Killer "Killer Examples" for Design Patterns

Carl Alphonce
Department of Computer Science & Engineering
University at Buffalo, SUNY
Buffalo, NY 14260-2000
alphonce@cse.buffalo.edu

Michael Caspersen
Department of Computer Science
University of Aarhus
DK-8200 Aarhus N, DK
mec@daimi.au.dk

Adrienne Decker
Department of Computer Science & Engineering
University at Buffalo, SUNY
Buffalo, NY 14260-2000
adrienne@cse.buffalo.edu

ABSTRACT

Giving students an appreciation of the benefits of using design patterns and an ability to use them effectively in developing code presents several interesting pedagogical challenges. This paper discusses pedagogical lessons learned at the *"Killer Examples" for Design Patterns and Objects First* series of workshops held at the Object Oriented Programming, Systems, Languages and Applications (OOPSLA) conference over the past four years. It also showcases three "killer examples" which can be used to support the teaching of design patterns.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education—*Computer Science Education*

General Terms

Design

Keywords

Object-orientation, Design Patterns

1. WHY TEACH DESIGN PATTERNS?

The underlying premise of this paper, and indeed of the workshops from which it derives, is that students need to learn skills and concepts which will be of long-term value to them even as the technology of the day changes. We believe that design patterns are an important part of a student's education in this regard.

However, giving students an appreciation of the benefits of using design patterns as well as an ability to use them effectively in developing code presents several interesting pedagogical challenges. This is especially true for instructors of introductory courses.

The first challenge is that students tend to focus on the input-output behavior of their programs rather than high-level properties of their code. While the input-output behavior of a program is measure of its correctness, there are other aspects of software design that are important. These include the ability of a software solution to scale from small problems to large problems, the degree to which the software is extensible, how robust the software is, and so forth. Workshop participants have observed that students do not pay enough attention to these properties of software in their coursework. Knowledge and use of design patterns highlight these broader issues because they are in large measure the *raison d'etre* of design patterns.

The second challenge to educators is that students often do not believe that what they consider to be very "abstract" design pattern solutions are used or even desirable in fast-paced real-world settings.

A third challenge is that examples which benefit from the application of patterns tend to be more complex (in the sense of involving more code and a "richer" domain) than typical textbook examples. This can make it more difficult for students to grasp the examples and discern their essential characteristics. Pattern catalogs, such as the classic "Gang of Four" [4], are great resources for faculty, but not necessarily for design pattern novices. Faculty need more accessible examples to support their teaching.

A final challenge in teaching design patterns is that examples which are constructed by faculty to demonstrate the power of patterns run the risk of lacking "street cred": students can easily perceive them to be ivory tower products with no grounding in real-world software development. For this reason real-world examples are valuable, since they drive home points better, but they are generally much too complex to present directly to students.

Erich Gamma, in an interview with Bill Venners [5], says of learning design patterns that "You have to feel the pain of a design which has some problem. I guess you only appreciate a pattern once you have felt this design pain." His point is that students will not appreciate design patterns if the are presented to them without an appreciation of the problem they solve. On reflection this makes pretty good sense since a pattern is a solution to a problem in a context. If you don't believe in the problem – if you don't own it– how can you ever appreciate a solution? No problem, no solution!

The "Killer Examples" workshop series was born of a desire to gather examples of design pattern use which address these challenges.

2. WHAT IS A "KILLER EXAMPLE"?

SIGCSE'07

We define a "Killer Example" to be one which gives overwhelmingly compelling motivation for something. The term is inspired by "Killer App", which is described by the *Jargon File* [1] as, "The application that actually makes a sustaining market for a promising but under-utilized technology."

The "Killer Examples for Design Patterns and Object First" workshops have been held at the OOPSLA (Object-Oriented Programming, Systems, Languages and Applications) conference annually since 2002. In the first four years we have had eighteen examples presented at the workshops. Approaches to teaching design patterns in various settings have also been discussed at these workshops.

This paper shares some of the general lessons we have learned from the workshops, as well as three examples presented at the workshops which we feel best demonstrate what it means to be a "Killer Example".

