.8)*.

1.1.3 Dependency problems

But *now*, problems arise as we get a compilation error. This is actually a good thing, since it gives us information about what we need to change, and we got that information from just naïvely writing some code.

The UI class is in the `mastercrupt` project and we must avoid a dependency to that project, since it still contains code that can't be shared between our two clients. Also, the UI class instantiates an application class, which in turn uses the Leeter for `mastercrupt`. This is best illustrated in the package diagram in *Fig. 1.5 (p.9)*.



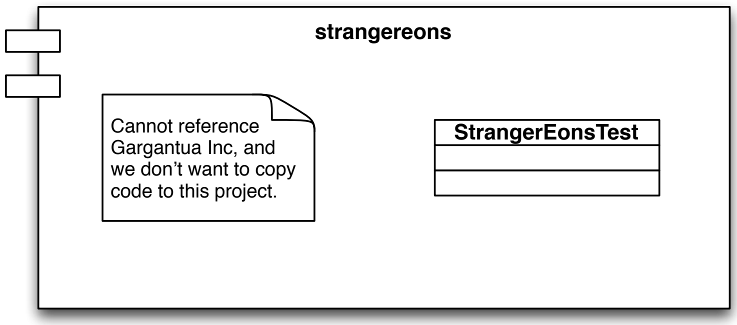

strangereons
Cannot reference Gargantua Inc, and we don't want to copy code to this project.
StrangerEonsTest

Figure 1.5: Problems with new project dependencies

In this simple case, this could probably have been spotted in advance by an attentive developer, but compilation of speculative code changes inside ones head is not feasible for a more complex system. In this case we made the change to see what happened, and left it to our compiler to tell us where it breaks.

The next natural step would've been running the tests but it isn't possible because the code doesn't even compile. If we would have been using a dynamic language, where it's impossible to lean on the compiler, the automated tests are even more important. They are the only way to consistently verify the effects of a change.

To resolve the compilation problems, one option is to duplicate the entire project. We won't do that because we think duplication is bad, as described in detail in *Don't Repeat Yourself - DRY (p.**??**)*. So, we need to change the chain of dependencies in order to allow us to add the test case to the project without any compilation errors.

> ***Why is it interesting to make changes that still compile?*** *IDEs can help out a great deal when it comes to refactorings that require changes to multiple parts of the codebase at the same time. For instance when the name of a method is changed, all places calling that method needs to be changed as well. A modern IDE can automatically do that for you. For a statically typed language, that support is highly dependent on compiling code in order to correctly find the places needed to change. As soon as the code doesn't compile, the automated refactoring support will have trouble finding the references to the altered code. The alternative is manual change which is really tedious, not to mention unnecessary and errorprone. Furthermore, it takes our focus off the real problem, which is doing the least amount of refactorings in order to be able to add more business value.*

1.1.4 A naïve resolution to the dependency problems

Now it's time to decide what to do about the dependency problems. `UI` has some common logic which we want to use in both projects. We choose to create a new project for the `UI` code and this project will be used as a common dependency, to share code between customers.

Decision: Put `UI` code in separate project.

Now, the compiler errors have provided us with valuable information which we need to consider. We *Write down the goals (p.**??**)* and add this as a step to the Mikado Graph in *Fig. 1.6 (p.11)*.

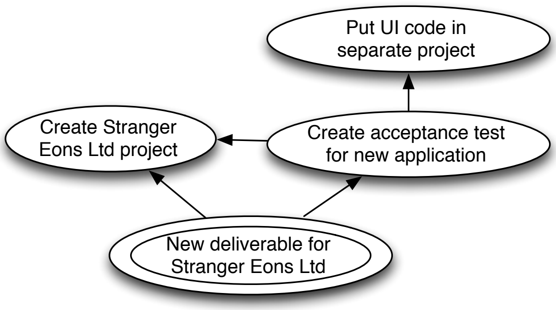

Put UI code in separate project
Create Stranger Eons Ltd project
Create acceptance test for new application
New deliverable for Stranger Eons Ltd

Figure 1.6: The decision to put `UI` code in a separate project

Time to fix the broken system.

1.1.5 Roll-back time

Now it's time for a non-intuitive step, but an important part of the method: *Back out broken code (p.**??**)*!

***Rollback**:* Roll back to the tag "Before new client".

The `strangereons` project has compilation problems and we don't want to do anything there, nor any place else. So, we roll back to the very beginning, which is the state tagged *"Before the new client"*. By doing so, the newly created project disappears and we have the code in its original state.

As a complement to rolling back, the changes that broke the code can be saved as a patch and stashed away until the prerequisites are implemented. When the prerequisites are met, an attempt to reapply the patch can be made. In this example, the changes are reproduced with little effort, so we won't use the stash-and-reapply approach.

1.1.6 Implementing the naïve resolution

We continue to *Fix the leaves first (p.**??**)*. To have the `UI` code in a separate project we need to have a project for it.

