From Hello World to Interface Design in Three Days: Teaching Non-technical Students to Use an API

George M. Wyner
Boston University School of Management
gwyner@bu.edu

Benjamin Lubin
Boston University School of Management
blubin@bu.edu

ABSTRACT

In this preliminary report we describe ongoing research aimed at developing an effective method for giving business students the experience of using an API. Specifically, we seek a method that is accessible and useful even to students with no technical background. The approach we have taken is to teach students a small but sufficient amount of the Python language and then provide them with the API to a restaurant information system that we have developed specifically for the course. Student teams use the API to implement a simple information system tailored to the specific work flow of a restaurant of their choice. Anecdotal evidence suggests that students have found the programming experience useful and that further investigation of the proposed method is warranted. This paper describes the approach taken and the reasons behind the approach in enough detail so that the reader can assess its potential relevance to his or her own curriculum and find some guidance on how to develop similar approaches.

Keywords

Management Pedagogy, Curriculum Development, Introduction to Programming, API.

INTRODUCTION

This paper describes preliminary research towards the development of teaching materials used to give MBA students with a non-technical background a hands-on introduction to the concept of an application programming interface (API). Our focus here is primarily on describing the approach we have taken to teaching these concepts, including extensive commentary on what we consider to be the key elements of this approach and our rationale for the key decisions we made in developing this material. What this paper does not do is to provide an evaluation of how our work compares to existing alternatives for conveying this type of technical content to non-technical audiences. Neither do we provide a rigorous evaluation of the benefits we ascribe to our approach, although as we will describe below, there is anecdotal evidence to suggest our approach was perceived as useful by the students who have experienced it. Our hope is that, while preliminary in nature, this research in progress will be of use to others engaged in the challenge of usefully conveying technical content to management students. Our goals for this paper are two-fold: First, to describe the approach we have taken and the reasons behind that approach in enough detail so that readers can assess its potential relevance to their own curricula; for those so interested we offer some guidance on the development of similar approaches. Second, we seek to provide a foundation for future empirical work in which we systematically assess the usefulness of our approach and its applicability to future business school curricula.

TEACHING GOALS

We developed this module in the context of a one week intensive course on business architecture which is taught as part of dual degree program where students concurrently pursue an MBA and an MS in information systems. The program is targeted at students with varying backgrounds, including those with strong technical skills and those with no previous experience with programming or managing information technologies. The goal of the program is to develop managers with a deep grasp of the capabilities of information technologies and the methods by which they can be deployed in the service of an organization. The program is not positioned as a "techno-MBA" in that the goal is not to train managers specifically for the technology industry but rather to prepare them to employ information technology to advantage in any industry.

In order to provide students an opportunity to complete the second degree in the same time frame as single degree MBA students, the program includes a number of intensive courses in which a full or half semester course is delivered in one or two weeks by means of full-day executive education style classes in which students are in the classroom from 8am until 6pm, Monday through Friday. These intensive classes need to be delivered when regular classes are not in session, specifically in early January, before the start of the spring semester, and again in late May, after the spring semester and before students begin their summer internships.

The particular intensive course for which we developed our approach occurs in January of the first year of the program and is the first contact students have with the technology content of their degree program. The course has an ambitious goal which is to take students from the lowest levels of architecture, such as the architecture of a central processing unit (CPU), to the highest levels of architecture, such as a business architecture in which information systems are but one component.

A central goal of the course is to teach students not only what an API (application programming interface) *is*, but also a sense of how APIs work and why they are important – and thus why analogous concepts across computing and web architectures are likewise important. Given the move to cloud computing, software as a service, and service oriented architectures, managers are now in a position to build solutions by plugging together diverse technologies. The potential business value of these technologies often rests in the formal description of the services they provide, which is to say the API that they expose.

THE CHALLENGE

We must confront the basic issue that our students have for the most part non-technical backgrounds and thus, for many of them, this is their first introduction to programming. Previous experience with teaching technical material to business students has suggested the efficacy of giving students hands-on experience with technology. However, given the intense nature of the class schedule and the students' limited technical background we must be selective so as to create a practical structure and scope of work. In doing this, we seek to create an experience that is compelling to students whose primary goal is not to learn to program, but rather to understand the business implications of the technologies presented.

PREVIOUS CURRICULA

In previous years we have had students obtain practical knowledge of an API by creating *Mashups* using Yahoo Pipes. This was in many ways a very successful exercise. *Mashups* are fun to create and are appealing as examples of relatively new web technologies. The graphical interface provided by Pipes and the set of built-in capabilities allows students to create impressive looking *Mashups* with a small amount of effort and with no programming. Ironically, this advantage is also a drawback to this approach: the graphical interface in Pipes makes it easy to use, but obscures what is going on underneath the hood. Yet, it is precisely some of these details that we want to make visible to the students.

