Behavioral Statement Classifier

Subtitle: **Software Development as a Team Process**

Material related to a presentation on Sunday 25 September 2021 @ 12:00 to the Slack channel, **Behavior Analysts Who Code (BAWC)**

This will change frequently up to the presentation date

Tom Donaldson

tom.donaldson@performanceally.com

Created: Sunday 22 August 2021

Overview: Software Development as a Team Activity

Contrary to common myth, software development is primarily a team activity. The most critical software development skill is "plays well with others".

That is not to say that solo efforts are not important. They are. Many, if not most, exploratory coding is solo. For example, this solo demo project that was done primarily to explore a possible way to facilitate learning about the **Donahoe, Burgos, Palmer (DBP) selectionist neural network model**:

- **BGL2015 Visualizer**

But creating software useful to a broader audience, and more generally applicable, requires a team. There is too much work required to produce software that actually works reliably, and then to maintain it and support users. More than one person is generally required, and these people must work together without stepping all over each other.

The focus of this presentation will be a few software development techniques and tools that support teamwork. I will talk about and demo some basic infrastructure for working together, but does not address the behavioral aspects of teamwork. That is more up to you, behavior analysts.f

How to use this document

There is a ***lot*** of material here. Consider this to be primarily a "survey" that will give you some broad notion of the lay of the land, and the specifics of a couple of areas. This document is a self study guide. This document is intended as a resource to be read before or after the session. I will cover this material very briefly, but links are provided throughout as a guide to more in-depth information.

In light of the ever increasing importance of software in all aspects of life, including in behavior analysis, my major goals:

1. Foster a deeper understanding of software development among behavior analysts to facilitate their own coding as well as make their interactions with professional software development groups more effective.

2. Contribute to the continuing development of an ecosystem of behavior analysts who are software-development aware.

3. Call out important aspects of software development and leave a trail of breadcrumbs (i.e., links and terms) to act as entry points to a deeper understanding for those who care to pursue them.

4. Contribute to the "behavioralization" of software development. I will strive to explain a few practices in behavioral terms. The most productive practices *should* make sense from a behavior analytic perspective. I may need help here. Applied behavior analytic research is needed here.

Future presentations

If participants are interested, the project may be continued in future Saturday workshops outside the usual **Behavior Analysts Who Code (BAWC)** meetup schedule. Again, the purpose is to promote software development skills within the behavior analytic community. It might even result in software that is useful to the behavior analysis community, owned by the behavior analysis community.

Such "workshops" would include online participation by attendees, including taking part in writing and debugging code (e.g., see "**pair programming**"). Ideally, the process would evolve to include Behavior Analysts Who Code members taking ownership of parts of the app, working offline (perhaps in pairs), presenting to the "team", and guiding online collaborative, cooperative, work.

Presentation constraints

The basic drivers for this presentation will be:

1. Wide ranging software development skills of the audience.

    1. Just starting.
    2. Computer science background.
    3. Professional (or retired) software developers?

2. Given the levels of audience experience with software development, this first presentation needs to be very general, and perhaps skimpy on details.

3. The "software development project" must be in some way relevant to behavior analysis. In this case: classification of textual statements as behavioral (or not).

General format

Duration of the presentation: 60 minutes, max. At least half of that will be live demo and questions/discussion. Of course, if there are no question, the presentation will be shorter.

I will cover the following topics superficially to provide a context for the demo. The text here includes numerous links to more in-depth information for those who want it.

For those who are ***really really*** interested, it would be useful to at least skim over the linked material before the presentation.

1. ***Critical Development Techniques:*** Two of the most important software development techniques that you have never heard of.

2. ***Scenario:*** Behavior analytic scenario as the motivation for the project.

3. ***Development Strategy:*** Overall (basic) development strategy in terms of incremental phases shaping the functionality of the code toward the overall goal.