***Step**: Create `UI` project.*

We also need to move the code to the new `UI` project, to *Seek things to try (p.**??**)*.

***Step**: Move `UI` code to new `UI` project.*

Now our graph looks like this *Fig. 1.7 (p.12)*.

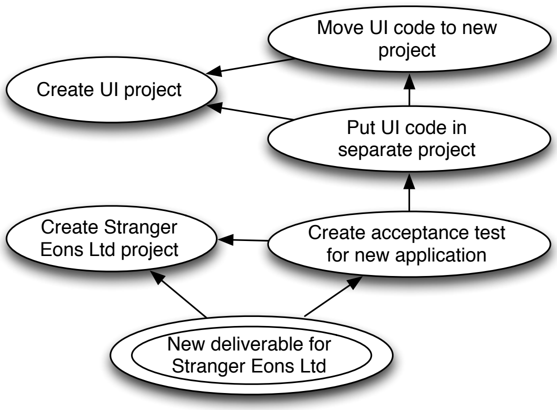

Move UI code to new project
Create UI project
Put UI code in separate project
Create Stranger Eons Ltd project
Create acceptance test for new application
New deliverable for Stranger Eons Ltd

Figure 1.7: `UI` code in a sep project and create `UI` project

This path is one of many that might work for us. There is a large set of possible solutions we can end up with, and depending on our decisions along the way, we implicitly select one of them. The choices we make will affect how easy or hard our path will be. Sometimes we choose paths that aren't feasible, but then we are likely to realize that from an ever growing Mikado Graph. On those occasions, we take a step back and reflect over the graph, and hopefully see where we took a wrong turn.

Adhering to good design principles often leads us right, and in *Ch.***??** *Guidance for creating the nodes in the Mikado Graph (p.***??***)*, some important principles on how to structure code are presented. Sometimes, we experiment to find the right abstractions. The Mikado Method helps us keep track of the needed changes, but technical skills, and domain and market knowledge, are also required to know the right direction to take.

1.1.7 Moving the `UI` code to a new project

According to Figure *Fig. 1.7 (p.12)* and our desire to *Fix the leaves first (p.***??***)*, we now need to create a new `UI` project and move the `UI` code there. That is easily said and done, but *ouch*: The code doesn't compile.

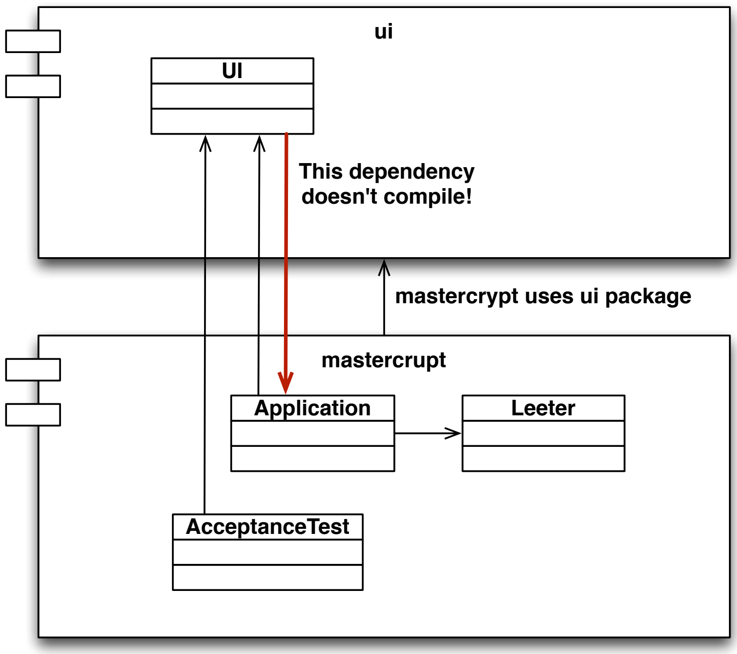


Figure 1.8: Problems with circular dependencies `UI-Application`

As we can see from *Fig. 1.8 (p.14)*, `UI` depends on `Application` and, unfortunately, `Application` depends on `UI`. It's a circular dependency, and we need to resolve that before we can move the code as we wish.

***Decision**:* Break dependency between `UI` and `Application`.

We often add decisions, like breaking dependencies, to the Mikado Graph even before we know exactly how to resolve them. Such items serve as a *decision node*, like in *Fig. 1.9 (p.15)*, and they help us defer commitment until the last responsible moment.

