|  |
| --- |
| Force Strategy Division |
| Marathon 3.1415926535897932384Design Documentation |
| Working Draft |
|  |
| **Tom Spoon** |
|  |

|  |
| --- |
|  |

# Introduction

The purpose of this paper is to provide a basis for understanding Marathon, and to shore up the argument for (somewhat drastic) changes to its design. The paper starts with a conceptual frame of reference for understanding Marathon. Using this frame of reference, the paper identifies inherent sources of complexity within the problem domain, and some proposed design philosophies that attempt to mitigate complexity. A discussion of Marathon in-theory follows, in which the major abstractions and “muscle movements” inherent to any Marathon implementation are enumerated. Finally, the reader will delve into implementations of Marathon, beginning with the current object-oriented implementation, and exploring how Marathon will look under functional and distributed computing paradigms.

**What is Marathon?**

Marathon is a mechanism for analyzing the effects of Army supply, demand, and policy variations, where supply is a set of potentially deployable units, demand is a set of activities requiring a unit, and policy is a collection of rules or constraints that determine a unit’s ability to fill a demand. As a design goal, Marathon seeks to validly simulate the physics of Army supply and demand, governed by policy, to analyze both the general behavior of such systems and the specific effects relative to changes in supply, demand, or policy. Ultimately, Marathon is an analytic sandbox for evaluating courses of action relative to the Army Force Generation domain.

**Army Force Generation**

Army Force Generation is a system for managing readiness, the ability for units to deploy to meet contingencies. In general, force generation is the structured progression of increased unit readiness over time, resulting in the periodic availability of trained, ready and cohesive units prepared for operational deployment in support of civil authorities and combatant commander requirements. The domain of Army Force Generation is enormous, encompassing the range of processes and resources necessary to man, equip, train, deploy, and sustain the Army’s supply of units.

Out of necessity, Marathon focuses on a subset of the Force Generation process, and generally holds many gross assumptions about the behavior of quite complex subsystems (such as training processes, manning, equipment, mobilization, etc.) Even with the Force Generation domain scoped to the unit level of detail[[1]](#footnote-1), and with complex subsystems like equipping and manning abstracted away, the variety of supply, demand, and policy options is still staggering.

**ARFORGEN**

In addition to the breadth of the force generation domain, the policies for managing the force generation process have historically ranged from various flavors of Tiered Readiness to a contemporary notion of Cyclical Readiness.[[2]](#footnote-2) Marathon was initially designed to analyze ARFORGEN, the Army’s contemporary cyclical force generation process. ARFORGEN seeks to synchronize individual sourcing, manning, equipping, and training processes to ensure a continuous supply of forces, and transitions the Army from a system focused on surging forces to war to a system for sustained operations. The goal of ARFORGEN is to provide a consistent and predictable supply of units ready to deploy to meet contingencies.

Under ARFORGEN, units constantly[[3]](#footnote-3) increase readiness as a function of time in a cycle. At the beginning of a unit’s ARFORGEN lifecycle, the Reset phase, the unit lacks equipment, personnel, and training, leaving it at the lowest relative state of readiness. Units accumulate resources and training as they accumulate time in the lifecycle, which propels them through the abstract ARFORGEN phases: Reset, Train, Ready, and Available. In the available phase, units are at the highest level of readiness, and are best suited to deploy. After the Available phase, units begin a new lifecycle, losing equipment, personnel, and readiness as they transition to the Reset phase. In theory, distributing units uniformly across ARFORGEN lifecycles – an ideal ARFORGEN state – simultaneously enforces equal opportunity for training, deploying, and resetting across the unit supply, and ensures a static (consistent) supply of units in each ARFORGEN phase.

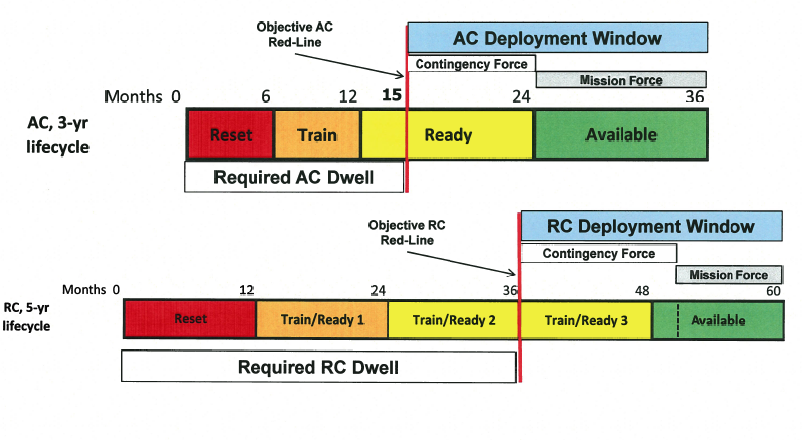
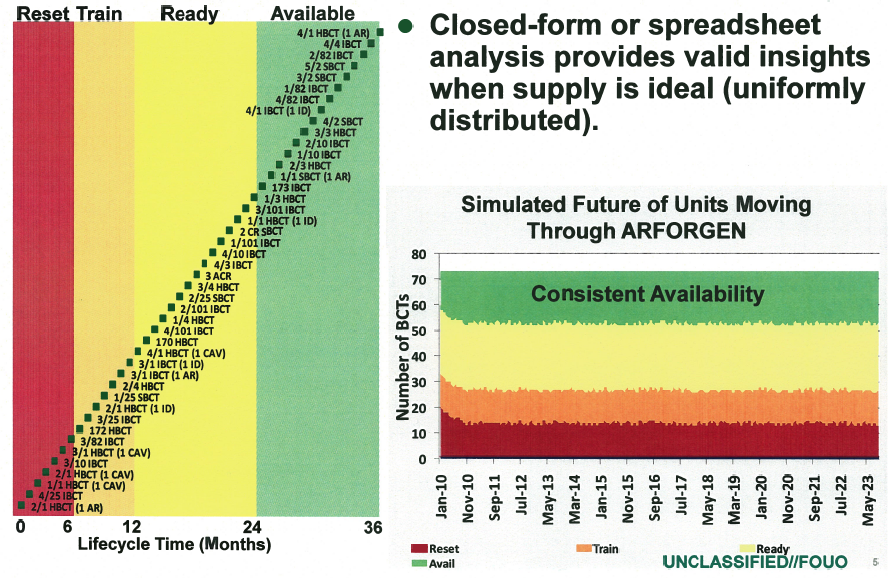


Figure : Notional ARFORGEN lifecycles for the Active Duty and USAR/ARNG

One should note that ARFORGEN is only one system of managing readiness, and that alternatives, such as Tiered Readiness, do exist. The relative value of a force generation system is affected by resource constraints (namely budget and end-strength), the magnitude of demand, and the duration of demand. ARFORGEN was enacted in response to the protracted wars in Iraq and Afghanistan, and under the auspices of large budgets and congressional mandates to grow structure. ARFORGEN may not be feasible or desirable under different circumstances.

