Refactoring Big Balls of Mud

Paul Adamczyk[*] Arturo Zambrano Federico Balaguer
LIFIA, Facultad de Informática
Universidad Nacional de La Plata
La Plata, Argentina
{Paul.Adamczyk, Arturo.Zambrano, Federico.Balaguer}@lifia.info.unlp.edu.ar

Abstract

This experience report describes a redesign of a large commercial system. The goal of the redesign was to break up the system into two parts without changing its external behavior. Such a task is essentially a refactoring. We describe our redesign process as steps of a refactoring called Extract subsystem to a separate process. *We believe that documenting large-scale refactorings is important, and necessary to make redesigning software easier.*

1 Introduction

Software systems that have many users tend to grow very fast to accommodate new requirements. Quickly, they outgrow the original architecture and get too big to understand. But because they are working systems that keep generating revenue, they are used, extended, and maintained far beyond their intended life span. Such systems are what Foote and Yoder call *Big Balls of Mud* – "haphazardly structured, sprawling, sloppy, duct-tape and bailing wire, spaghetti code jungle" [5]. Other characteristics to add to this list are dead code, scattered functionality, and unusable documentation artifacts. Foote and Yoder claim that Big Ball of Mud, or BBM for short, is the most typical software architecture. But it's important to note that we do not condemn BBMs. Such systems are useful, even if it is increasingly more difficult to extend them, and at some point it becomes economically infeasible.

This report describes our refactoring, called *extract subsystem to a separate process*. We have performed this refactoring on a large software system with BBM characteristics. We will describe it here as a sequence of five distinct steps. First, it is necessary to study the code and other artifacts to learn as much as possible about the system. The second step is to cleanly separate the responsibilities of the two processes by dividing the code between them and identifying which code might be needed by both processes. The next step requires implementing enough code to successfully compile and build an executable of each process. Next, the communication between the processes must be implemented. Until, finally, all the prior functionality is restored, one feature at a time, in the new, distributed system.

We performed this refactoring on a casino game engine, written in C++. A casino game engine (CGE) has three key parts: game-playing logic, game graphics, and reporting/monitoring functionality. Roughly, these parts correspond to the needs of three groups of stakeholders involved with casino games – owners, players, and regulating agencies (i.e. government). The owners need to control how much money machines pay out to maximize their revenue. The players select which games to play based on their appearance (but also based on the potential payout). The government, also to maximize its revenue, is interested in correct applications of the many laws regulating the casino gaming industry. Slot machines are monitored and controlled by systems we will refer to as *report servers*.

This system has become a BBM through piecemeal growth [5]. It contains about 1.5 million LOC in C++, not counting the scripting code for game graphics. One of its problems is lack of modularity – its reporting and monitoring functionality is intimately intertwined with the game-playing logic. The system's core and the reporting functionality are evolving at different pace, so it became necessary to separate them. Since the reporting functionality must meet real-time requirements, it was decided to move it to a separate process, so that it could run even on dedicated hardware. Note that because this is a proprietary system, we cannot disclose specific details.

Many software systems benefit from large-scale redesign, which helps to keep them in use longer. Many system fall out of use, in part, because they are not redesigned in time. In this report we share our experience of refactoring a BBM and the challenges we faced. The currently available redesign techniques do not provide adequate sup-

[*]also affiliated with the Department of Computer Science, University of Illinois at Urbana-Champaign.

port for performing this complex task. We describe our experience with a large-scale refactoring in order to initiate a discussion about advancing the state of practice in this area.

This report begins by motivating the need for large-scale refactorings. Then the *extract subsystem to a separate process* refactoring is presented in detail. The description of some tools that helped us execute this refactoring follows. A discussion of the challenges of redesigning and refactoring BBMs in general comes next. The report ends with a short evaluation of our approach and lessons learned.

2 Toward Large-scale Refactorings

Redesigning large systems is hard. Refactoring, i.e. altering the internal structure of code without changing its external behavior [6], turns a redesign into a sequence of small steps. Refactoring research today still focuses on such small refactorings and tools for implementing them. Current refactoring techniques don't scale to the size of large-scale redesign.

At the age of multi-core processors, breaking up a monolithic system to run as multiple processes is a recurring redesign problem. There are many variants of this task: turning a standalone system into a distributed one, splitting a system among multiple processors or multiple processes on the same processor. We are not familiar with any work describing the steps required to perform such refactorings. Other examples of BBM-scale refactorings are *extract behavior to new component* and *extract a library* – both focusing on improving system modularity. Another example is *restructure the API of the system* to adapt software to meet the constraints of a larger distributed system, for example the Web. (An example where this refactoring is relevant is outlined in section 6.2). Doubtless there are other large-scale refactorings to consider, but this report focuses only on the one refactoring we have performed.

It would seem that a large-scale refactoring should be just like a long sequence of small refactorings. This would be true if the software system to be redesigned was modularized, i.e. if it were *not* a BBM. The system we worked on, like most real systems, had bad modularity; its original design has eroded, and its critical functionality was intertwined with the rest of the code. Because of the code size and the number of inter-dependencies, it was not possible to redesign it by applying small refactorings sequentially.

To begin the *extract subsystem to a separate process* refactoring, a BBM must be cut at its core. Refactoring tools offer limited help in achieving this step, because their primary goal is to preserve the behavior of the system after applying each automated refactoring. When a BBM is cut in half, the system is broken and it cannot be run or tested until the cut is healed. This may mean going weeks or months without the cushion of passing tests. But regardless of how much one would like to simplify this step, cutting out a set of classes and moving them to a new module *is* the smallest possible first step of this refactoring.