4. ***Major Software Components:*** The logical units of software which implement specific behaviors. Taken as a whole, an application (or model) can be viewed as an emergent arising from all of the components interacting.

5. ***Initial Development Process:*** A tentative set of steps in using the tools to meet the requirements specified for the application scenario.

6. ***Tools:*** Software that will be used in the initial "simple prototype" phase, plus some possible tools for future directions.

7. ***Demo: Correct and Bulletproof:*** Demo with live development project (will have basic framework in

place, then do some coding to illustrate the “Critical Development Techniques”.

8. ***Where to go from here?:*** What might a next step be? Do it in another presentation?

Critical Development Techniques

I am making up my own “behavioralized” names for these techniques, but will, of course, tie them back to the more standard software development names (such as there are). Please suggest better terms.

These are two development techniques that changed my life as a software developer when I was “forced” to start using them. They are techniques that are rarely, if ever, taught in computer science classes. They are interrelated, but each has a different target population. After using them, my reaction was, “How did I ever develop software without this?”

1. Continuous Implementation Feedback. The target population is developers ***implementing*** a unit of functionality.

2. Clear User Expectations. The target population is other developers ***using*** a unit of functionality.

As for all software development practices, whether and how they are performed are very idiosyncratic to the organization, team, individual.

I will give examples of both in the demo, but first, a preview here.

Continuous implementation feedback

The term “Continuous Implementation Feedback” is a behavioralization. The technique is actually known within the software development community as **Test Driven Development (TDD)**.

With respect to “teamwork”, the technique is very important in quickly (relatively) producing working code, and sharing it with team members. A project in which team members do their share of the work, and do not interfere needlessly with other team members, is more likely to succeed than if team members trip each other up. Before sharing code, ***the code itself must provably work, and it must work with the code of the other team members***.

The primary features of this technique are:

1. Define requirements for a unit of functionality.

2. Create some skeleton code to implement the unit.

3. Create some skeleton code to test the unit.

4. Implementation one micro-goal at a time:

    1. Encode the next (or first) micro-goal within the unit test to specify some “vital behavior” that the implementation must master.
    2. Work the implementation until the unit test says that the micro-goal has been mastered.
    3. Rinse repeat until all requirements for the functional unit are fulfilled.

Naturally, TDD is more complicated than this. Or can be. But the essence is breaking a coding task down into micro-goals and achieving them with nearly instantaneous feedback.

The process results in two work products:

1. The main code that “works as tested”, that is, provably provides particular functionality. So long as the requirements for using the code are met, then the code will provide the specified service.

2. The unit test(s) form a suite of tests that can be run at any time. This is critical because any change to one object can affect any number of other objects in completely unexpected ways. Bugs happen, where

"bug" is a programmer error. The classic lament after an application stops working is, "I only changed a comment!" The entire suite of unit tests MUST be run and PASSED before sharing code.

Clear user expectations

This technique is generally some combination of **design by contract** and **defensive programming**.

With respect to teamwork: this technique provides guarantees to your team members, or anyone who uses your code, regarding its behavior under conditions that you, as the developer of a unit of code, specify. You say: do this before calling this code; call it with these arguments; and here is what I will do for you. You are also guaranteeing that if the person using your code does not follow the rules, your code will not give a pass, but will throw an error of some sort.

The Contract: antecedents

The idea here is to clearly specify the conditions for calling a function (method, message, procedure, whatever), and the consequences of doing so correctly or incorrectly:

1. the environment in which a piece of code may be used (e.g, in terms of previous actions);
2. what information the code object must already have;
3. what information must be supplied to the code object that you are calling; and
4. the consequences of fulfilling these expectations, or not.

In developing code, you will write many "public" functions. Public functions are ones that other programmers are permitted to use. There are also "private" methods that only you, as the developer of the code, may use.

In private functions, you control how the function is called (because it is your code hidden within your objects). You can make whatever assumptions you like about how it is used. As long as you don't stab yourself in the eye (metaphorically), the code should behave as expected.