**Static Rotational Analysis and Ideal ARFORGEN**

Rotational analysis is the notion (circa 2003-2012+) of units deploying for a fixed interval to fill demands, with a supply system producing new units to backfill demands under a cyclical readiness policy. This typically equates to an ARFORGEN force generation process.[[4]](#footnote-4) Fortunately, ARFORGEN (in the ideal state) is amenable to analysis due to its static nature. In the ideal state, units are evenly distributed across ARFORGEN lifecycles: 

Units are not deployed “out of cycle”, which keeps the quantity of units in each ARFORGEN pool constant (or static). The **theoretical capacity of the supply** – the expected number of units available at any time, relative to a total rotational supply and a unit lifecycle - can be calculated by multiplying a rotational discount by the total number of units in the supply. The **rotational discount** is a dimensionless quantity, ranging between 0 and 1, which represents the expected proportion of a unit’s available time relative to lifecycle time. Summing the factors that increase available time (e.g. time in the available phase) or decrease available days (e.g. time required to mobilize) provides a numerator, while the denominator is the duration of a lifecycle.



Theoretical capacity is built upon ideal assumptions like perfect availability, and does not reflect the complexities of unit histories, transformation periods, changes to rotational policy, and lifecycle sequencing. **Static rotational analysis (or static analysis)** examines force generation through the calculation of theoretical capacities and variable rotational discounts, where idealized assumptions guarantee the system is in a static, uniformly distributed state. Static analysis is a highly useful tool for determining feasibility and providing upper bounds on the capacities of various ARFORGEN schemes.

**Dynamic Rotational Analysis and Real ARFORGEN**

Ideal ARFORGEN has yet to be achieved[[5]](#footnote-5), let alone maintained, so static rotational analysis cannot validly answer questions about real ARFORGEN, where the state of supply, demand, and policy is dynamic or changing. **Dynamic rotational analysis (or dynamic analysis)** accounts for changes in the force generation system and non-ideal states, to bridge the gap between theoretical ideal and the empirical reality. Dynamic analysis enables observation of supply, demand, or policy scenarios that vary as a function of time or event, and often illuminates unforeseen consequences via second and third order effects.[[6]](#footnote-6)

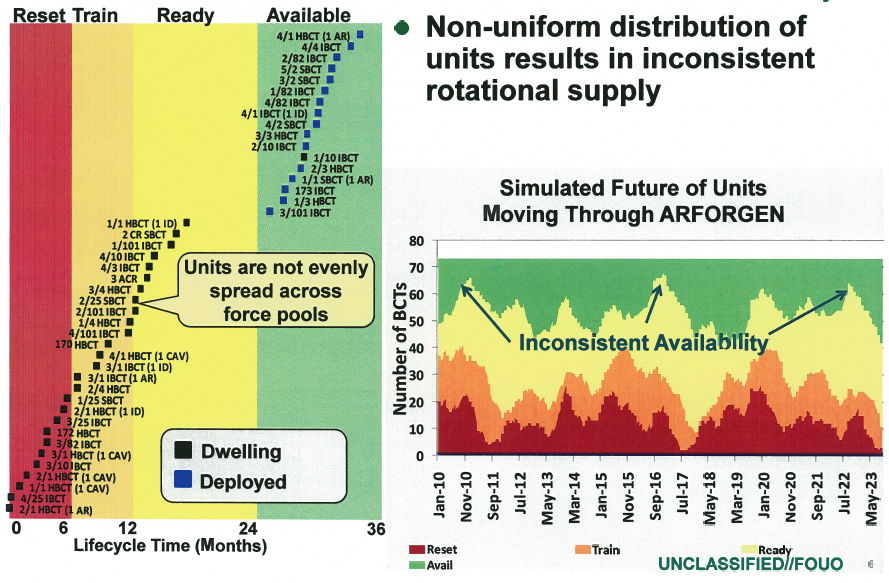


Figure : Reality is not ideal.

Dynamic analysis utilizes some form of simulation, optimization, or difference equations to generate multiple sequentially-dependent states of the rotational system. The sequence of states resulting from the application of a transition function to an initial state forms the “history” of the system. System history is the primary output, and serves as the basis for multiple forms of focused analysis. Incorporating dynamics allows for a more nuanced and contextual analysis (particularly when analyzing temporal phenomena[[7]](#footnote-7)). 

Figure : A possible workflow for dynamic analysis.

CAA created Marathon to analyze dynamic force management under ARFORGEN. Marathon performs dynamic analysis through a combination of discrete-event simulation and on-line optimization[[8]](#footnote-8). Specifically, Marathon simulates the deployment of rotational operating force units to meet operational demands over time, based on specified ARFORGEN rotation policies. Marathon illuminates the differences between theoretical rotational capacities, and rotational capacities stemming from non-static system conditions, such as historic deployments and surges, and dynamic changes in supply, policy, or demand.

**How Does Marathon Work?**

Marathon typically simulates the force generation process through a coordinated set of supply, demand, and policy simulations.[[9]](#footnote-9) The supply system acts as a coordination point for polling unit availability, a dissemination channel for simulation supply events, and a general container of units. Thousands[[10]](#footnote-10) of unique unit entities follow rotational policies that are either global (shared) or local (unique to the unit), and are directed by one or more supply systems to execute the “supply physics” dictated by the corresponding policy. Each unit’s simulated history can be traced, recorded, and reacted to within the simulation ecosystem.