3. LESSONS LEARNED: THE PEDAGOGY OF "KILLER EXAMPLES"

Many lessons have emerged from the workshops. The most important and recurring ones are described below.

Context Design patterns cannot effectively be taught independent of an application of it. Patterns must be presented in a context which clearly demonstrates the usefulness of the pattern in comparison to the software built without the pattern.

Accessibility Design patterns cannot effectively be taught if the examples used to demonstrate the benefits of the patterns is too complex or too far removed from the experience of students to be meaningful to them.

Real-world Design patterns cannot effectively be taught unless the examples which demonstrate their application and benefits have a real-world grounding. Since patterns are mined from practitioner code, this is important.

Clear benefits Design patterns cannot effectively be taught unless their benefits in terms of desirable high-level properties of software, such as scalability, robustness, extensibility, flexibility and maintainability are clearly evident.

3.1 Intra-Pattern considerations

Although a single "killer example" may demonstrate the use of several design patterns, it is important that for each pattern students move through a sequence of stages of exposure to a single pattern – we therefore refer to these as *intra-pattern* considerations. These stages are motivated by the "read-before-write" pedagogical pattern.[2]

Use it Students should gain an appreciation of the usefulness of a pattern by using an implementation of it. For example, when learning the Iterator pattern students should gain experience by using an Iterator to traverse some collection.

Conceptualize it Students should be engaged in a discussion of the general architecture of a given pattern. For example, when learning the Iterator pattern students must come to understand the concept of an iterator; alternate approaches, such as a cursor, must be discussed.

Build it The next gain in understanding comes from a student's implementation of a pattern. When learning the Iterator pattern students must next create a class that is an iterator over some collection.

Analyze/study high quality code A deeper understanding of any pattern comes from studying a variety of high quality implementations of the pattern. In the case of the Iterator pattern it is perhaps at this point that students begin to truly grasp the beauty of having a separate iterator which can access private parts of a collection; in Java this is achieved by defining a class's iterator as a public inner class.

3.2 Inter-Pattern considerations

At some point the focus must shift from a single pattern back to a system of mutually supporting patterns, as demonstrated in a killer example. At this *inter-pattern* level of experience, we find the following stages:

Design and construct Students must at some point apply their knowledge of patterns to design and construct software. Killer examples can serve as useful exercises for students also in this regard.

Evaluate A final step in the process of learning to use patterns comes in being able to evaluate and critique the use (or lack of use) of design patterns in software.

4. FIRST EXAMPLE: FRAMEWORKS

Software reuse, after decades of unfulfilled promises, is beginning to become true in the form of object-oriented frameworks.[1] Industrial developers can build large, complex software systems that are reliable and computational efficient because they do not build from scratch; the reuse the vast effort invested into software frameworks such as the Java 2 Enterprise Edition, Java Swing, or Remote Method Invocation (RMI).

4.1 Why Frameworks?

Good object-oriented frameworks are unique examples of the strength of the object-oriented paradigm. Looking behind the scenes of good frameworks shows how careful modeling of domain concepts, use of polymorphism, and the use of design patterns makes a piece of software highly flexible and demonstrates the power of low coupling and high cohesion. It is simply a brilliant case study to learn from, and as such the ultimate killer example of the use of design patterns. The framework we present is developed specifically for educational purposes at the introductory level; the framework encapsulates the MVC design pattern.[3]

4.2 Framework Essentials

The essential characteristics of software frameworks are inversion of control and hotspots.

Inversion of control Typical novice programs consist of a number of interacting objects and a single driver that does the setup and defines the main flow of control. The novice programmer applies services provided through classes that are part of the program or through

[1]This example is due to Michael Caspersen. It was presented at the 2003 workshop.

library classes (e.g. collection classes). When programming using a framework, the main flow of control is out of the programmer's sight; it is dictated and controlled by the framework. The novice programmer's task is to supply code that implements interfaces or specializes (abstract) super classes. This is also known as the Hollywood principle: Don't call us, we'll call you.