With public code, on the other hand, you have no control over its usage. You cannot known when it will be called, by whom, or where it will be called. Thus, either you must be prepared to handled any and all circumstances, ***or*** you can limit the scope of what you must consider by specifying the "three term contingency" for a method call. This is the contract described above.

Implementation of consequences

The contract (above) is text that attempts to set the occasion for certain behaviors on the part of the software developer. But it is just text. Something must implement the specified contingencies.

As described in the linked pages on design by contract and defensive programming, your code must implement the contingencies. There are various ways to do this, some of which will be illustrated in the demo.

There are two primary ways to enforce the rules:

1. Throw an informative error that may be "caught" and handled. You would generally do this in the case of a runtime error that might not be under the control of the programmer calling your code. For example, a file has gone missing.
2. Deliberately crash the program (that is, "assert fail"). Generally, you do this if you consider the situation to be a simple programming error that should be corrected by the programmer calling your code. For example, you have specified that an argument, such as `first_name` must be supplied, but the programmer called your code with a no value for `first_name`.

This is complicated by the fact that different languages use different conventions with regard to "production" code versus "debug" code. Generally, "assert fails" are NOT included in production code, but errors (a.k.a., exceptions) are. This makes sense when you consider that most code calling issues will occur in development, and by the time you put a code out for other people to use, there ***should*** be few if any coding errors, only runtime resource errors.

The other way to look at it:

1. Assertions provide feedback to programmers during development.
2. Errors/exceptions provide feedback to the users of your system.

Scenario

What is the behavior analytic purpose?

The purpose of the software is to collect and evaluate statements of behavior as part of identifying behaviors important in producing some result, otherwise known as, "pinpointing vital behaviors". Such statements would be specific to particular domains and perhaps settings within those domains.

Thus, the issue/setting being addressed by the statements would have to be identified.

Besides the general need of classifying behaviors, we also want to collect data that will support efforts to improve the entire process: train users to classify statements; the criteria used in classifying statements; the explanations of the criteria; how the user interacts with the system; and so on.

General need

Would like software that would facilitate the process of training people to do the classification, and to collect classifications of large volumes of behavior statements.

For a statement to describe a behavior, for our purposes, the statement must meet the following criteria (see articles below for more/alternate information):

1. Measurable: Must be observable by others than the performer: seen, heard, felt, smelled, tasted.
2. Objective: Must be confirmable by others.
3. Active: Only a living organism can do it (see also: **Dead Man Test**)
4. Neutral: Non-biased, non-judgmental, non-emotional.

Articles on pinpointing

A few relevant articles:

1. A "must read" by Critchfield and Shue: **The Dead Man Test: a Preliminary Experimental Analysis**
2. **How to Pinpoint Behavior Effectively**: More inclusive than our current project. We will focus on the "Movement Cycle" (action verb and object).
3. **Vital Behaviors**

Main data collection need

There are two main use cases relating to data collection:

1. Classifying statements.
2. Training users to classify statements.

It is a classic "chicken and egg" problem: to train users to classify, we need a training set of statements that are already classified so that we may provide feedback to learners, but to get a training set of statements we need people who are already proficient and agree with one another.

We will start with the most basic requirement: collecting classification ratings by user. One possible procedure would be to have a group of users do classifications until there is a high inter-rater agreement on a large percentage of the statements, say, 95% agreement on ratings of statements. Should probably also have some fluency requirement, or at least require some steady state for latencies, inter-response times, overall rates.

Final product

There are literally infinite directions we could take in future versions. A combinatorial explosion of features.

But if we were to carry the project through, one possible finished system:

1. Can be used by large numbers of users to classify statements within different domains.
2. Some users (admins?) would be able to define new domains, feed streams of statements into the system, along with descriptions of what issue(s) the statements are addressing.
3. Users would get feedback on their "correctness" and fluency scores (when?). System would be used to train and maintain rating behaviors. Work product would be domain specific sets of statements classified as to how behavioral they are, and in what ways the "non-behavioral" statements differ from what is desired.
4. Graphs, tables, lists, of data.