Unit rotational policy generally consists of a directed sequence of states and durations.[[11]](#footnote-11) Units also have a behavior, which interprets policy to implement the desired supply-side and deployed actions. Policies are entirely modular and variable, as are individual unit behaviors. The decoupling of behavior and policy allows for both homogenous sets of units that appear to behave identically, as well as a diaspora of independent singletons that can apply similar behavior to different policies or interpret the same policies (via different behavior) to simulate radically different populations. The potential for unique entities allows Marathon to flexibly and modularly account for the legion of subtleties and corner-cases in the force generation problem domain.[[12]](#footnote-12)

Demands are activated, and slated for filling, based on a - potentially sophisticated - user-defined priority function. A fill system matches the highest priority demand to the most suitable supply as needed, and directs the transition of units from the supply system to deployments or other states. The fill system also accounts for potentially complex unit substitution rules, demand preferences, and almost any value function associated with the selection of units to fill demands.

Finally, a policy system accounts for changes to policy (such as ARFORGEN suspension, variation in lifecycle length, and changes in deployment time) by enacting system-wide policy changes in response to either time or event.[[13]](#footnote-13) Policy changes automatically filter down to subscribing units, enabling a rich and diverse simulation of the supply-policy-demand dynamics.

**Marathon Design: Tending Toward the Abstract**

**Dealing With the Fact That Marathon is Complicated**

The primary concern in this paper is the design of Marathon, an analytic process that makes force generation analysis comprehensible. Force generation is complicated, thus Marathon is complicated.[[14]](#footnote-14) To cope with Marathon’s complexity, we develop a thoughtful design philosophy. We initially refer to “the design”[[15]](#footnote-15) of Marathon to communicate an understanding of its governing dynamics, or the general precepts associated with an abstract notion of Marathon, or any dynamic rotational analysis. Having attained a solid conceptual basis for these thematic elements, the task of “engaging in the design”[[16]](#footnote-16) of Marathon comes into focus.[[17]](#footnote-17) Openly illuminating this design philosophy facilitates reasoning about existing designs, and significantly aids the process of extending or creating new designs.[[18]](#footnote-18)

**Why Marathon is Complex**

Marathon inherits much of its complexity from the force generation domain. Force generation is **really** complicated. Validly accounting for the coordinated act of equipping, manning, and training thousands of units, involving hundreds of thousands of individual soldiers, and millions of pieces of equipment is not trivial.[[19]](#footnote-19) Assuming away confounding details like synchronized training, equipping, and manning , and looking only at whole units still involves many hundreds of different unit types, spread across multiple components (e.g. Active, USAR, ARNG) with heterogeneous policies. Finally, the domain is rife with corner cases and special concerns which cannot be assumed away, as they are often the focus of analysis.

The dynamic nature of force generation also contributes to complexity. Many interesting phenomena emerge from the reactions of a dynamic system responding to time and events. System dynamics inject additional complexity in the form of indirect effects and reactions, and increase data generation and collation requirements. Simply stated, a system with dynamic interactions is inherently more complicated than a static one, particularly when the history of the system must be retained for analysis.

The volatility of force generation data further increases complexity. Unlike the natural sciences, there are very few constants in the force generation domain. Force generation data is frequently used to drive experimentation, and tends to come in multiple flavors. Supply and demand are subject to change from study to study. Even within a study, there are generally multiple excursions related to different courses of action. Sometimes Force generation analysis requires a laboratory, rather than a deterministic computational process.

Out of practicality, Marathon adds its own incidental complexity. The process of boiling down various domain-specific input data, verifying the data[[20]](#footnote-20), pushing the data through a computational representation, analyzing the large and diverse set of domain-specific results, and reducing the results into analytic insights requires processes that increase complexity.[[21]](#footnote-21)

Marathon exists to abstract much of the details away from the end-user. Marathon facilitates force generation analysis, and ultimately enables comprehension of the effects of a variety of changes to the force-generation domain. In a sense, Marathon provides a sand-box in which force generation policies may be dynamically constructed, examined, and changed. For the end-user, the “magic” behind the sand-box is irrelevant; they only care that Marathon allows them to easily[[22]](#footnote-22) commit ideas and data to an analytic canvas.

**Methods of Design: Building Up From Atoms, or Coming Down From the Clouds?**

Our task is not to build a user’s guide, which may vary with any particular implementation of Marathon, but to address the needs of the developer: to fundamentally understand any “magic” that is needed and how the “magic” works. More importantly, to design new versions of Marathon, or to expand an existing version, we will benefit from a structured way of thinking about Marathon; a design philosophy.

In the author’s opinion, there are two obvious approaches to building and maintaining a complex system like Marathon.[[23]](#footnote-23) One can conceptually start from the lowest logical level of the system, the “bottom” or atomic level, and build “upwards” by creating ever more sophisticated arrangements of atomic components. Pushing atoms around is not unlike moving cement blocks into the shape of a pyramid. This is often attractive, because the atomic components are often well-understood and easy to manipulate, and suitable arrangements may quite simply be obvious. *Furthermore, the simpler a task, the more easily it scales.*

Unfortunately, such a design philosophy lacks long-term staying power: it has little to no useful abstractions, focusing only on operations for manipulating atoms. In other words, one trades the relative micro-simplicity of manipulating atoms for an unintended growth in macro-complexity. The result is a multitude of intricately ordered and dependent atoms, produced by an equally intricate ordering of instructions to push said atoms into the final arrangement. The more an arrangement grows in complexity, the more complicated the sequence of primitive operations required, tending toward obfuscation of the structure of the operations. *The primary results are erroneous arrangements, or increasingly vast, brittle designs, which resist change and defy reasoning due to fear of error.*

One is entirely capable of creating sophisticated arrangements via simple operations, but the size and complexity of the resulting design increases with the sophistication of the desired arrangement. Sophisticated designs composed of simple operations also tend to become fairly rigid, due to the need to eliminate error.[[24]](#footnote-24) Additionally, it may not be obvious what a group of blocks does, or where a functional set of blocks starts and stops. As the design becomes more sophisticated, it also leads to inflexibility; if you need to change some blocks, it is harder to reason about the effect on other dependent blocks (or later operations), or the limitations on future growth. As with dynamic systems, the process used to arrange the atoms is further complicated due to subtle interactions lost amid large sets of operations.

For certain use cases, like early prototyping, pouring such intellectual concrete is perfectly adequate, if not necessary. Like a sculptor’s use of clay, using only primitive operations and atomic elements allows one to quickly mock-up and approximate an initial design. In some cases, this may be all that is needed (i.e. the atomic elements and operations are suitable abstractions for implementing the design). However, one must objectively evaluate the derived design[[25]](#footnote-25) and determine if the means used to achieve the design were best suited. In essence, using the bottom-up approach, which facilitates an experimental design, we need to assess our end-result. *Central questions include: “If we had to do this again, would we do it the same way? Can we change the design to accommodate different inputs? Can we change the design to produce different outputs? Can we explain the design in two sentences?”*

Historically, Marathon’s design philosophy centered on the idea of concrete, bottom-up design using readily-available [primitives].