Even more problematic, the most interesting (business relevant) examples of an API in the Pipes environment involve connections to external sources of data such as web pages and newsfeeds using the Really Simple Syndication (RSS) protocol. In this case the API is embedded in the construction of the Uniform Resource Locator (URL), either as a path to a resource as when using the Representational State Transfer (REST) approach (Richardson and Ruby, 2007), or as a set of parameters. This use of the URL as the API to a website not only obscures the notion of an API as a request for service but is quite opaque to students with limited experience with the technical aspects of the world wide web, which is to say most of our students. Using actual web services would be one way to make the API more visible but Pipes has limited support for interacting with web services that are defined using the Web Services Description Language (WSDL), and this approach to web services would be potentially even more impenetrable to a newcomer than the programming-based approach we describe in the sequel. Finally, the graphical nature of Pipes obscures the implicit program that is being described, and creates a substantial pedagogical gap with the hands-on programming exercise we eventually gave the students, namely a self-contained use of a scripting language like Ruby or Perl.

Due to these drawbacks in the use of *Mashups,* we have sought a way to enable students to *quickly* be able learn about architecture and APIs in the same environment where they learn simple programming: a text editor and an interpreted language. This change lets us maintain continuity between the student experience in our hands-on programming labs, and their exposure to architectural considerations at the API level and beyond. In particular, we teach students about functions and their parameters and then introduce the API as a set of such functions. However, rather than building up an API slowly, as may happen in most traditional courses, we instead create a complete API for an imaginary system, enabling our students to focus on complete interfaces early, rather than on the programming details. The trick is to keep this early exposure to complexity from becoming confusing.

The API we provide consists of a flat set of self-contained functions that are called by the student when writing short programs. While not as exciting as accessing a well known API like that of Amazon or Twitter over the web, this has fewer moving parts and enables us to carefully manage the complexity of the API and provide documentation tuned to the students' background and the context of the course. Students thus obtain a toolkit which is well integrated and complete, and which they can thus more easily comprehend as a whole.

In addition to our prior experience with *Mashups*, another area of previous curricula has influenced our decision to adopt this approach. Consider two options: 1) teach students a complete but hugely simplified machine architecture with a ten op-codes or 2) teach students a tiny subset of a high-level scripting language like Ruby. We have found that students are far more at ease, and far more successful with the material in option 1, even though the machine language instructions are far more technical than the more friendly high-level language in option 2. We hypothesize that students are more comfortable because they can quickly understand the complete instruction set (ten simple commands) as opposed to the richer set of options provided by a high-level language. Based on this admittedly untested hypothesis we foresaw a potential advantage to our small home-grown API approach over larger real-world examples: if we limited the size of the API and provided complete and readable documentation, then students would be able to digest the possibilities afforded by the *complete* API and think creatively about how to deploy it to solve a problem.

Finally, the rough hewn nature of the API is itself a potential advantage. We make clear to students that this API is being built just for the assignment and may have shortcomings. We encourage students to report bugs and request enhancements. We believe that if we can take non-technical students to a point of engagement with an API where they are able to identify non-obvious shortcomings, then they have internalized the concepts in a way that will well position them to subsequently take this same critical perspective to the selection and use of real world interfaces. When this happens, we consider it to be a great outcome for the course.

A desire to enable a rapid response to such change requests has led us to develop and deploy the API via an industry standard Subversion (SVN) version control system (hosted on Google Code). This not only facilitates remote collaboration among those creating the API, it also enables us to expose students to a version control system and concretely explain why such platforms are useful in development. In the 2011 version of the course, we released several revisions to the API during the exercise including updates to both code and documentation.

THE TEACHING MATERIALS

This analysis suggested that we present students with a series of labs that embody this API-based approach. The labs introduce students to programming, including all the concepts they need in order to work with the API we subsequently provide. This includes: how to write code into a text file and execute the resulting program, the use of variables to store information, basic input and output, and simple mathematical operations and manipulation of text in the form of strings. Finally students need to learn some control flow, including looping and if statements and, most critically for the concept of an API, about functions, their parameters and return values.