Hotspots Frameworks define core functionality, control flow, and object collaboration patterns. Application programmers refine frameworks to specific domains by adding code at well-defined points: the hotspots (also known as hooks or variability points). Hotspots can be realized in a number of different ways: call-back methods, delegation to objects implementing interfaces defined by the framework, or subclassing.

A killer example framework must demonstrate the essential characteristics in a simple and convincing way; it must be simple for novices to use and it must be flexible, i.e. allowing a number of distinct, sensible, and interesting instantiations.

4.3 Example: Presenter Framework

The killer example we have chosen is a presenter framework. The presenter framework facilitates construction of multi-media presentations of a domain where the compass-directions are a suitable metaphor for user navigation; Figure 1 demonstrates an instantiation of the framework that shows a presentation of the tomb of Tutankhamon. Using the compass-directions it is possible to visit the different parts of the tomb while pictures and text is being presented to the user.

The presenter framework provides the application programmer with the simple interface shown in Figure 2. This is the hotspot of the framework; an abstract class which the application programmer must specialize to a specific application.

The presenter framework provides the backbone functionality: a large area for displaying images, a smaller one for displaying text, and the four buttons labeled North, East, South, and West. The buttons respond to user clicks by invoking one of the four abstract methods in the abstract class Presenter which the application must specialize.

Instantiating the framework is a matter of redefining the four abstract methods in class Presenter (see Figure 3).

4.4 Model-View-Controller in Action

The presenter framework encapsulates the MVC design pattern by defining a View and an abstract Controller which can be plugged with a concrete Controller and a Model to provide a full application. The overall architecture is sketched in Figure 4.

4.5 Discussion

Frameworks can serve in teaching in several ways. At the introductory level frameworks may serve as a black box that makes even a small student effort into a rather impressive program. Later, the black box can be opened to demonstrate how good frameworks are structured. The presenter framework is also used as a stepping stone toward learning more advanced frameworks; the simplicity of the presenter frameworks makes it easier to grasp and understand the

Figure 1: Instantiation of the Presenter Framework

```
public abstract class Presenter {
    public void showImage(String filename) { ... }
    public void showText(String text) { ... }

    public abstract void northButtonPressed();
    public abstract void eastButtonPressed();
    public abstract void southButtonPressed();
    public abstract void westButtonPressed();
}
```

Figure 2: The abstract class Presenter

```
public class TutankahmonPresenter {
    public abstract void northButtonPressed() {
        guest.move(NORTH);
    }
    ....
}
```

Figure 3: Specialization of the abstract class Presenter

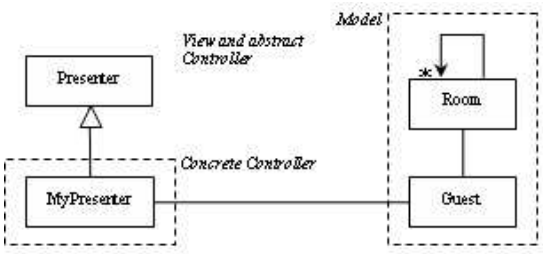

Figure 4: Software architecture

essential characteristics of frameworks (inversion of control and hotspots) and provides a solid ground for working with more complex frameworks (e.g. the Java GUI framework Swing).

We claimed that a killer example framework should allow a number of distinct, sensible, and interesting instantiations. Here are a few such instantiations for this framework:

Virtual museum tour An application which collects pictures of paintings and other artifacts from the Internet and presents a user with a virtual tour of a museum.

Presentation tool The framework forms the core of a presentation tool, along the lines of PowerPoint.

Map navigation Rather than having user interaction generating buttonPressed events, one can have them generated indirectly from a GPS receiver, such that if the coordinates change sufficiently much in a given direction, a buttonPressed event is generated.

While this example does not come directly from a real-world application, the connection to real-world applications is clear and therefore compelling to students.