Concrete thinking tended to pollute design. Concrete design leads to inflexibility. Like concrete, might be quick to pour, sets strong, but hard to change and is harder to adapt to new circumstances. Start up from high in the design stratosphere, incrementally defining lower layers.

* + - AKA Stratified design.
    - Implementation is delayed.
      * Avoid the concrete details until necessary.
      * Allows for a more flexible design.
    - Layered abstractions implicitly define a domain-specific language.
      * Only the important stuff for a particular strata of the design.
        + Primitives in the strata.
        + Means of combining primitives in the strata.
      * Primitives can unfold additional, lower abstraction layers.
        + Automatically compartments relative details (abstract primitives and means of combination) in each strata of the design.
        + Isolates the effects of lower-level strata on higher level strata.
        + Explicitly defines barriers between abstractions.
        + Allows one to change / extend implementations.
* From this point forward, Marathon is to be understood via stratified design.
* Marathon has adopted stratified design as its design philosophy for future design changes.

The running theme of this whitepaper will be the use of layers of abstraction, increasing in specificity until the gradual and eventual resolution to the concrete (where applicable). The underlying and complementary message is the desire to delay concrete thinking until absolutely needed. The intent for thinking this way is to alter an implicit Marathon design philosophy that has prevailed – and in many ways failed[[26]](#footnote-26) - for years, and in the process, avoid the consequences of pouring intellectual concrete without justification. Indeed, through historical example, the consequences of concrete thinking will serve as the intellectual fulcrum for leveraging change in Marathon’s design philosophy and architecture.

**An Abstract Notion of Marathon**

**Definition of *Chimera*:[[27]](#footnote-27)**

* 1. a fire-breathing she-monster in Greek mythology having a lion's head, a goat's body, and a serpent's tail.
  2. an imaginary monster compounded of incongruous parts.

1. an illusion or fabrication of the mind; *especially* an unrealizable dream <a fancy, a *chimera* in my brain, troubles me in my prayer — John Donne>.
2. an individual, organ, or part consisting of tissues of diverse genetic constitution.

To develop a strong foundation for understanding, if not designing Marathon, we must have some way to begin. Like the mythical Chimera, Marathon is many things. This is a consequence of the complexity and variability of the force generation problem domain. To further compound the issue, the process of applying Marathon during a study introduces practical concerns that have nothing to do with force generation: input data validation, experimental design, discrete-event simulation, optimization, run-time scripting, database management, analysis of algorithms, interactive visualization, exploratory data analysis, and some statistics.[[28]](#footnote-28) As a consequence, Marathon is simultaneously an idea, a specification, an interactive tool, a deterministic number cruncher, a mechanism for handling uncertainty, a visualization suite, a data validation mechanism, etc. The combination of technical requirements and rich domain specific information has elevated Marathon into an analytic process that consumes and validates force generation data, produces relevant domain metrics, quantitatively analyzes and summarizes vast amounts of output, and generates effective visualizations to communicate higher-order information with sponsors. Marathon seemingly defies simple abstraction.

On the other hand, Marathon, like Chimera, is also an abstract joining of three things: **supply**, **demand**, and **policy**. The earlier conceptual frame of reference made this very claim. Despite the apparent complexity in the preceding paragraph, almost every piece of Marathon can be compartmented and understood along the axes of supply, demand, policy, or combination of the three. This is an example of a useful abstraction, because it allows our minds to comprehend a subset of – what could be- a large problem space. Consequently, when one thinks of Marathon[[29]](#footnote-29), it helps to think in terms of abstract ideas and questions. The Chimera example conveniently, if not coincidentally, alluded to a simplified mental model indicating a relation between supply, demand, and policy. We can continue to benefit by projecting that mental model onto pieces of Marathon, and by identifying other convenient abstractions.

In general, thinking about the “what” instead of the “how” is extremely useful[[30]](#footnote-30) for gaining an initial understanding, or breaking new mental ground. “What happens when….?” “What happens if….?” “What could be accomplished with this many?” “What should we build for….?” “What capabilities do we need?” “What is time?” Questions initialized with “What…?” tend to ignore or demote the importance of implementation, or how something is accomplished. Every piece of Marathon can be traced back to these kinds of questions.[[31]](#footnote-31) These questions serve as excellent entry points for deriving abstractions, because they tend to point immediately to how the mind views a problem or a phenomenon. Once we have our abstractions in hand, we may confidently ask “How…?”, because then – and likely only then – are the details of how to accomplish something actually important. If you don’t know what a house is, does understanding that boards can be combined - perhaps by hammering a nail – get you any closer to fundamentally knowing what a house is?[[32]](#footnote-32)

Furthermore, the abstraction of higher level concepts allows users and developers to communicate needs, expectations, and capabilities in a purer form, with a much lower signal-to-noise ratio. If I were to tell a sponsor that Marathon is a discrete-event simulation, modeling atomic entities as finite state machines whose transition probabilities were functions of policy and behavior, they would probably have tuned out after “discrete”[[33]](#footnote-33) due to their perception of technical noise. More often than not, when non-developers want to understand Marathon, they want to know “what” and not “how”.[[34]](#footnote-34) Abstraction allows for analogy, and analogy (if one finds the right example) becomes an indispensable tool for communicating otherwise complex technical or policy information.[[35]](#footnote-35)

**Marathon in Theory**

Marathon is a process that combines a set of units **S**, a set of demands **D**, and a set of policies for relating units and demands **P**, into a dynamic system open to observation and analysis **M**. Units are unique entities that collectively form an abstract supply. Demands are unique entities that require a relation with some amount of supply. The abstract notion of relating a demand with one or more units from a supply is to fill a demand. The filling of demands, with units from supply, is constrained by policy. The dynamic system produced by Marathon ultimately provides a set of fills, **F**. Thus, Marathon is a function that maps supply, demand, and policy to a set of fills:

Without additional information, we could probably apply the description above to a host of problems that have nothing to do with the use case Marathon was designed for; this is exactly the purpose of starting with such a general abstraction. At this level, we are intentionally “in the clouds” due to the generality of our abstract notion of Marathon, so general as to be almost meaningless. However, we have a framework for building additional, lower layers of abstraction, which will allow us to gradually add more detail in a controlled manner. The beauty of this approach is that, by tending toward generality, we implicitly retain maximum flexibility at each layer of abstraction. Who is to say what concrete implementation we should choose at this point to fulfill the conditions implied by Marathon (M)? We could develop a simulation, an optimization, a spread-sheet, or simply a random-fill generator. All could have radically different implementation details. However, in any of the implementations, we will have a supply, a demand, policies that constrain fill, and a resultant set of fills.