There are many other reasons why large-scale refactoring is harder than its automated, small-scale counterpart. We will return to some of them in section 5.

3 'Extract subsystem to a separate process' refactoring

The procedure for performing this complex refactoring along with the tools we developed to help us with the process (described in the next section) is the core of this experience report. We regret not being able to illustrate this section with more specific examples but, as already noted, we cannot present details of the system.

We split our BBM into two independent processes. However this procedure could be applied to split one process into *n* processes, where *n* is small. The refactoring has five steps (also shown in Figure 1):

- study the existing system
- split the functionality between processes
- define logical abstractions necessary to compile the code of both processes separately
- define shared data structures and implement the shared behavior to run basic functionality
- port features, one at a time

We began each step in the in order listed. These steps correspond roughly to the software development lifecycle – analysis, architecture, design, implementation, and maintenance, respectively – but they aren't as neatly delineated. We have often found ourselves working on multiple steps in parallel. In fact, in the last step, it is necessary to retrace the earlier steps iteratively in order to re-introduce existing features into the system. Following the style of documenting refactorings, we present these steps as imperative commands, augmented with some necessary commentary.

3.1 Step 1. Exploration

The purpose of the first step is to develop a mental model of the system. On the outside, BBMs look like well-designed systems – they seem to have structure and all the code appears to have a purpose. The insides are messy. To explore the insides, start playing with the system – run it, read the code, take notes on the code structure, make small code changes – to get a feel for the system.

This step involves understanding the current code and drawing new borders between the emerging components. For this step code analysis tools can be used, for example

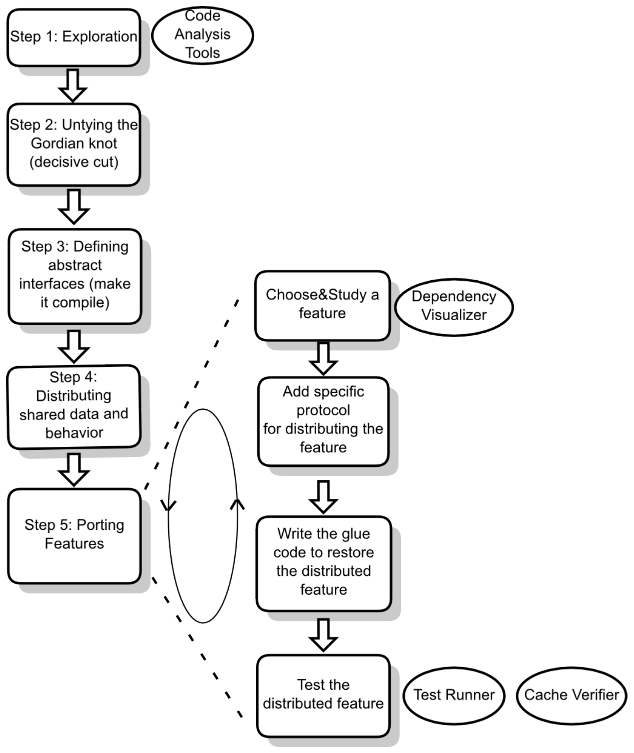
Step 1: Exploration
Code Analysis Tools
Step 2: Untying the Gordian knot (decisive cut)
Step 3: Defining abstract interfaces (make it compile)
Step 4: Distributing shared data and behavior
Step 5: Porting Features
Choose&Study a feature
Dependency Visualizer
Add specific protocol for distributing the feature
Write the glue code to restore the distributed feature
Test the distributed feature
Test Runner
Cache Verifier

Figure 1. Flow of the "extract subsystem to a separate process" refactoring. Rounded boxes are steps. The second column shows the substeps for adding individual features. Ellipses show tools applicable to each step.

CodeSurfer [13]. Annotating the code also helps in preparing the terrain for the next step. Tools like TagSea [11] help to organize the metadata in the code.

The only wrong thing to do at this stage is to trust the existing design documents and the code comments – they seldom look obviously wrong at the first glance, but they are rarely up to date. They do, however, provide interesting insights into the history of the code changes, so they shouldn't be ignored. Just don't accept them on faith.

Figure out which parts of the system need to be separated. In our CGE, we were moving all the reporting functionality out of the game process. Of course, our system reports the game data, so the separated processes shared a lot of data and behavior.

This is the good time to see if you can change the structure of your system by making small changes. Perhaps your system is malleable and it can be fixed by making small design changes. You might be able to separate the code cleanly by moving classes, one at a time, and compiling the code after each move. But it's more likely that after moving 3 or 4 classes, you will be forced to resort to tricks – like defining ad hoc interfaces – to make the code compile. This is precisely why such systems are called Big Balls of Mud – all the code is connected together. Another quick exercise to try here is applying small refactorings to break up the system slowly (Michael Feathers calls this a *scratch refactoring* [4]). Again, you'll determine quickly whether your system is a BBM. We tried both of these approaches first, but failed. This convinced us that a BBM redesign cannot be accomplished in small steps.