5. SECOND EXAMPLE: HARDWARE AND SOFTWARE TESTING

Since Design Patterns have grown from the OO community, there are many outside of that community that have difficulty accepting design patterns as applicable to other domains.[2] This is especially true once you leave the software domain and travel to the lands of hardware development, embedded systems, or distributed real-time systems.

In the software domain, and when students study software engineering, an often discussed topic is the idea that when developing large software systems, their development is broken into modules. Those modules are often developed concurrently. The different modules often need to communicate with one another, but development of one module can not stop to wait for another module to be completed.

The same is true in the hardware domain, except some of the modules are software pieces while others are hardware components and their drivers. Developing the software after the hardware is available is often impossible, and both pieces need to be developed and tested concurrently. However, without the hardware to use in the tests, test-driven development, which has shown to be a useful development methodology, can be a challenge.

This example is an industrial example that has been used by a company that develops real-time and embedded systems. They needed to devise a way to develop and test their entire product, the software components and the hardware it will run on concurrently.

5.1 A First Attempt

A naïve attempt to solve this problem is shown in figure 5. A test case is written (TestCase) to test the class/component (ClassUnderTest). ClassUnderTest requires one or more of the hardware components controlled by drivers A, B, and C. The problem with this design stems from the fact that the hardware components are still under development and therefore their drivers are not available. Therefore, testing the class would not be possible until the drivers become available. However, we can introduce a solution which allows us to program to an interface, not an implementation and complete the testing of ClassUnderTest.

[2]This example is due to Bruce Trask and Angel Roman. It was presented at the 2005 workshop.

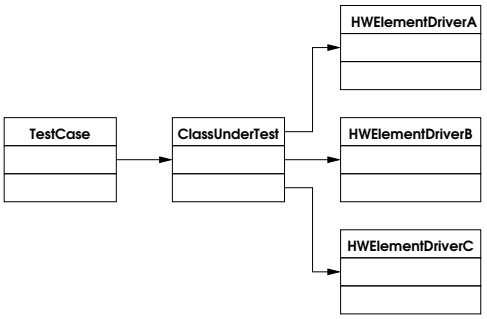

Figure 5: Naïve implementation

5.2 The Strategy Pattern

If we introduce the strategy pattern to this problem, we create interfaces for each one of the hardware driver elements. The drivers themselves have yet to be written because the hardware components are not yet completed. The introduction of interfaces enforces what the drivers will look like (i.e. what methods the drivers will contain). Then, testing can be completed of ClassUnderTest before the hardware is ready. Also, when introducing this pattern, we allow for differences in the underlying implementation of the drivers (i.e. multiple classes that implement the driver interface but actually connect to different hardware implementations).

5.3 The Abstract Factory Pattern

Introducing the Strategy pattern allows us to test the ClassUnderTest independent of the hardware or hardware driver implementations. However, we could have introduced a potential problem. Suppose that some implementations of the driver for hardware component A, only work with certain other configurations of hardware components B and C. We need a way to ensure that the correct configuration of hardware components and drivers are tested. Thus, the introduction of an abstract factory becomes necessary to manage the configurations of the drivers for the hardware components. Then, the TestCase can interact directly with the factory to invoke the proper configuration of the hardware when testing the ClassUnderTest.

Applying both these patterns results in a design as shown in figure 6.

5.4 Why this example is Killer

The applications of the patterns to help solve this problem are not buried in the complexity of the solution to understand. It illustrates the fact that patterns do not always exist in isolation and the introduction of one pattern often necessitates the introduction of more. This example also illustrates that design patterns are not limited to organizations that strictly develop software, but can be used to work with embedded and real-time systems development. It also shows how design patterns can support the test-driven development methodology.