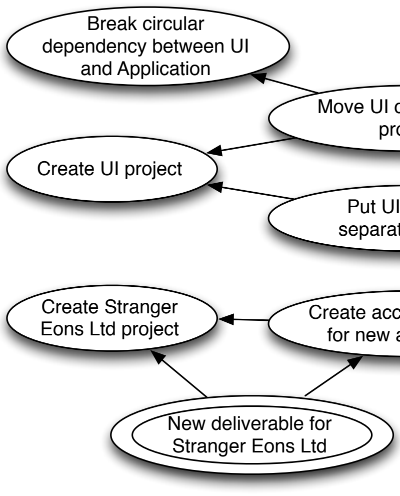

Break circular dependency between UI and Application
Move UI code to new project
Create UI project
Put UI code in separate project
Create Stranger Eons Ltd project
Create acceptance test for new application
New deliverable for Stranger Eons Ltd

Figure 1.9: Break circular dependency `UI - Application`

After we've added the prerequisite, we roll back our changes again, since we always *Back out broken code (p.**??**)*.

***Roll back**: All the way*

Codewise, we're back where we started, but we have built up a body of knowledge about the application and how we want to change it. We are again in the position to take care of a new leaf in the graph, to *Fix the leaves first (p.**??**)* without any errors in the code.

1.1.8 Resolving the circular dependency

A common way to break circular dependencies is to introduce an interface for one of the classes involved in order to change the direction of the dependency. By doing so, we adhere to the *Dependency Inversion Principle*. See *Dependency Inversion Principle - DIP (p.**??**)* for more details.

We choose to introduce an interface for the `Application` class, the `ApplicationInterface`.

Step: Extract `ApplicationInterface`, including method `leet(...)`

The graph now looks like *Fig. 1.10 (p.16)*.

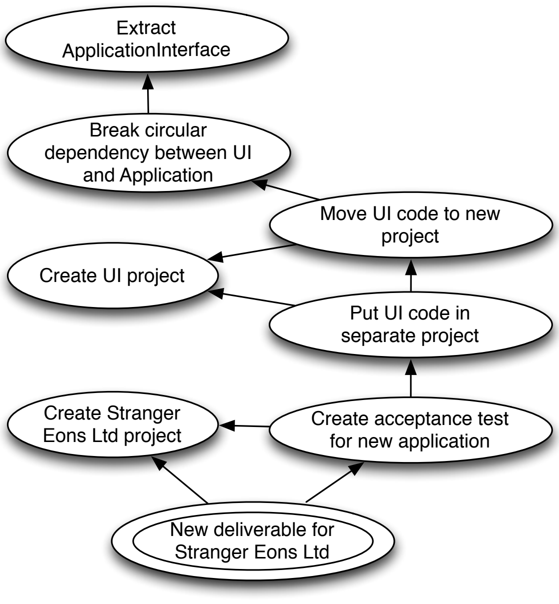

Extract ApplicationInterface
Break circular dependency between UI and Application
Move UI code to new project
Create UI project
Put UI code in separate project
Create Stranger Eons Ltd project
Create acceptance test for new application
New deliverable for Stranger Eons Ltd

Figure 1.10: Application interface

We have a new leaf in the graph, so we try to perform that step without hesitation. This means creating the interface, let `Application` implement it, and replace all declarations of `Application` where we can use `ApplicationInterface` instead. By hand or with an automated IDE tool, this can actually be implemented without any errors. The affected code looks like *Listing 1.6(p.17)*.

```
public interface ApplicationInterface {
    public abstract void leet(String string, UI ui);
}
public class Application
        implements ApplicationInterface {
    @Override
    public void leet(String string, UI ui) {
        ui.setLeeted(Leeter.leet(string));
    }
    . . .
}
public class UI {
    ApplicationInterface application = new Application();
    ...
}
```

Listing 1.6: The extracted `ApplicationInterface` and its usages

This is good, but we still have the use of `new Application()` in *Listing 1.6(p.17)* that upholds the circular dependency.

1.1.9 Should we check in now?

We'd rather check in every small step than keep a batch of uncommitted changes. In reality, we might need to run a test suite locally before checking in. If that takes time, we might try to bite off a little more for every checkin. This again stresses the importance of having fast tests in order to shorten development cycle times.

That said, at the moment we are not sure if this path will actually break the circular dependency, thus we refrain from checking in. This is a judgment call which requires a bit of training to get the hang of it and at this particular time we don't think it makes enough sense to check in.

1.1.10 Adding dependency injection

We have only broken half of the circular dependency. There is still the problem of the `UI` class creating an instance of the `Application` class, we saw in *Fig. 1.11 (p.18)*.

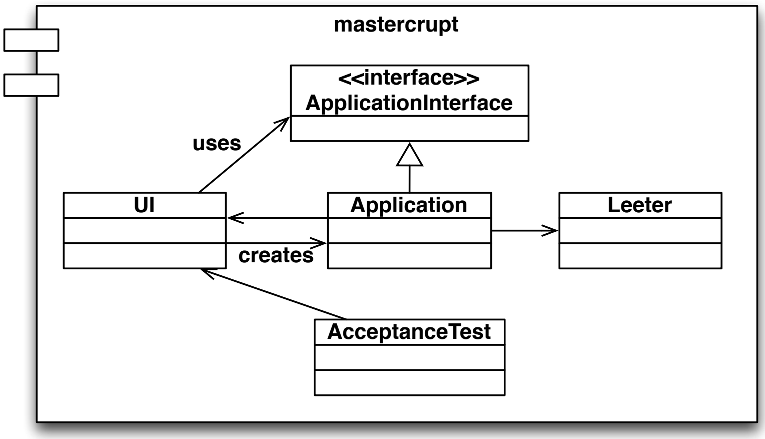


Figure 1.11: The `UI` still creates the `Application` instance

If we want to break the circular dependency chain completely, we need to move the instantiation of the `Application` class outside of the `UI` class, and inject it as an `ApplicationInterface` instance into the `UI` constructor.