Automated testing is the other critical ingredient of a successful redesign. Our GCE is a reactive GUI system. Most tests are initiated by performing the player activities, e.g. starting a game. This means that every non-trivial test is a *system test*. When we began the redesign, the system had no automated tests. System tests, complex and cumbersome to execute manually, are also hard to automate. For our purposes, it did not make sense to write subsystems tests – it wouldn't make sense to test one process in isolation, because neither process can do much by itself. Setting up the infrastructure for running unit tests in C++ looked even more difficult. To be able to test our changes effectively, we built a system integration tool for automating system tests. We describe the tool in some detail in section 4.3.

But even without automated tests, testing is crucial. It's the most important self-check mechanism for a redesign. Since nobody understands the behavior of the entire BBM, the only way to determine the correct behavior is to run the original system and observe the results. Learn to run the basic tests. Keep a running copy of the original system to compare the results of running the same tests. Use a source control tool – have one branch for exploration and prototyping and a separate one for new development; make sure to tag the original code in the source control so that you can find it easily. Use the original system as the guide even if its results appear to be wrong. Let other developers fix bugs in the original system, in the main development branch.

The essential problem posed by BBMs is not the functionality, but the way different parts of the system are connected. These connections (or *glue* code) adapt to the code around it and inherit its assumptions. In the process of separating functionality, one often finds that the glue code includes some domain functionality that needs to be preserved when the connection is broken.

It may turn out that it's not possible to redesign the system in the way originally envisioned. If the functionality to be moved is entangled intimately with the core functionality, the redesign may be too difficult, too expensive, too time consuming, or too risky. Talk to management and quit early, if the task is too hard. Perhaps instead of a clean redesign, the best you can do is to break the BBM into two smaller, but still messy, parts that may be redesigned further at a later time.

One key exploration required for our refactoring was to study candidate inter-process communication models. When a system consisting of a single process is being redesigned, it has no IPC functionality to reuse. Build toy processes and try different models of communication to see which one suits better. It's not critical to select the best model at first. It's more important to try out multiple options to get some idea about the tradeoffs between them, in case you need to use a different model later.

At the end of this step, you will have a high-level understanding of the system. Armed with the knowledge of the current system and the lessons learned from prototyping, present the new system architecture to other stakeholders.

Learning and exploring doesn't end when this step is completed. As each feature is ported, it must be first studied in depth as we will see in section 3.5.

3.2 Step 2. Cutting the Gordian knot

After doing some exploration, you will feel ready to get your hands dirty and to start coding. Then and only then should you begin the next step of the refactoring: cutting the functionality that belongs in the new process out of the main process. This task requires Alexandrian skill and ruthlessness, because the cut must be sharp and clean.

Move only the essential code to the new process and disregard all the glue code, even if it contains domain-specific functionality. Pay the most attention to places where the core functionality of the main process and the essential functionality to be removed are intertwined in the code. In our CGE, the game functionality and the reporting functionality intersect at the places where the game generates the data to be reported. In the original design, the Game object indicates that new data is available by notifying its observers using the Observer pattern [7]. Since you are moving these objects to a different process, the notification mechanism needs be replaced. But for this step, mark in the code where logical connections are broken and move on – these breaks will be fixed later.

After dividing the functionality between the two processes, factor out the code which is common to both processes and place that code in a separate library accessible to both processes. The two processes are now two separate executables, so they will need to be compiled independently, but they can both depend on the same shared libraries. Figure 2 shows both processes sharing common code, and the New Communication Protocol needed to support the distributed functionality. Note that any process-specific code that depends on the code of the communications protocol also has to be placed in the shared library. The figure depicts it as Game Specific and Reporting Specific Code.

At this point, the code does not even compile, but this step sets the table for doing bottom-up compilation, one class/file/module at a time, until all the dependencies that we broke in this step are fixed. Even though nothing is working, the most crucial work is finished.

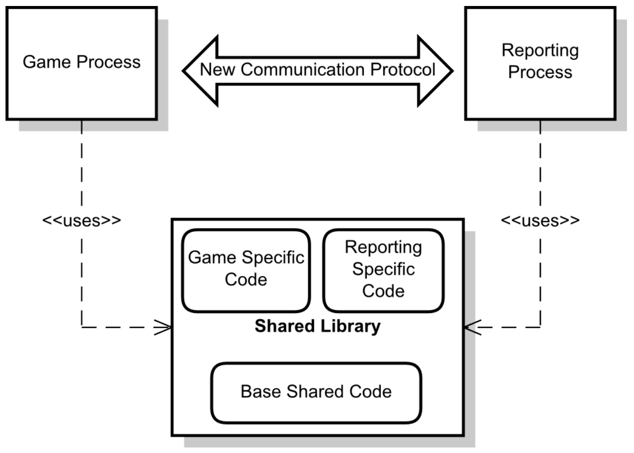

Figure 2. Game process, extracted Reporting process, and the Shared Library in the GCE.

3.3 Step 3. Defining abstract interfaces

The purpose of this step is to compile the code for the first time. This is accomplished by defining a set of new interfaces that declare the shared functionality of both processes.

To determine what interfaces are needed, identify which objects need to be active on both processes. Let's call them *key objects*. Some ideas are global to the system; they cannot be tied to one process. Step 2 works only because the cut is sharp and decisive; if every class or module needs to be sliced, this refactoring will not work. While the behavior associated with the key objects needs to be executed on both processes, the objects themselves reside on one process and have proxies [7] on the other one. In our CGE, the reporting process communicates with the *report servers* that can initiate requests to modify the internal state of the game, e.g. take the slot machine offline, so the Game object and game-specific data must be shared, so that both processes have read and write access to the data.