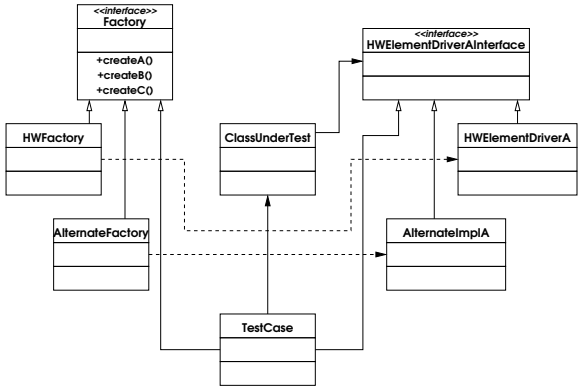

Figure 6: A more flexible approach

6. THIRD EXAMPLE: INTERACTIVE PROGRAM GUIDE

An interactive program guide (IPG) allows a user to browse television (cable/satellite) content in various ways, such as by channel, title, timeslot, and genre.[3] Some systems provide access to weather forecasts. It is also possible to use the IPG to set subtitle or closed captioning options. To control the IPG a user presses keys on a remote control. The remote control typically has a small number of buttons used for navigation and selection. Depending on the current state of the IPG system, different things might happen when a given button is pressed.

For example, selecting a program to watch in the normal TV mode will switch to the indicated channel. However, in pay-per-view (PPV) mode some additional level of confirmation is required, so that a user does not accidentally incur a charge for a program they do not wish to pay for.

Similar systems are used in hotels to present guests with various kinds of information. For example, hotel systems allow guests to order things as diverse as movies and room service. They typically also allow guests to view their hotel bill on-screen and also to check out.

This example is especially interesting because it is a real-world example combining a large number of patterns which nonetheless is accessible. Among the many patterns incorporated in this example are state, model-view-controller, observer, iterator, composite, command, singleton, and proxy. The role of a few of these patterns in the example is presented below.

6.1 Iterator Pattern

The iterator pattern is used to allow the IPG system to traverse a variety of data structures, representing things such as channels, groups of channels, programs, etc. The IPG system maintains a "current" position during browsing, something that lends itself to implementation using a bi-directional iterator.

6.2 State Pattern

An obvious design issue is that the system is *state-based*. In other words, its behavior is governed by the particular state that it is in. Indeed, the behavior associated with all the buttons on the controller change together as the state of the IPG changes. This is modelled this using a state pattern.

Using the state pattern in this example helps to ensure robustness: the behavior of the system is always coherent, since the behaviors associated with a collection of buttons is changed *en masse*.

6.3 Command Pattern

The command pattern is used to represent the behaviors associated with particular buttons on the controller. Because these behaviors are "objectified" as command objects the system retains the flexibility to easily accomodate new menus with new features.

6.4 Mediator Pattern

The mediator pattern is used to maintain loose coupling between components in the case where the IPG displays category information in one pane and element information in another, and changes to the category must result in changes to the set of elements displayed.

6.5 Discussion

This example has demonstrated the potential application of a handful of design patterns in a real-world software system. The beauty of this particular example is that it is one that is familiar to most, if not all, students. The domain of the problem is therefore immediately accessible to them.

7. CONCLUSION

In this paper, we have discussed three "Killer Examples" that introduce students to problems that lend themselves nicely to solutions using design patterns. Many of the complaints of instructors about teaching design patterns stem from the inability to find examples that show the utility of patterns. Many examples are of "toy problems" that do not show the usefulness of the pattern in a larger context, or the examples involve a system that is too complex to break down. A unique balance has been reached in these three examples that allows an instructor to provide a problem and a problem domain that is accessible to students that points to where design patterns can be useful and beneficial to the overall system.

The example used to illustrate patterns is arguably the "make or break" point in a student's pattern education. If patterns are presented as some lofty educational-only idea, students will not see them for their usefulness in real-world software development settings. If patterns are viewed and presented by educators as a real-world-only problem, then students will miss out on an opportunity to be exposed to a beneficial tool for software engineering early in their careers.

8. REFERENCES

[1] The jargon file. http://catb.org/~esr/jargon/.

[2] J. Bergin. Some pedagogical patterns. http://csis.pace.edu/ bergin/patterns/fewpedpats.html.

[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture*. John Wiley & Sons, 1996.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[5] B. Venners. How to use design patterns – a conversation with Erich Gamma, part I. 2005.

[3]This example is due to Asher Sterkin. It was presented at the 2003 workshop.