***Step**: Inject `ApplicationInterface` instance into the `UI` constructor.*

This is often referred to as *dependency injection (DI)*, and is described more in detail in *Dependency Injection and Inversion of Control (p.**??**)*. This step obviously require us to have the `ApplicationInterface` in place and after we have added another node to the Mikado Graph, it looks like this: *Fig. 1.12 (p.19)*.

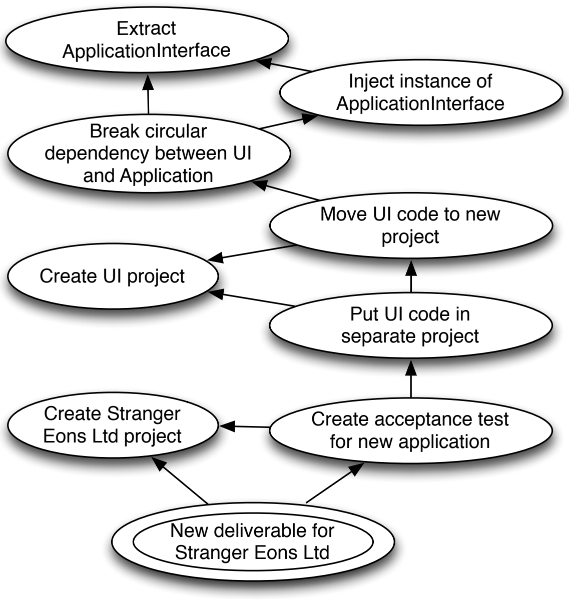

Extract ApplicationInterface
Inject instance of ApplicationInterface
Break circular dependency between UI and Application
Move UI code to new project
Create UI project
Put UI code in separate project
Create Stranger Eons Ltd project
Create acceptance test for new application
New deliverable for Stranger Eons Ltd

Figure 1.12: Injection `ApplicationInterface` instance in `UI`

The graph now reminds us of our next step in our refactoring and with the extracted `ApplicationInterface` in place it is time to decouple UI from Application. In order to loosen the tight relationship between them we turn our attention to the `UI` class and decide that a simple *Change method signature* will suffice.

Instead of creating an `Application` instance inside the `UI` class we inject the abstraction, an `ApplicationInterface`, through the constructor. This will effectively dissolve the coupling between the two implementations and we are a step closer to our goal. The code changes slightly and now looks like this: *Listing 1.7(p.20)*

In `Application` we change the main method so it looks like figure *Listing 1.8(p.20)*.

The test case is changed in the same way, as in *Listing 1.9(p.20)*.

```
public class UI {
    private ApplicationInterface application;
    private String leeted;

    public UI(ApplicationInterface application) {
        this.application = application;
    }
   . . .
}
```

Listing 1.7: UI class with ApplicationInterface injected in constructor

```
...
    public static void main(String[] args) {
        UI ui = new UI(new Application());
        System.out.println(ui.leetMessage(args[0]));
    }
...
```

Listing 1.8: Create the instance to inject in the main method

```
...
    @Test
    public void testLeeting() throws Exception {
        UI ui = new UI(new Application());
        assertEquals("Leeted:␣S3cr3t", ui.leetMessage("Secret"));
    }
...
```

Listing 1.9: Creation of instance in test case

Everything compiles and all the tests run! Now the class diagram has no circular dependencies. Note that the *realize arrow*, the triangular arrow, in the UML diagram in *Fig. 1.13 (p.21)* is also a dependency arrow.

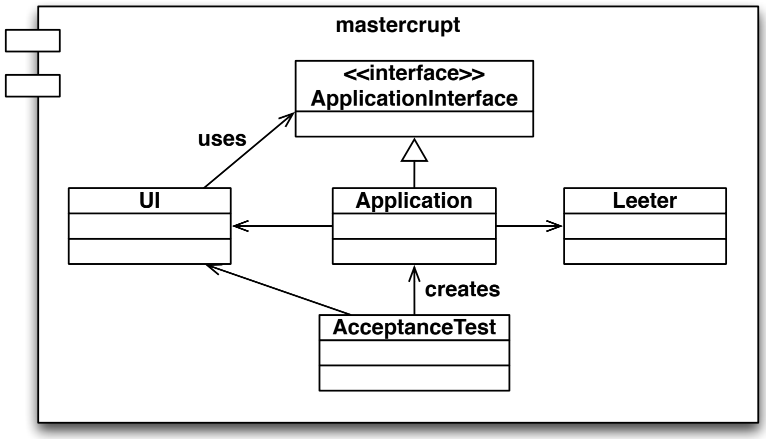


Figure 1.13: Circular dependency resolved

1.1.11 First checkin

We have broken the circular dependency in the code, and that *really* makes sense. It is time to check in! We use the label *Broke circular dependency between* `UI` *and* `Application`.