For each class that needs to be shared, make it a subclass of its new interface class. Declare all the shared methods in the interface class and make them abstract. This is the *extract interface* refactoring [4]. In these interfaces include all the methods that the key objects need to make available on the other process, where they are not present. For example, a GameInterface class provides all the getters and setters used by the reporting functionality. Its dummy subclass is instantiated on the reporting process and provides default implementation of all the required the behavior. All the existing code moved to the reporting process that has previously referred to the Game class is changed to point to GameInterface by using the *rename class* refactoring [6].

Now the key objects can stay in one process and be known on the other process only by their interface name (and a dummy implementation). Place the code of the interfaces in the shared library. This trick makes it possible to compile the code for the first time.

These dummy implementations serve another purpose: they indicate the project's progress. The number of interface methods left to be implemented is a pessimistic indicator of the amount of remaining work. Why pessimistic? In the end, some of the interface methods will turn out to be unnecessary. When they are deleted, the progress metric will improve immediately with little extra work.

At the end of this step, the whole system finally compiles. The real functionality is not implemented, but being able to run each process idly is a sufficient sanity test. In our CGE, we can run the game standalone, play it, and have it generate reports (which are not sent out by the system). Similarly, we can run the reporting process, standalone, sending reports with all the data set to dummy default values and accepting messages from the report servers.

3.4 Step 4. Distributing shared data and behavior

At this point, the pieces are in place to start writing new code to replace the dummy default implementations introduced in the previous step. Some new code is needed to facilitate communication between the two processes. Other important part is the code that cleanly divides data and behavior between processes to eliminate unnecessary duplication. Each shared key object, although it resides on one process, is logically present on both. Make one process responsible for keeping its state consistent, while the other process is a passive user with limited ability to modify the internal state of the key object.

There are many ways for two processes to communicate. We will call the data-carrying messages passed between them *commands*. Commands may be implemented as binary messages, serialized objects, distributed observers. Choose only one implementation – mixing multiple approaches is error-prone and can produce strange bugs, especially if the processes use multithreading. The choice of the implementation depends on the needs of the system, but the overall goal is to pass data between the two processes as efficiently as possible. It is important to minimize the amount of data to be passed, but another important consideration is the amount of code to write. The command-passing code is merely glue code, so it should be kept small.

There are different ways to *fake* the presence of key objects on both processes while keeping most of the original code unchanged. Here are three ways of providing the behavior and up-to-date data of key objects on both sides:

Duplicate the behavior. Large portion of the behavior shared by the two processes will be similar on both processes, but not exactly the same. Place the common code that both sides share in the interface classes. Concrete classes that represent key objects on both processes will extend the common code by overriding the interface methods.

If the same feature can be initiated by either process, they both need to keep track of the originator. In multi-command exchanges (i.e. transactions), both processes must know the current state of the transaction. For example the process of passing game credits to the player, i.e. *cashout*, can be initiated by many entities in the casino: the player, the game when a predefined threshold is reached, or any report server. Regardless of who initiates the cashout, the basic processing is the same – game credits are transferred from the slot machine to the player – but the division of responsibilities between the two processes is different in each case.

Duplicate shared data and update it in lock-step. Keep duplicates of the critical data that both processes need to access often on both processes. When a key object storing the data is modified on one process, apply the same methods with the same parameters to its copy on the other process. Pass the data needed for the call in a command.

When data is duplicated, either process can provide it to the outside world as soon as requested thus improving the responsiveness of the system. Note that the relationship between the data of the two processes is somewhat complex – there is no master copy and backup copy. The responsiveness comes at the price of code complexity required to keep the distributed data synchronized. The first process to receive the update applies it and then sends the command with the update to the other process.

The lock-step update is ideal for simple numeric data, which is changed by incrementing and decrementing. It doesn't work with more complex data structures that are sensitive to the order in which changes are applied. For those cases, it's better to consider the next option.

Cache shared data. Designate one process to be the owner of some data. In the other, *cache* process keep a read-only copy of the data. Pass the delta of the change from the main process to its cache on each relevant change.

This might seem inefficient, for example, if the cache process generates an event that will result in changing its data. It might seem easier to update the cache first and to notify the main process next, but this may lead to timing bugs. Let's consider error cases that put CGE out of service. There are many causes of this condition. If the reporting process detects a problem first and updates its state, the report server might query it for the status and learn that CGE is out of service before the game process is notified. But if the game process is rendering an animation, it won't process the update command from the reporting process until the animation is completed. In the meantime, the game process might detect a different condition that puts it out of service. In the end, both processes will see two errors, but in different order; and as the ordering of events is important, they will end up with inconsistent states.

When it is important to preserve the timing and ordering of the events so that they can be applied in the same order on both sides, designate one process to always apply all the updates. When an event occurs on the cache process, have it first notify the main process, which will process the command and send out an update with a consistent new state. It's better to incur the extra overhead of waiting for the update, because this guarantees that both processes see the same events in the same order.

We applied these three techniques to different parts of the system. You can use more than one of these techniques to distribute data and behavior between processes, as long as you apply only one technique to a key object, or other piece of data.

At the end of this step, the basic functionality is in place. Test it. In our CGE, at this point we can, for example, play a game and verify that the result of the game is sent by the reporting process to the report servers.