**A Marathon Algorithm Based on Simulation**

This section will simultaneously promote a general understanding of Marathon and introduce the inherent complexity that must be mitigated via design considerations. We build on abstractions from previous sections to continue to develop a thorough basis for Marathon. Going forward, we assume that the scaffolding for Marathon is some form of simulation (note the generality…), in which there is an abstract notion of time **t**, where the beginning of time **t0** occurs with an initial supply, demand, and policy state (**S0**, **D0**, **P0**), which undergoes a sequence of ordered evaluations (applications of the Marathon function, or **M**) which result in a simulated history of (possibly changing) state. An abstract termination function **Continue?** maps simulation state to a Boolean value (true or false) to indicate when the sequence should stop. Since we must account for fills, we need another function **Fills** to transform our history (the ordered sequence of simulated state) into a set of fills.

**[Insert Functional Representation here]**

We can try understanding this less-general Marathon by rephrasing the preceding description into an easily-comprehended set of questions. Note that the questions intentionally begin to introduce, perhaps pre-suppose, some domain-specific concepts (e.g. ARFORGEN, the idea of rotation having an effect on the state of supply, suitability of supply, priority of demand). By answering these questions, we implicitly outline an algorithm for computing the simulation, which we can then implement to varying degrees of efficiency.[[36]](#footnote-36) Elements of the questions that are not formally accounted for in our current abstraction indicate new possible layers of abstraction to explore and define.

The questions that outline our algorithm are:

* Is the simulation over?
  + Are there demands to be filled?
    - What is the highest priority demand?
    - Are there units that can fill demands?
      * What is the unit most suitable to fill the highest priority demand?
        + What happens when the demand is filled?
      * Are there more demands to be filled?
  + Are there units that have not been used for fill that can be rotated in ARFORGEN?
    - What happens to units in ARFORGEN when they rotate?
* What is the final state of the simulation?
  + Which units filled demands?
  + When did fills happen?

As evidenced, the general algorithm is quite simple, and it uses descriptive questions to guide the flow of computation. Another way to look at the algorithm would be from a declarative perspective…

* While the simulation is not over
  + Get Fills Resulting From
    - Rotating remaining units after
      * Determining the effects of filling a deployment
        + With the most suitable unit to fill a demand

Of the available units

To the highest priority demand

Of the unfilled demands.

We turned the series of questions into declarations about the aspects of the computation that we care about. The algorithm is still general, and quite readable. Furthermore, it becomes more obvious that the context of remaining units is dependent on the context of units used to fill demands, whose context is dependent on most suitable units, etc…ultimately ending with unfilled demands.

The nice thing about the declarative approach is that it retains the clarity of the algorithm, but also shines a light on dependencies. This implies that if we can solve dependent steps of the problem, we can re-use them later in other steps. Conversely, we can assume that the sub-problems have already been solved when architecting our implementation, which allows us to create highly modular, reusable code. This is the functional approach to computation, and it is heavily embraced in Marathon’s design.

**Sources of Complexity**

For some of the questions above, the apparent simplicity of the preceding algorithm corresponds to a simple implementation. Specifically, choosing to continue simulating, and reporting the final state of the simulation are typically “easy” questions to answer, since they tend to naturally surface during the course of answering much more difficult questions. At this point, assume that choosing to continue simulating and obtaining the final state of the simulation are trivial to implement, and are relatively immune to changes in design. The remaining questions are far more nuanced, and context-sensitive. They require more attention and forethought prior to implementing a solution.

For example**, prioritization of demands** and **determining the suitability of a unit to fill a demand**, are potentially “difficult” concepts that require much thought. It should be noted that the difficulty is entirely a function of the complexity of the questions. Sometimes analysis calls for a very simple answer to a very limited question. Indeed, implementation of a relatively simple function to fill supply with demand is trivial. If we live in an arbitrary world, as determined by a set of very relaxed policies, we could arbitrarily collect unfilled demands in a list, and use the ordering to define some notional priority (no actual priority). Taking the first element of demand, we then try to fill it with a suitable unit of supply (no actual suitability). Assuming that available supply is known or can be calculated, we could just as arbitrarily order the supply and take the first element from the resulting list, completing the fill. This notion boils down to “find an unfilled demand **d**, find an available supply **s**, fill **d** with **s**.”

**The Inexorable Growth of Complexity**

Unfortunately, the actual rotational analysis associated with Marathon is rarely of an arbitrary nature. In the “real world”, decisions often demand many dimensions of input, and the result of a decision is similarly assessed across multiple output dimensions. The enduring desire to more closely approximate reality[[37]](#footnote-37) tends to inject multiple dimensions of complexity. There is an inherent attractiveness to complexity as well. Added complexity allows for a great degree of flexibility, expression, and clarity.[[38]](#footnote-38) For the analyst, a complex system allows more avenues for expression and more “tools” to apply to a problem. For the sponsor, a complex system enables the analysis of more interesting phenomena. In Marathon, the notions of “suitable” supply, “highest priority” demand, and “fill” are typically the centers of gravity for complexity.[[39]](#footnote-39)

The combination of sponsor-driven and analyst-driven complexity creep manifests in **change**. Historically, Marathon development underwent ad-hoc changes in the contemporary design – **patching** - to respond to immediate complexity creep. These patches did not tend to anticipate or mitigate future complexities, but were short-term fixes. Inevitably, performance problems or changes in sponsor requirements meant the patches were no longer feasible, maintainable, or useful, and resulted in drastic redesigns to meet or exceed new complexity standards. Redesign generally resulted in removing the existing architecture –**bulldozing** – and constructing a more capable architecture to encompass the features of the old and address new complexities – **rebuilding**. Unfortunately, Marathon design continued to inherently ignore the high probability of new changes stimulated by the next phase of inevitable complexity creep.[[40]](#footnote-40) This effectively guaranteed perpetuating the cycle of **patching**, **bulldozing**, and **rebuilding**.[[41]](#footnote-41)