Checkin: Broke circular dependency between `UI` and `Application`

Now, we can tick off some nodes the graph, including the decision node for breaking the circular dependency since it is completed when all of its dependencies are completed. The graph looks like *Fig. 1.14 (p.22)*.

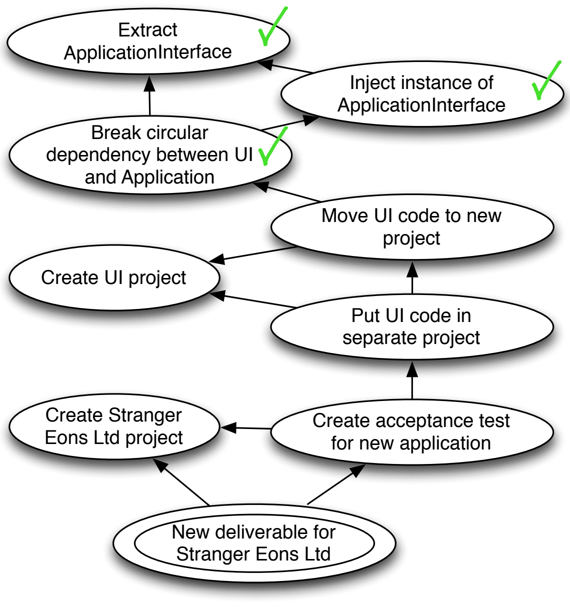

Extract ApplicationInterface
Inject instance of ApplicationInterface
Break circular dependency between UI and Application
Move UI code to new project
Create UI project
Put UI code in separate project
Create Stranger Eons Ltd project
Create acceptance test for new application
New deliverable for Stranger Eons Ltd

Figure 1.14: Circular dependency resolved

1.1.12 Moving the code to new projects

After dealing with some prerequisites, we now seem to be in a position where we, according to the graph, can create the new UI project.

Step: Create new UI project

Now we are able to use the *Move class* refactoring which moves the UI class to the new UI project. The ApplicationInterface is now a part of the UI code, and has to follow along as well.

Step: Move UI and ApplicationInterface to UI project

With an IDE that eagerly compiles code as we edit, it is extra important to add any projects that `mastercrupt` will depend on before we start to move classes. If we don't, the project can't be compiled and we end up with a broken code base after the move.

We could add this step as a node to the Graph to remind us about the project dependencies, but in this case we don't, because a node like that would be too short lived.

Adding a lot of unecessary details and steps to the Graph only makes the Method feel like a cumbersome process and we would end up with a lot of overhead and ambigious graphs, which is exactly what we are trying to avoid.

Let this brief example serve as a reminder that sometimes obvious details or short live nodes can be left out of our graphs if that makes them more effective.

It is now possible to move `UI` and `ApplicationInterface` *without any compilation errors*, and the result is shown in *Fig. 1.15 (p.24)*.

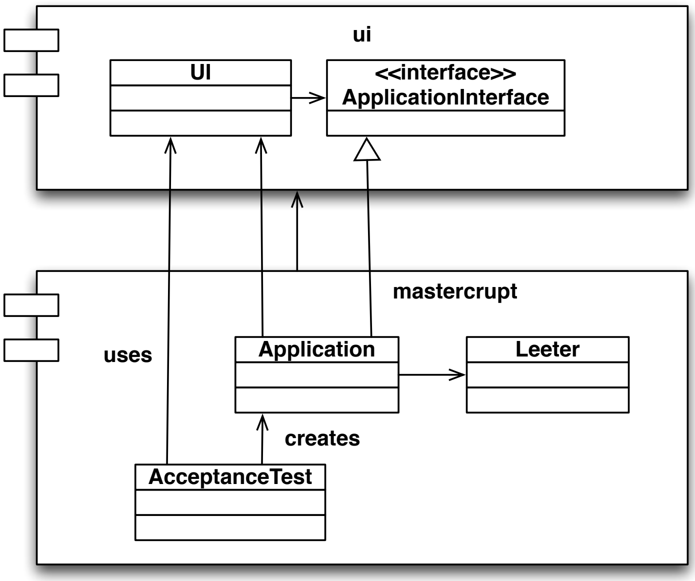


Figure 1.15: Classes in new UI project

The tests run, so we check in.

Check-in: UI-code in separate project

Of course we also tick it off in the Mikado Graph in *Fig. 1.16 (p.25)*.