3.5 Step 5. Moving features

Although this is the last step, performing it means revisiting all the previous steps, iteratively, for each feature. In this step we fix all the remaining broken code we marked in step 2.

Once the core of the system works, port existing features, one at a time. Pick the most challenging features first, because they are likely to add some insight into the emerging design. Explore and understand the existing code of each new feature before writing new code.

The new design is still malleable at this point, because it is not constrained by the glue code. The initial split of functionality in step 2 was coarse and not likely to be perfect. As you iteratively add features by re-introducing the old code, you will find problems or missing pieces in the communication model and the definitions of the key objects. Change the design as needed, as soon as you know how.

As soon as some interesting features are re-introduced into the system, you can start evaluating the new system (e.g. measure its performance). In our CGE, it's easy to see performance improvements – asking the game for the the last game won by the player no longer requires waiting for the game animation to finish. The reporting process can reply immediately with its current result.

As an added benefit of having two processes, their deployment can be done on separated computers. This decision guides the distribution of the behavior. Recall that in Step 1 all the code related to reporting was disabled to avoid compilation problems. But now that the existing features must be reenabled, it is necessary to decide *where* to enable each piece of behavior – in the game or on the reporting process. Let's look at an example. Some slot machines use vouchers instead of coins. When a player does a cashout, the machine prints a voucher, which can be inserted in another machine to transfer the credits there. For security reasons, vouchers have no money codified, instead they have an identifier that is mapped to an amount of credits in a report server. A slot machine needs to notify the server when a voucher is printed – so that the server keeps track of that money – and also when a voucher is inserted – to ask the server for the associated credits. Porting of this feature is constrained by the fact that the associated hardware (a bill acceptor capable of reading vouchers) is on the game side, so the functionality of reading the voucher needs to be on that process. After reading the voucher, the slot machine needs to validate its code. But every report server could potentially have a different type of voucher, so the voucher validation should take place on the reporting process, which keeps track of the different servers. The reporting process notifies the appropriate server about the voucher and receives the acknowledgment from the server. Now the credits can be shown to the player, so the reporting process tells the game process to accept the credits associated with the voucher. In short, one important heuristic for distributing the behavior is to keep the behavior in the process which has access to the needed data, hardware or connectivity.

In parallel with this refactoring, the original system is being maintained, so many fixes are applied to the code. Don't try to incorporate these fixes in your code branch. This is a refactoring – the existing behavior, even if it's wrong, must be preserved. After the refactoring is completed, incorporate all the code changes in other branches into the system before attempting to merge the code into the mainline.

4 Tools

In this section we describe some tools we have built to make our refactoring easier. We built a simple tool that finds subject-observer dependencies. To locate inconsistent data updates we built another simple tool. Finally, we built a large automatic integration testing tool to run regression and system tests.

4.1 Dependency visualizer

Our system was using the Observer pattern heavily. The application of the pattern was correct, but we ran into some problems trying to undo some dependencies. We describe this problem in more detail in section 5.4.

It's difficult to keep a consistent image of the system that has dozens, or hundreds, observers. We developed a simple tool for locating the uses of the Observer pattern in the code. We used the tool to find all observers of a subject. Given this simple information, we could identify which subject-observer relationships should be broken up, e.g. because the subject object was big, but the observers were interested in a small portion of that object's state. The tool also helped us find all subjects that have no observers (or only one) suggesting which observers may no longer be necessary.

The dependency visualizer is a simple static analysis tool. The tool reads the code to find connections between observers and subjects at the class level. It is looking for specific patterns in the code, e.g. class definitions where Subject or Observer are listed as superclasses. These patterns are matched using regular expressions.

Having fewer dependencies to account for makes it easier to understand the run-time behavior of the system. As an additional benefit, this tool helped us locate dead code. Over time, many developers have introduced new observers, but failed to remove obsolete dependencies.

4.2 Cache consistency verifier

The lock-step cache update approach (described in section 3.4) is effective only if all the updates are performed diligently and timely on both processes. While a refactoring is in progress not all the code is working making it difficult to ensure that every code change is applied consistently on both processes. We have built a periodic synchronization tool for detecting when the shared data gets out of sync.

At predefined time intervals, the cache process asks the main process (via a command) to send it the current version of the shared data. For better performance, the data was sent as a digest (compressed XML).

Here is the algorithm used when the digest is received:

- Calculate the digest of the cache's current data. If the sizes of the two digests don't match, the data is obviously wrong. Goto (3).
- Compare digests, byte by byte. If they are the same, the cache is up to date, exit.
- Unpack the digest, and replace the contents of the cache with the new value.

There are some conditions when this synchronization step should be skipped. The cache process should not ask for an update when it is processing a transaction, because its data may be inconsistent. For the same reason, the main process should not send the response if it processing a transaction. Because of these conditions, neither process should depend on the timing of the periodic synchronization command, but both need to process it whenever it arrives.

We incorporated this tool into the system and used it to verify that both processes were updating their data correctly. This tool was used during the debugging stage to detect places where code was not updated and to find newly introduced race conditions. Now that the system is in deployment, this code is disabled.

4.3 Test runner

As noted earlier, typical refactorings benefit from automated unit testing [1] – whenever code is changed, tests must be run to ensure that no functionality is broken. In the case of large scale refactoring, unit tests are not enough – automated integration testing is critical. System testing is complex, mainly because the system needs to be tested for all the use cases. There are myriads of tests to write and run, since it is desirable to cover as many execution paths as possible.