Development Strategy

How to break it down into digestible chunks?

Development resources, people

The main problem in developing such an application is, as always, resources:

1. resources to more completely define the problem,
2. resources to design the system,
3. resources to implement it,
4. resources to evaluate the implementation, and
5. resources to deploy each application version.

In this demo resources are especially constrained.

Phases

Thus, I will break the development down into excruciatingly small chunks.

1. This phase: develop a very basic "model" (see below).
2. Second: add a basic desktop GUI ("view controllers") for interaction.
3. Third: add basic graphs and table displays for simple data analysis.
4. And so on.

Potential future phases

If the project garners interest among the Behavior Analysts Who Code community, then we may continue it in later sessions. Ideally, members of the community would volunteer to help with coding, test data, data collection, and analysis.

This may only ever be a toy application. It may not be useful to anyone. But, it also might be a good framework for learning about building an application as a team scattered across the galaxy. Or at least the planet.

Major Software Components

What code will I develop?

Software components may go by many names. We will focus on objects, as in **Object-Oriented Programming (OOP)**.

Software development is generally guided by **software design patterns**. But they are a bit like patterns in sewing: everyone adapts them to their needs.

The following pattern will guide the overall structure of the application. While most **end users** picture an application as a **graphical user interface (GUI)**, the reality is generally quite different.

Which raises the issue of breaking software up into manageable units with well defined responsibilities. This process goes by many names, including:

1. **Decomposition**
2. Factoring: This is the more commonly used synonym of decomposition.
3. **Refactoring**. For example: **rule of three**.

Design pattern: MVC

The most common, and most basic, application software design pattern over the past few decades is known as **model-view-controller (MVC)**. It goes hand-in-hand with **object oriented programming**, and seems to originate with the development of the **Smalltalk language** in the late 1960s.

The "M" is the model, in the sense of modeling or simulating some active process, mechanism, or organism.

The "V" is the view, or set of views. *It is part of the user interface.* Views allow you to see aspects of the model. If you can see something in a software application, you are looking at views. It is how the model talks to you. Things like text and images.

The "C" is the controller, or set of controllers. *It is part of the user interface.* Controllers allow you to talk to the model, tell it what to do. Things like buttons, sliders, text boxes.

Model: The behaving organism

The model is the "M" in "MVC".

It is the living breathing heart of the application.

The model is the core engine of an application. This is where the main logic and data are encapsulated, and where the core behavior of the application is defined. The model defines the full set of possible interactions with, or actions on, the data. It is generally embodied as a code module (a.k.a., a library, a package).

The primary user user for a model is generally a programmer, and the interface is an **application programming interface (API)**. The end user interface is built upon the model, perhaps through other intervening layers of software.

Functional model: a process simulation

The model owns, encapsulates, protects, provides access to, all data. The model provides all behaviors that affect the internal state of the model, and which provide services to the “outside world”.

The behavior requests are generally referred to as messages, but more commonly as **methods**, which should be viewed as short for “method by which some action is accomplished”. We might also consider a method a software version of a vital behavior that impacts some particular outcome.

The model defines:

1. Under what conditions (previous behaviors, request parameters, etc.) a particular behavior can be requested.

2. The effect on the application state/environment if the request succeeds (or does not).

3. The values returned in response to the request, if any.

Structural data model: the proverbial “Dead Man”

Besides behavior, the model includes data, which is the internal state of the organism, its “guts”. These guts are part of the functional model, but are frequently externalized as a storage format, such as a **relational database entity-relationship model**.

The data model only defines a bare outline of the data items in the model, and to some extent, how the data items are related. It does not, and cannot, define behavior. Just be aware that the ***data*** model is not ***the*** model; it does not include behavior.