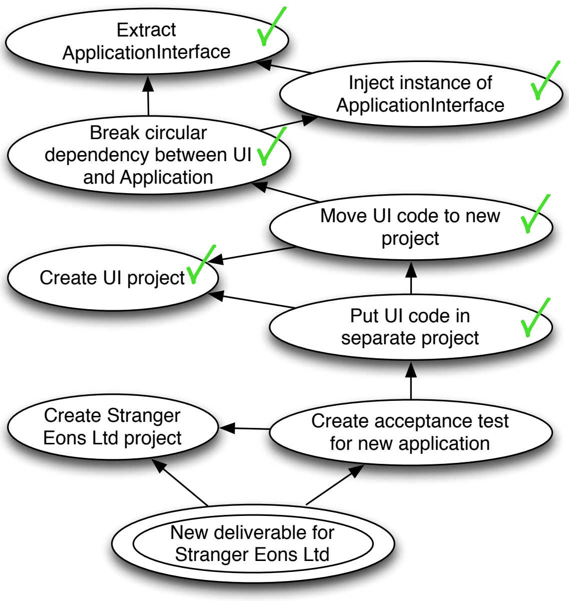

Extract ApplicationInterface
Inject instance of ApplicationInterface
Break circular dependency between UI and Application
Move UI code to new project
Create UI project
Put UI code in separate project
Create Stranger Eons Ltd project
Create acceptance test for new application
New deliverable for Stranger Eons Ltd

Figure 1.16: UI project established

1.1.13 Creating the new deliverable

If we look at the graph, we can see that the next leaf is almost where we started out, to create project for Stranger Eons Ltd. We also need to make it depend on the UI project, since we do want to actually use the UI project that we have just extracted from the codebase.

The test case looks like the one we created earlier, with the addition that the UI constructor takes an argument. The result is in *Listing 1.10(p.26)*.

At first, the new project doesn't compile since there is no local Application implementation of the ApplicationInterface. We implement the interface with an empty implementation, as in *Listing 1.11(p.26)*.

```
...
    @Test
    public void testLeeting() throws Exception {
        UI ui = new UI(new StrangerEonsApplication());
        assertEquals("Leeted:␣5ecret", ui.leetMessage("Secret"));
    }
...
```

Listing 1.10: The testcase for Stranger Eons Ltd.

```
...
public class StrangerEonsApplication
        implements ApplicationInterface {
    @Override
    public void leet(String string, UI ui) {
    }
}
...
```

Listing 1.11: Make the Stranger Eons Application instance implement the ApplicationInterface

Now we can run the tests, but they fail since the implementation of `ApplicationInterface` is empty. Let's fill the implementation with something like that in *Listing 1.12(p.27)*. We also added the main method, like in the Gargantua application.

1.1.14 Done and delivering!

Now the tests pass! We're done and can deliver the system to Stranger Eons Ltd. The system now looks like *Fig. 1.17 (p.27)*

```
...
public class StrangerEonsApplication
        implements ApplicationInterface {
    @Override
    public void leet(String string, UI ui) {
        ui.setMessage(string.replace('S', '5'));
    }
    public static void main(String[] args) {
        UI ui = new UI(new StrangerEonsApplication());
        System.out.println(ui.leetMessage(args[0]));
    }
}
...
```

Listing 1.12: Make the test pass

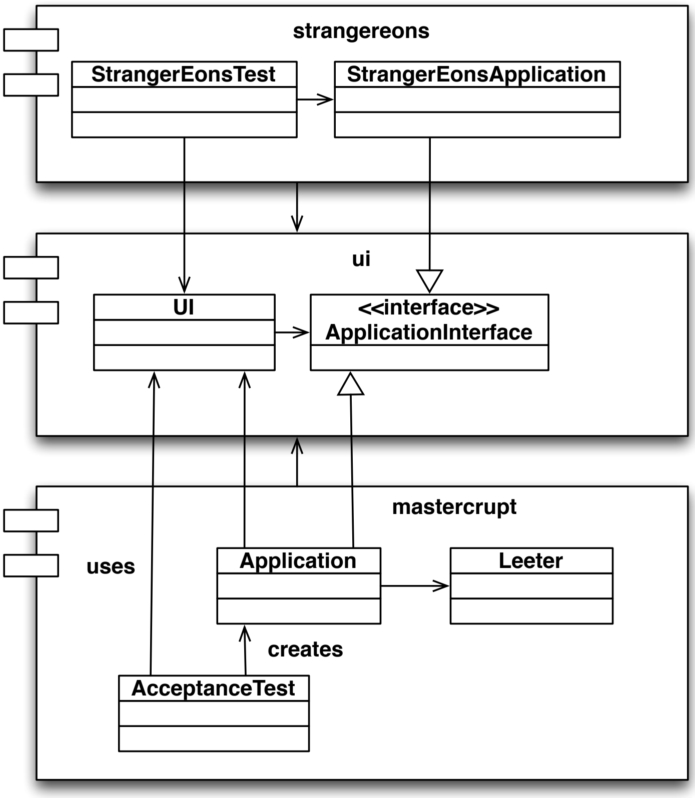


Figure 1.17: The final result. The system is ready for delivery!