We use Python as our programming language for several reasons. Since Python is an interpreted language, students can try things out using an interactive command line. Python has a very gentle learning curve. While Python is object oriented in nature, this can be hidden from view (unlike in Java or Ruby) so we can give students an introduction to procedural programming without the need to master the notion of objects. Objects are *not* the goal of this particular course, and thus rather than the "objects first" or "objects second" styles some courses use, we prefer to avoid the question entirely in this course with an "objects never" approach. Finally, Python uses a particularly simple syntax with a minimum of special characters like ";' at the end of commands or "{}" to enclose blocks of code. While Python's peculiarity of using white space to denote a code block is perhaps controversial to aficionados of other languages (which studiously avoid making white space significant), we find the resulting visual structure and enforced style compliance to help our neophyte programmers, who can intuit what a code block is almost without being told, and who find the cleanliness of the syntax welcoming.

We chose to have students use Eclipse as an integrated development environment for their Python programming. While Eclipse was originally developed for Java development, it has a very usable Python plug-in which we have students install. We believe it is useful for students to see what an integrated development environment (IDE) is like. While Eclipse provides students assistance in the development process, the operating parts of the system continue to be very visible: students see both the code they write and the results of its execution with no graphical user interface (GUI) intervening to obscure this connection.

We have structured the course such that the first three labs introduce students to Python programming, while the fourth and final lab has students working with our API. The labs are all required but ungraded. Our goal is to take away the anxiety of having non-technical students working with this material by allowing them to collaborate informally during the lab. Our one requirement is that each student needs to get each lab to work on his or her own laptop. While the first three labs are essentially individual projects (although the students are encouraged to help each other work on them), the fourth lab is a team exercise as described below.

Labs 1-3: Programming in Python

Lab 1: Getting Started with Python

The goal of Lab 1 is to have students install Eclipse and write their first program. This is the traditional "hello world" program, which simply prints the message "hello world" on the user's screen. The complete code for this program in Python is:

```
print "Hello World!"
```

In this case we give the code to the students, and ask them to download, install and get comfortable with the programming environment.

Lab 2: Algorithm Implementation

In this lab students are asked to write a series of simple Python programs. The programs build on each other, culminating in a simple cash register program. We choose this application for two reasons: first, a cash register is nothing if not business-relevant and second, the cash register is very simple and allows us to introduce three important programming constructs: a loop with simple arithmetic, branching (decision making), and user input and output.

A lecture precedes the lab and introduces all of the necessary concepts. During the lecture students are encouraged to try out the syntax being described using a Python interpreter on their own laptop.

The students are asked to build the following programs in the lab:

- o Prompt the user to enter a series of numbers and then print out the total, using a simple loop and basic input/output.
- o Next add the following functionality: calculate sales tax (fixed at 5%) and print a grand total. This step requires students to make more extensive use of variables and adds incrementally to the complexity.
- o Finally, add the ability handle tax exempt customers. This requires students to understand and use an if-then-else construction. A sample output of this third program is included in Figure 1.

```
Tax exempt? (y or n):  n
Item: 3.95
Item: 6.95
Item: 7.95
Item:
Subtotal: 18.85
Tax: 0.9425
Total: 19.7925
```

Figure 1. Sample Output of Lab 2, Part 3

Lab 3: Using Functions and Lists in Python

This Lab introduces two new concepts: function calls and lists. Again the material is discussed in advance in a lecture. We choose to introduce lists so that students can begin to see how a data structure can be built up, the list being perhaps the most basic example of structured data. Information systems move data around in chunks, via objects, database result sets, JavaScript Object Notation (JSON), Extensible Markup Language (XML), and so on. This use of structured data is essential to all but the most trivial APIs and we thus consider it an important topic not only for grasping the notion of an API, but for subsequent course work as well. In the API provided in Lab 4 we make occasional use of more elaborate data structures; armed with their understanding of lists, students are able make sense of these structures by analogy to lists, implicitly learning something of encapsulation in the process.

In lab 3 students are asked to recreate the third program from lab 2, now dividing the functionality into three separate functions. We provide students with a definition of what each function ought to do. Students are asked to add in the code needed to make the functions work. For two of the functions we give students the function declaration, and ask them to code the body (based on their work in lab 2). For the third function students have to provide the function declaration as well as its definition. Students' use of lists is enforced by stipulating lists in the arguments of the provided function declarations.

In working on lab 3, students are encouraged to refer back to the previous lab and reuse their already written code. Students can thus see how introducing functions can make existing code more modular. This is connected to a lecture on modularity which introduces students to these concepts (Baldwin and Clark, 1999; Simon, 1981; Messerschmitt 2000). This opens the possibility of a high level discussion about the desirability of modularity and modular design.