We developed a tool for automated system testing to aid our redesign. Given the real-time nature of the CGE, the testing tool has been organized into two layers: infrastructure (the bottom layer), and high-level scripting. To successfully test a distributed system, it is necessary to test the inter-networking between the subsystems, the communication with other entities in the network, and the events that occur at the application layer. To test any scenario, the report servers, the network connections, and the player interacting with the user interface must be simulated. The bottom layer of the test runner is responsible for doing the low-level generation, delivery, and handling of events (e.g. network messages, user clicks, timeouts) that occur outside of the game engine, in real time. It is implemented in C++. The top layer, written in a scripting language, provides a higher level abstraction for writing and executing tests. The upper layer makes it possible to write complete tests (setup the test scenario, perform some actions on the game, and check the results by executing a series of assertions on the test data) using domain specific vocabulary and without recompiling the code of the bottom layer.

The development of this kind of tools can also benefit from other large scale refactorings. For example *extract a library* refactoring can help reuse existing classes, in both the new process and the testing tool.

This new testing tool was developed in response to the needs of our refactoring. After the refactoring was completed, the tool became an added asset for the system. Having this tool will facilitate the testing of future refactorings of the system.

5 Challenges of refactoring BBMs

Having presented the details of our refactoring, it's time to step back and consider some of the challenges of doing large-scale redesign. The biggest challenge is that current refactorings are too small to be useful at this large scale. Therefore it is important to convince the refactoring community that (1) large-scale refactorings, like the one we have described, fit the classic definition of refactoring and (2) more work on this topic would be beneficial to advance the state of practice. Other tasks we found challenging were recovering the conceptual integrity and, surprisingly, dealing with software patterns. Another typical challenge to consider is getting the support of the management.

5.1 Refactoring today

The industry has began to accept the idea of refactorings once the tools for performing them have become commonplace. The tools make the results of applying refactorings predictable. Simple refactorings, such as *rename* or *extract method* [6], are easy to understand and very useful in daily development. The clinical precision of modern refactoring tools comes at stark contrast with the messy code of typical large systems. They don't just seem different, they seem to be opposites, which makes it difficult to even attempt to bridge the gap.

The largest refactorings described in the literature are *extract class hierarchy* [6], or *move a class within the inheritance hierarchy* [10]. They are presented as a sequence of smaller, fully-automated refactorings so that a complete redesign is performed in small, safe steps. This approach did not work in our case, because the change was too big and too complex to plan it as a sequence of small refactorings.

An important challenge posed by refactorings at the BBM scale is the fact they require a complete understanding of the system. This might sound obvious, but it's not required for typical refactorings. For example: a developer with little knowledge of the existing system can apply some tool-supported refactorings, because their effects are well-known and usually there are unit tests to prove that the system still works. Refactorings of BBMs do not offer that safety net, because they lack tool support and passing unit tests are not sufficient to show that the change was applied correctly. Having automated system testing is very important to ensure that large-scale refactorings preserve the behavior of the system.

5.2 Is this refactoring?

The task of breaking up a BBM into smaller, well-structured pieces is clearly a redesign. Sometimes the term refactoring is used as a synonym to redesign. But we are using refactoring in its original meaning.

By definition, refactoring requires that there are no changes in the observable behavior of the system after the refactoring; that the change is *behavior-preserving*[*]. This is difficult to guarantee, for example, when a single process system is refactored into a multi-processed system.

A successfully applied *large-scale* refactoring produces a system with a modified internal structure that still passes all the original test cases. The fact that the behavior is not preserved at every point during the refactoring doesn't mean that the final result isn't behavior-preserving; it merely indicates that individual steps of the refactoring may be not behavior-preserving. For this reason, we believe that it is possible to define general purpose large-scale *and* behavior-preserving refactorings.

[*]It is desirable that refactoring improve non-functional requirements, such as maintainability or performance.

5.3 Conceptual integrity

The key idea behind redesigning a BBM is recovering the *conceptual integrity* [2] of the system. Conceptual integrity is more than the architecture of the system. It provides a common metaphor for discussing the architecture. That metaphor can guide developers when adding new functionality that was not originally envisioned in the system. If a system has conceptual integrity, it's easier to add new things without breaking the overall architecture.

When the original designers of the system leave, they take the undocumented architecture of the system with them. Knowledge that isn't documented at that time, takes on the form of myths: "Bob wrote this part this way, because... I don't remember, but we need to keep it that way." When design decisions are not shared, adding any code increases the complexity of a BBM and obscures any conceptual integrity that the system still might have.

Recovering conceptual integrity of a system is hard. In our case, it was a process that spanned the entire refactoring. During the exploration stage, we identified some concepts that seemed important and tried to map the major pieces of the existing code to these concepts. Next, to facilitate inter-process communication, we extended the architecture with some new concepts, such as commands. As we were re-introducing features back into the system and refining the design, new architectural elements emerged and they fit nicely into the design. The Systematic untangling of the BBM has made it easier to recognize them.

Protecting the conceptual integrity of a system is equally important. We have developed an attachment to the architecture, because we recovered it. By participating in the process of recovering the conceptual integrity, the current developers feel like the owners of the architecture and they are more likely to protect its integrity as the system grows. Giving meaningful names to architectural elements, careful documentation, and discussions of design tradeoffs will help to preserve the architecture for posterity.