**Managing Change and Complexity Potential**

Refocusing the underlying Marathon design philosophy toward layered abstractions is one way to safely and flexibly accommodate change, allowing an unlimited[[42]](#footnote-42) **addition** of features to address complexity **as needed**. Rather than reactively shoring up contemporary design shortcomings to meet immediate complexity creep, Marathon’s new design is fundamentally dynamic and extendable. Design-by-extension allows future developers to address unforeseen limitations in contemporary designs without disrupting the existing Marathon feature set, or previous analytic projects.[[43]](#footnote-43) By accepting *change as a core tenet* of the design philosophy, and developing using **stratified design**[[44]](#footnote-44) (multiple hierarchical layers of abstraction representing primitive elements and means of combination), future versions of Marathon should have the **potential** to be as **complex** or as **simple** as needed, retaining an expandable core feature set from version to version. This philosophy can be summed up succinctly “*Marathon should give the developer and analyst as many tools as possible to answer questions of varying levels of currently known complexity, as well as the ability to compose new tools to facilitate expected changes due to future complexity creep*.”

**Flexible Design Strategies**

To accommodate these lofty goals, several strategies have been adopted.[[45]](#footnote-45)

1. **Develop with abstractions using Stratified Design[[46]](#footnote-46)**. Layered abstractions are a running theme in this paper for a reason. Implementation is just a detail, and delaying the concrete details allows greater flexibility in design.[[47]](#footnote-47) In fact, if we work from the top down, developing “towards” the less abstract, we gain a great deal of implicit modularity along the way. Abstractions, particularly data abstractions, are the real payoffs. By formalizing our abstractions, we can build many, many functions of abstract data types. The result is instantly re-usable, generic, and composable code. [[48]](#footnote-48) Abstractions also enable interfacing between disparate languages and platforms in a consistent manner (XML and JSON are decent examples).
2. **Anticipate change**. Implementation should not box us in, or else we will have to **bulldoze** and **rebuild**. Since change is guaranteed to come with time and new problem domains, we should allow for core abstractions to be extended without invalidating previous versions. Computational objects and functions help us do this, and create plenty of room for safe, directed growth. We can create better implementations without endangering the underlying abstraction framework. We can also extend the underlying abstractions without endangering existing implementations.[[49]](#footnote-49) Marathon developers should question the generality of anything added to the core feature set, and should instinctively isolate ad-hoc, experimental development branches.
3. **Separate content from logic**. Content generation and analysis are orthogonal to Marathon’s logic. Marathon is concerned with prosecuting a discrete event simulation to provide some form of output. Input and simulation-generated output are both forms of content. We generate input for Marathon to process, and in turn analyze the output processed by Marathon. By separating these concepts, we can use our abstractions to generate concrete interfaces necessary to permit the flow of data. This separation allows developers additional freedom to develop independent tools and processes suitable for the specific domain, and to extend to new domains as needed.[[50]](#footnote-50) Again, Marathon developers should instinctively separate data (or content) from application logic. Data embedded within logic (with the exception of default values and universal constants) should be considered toxic, since it obfuscates a clean separation.
4. **Find durable data abstractions.** Data abstractions allow us to build a stable, vibrant ecosystem “outside” of Marathon, and to retain interoperability between different Marathon implementations. We formalize the role of data by identifying orthogonal, domain-specific data where possible, and providing contracts, or explicit elements of the data. Rather than repeat the historic ad-hoc generation of datasets, which changed in each iteration of patch-bulldoze-rebuild, we aim for explicitly structured data that changes much more conservatively. Consistent, structured data provides an invaluable interface between (possibly) disparate logical implementations, and allows newer implementations (or even cross-platform implementations) to maintain compatibility with older data. Finally, promoting data to a first-class citizen allows us to design a plethora of content-management or production tools dealing specifically with the data. First-class data also facilitates independent data-verification processes.

**Complexity** **Example: Conceptual Fill Algorithm**

The aforementioned “centers of gravity” for complexity revolved around the nuances of filling demand using supply (the fill algorithm). Particular dimensions of complexity include but are not limited to:

1. Fill Paths – Which supply can fill which demand?
   1. Function of substitution rules (A can sub for B).
      1. Are substitutes prioritized?
         1. Can priorities change?
   2. Equivalence rules (A is equivalent to B, C, and D).
      1. Can equivalences change?
2. Priority of Fill – Which demand/fill rule should be filled first?
   1. Is there an absolute notion of priority?
   2. Is priority determined or changed during the simulation?
      1. What are the components of priority?
      2. Are they always the same or do they change from study to study?
3. Suitability of Supply - Given a fill rule, which supply best fits it?
   1. Which supply can match the fill rule (a demand)?
   2. Is there a cost or value associated with using this supply that makes it preferable?
      1. What are the components of cost?
      2. Does cost change?

Notice the amount of added detail that must be addressed relative to our simple high-level abstraction. This is a small sampling of the complexities encountered from previous studies. Most of the elements listed are dynamic (or should be dynamic). Previous implementations adhered to a more concrete fill algorithm by holding many things constant. For instance, substitution rules were generally limited to primary, tertiary, and secondary ordinal values and could not be composed. Dependencies between fill-rules were not accounted for. Equivalence rules did not exist (unless modeled using substitution rules). Demand priority was relatively unknown. There were a set of hard-coded supply prioritization schemes that provided a limited suitability function. The net result was a somewhat sophisticated, but inherently static rule set that described the fill algorithm, implemented with a mixture of logic and content.

Without the capacity to change rules dynamically, previous versions of Marathon lacked the capacity to make decisions on-line. The net result was a sophisticated Plinko machine with a small, predetermined set of possible outcomes.[[51]](#footnote-51) Furthermore, adding novel new combinations to the Plinko machine meant rebuilding most of the machine; a non-trivial task. Our goal is to allow the analyst to vary the sophistication as needed, and to dynamically change the rules/decision process as needed. Essentially, allow Marathon to change its behavior on-line, and to even make intelligent (or directed) decisions in an optimal or approximately optimal manner (or to follow apparently static, prescribed rules as before).