That is, the data model is structural; the overall model is functional.

View-Controller: The User Interface

This is the “VC” in “MVC”.

In our case, the “view-controller” components of the MVC pattern will be combined into a simple **graphical user interface (GUI)**. This intermixing is pretty standard, unless you are the person who is developing the individual “views” and “controllers” (and sometimes even then).

“The” user interface is difficult to specify. Models typically are presented to different audiences via different end user interfaces. For example:

1. A **command line (CLI)**.

2. An **application programming interface (API)**.

3. Multiple **graphical user interfaces (GUIs)** on a variety of platforms (e.g., iOS, macOS, tvOS, watchOS, MS Windows, Xamarin, browser interfaces). All of these interfaces could be implemented against the same model.

Initial Development Process

What are the steps in using the tools to develop the major components and verifying that they work and will continue working?

Development processes are very idiosyncratic to individuals, teams, development organizations, and so on up the ladder. These days “Agile” is a reasonable starting point: **Manifesto for Agile Software Development**. Whether or not it has ever been carried out as intended is controversial. But the ideas are interesting.

My idiosyncratic process for this project starts with the preceding steps, which are not necessarily sequential, and probably should not be:

1. General requirements
2. Overall software pattern
3. Development strategy

Then design and implementation. In large projects involving multiple teams, one would likely need a more heavy-weight process involving many many lengthy meetings before any real design or coding begins.

In smaller projects (my preference), meetings and documentation should be minimal, and design and coding should be of a rapid-turnaround nature, with involvement of the "customer", frequent feedback, and so on.

My general preference is to simultaneously, iteratively:

1. Work out the internals of the core engine, that is, the model.
2. Work out the rough outline of the primary user interface of the model.
3. Work out the rough outline of the end user interface (if different from the model's interface)
4. Continuously integrate, evaluate, refactor

I will ***not*** be designing a graphical user interface (GUI), at least not now.

I will develop the basic application model, and for me, that means:

1. sketching out the major classes with extensive commenting.
2. Once I have some idea what the classes should be, start implementing.
3. The implementation includes at minimum at least two files for each class: the class definition/implementation, and the goal-setting unit tests.
4. Refactor (**behavior preserving transformations**) as the need is discovered.

Tools

What software will I use to develop?

There is a very large number of choices among development tools. I will focus on the ones that I use, and which serve well in a "pro" environment. But the choice of tools is often dictated by an organization. When there are no mandates, then the choices tend to be very personal.

Current Phase

This is a very cursory overview of the tools I will use in this current simple prototype. They are also the tools I use the most in any Python project.

1. ***PyCharm:*** This is a full featured **integrated development environment (IDE)**. It includes editors, project management (technical, not business admin), debuggers, and so on. EVERYONE on a team must use the same development environment (as practical) to avoid needless incompatibilities and and conflicts.
    1. As for many such "pro" level tools, it is highly customizable. It makes complex tasks easy (and some would say, vice versa).
    2. PyCharm is on a par with **Apple's Xcode** and **Microsoft's Visual Studio**.
    3. The free "community version" is very capable. If you do use it on a continual basis, you may find that there are features that require the paid version. The individual license is very

reasonable: $89/year.

2. ***GitHub:*** GitHub is an online means of sharing *versioned* project materials. It is one of the many cloud versions of the popular **git source control system**. It has features to support parallel work on the code by different team members and even separate teams in a controlled, yet flexible manner. Version control systems are how developer can work on the same code without destroying each other's work.

    1. This is where the materials for this presentation are stored: **Ally-Assist/For-BehaviorAnalystsWhoCode**. If you are reading this online, you are probably already on the GitHub site.
    2. Note that this source code repository is a public, open source, space under the Performance Ally, LLC, account. All materials in this repository are licensed under the **Apache License version 2**.
    3. You ***CAN*** use command line tools, but I do not. If you have to ask what a command line tool is, then command line tools are probably not well suited to your purposes. Instead, try **GitHub Desktop**.