Lab 3 gives students an initial experience with an extremely basic API: one that consists of exactly 3 functions. It thus sets the stage for the fourth lab, where students work with a far more elaborate API.

Lab 4: Building a Restaurant Information System

The fourth lab is designed as an extended exercise in using an API to deliver meaningful business functionality. As noted above, a key element of our pedagogical strategy is to carefully design our own custom API to precisely fit the needs of our students, rather than forcing them to confront an overly complex real-world system. This enables us to explicitly seek a balance between minimizing the complexity of the API and providing students with an opportunity for creativity in how they use the API.

Given our goal of relating programming interfaces to higher level business decision-making, we focus on a situation that explicitly involves process design. Specifically, we choose a restaurant domain because of its familiarity to the students and because of the tremendous variation in how the work of a restaurant can be organized (Salancik and Leblebici, 1988; Whyte, 1949). We present the students with an API that provides support for the creation of a basic restaurant information system. Our materials describe it as follows:

> The concept is of a restaurant with a touch screen embedded in each table along with a credit card reader. The idea is that the touch screen will be used for ordering food and paying for the food. The API also includes support for managing the flow of orders through the kitchen, both alerting the kitchen to new orders and alerting wait staff to the availability of "platters" of food to be served at the table.

In reality, the code we provide to students is more of a simulation of such a restaurant system, in that we don't actually have them using touch screen displays. Instead this behavior is represented to them on the console: e.g. students can display a (text) menu on the screen, and have "the customer" interact with it.

As Salancik and Leblebici have discussed in their work, there is tremendous variation in how a restaurant might organize the delivery of food to a customer in exchange for a payment (Salancik and Leblebici, 1988). The API is designed to support as much of this variability as possible while capturing the various constraints about the sequence in which tasks can be performed. For example, functions relating to serving food cannot be called unless the food involved had been previously prepared. See Table 1 for a selection of the functions included in the API along with a brief description of their use.

Function	Description
`start_meal(restaurant_name, greeting)`	Initiates the system for a new group of diners. Does not return anything. Displays restaurant name followed by greeting on the line below.
`end_meal(restaurant_name, message)`	Closes out the session for the current group of diners. Does not return anything. Displays message followed by restaurant name.
`show_logo(restaurant_name)`	Displays restaurant name surrounded by a box. Used mainly by start_meal and end_meal but you can use it any time you want to display a logo.
`get_menu(file_name, title)`	Reads in menu information from a CSV file and returns a menu with the specified title. The title parameter is optional with a default value of 'Menu'.
`show_menu(menu)`	Display a menu on the screen
`create_order(waiter, table)`	Create a new Order and return it
`get_choice(menu, order)`	Displays the menu to the user and updates the order to reflect the choices made by the customer.
`show_order(order)`	Print out an Order on the display
`prepare_food(order)`	Notifies the kitchen that there is food to be prepared. Returns a LIST of tickets; one for each plate of food to be created.
`pickup_food(ticket)`	Pick up a new plate of food for a given ticket. Returns the plate of food.
`serve_food(plate)`	Serve a the given food, returns nothing
`new_bill()`	Creates a new empty bill and returns it
`add_to_bill(bill, order)`	Add an order to a bill
`show_bill(bill)`	Display a bill
`merge_bills(bill1, bill2)`	Merges two bills and returns a NEW merged bill
`get_payment_type(bill)`	Asks the customer what type of payment it wants to make and returns

	this
`collect_payment(paymentType, bill)`	Collects payment from the customer

Table 1. Restaurant API

The functions which comprise the API are made available in a set of Python modules (that is, files containing Python code) which students import into their own program. The lab itself consisted of the following assignment:

> In this lab you will work in small teams to develop a restaurant information system using an API. Your team should identify a high level (summary) use case for your restaurant that reflects the nature of the business and how the dining experience will unfold. Is this a fast food restaurant where payment is collected up front? Or a fine dining establishment where a menu is presented at several points in an elaborately choreographed evening of gustatory delight?
>
> The use case does not need to be turned in but you will want to have it noted down someplace for reference. You will all be using this use case as a point of departure for your individual use case assignments and it will be an important reference point in designing your system.

Students are previously taught about use cases as a method for capturing the user experience of a system (Cockburn 2001). For this lab students are asked to create a use case to summarize the customer experience they are hoping to provide in their restaurant. Students then build a working system guided by this use case.