Small automated refactorings need to be applied several times to improve a design. Similarly, many large scale refactorings may be needed to recover the conceptual integrity of a system.

5.4 Problems with patterns: Observer

Refactoring goes hand in hand with design patterns. In the process of refactoring the code, one often introduces patterns into the system to improve the design [8]. Some patterns can cause to trouble if they are overused. Cinnéide and Fagan discuss problems with Singleton, Abstract Factory and Facade [3]. One can also overcomplicate a design by applying Mediator or Visitor too often, but most patterns are beneficial when applied judiciously in the code.

Ironically, BBMs have an uncanny ability to caricature even the most useful design patterns. Consider the Observer pattern [7], used to decouple event producers (subjects) from event consumers (observers). The use of Observer enables systems to grow much faster, because it decouples different parts of the system. But an unrestrained growth of observers can lead to problems. We have observed several problems caused by adding subject-observer dependencies blindly, without checking the existing dependencies and removing ones that are no longer used. When nobody understands the system architecture completely, but many developers are updating it in parallel, observers are added indiscriminately. While these problems have only marginal impact on working systems, they make breaking up a system that has many such dependencies much harder than necessary.

Here are some problems we discovered, along with possible solutions. We gave these problems pet names to make it easier to refer to them during the refactoring.

- *My 15 seconds of fame.* In a system with hundreds of subjects and observers, a single, seemingly trivial, `update()` call can bring the system to a sudden halt, if the call causes other, marginally related, updates be called *just in case*. Eliminating these update chains requires having a global view of the system. They can only be removed one at a time, and carefully tested.
- *Miss Universe.* Large, important objects may over time become the subjects of many objects in the system. When they change, the whole system is flooded with change notifications. But most observers are likely interested in a specific aspect of these objects. Make these objects announce changes of their different parts separately, by calling different `changed()` methods.
- *Telephone game* (or *My observer's observer's observer*). Object A observes object B, but it is not directly interested in the update. Instead, when object B changes, object A notifies its observers (object C) that it (A) has changed. When C is notified that A has changed, it deduces that B must have changed. The obvious solution is to make C observe B directly.
- *Too shy to hold hands.* Sometimes the subject-observer relationship is added between objects that are already coupled. Two objects keep direct references to each other, but instead of communicating directly, they broadcast updates via the change notification mechanism. Removing the unnecessary `update()` call improves performance, but more importantly it makes the code easier to understand.

Bigger systems are more intimidating and harder to change, because they require more effort and more care to undo small things that grow unwieldy over time. This uncontrolled growth of observers might be caused by the fact that the Observer pattern has good reputation, so developers that do not fully understand the system see it as a safe way to add functionality without going through the trouble of understanding the existing code.

5.5 Fear of big changes

We were lucky to enjoy full support of our management throughout the refactoring process. But often the biggest obstacles to redesign aren't technical.

Working software is rarely brought to an overhaul. A large-scale refactoring is a serious overhaul, even if the refactoring is performed in a separate code branch. At some point, the code needs to be merged and the sheer size of the changed code will make it time consuming and difficult.

Managers tend to discourage big redesigns, because they know that the system will be broken temporarily. Even when a refactoring task is underway, the uncertainty about when the system will be working again adds extra pressure for developers, especially if the current team is not the one which has developed the original architecture. As they do not know the details of the system, the developers are afraid of making big changes. Therefore refactoring tasks need to be planned in order to mitigate risk. Roock and Lippert [10] provide a lot of good advice on planning refactorings.

When management discourages good practices like refactoring, developers get uncomfortable with the code they maintain. They know the code is troublesome, so they want to fix it, but there is no time or resources allocated for that. As the system evolves, the discontent of the developers grows and makes the work frustrating.

6 Evaluation

There are many lessons to learn from performing a large-scale refactoring. Most of our advice is sprinkled throughout this report. In this section, we present some summary observations about our project. We also consider it from a broader context of software engineering – what insight we have gained about redesign in contrast to reuse, and about modularity.

6.1 Outcome of the refactoring

Our refactoring was a success; the redesigned system was delivered on time. The main business goal, improving the response time, was achieved; the response time of the system improved about 3-fold. The main technical goal, improving the system's design, was also achieved. The system has regained a better-focused architecture and the reporting subsystem is clearly separated from the main logic. Moreover, we have produced some design artifacts, such as high-level documentation, that didn't exist before.

The redesign took about six months of full-time work of the first two authors. Most of the time we were pair-programming. We spent the most time working on step 1 and step 5. Step 1 lasted for about two months. Step 5, combined with the integration testing, took about the same time. We needed two weeks to add the first large feature, and one week to add the second one, but once we figured out how to perform that step, adding the remaining features was much faster.

Most of the integration testing was done by two other developers. Our refactoring did not affect the entire codebase, we touched about 300 KLOC. Not counting the code of the new tools, the redesign *decreased* the size of the codebase by about 5 KLOC. This result is worth repeating: despite adding a lot of new code to facilitate distributed processing, we removed more code than we added.

One important lesson we have learned was not directly related to refactoring. It turns out that redesign, even on the BBM scale, is relatively easy compared to the difficulties posed by multithreading. We have spent a lot of time coding, refining, debugging, and rewriting the interaction between threads. Our cache verifier tool helped us detect race conditions. In the end, our documentation describing how the threads ought to interact will probably be the most often used design artifact we have produced.