**Managing Complexity via Abstraction: Dynamic and Extensible Fill Algorithm**

Relative to the fill example and working in-line with the stated Marathon design goals, one way to address the complexity of any fill algorithm is to formalize the notion of filling via a useful abstraction. A useful abstraction would allow us to dynamically express the complex relation between supply and demand (substitutions and equivalencies). Further, the underlying concrete representation of the abstraction may vary drastically without impacting the descriptive utility of the abstraction. An abstract graph fulfills these conditions for utility, and more.

The vaunted graph is one of the most useful recurring abstractions in Mathematics, Computer Science, and Operations Research. A graph G is a set of Vertices (V) and edges (E), where edges contain information describing the connections of the vertices (or nodes).[[52]](#footnote-52) Google Maps is a wonderful example of a graph, where cities are on the map are nodes and the lines connecting cities are edges. Graphs are inherently dynamic and extendable because they are composed of nodes and edges (relations between nodes). To change the structure of the graph, or to define a new graph, we just connect a bunch of nodes with the appropriate edges.[[53]](#footnote-53) We can add supplemental information to the graph, associate a weight or cost with each edge (distance between cities in Google Maps), or even associate non-numerical values. We can also associate a number of supplemental data with nodes, if we need to. Graphs serve as a highly useful format for storing information about related things. With some tweaks, and some well-studied algorithms, we can use them to help us formulate substitution rules, relate supply to demand, prioritize fill-rules or demands, determine co-dependencies between fill-rules and sources of supplies, and a host of other useful things.

Utilizing graphs provides one keen advantage that will support flexible adaptation and change to mitigate complexity creep: they are not hard-coded logical constructs, but dynamic content. With the appropriate logic in place, we push the definition of a number of highly variable, concrete items such as substitution rules, prioritizations, suitability, etc. into the realm of content creation. Marathon then builds almost the entire basis of its fill context from content that ultimately describes an abstract graph. This allows a massive degree of flexibility in the **definition** of the fill context, grants the ability to **dynamically change** the fill-context, and enables **on-line decision making**.[[54]](#footnote-54) It also delegates the processing of these graphs to well-known, thoroughly studied algorithms. We will revisit this topic later, as we talk about the existing Marathon implementation.

**Interlude**

The remainder of the paper is devoted to answering more subtle questions on complex subsystems, such as the behavior of Units and Demands, how to manage supply, manage demands, script events, handle events, etc. Much of the prose begins with an abstract layout, followed by actual implementation details relevant to the current implementation of Marathon. Thankfully, the reliance on well-formed abstractions and stratified design means that the vast majority of the paper should still be relevant through future iterations. Even drastically different future implementation strategies, perhaps using alternate optimal datastructures, should not affect the overall design. Keep in mind that this is a working document, and is subject to change.

# Implementation

**High Level Objects**

The current implementation of Marathon (not the prototype 3.0 array-based model) is an object-oriented framework, with some functional programming idioms mixed in. The top-level object is the **Engine**, which coordinates several child objects to delegate tasks and processing during the simulation. The engine’s primary purpose is to serve as connective tissue, or a backplane, for all of the child objects. In the cases where children need to access data from other children, they can always go through the parent. This is very similar to most parent/child object hierarchies (the Workbook/Worksheet/Range/Cells object model of Excel for instance).

The object model also serves to delegate tasks and responsibilities to various objects. This is not true functional form, but the object oriented VBA implementation is not intended to be a purely functional implementation (F# and Clojure implementations are different matters). Children of the Engine are all **Managers**. Their purpose is to act as an intermediary between atomic components (such as data and primitive objects), to abstract away implementation details from the calling code. In the words of Abelson and Sussman, they are abstractions that allow for stratified design. At the engine level, we do not care about “how” supply is managed. We only care that, at a given time, the SupplyManager manages Supply. This allows us to push down concrete implementations to the lower level (many of which are not even handled in the managers, but lower in the hierarchy).