We can tick off the last nodes of the graph in *Fig. 1.18 (p.28)*.

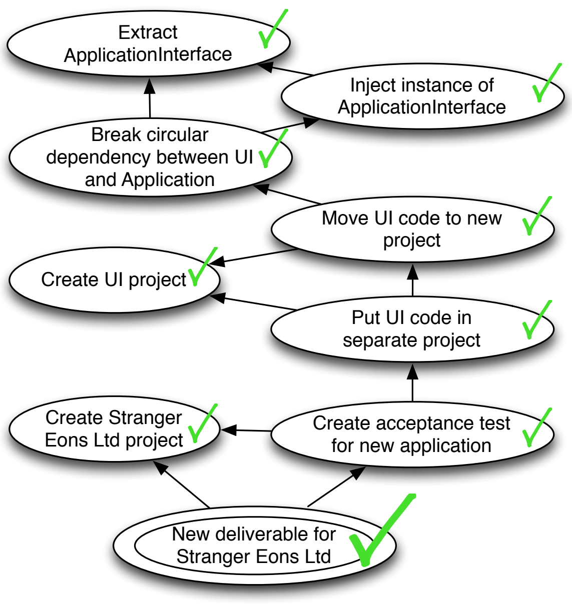

Extract ApplicationInterface
Inject instance of ApplicationInterface
Break circular dependency between UI and Application
Move UI code to new project
Create UI project
Put UI code in separate project
Create Stranger Eons Ltd project
Create acceptance test for new application
New deliverable for Stranger Eons Ltd

Figure 1.18: Done!!

1.2 Conclusion

We've managed to morph the code from one state that didn't allow us to do what we wanted, to one that actually does what we want. We did this by using the Mikado Method to *Write down the goals (p.**??**)*, *Seek things to try (p.**??**)*, *Back out broken code (p.**??**)* and *Fix the leaves first (p.**??**)*. We used the Naïve Approach combined with some simple analysis. Along the way we made use of some test-driven development, and atomic and composed refactorings. By focusing on doing small changes, step-by-step to isolated pieces of code, we kept the system in a working condition all the time. In addition to that, we also wrote some brand new code to implement the new functionality.

Finally we ended up with a restructured system that is more suited to the needs of Pasta Software and their clients. The code might not be perfect, but it does the job.

Our attention was focused on actual problems which arose during the initial creation of the graph. After that, we quickly arrived at a feasible solution.

When we first started out, we didn't know the single most effective way to reach our goal, but this is very seldom required. The goal is to reach a *good enough solution* to a particular problem. There are often several ways to reach the same goal and perhaps better ones as well in this case. If we had the time, or the problem was small enough, we could reiterate the whole or parts of our refactoring and see what decisions make for what results.

> *For this particular problem, the authors have refactored the code more than a dozen times, and come up with several different, and working, solutions each time. The Mikado Method may produce different paths each and every time but don't worry, it always leads to the goal. It's an inherent feature of the Method which makes it very pragmatic and suitable for changing a system.*

If we find more things we want to change, the same method can be used over and over for different goals.

1.3 USING THIS EXAMPLE AS A SOFTWARE KATA

The term *Kata* comes from Japanese martial arts and is the name of a type of exercise that aims at practicing a particular set of actions, over and over again. The example above can be used as such a Kata for practicing the basics of the Mikado Method, and the URL to the example can be seen in *Try This*. There is a short appendix on Katas in *App.A Software Katas (p.31)*.

1.4 Summary

Jake:	Wow, we did it! We actually did it!
Melinda:	Yes, we did! Great feeling, huh?
Jake:	Very much. The Naïve approach stopped me from spending a lot of time analyzing, it just took me straight to the problems.
Melinda:	Exactly.
Jake:	It was also a good feeling to first capture the knowledge in the graph, and then start checking the nodes off as we worked our way back.
Melinda:	Did you notice that we mostly ended up doing simple, atomic refactorings, except for implementing the logic for the new customer?
Jake:	Yes. And you can apply the same principles for even larger systems, right?
Melinda:	You follow exactly the same flow in a larger system. The main difference is that you might change hundreds, or thousands, of files at a time... of course with the help of automated tools.
Jake:	That sounds cool, and scary.
Melinda:	It can be, but more often it is actually rather comfortable.

1.5 Try this

- Get the code for the example from
  `https://github.com/mikadomethod/kata-java`
  or
  `https://github.com/mikadomethod/kata-dotnet`
  and do the steps yourself. Start with doing the exact mechanics as above, and draw the graph exactly the same way.
- Try to solve the same problem in your own way. What did you do differently?
- Try the method on some of your own code. How does it feel? Did you find a reasonable path?