3. ***PyTest:*** PyTest is a testing framework that is integrated into the PyCharm environment. There are other testing tools that could be used with PyCharm, but PyTest is the one that I use. I call it out because the demo, **Demo: Correct and Bulletproof**, is based on PyTest. Note that most "pro" level development environments will have **some sort of testing framework**. Testing, especially within developing (vs after), is your friend. It tells you when you have accomplished your goals, and when it is safe to share your code.

4. ***Others:*** As the project progresses, other Python packages will be pulled in for specific purposes. A rule in Python, and many other languages, is that before designing and writing new code, check to see what is available first. There is a large quantity of high quality free code available through the **Python Package Index (PyPI)**. PyCharm has **facilities that make adding such packages very very easy** (this is also a typical IDE feature).

Future Directions

1. ***Doxygen:*** Probably the most popular documentation generator, ever. As long as you follow certain minimal language specific commenting practices, you can generate "online" for your code. The tool is a bit clunky, but so useful that it remains popular. Example: **Documenting Python Programs With Doxygen**

2. ***SQLAlchemy:*** This is a fairly high level Python **Object Relational Mapper (ORM)** relational database API that provides access to a variety of relational database products. In addition to providing access to remote database servers, it also works with SQLite for local storage.

3. ***SQLite:*** Local relational database storage. You use SQLite every time you interact with your mobile devices, desktop computers, almost any web browser, and maybe even your refrigerator. It is used almost universally for local storage of complex (and even not so complex) application data. We would use SQLAlchemy to work with SQLite, but Python has a much lower-level interface built in, also: **sqlite3**.

4. ***Matplotlib:*** Graphing. **Examples**

5. ***Pandas:*** Data analysis tools. If we use Pandas, it will most likely be for its **DataFrame** which can read from the SQLite database, and which can be fed to Matplotlib graphing objects.

6. ***Desktop Python GUI with Tkinter:*** Bare bones GUI library generally included in Python.

7. ***Web/Browser Python GUIs:*** The two primary choices for developing a web app in Python.

Demo: Correct and Bulletproof

How will I use the tools to create "provably correct" and bulletproof software?

To set the stage: the code for this initial phase of development will have already been developed up to a point. This demo will complete the functionality for this phase (I hope) in a manner that illustrates the **Critical Development Techniques**.

The Code Project

Resources specific to this project:

- Public GitHub repository: **Behavioral Statement Classifier**
- The open source Apache License Version 2.0 is here: **LICENSE**
- The PyCharm Python source code is *(will be)* here: **source**

The Model

TO BE DONE

Main Demo

Most of the code will already exist as a Python in the **project repository**.

I will leave at least one critical model component, and one view-controller component, unfinished. I will have a set of behavioral requirements for that component that the component must satisfy. As part of demonstrating Test Driven Development, I will incrementally complete the components using the Pytest **unit testing framework**.

1. At each increment,
    1. I will set a goal specifying a sub-requirement for the component in the form of a **Python assertion in a test under Pytest**.
    2. The assertion will fail until I produce the code that satisfies that goal.
    3. I will set another goal (i.e., assertion), then engage in software development behaviors that will be reinforced by passing the criterial specified as an assertion.
    4. Rinse repeat until all behavioral application requirements for the component are satisfied.
2. The full test suite for the project will be run to find any other problems I introduced during implementation of the component. Any errors must be fixed before we can say that the component is complete.
3. Once the project is error free, the test coverage analyzer will be run.

Optional

Assuming Ryan Cole is present, Ryan and I may demonstrate "**paired programming**", which might better be thought of as cooperative problem solving.

If Ryan is not available, might this be time for "audience participation"?

Where to From Here?

TO BE DONE

Is the end goal still the same? What is the next enhancement toward the goal?