Since we have only applied this refactoring to a single system, we don't know how widely applicable it is. To evaluate our approach thoroughly, we should apply the same refactoring to other projects. We're looking forward to doing that in the future.

6.2 Redesign vs. Reuse

Refactoring techniques help to redesign the structure of software systems. But most advances in software design techniques (objects, then components, now services) are driven primarily by reuse. These advances seem to suggest that doing big redesigns is not a good idea, that the best way to build software is to connect the existing pieces together. But is reuse always better? in every context?

We hope to have shown in this report that large-scale redesign is feasible and should be done. The question of redesign vs. reuse is relevant in other domains too. For example, in the Web services world, both approaches, redesign and reuse, are tried side by side. RESTful Web services champion redesign, while the WS-* services focus strictly on reuse. So far, there is no clear winner [9].

The WS-* (a.k.a. *SOAP*) Web services promote the reuse of existing systems. The Web service simply exposes the API of the existing system via XML and makes the system Web-accessible without making any changes to the code. Some IDEs offer a push-of-a-button conversion of the API into the WS-* services. But as the ever increasing numbers of software security vulnerabilities indicate, most systems cannot be converted into Web services automatically; some redesign is required.

In contrast, the REST model suggests rewriting the API of a system before it is Web-enabled. The API needs to be aligned with the architecture of the Web, which is based on universal resources accessible through a small set of methods. Changing the public interface of a system to fit the constraints of URI and HTTP standards is also a large-scale refactoring, but the result of turning existing systems into RESTful Web services is more predictable. These services fit the architecture of the Web and obtain many of its benefits for free, e.g. addressability. In short, RESTful Web services *reuse* most of the underlying implementation, but they *redesign* the public API.

6.3 Modularity

As a system ages, the borders between its architectural components erode. In our case, the erosion occurred because the requirement to support the reporting functionality was added after the original game was already built. Accommodating this new requirement into the existing system resulted in scattering new code throughout the existing modules. In this way the reporting feature could not be completely modularized, compromising its own maintainability and the maintainability of its host (the CGE).

This problem is known in the *separation of concerns* community as the dominant decomposition dimension problem [12]. It originates crosscutting concerns. Some concerns are crosscutting to a particular decomposition – in this case, to the original decomposition of the CGE. This first decomposition is dominant and a requirement that comes after and does not fit completely, becomes crosscutting. Its implementation gets scattered in other modules, and the code in other modules is tangled with the code implementing the new requirement. That is what happened with the implementation of the reporting functionality being added to our CGE.

Performing the *extract subsystem to a separate process* refactoring helped us recover the modularity of the system by extracting a new component which was previously buried in the code of the CGE. Modularity gained from this refactoring facilitates the evolution and maintainability of both processes, especially of the reporting subsystem that now encapsulates all the reporting logic. This refactoring was suitable to improve modularization in this particular case. Other large-scale refactorings should be applied in other contexts. If having a separate process wasn't our primary focus, we could have simply extracted the reporting functionality into a separate library, which is another potential large-scale refactoring.

7 Conclusion

Refactoring techniques are great aid in keeping software lean and fresh, but they are not prevalent in industry. One reason is that popular refactorings are small and they do not offer immediate impact in redesigning large systems, ones that could really benefit from refactoring. Many large systems are Big Balls of Mud: they seem to be working, have many users, but are very difficult to change. Our experience indicates that BBMs can be refactored, but to do so they require equally big, large-scale refactorings. As of yet, no descriptions of such refactorings are available and this report is an attempt to fill this void. We hope that our experiences will be helpful to other developers who are facing a similar task.

Acknowledgments

The authors would like to thank Eloy Colell, Juan Antonio Zubimendi, and Javier Búcar for their help during the project as well as during the writing this report. We also would like to thank Baris Aktemur, Munawar Hafiz, Andrzej Leszczynski, and Diego De Sogos for reviewing an earlier draft.

References

[1] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison Wesley, 2004.

[2] Frederick Brooks. *Mythical Man-Month: The Anniversary Edition*. Addison Wesley, 1995.

[3] Mel Ó Cinnéide and Paddy Fagan. Design Patterns: the Devils in the Detail. *PLoP*, 2006.

[4] Michael Feathers. *Working Effectively with Legacy Code*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[5] Brian Foote and Joe Yoder. Big Ball of Mud. *Pattern Languages of Program Design 4*, pages 277–288, 2000.

[6] Martin Fowler. *Refactoring: Improving The Design of Existing Code*. Addison Wesley, 1999.

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[8] Joshua Kerievsky. *Refactoring to Patterns*. Addison Wesley, 2004.

[9] Leonard Richardson and Sam Ruby. *RESTful Web Services*. O'Reilly, 2007.

[10] Stefan Roock and Martin Lippert. *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*. Wiley, 2006.

[11] Margaret-Anne Storey, Li-Te Cheng, Ian Bull, and Peter Rigby. Shared waypoints and social tagging to support collaboration in software development. In *CSCW '06: Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, pages 195–198, New York, NY, USA, 2006. ACM Press.

[12] Peri L. Tarr, Harold Ossher, William H. Harrison, and Stanley M. Sutton Jr. degrees of separation: Multi-dimensional separation of concerns. In *ICSE*, pages 107–119, 1999.

[13] Gramma Tech. Codesurfer.