Appendix A

Software Katas

A.1 KATAS IN MARTIAL ARTS

The term *Kata* comes from Japanese martial arts and is the name of a type of exercise that aims at practicing a particular set of actions, over and over again. A Kata is usually a choreographed set of blocks, kicks, punches and more, that are used for practicing those moves. They can be practiced individually, in pairs or in groups. Katas also come in many shapes and difficulties, some are relatively simple, and some more advanced. The simple Katas are more suitable for beginners, but even advanced practitioners practice simple Katas to achieve perfection.

A.2 KATAS IN SOFTWARE DEVELOPMENT

Software Katas are the same thing, but instead of kicks and punches, a programming problem is to be solved. Historically, software katas have been about learning test-driven development and pair-programming, but any kind of standardized exercise can be used.

A.3 Coding dojos

The training room in (Japanese) martial arts is called the *Dojo*. The concept has been transformed to programming as a name for a place where people gather to practice programming together. This is called a Coding Dojo.

Just like in a martial arts Dojo, many forms of practice can be performed in a Coding Dojo. Katas is groups, pairs or individually are some ways of practicing, but one can also do sparring or individual training on certain techniques.

When doing a Kata in a Coding Dojo, it is common to do it *Randoori style*, meaning that one pair at a time is doing the programming in front of the whole group, preferably using a projector on a wall.

Other ways of doing it is that everyone is doing it synchronized, led by the trainer. Another is where everyone is practicing in pairs, at their own pace.

Bibliography

[1] Michael Feathers *Working effectively with legacy code*

[2] Dave Thomas, Andy Hunt *The pragmatic programmer*

[3] Eric Evans *Domain-Driven Design*

[4] Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts *Refactoring: improving the design of existing code* ISBN 0-201-48567-2

[5] Joshua Kerievsky *Refactoring to patterns*

[6] Eric Gamma, Richard Helm, Ralph Johnson, John M. Vlissides *Design Patterns: Elements of Reusable Object-Oriented Software*

[7] Robert C Martin *Clean Code*

[8] Kent Beck *Implementation patterns*

[9] Kent Beck *Test-driven development*

[10] Martin Fowler *http://martinfowler.com/bliki/StranglerApplication.html*

[11] Larry LeRoy Constantine *http://en.wikipedia.org/wiki/Larry_Constantine, reference International Bibliographical Dictionary of Computer Pioneers*

[12] Robert C Martin *Agile Software Development: Principles, Patterns and Practices.* Pearson Education. ISBN 0-13-597444-5

[13] Arthur J. Riel *Object-Oriented Design Heuristics*

[14] Ward Cunningham *The WyCash Portfolio Management System (http://c2.com/doc/oopsla92.html)*

[15] Martin Fowler *http://martinfowler.com/bliki/TechnicalDebtQuadrant.html*

[16] Kent Beck *http://c2.com/xp/CodeSmell.html*

[17] http://www.catb.org/ esr/jargon/html/Y/yak-shaving.html

[18] http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis

[19] http://en.wikipedia.org/wiki/Code_coverage

[20] Mike Cohn *Agile estimating and planning*

[21] Dr. Eliyahu M. Goldratt *The Goal*

[22] Gordon E. Moore *http://en.wikipedia.org/wiki/Moore%27s_Law*

[23] *http://en.wikipedia.org/wiki/Side_effect_%28computer_science%29*

[24] Johanna Rothman, Esther Derby *Behind Closed Doors*

[25] Kent Beck *Extreme Programming Explained 1st and 2nd editions*

[26] Staffan Nöteberg *The Pomodoro Technique Illustrated*

[27] James Q. Wilson and George L. Kelling *Broken windows*

[28] Mary and Tom Poppendieck *Lean Software Development - An Agile Toolkit*

[29] Joel Spolskys blog *http://www.joelonsoftware.com/articles/fog0000000069.html*

[30] Ken Pugh *Prefactoring - Extreme Abstraction, Extreme Separation, Extreme Readability*

[31] Christopher Alexander et al *A Pattern Language - Towns Buildings Construction*

[32] Leo Brodie *Thinking Forth - A Language and Philosophy for Solving Problems* ISBN 0-9764587-0-5

[33] Linda Rising and Mary Lynn Manns *Fearless Change - Patterns for Introducing New Ideas* ISBN 0-201-74157-1

[34] Russell L. Ackoff *Idealized Design: How to Dissolve Tomorrow's Crisis...Today* ISBN 0-13-196363-5

[35] Brian Foote and Joseph Yoder *Big Ball of Mud Fourth* Conference on Patterns Languages of Programs (PLoP '97/EuroPLoP '97) Monticello, Illinois, September 1997

[36] Jez Humble and David Farley *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation* ISBN 0-321-60191-2

[37] Gojko Adzic *Specification by Example: How Successful Teams Deliver the Right Software* ISBN 978-1617290084