RESULTS

While we have not yet formally evaluated the approach just described, we do have anecdotal evidence that it has been well received by students and provides something of value to them. Here we describe the results of having used this method to teach an intensive course in January 2011:

First, the students were highly successful in understanding and using the API both conceptually and practically. All of the student teams were able to present a working system at the end of the afternoon lab, with one exception. The one student team that did not create working code had deliberately given themselves a harder task to complete: they broke the code into entirely separate coding projects which they then sought to integrate. While they did not have sufficient time to complete this work, their even more modular approach to the project suggests that they had internalized some of the key concepts presented during the previous days.

Our informal conversations with students uniformly suggested that they found the material useful. Several students commented that as a consequence of the perceived usefulness of the course material, they were looking forward to the more extended technical intensive scheduled for this coming summer, although it would be a stretch to ascribe this attitude entirely to the novel approach we took.

Perhaps stronger (though still informal) evidence for the utility of our novel approach comes from the following outcome: for the first time in ten years of teaching technical material to management students, a group of ten students exposed to this new material decided to self-organize into a study group in order to learn more Python on their own. This represents nearly a third of all the students who took this course. Given that in previous years we have had students arguing that there should be less programming taught, it is significant that this year ten MBA students felt the opposite so strongly, they took it upon themselves to create a venue for learning it. One should certainly be cautious leaping to the ascription of a causal link here, but this is certainly encouraging evidence for pursing this new approach further.

DISCUSSION

While our approach has been well received by students and we have anecdotal evidence that it may have value, it is important to note the limitations of the current exercise. The API we developed was of necessity very simple and thus is not usable in a practical setting. Students have therefore not gained experience with a realistic and widely used API. We believe this limitation to be outweighed by the benefits of a more approachable programming experience, especially given the primary goal of understanding how an API works rather than gaining a marketable programming skill. However, the exercise might be even more effective if an established real world API could be adapted for this purpose. A second possible limitation is the use of Python, which is not as widely used in the classroom and in industry as Java or .NET programming. Again, we felt that the benefits of Python's accessibility made it a good choice for this exercise, despite its smaller marketshare. Moreover, Python has been getting increasing attention because of its central role in Google's new App Engine development environment (Sanderson, 2009). One possibility would be to develop a second and subsequent course which leverages students' familiarity with APIs and Python programming to work with a real-world example based on Google's App Engine platform.

CONCLUSION

As noted in the introduction, we have two primary goals for this paper. First, we described the challenge we face in teaching technical material to dual degree business students without a technical background and offered a particular approach to addressing this challenge: giving students a hands-on experience with the use of an API to create business value. We have described both the teaching materials we have developed for this purpose, and the third party tools (Python, Eclipse, and Google Code) which we employed.

Our second goal is the building of a foundation for future work. By providing some modest anecdotal evidence that our approach was effective, we have hopefully made the case that such further work is warranted. Specifically, we propose to collect data to assess the effectiveness of our approach in conveying an understanding of core programming concepts, controlling for the wide range of technical backgrounds and career interests in our student population. One intriguing question is why almost one third of the students exposed to Python in this way decided they wanted to learn more about programming. What do they hope to learn and what in their experience with this material made them see this need? It would appear that at least some of the students who encountered an API in this hands-on way have come to believe that understanding code is a potential specific source of advantage for a manager, which, if true, might indicate the utility of providing a range of additional technical material using the methods proposed here.

ACKNOWLEDGMENTS

The authors would like to thank the Boston University MS-MBA class of 2012 for their enthusiastic participation in this programming experience, and especially to thank Thomas Eisner for his contributions to the development of the API.

REFERENCES

1. Baldwin, C. Y. and Clark, K. B. (1999) Design rules: The power of modularity volume 1, MIT Press, Cambridge, MA.

2. Cockburn, A. (2001) Writing effective use cases*,* Addison-Wesley Publishing Company, Upper Saddle River, NJ.
3. Messerschmitt, D. G. (2000) Understanding networked applications: a first course, Morgan Kaufmann Publishing, San Francisco.
4. Richardson, L. and Ruby, S. (2007) RESTful web services, O'Reilly Media, Inc., Sebastopol, CA.
5. Salancik, G. R. and Leblebici, H. (1988) Variety and form in organizing transactions: A generative grammar of organization, in N. DiTomaso and S. B. Bacharach (Eds.) *Research in the Sociology of Organizations*, 6, JAI Press, Greenwich, CT, 1-31.
6. Sanderson, D. (2009) Programming Google App Engine, O'Reilly Media, Inc., Sebastopol, CA.
7. Simon, H. A. (1981) The sciences of the artificial (2nd Edition), MIT Press, Cambridge, MA.
8. Whyte, W. F. (1949) The social structure of the restaurant, *The American Journal of Sociology*, 54, 4, 302-310.