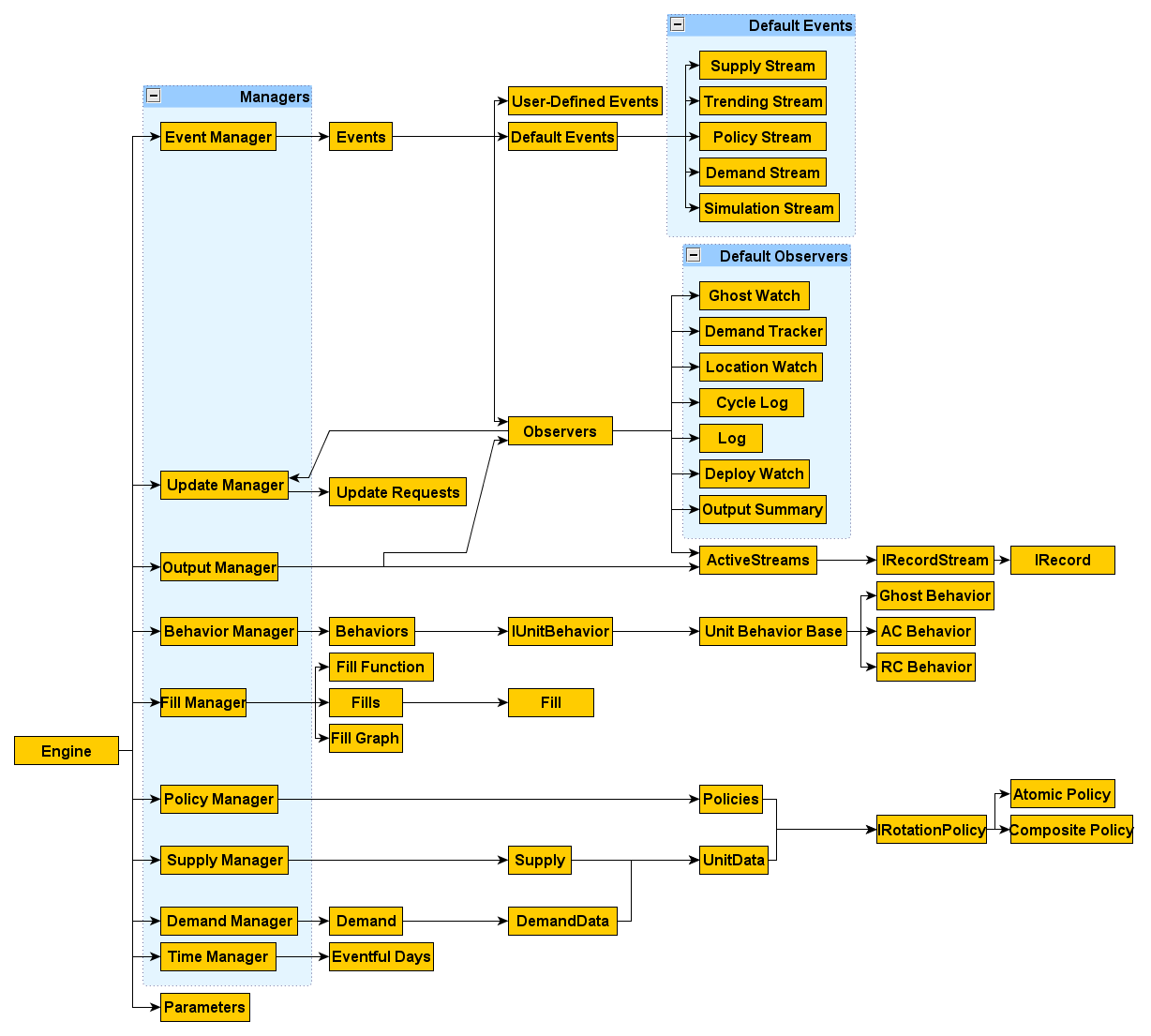


Figure : A very high-level look at Marathon. This is not a formal UML diagram, but a “general” relation of highly-visible components.

A formal listing of managers and their responsibilities follows:

* TimeManager
  + Maintains current time.
  + Allows the adding of eventful times priority queue.
    - May be set to filter out non-unique times.
  + Provides time elapsed.
  + Provides next time.

The TimeManager is fundamental to almost every Discrete Event Simulation (DEVS) framework. Since time drives much of Marathon’s simulated phenomena (deployment length, dwell time, scheduling demand activities, changing policies), the passage of time is elevated to first-class status. Time is the fundamental control and scheduling mechanism for all simulated events. The TimeManager maintains a list of “eventful days”[[55]](#footnote-55) which serve as the primary flow-control mechanism for the simulation. If there are no more eventful days (i.e. time has run out), the simulation will stop.[[56]](#footnote-56) The current VBA Marathon implementation maintains a rather coarse view of events, in that there are relatively few events “enqueued” on an event queue. Rather, Marathon’s subsystems schedule internal updates (or their own eventful days), and request to be updated at specific times. Some DEVS kits take a more fine-grained approach, where every microscopic change is modeled as an explicit event on a time-based priorityqueue. Marathon accomplishes the same functionality by coordinating the TimeManager, EventManager, and UpdateManager to ensure that only eventful days (or times) are simulated, and that only entities with changes are updated on eventful days. The result is an efficient blend between coarse-grained discrete events, and a pipelined updating of simulation state by specialized subsystems. For VBA, this provides the best balance between performance, simplicity, and flexibility.

* SupplyManager
  + Initializes, stores, and **directs** the updating of state data for pieces of supply (UnitData objects).
  + When called, it passes the current time to units in scope, and instructs them to update themselves.
  + Logs significant supply events, such as moving, deploying, returning home.
  + Maintains sets of available and unavailable units.
  + Provides an implementation for finding the most suitable unit for a given demand.
  + Provides an implementation for filling a demand.

The SupplyManager handles macro-level details for registering, tracking, updating, and deploying unit entities. It also maintains a significant amount of meta-data for each entity, as well as a host of cached information to facilitate searches for suitable supply.

* DemandManager
  + Initializes, stores, and **executes** updates to state data for pieces of demand (DemandData objects).
  + Registers activations and deactivations.
  + Activates and deactivates demands.
  + Maintains prioritized collection of unfilled demands.
  + Logs significant demand events, such as filling, unfilling, activation, deactivation.
  + Provides an implementation for filling a demand with a piece of supply.
  + Provides an implementation for sending a piece of supply “home” from a demand.

The DemandManager schedules, activates, and updates (requests fills) demand. Like the supply manager, it is a general container for demand entities, and it maintains useful statistics for prioritization of demand fills.

* PolicyManager
  + Initializes, stores, and **directs** the updating of state data for pieces of policy (Policy objects).
  + Provides a stock set of policies (default ARFORGEN policies).
    - Provides a set of atomic policies (policies which do not change).
    - Reads a set of composite, user-defined policies, which are derived from the CompositePolicyRecords table.
      * Composite policies map a variable policy to one or more period-specific atomic policies.
      * Reacts to period-change events, causing units following composite policy to change their policy (possibly their state).
  + Provides implementation for scheduling policy changes.
  + Provides implementation for executing policy changes.
  + Interprets time-periods from PeriodRecords table to schedule known period-changes events.
    - Affects period-dependent composite policies.

The PolicyManager interprets, schedules, and updates various unit policies. Its special function is to facilitate the changing of composite policies in response to events, or changes in the simulation period. It provides a mechanism for adding new policies programmatically, or via raw data. Default ARFORGEN policies and other useful policies are included, but more policies can be trivially extended. The policymanager also compiles information about substitution and equivalence rules from source data, which informs the FillManager’s rule set.

* FillManager
  + Builds one or more Fill Graphs from supply, relations, and demand tables.
    - Utilizes graph operations to identify unreachable supply, unfillable demand.
      * Notifies user (and halts by default) if there are possible data errors due to scoping results.
    - Prunes irrelevant paths from the raw Fill Graph.
  + Maintains one or more fill functions that define ways to prioritize sources of supply relative to demand.
  + When active demands require units, applies its fill functions to maximize the number of fills and the suitability of fills.
  + Coordinates deployment events.
  + Scopes demand and supply via graph algorithms

The FillManager has several complicated, but vital tasks. It compiles a FillGraph from the user-provided supply, demand, and relations (substitutions/equivalences). It then analyzes the FillGraph to determine which classes of units are independent, to facilitate dividing the simulation into sub-simulations, and to determine possible errors in the data. If “islands” are found (e.g. unusable supply, or unfillable demand), the FillManager warns the user, reports the data, and prunes the islands from consideration to improve efficiency. Finally, the fillmanager coordinates services fill requests from a DemandManager, in conjunction with a SupplyManager, using a stock or a user-defined fill-function, to find the most suitable supply for the highest-priority demand.

* EventManager
  + Currently serves as a unified interface for logging events, using a common vocabulary derived from enumerated types.
  + All calls for logging eventually pass through EventManager.
  + Output is generally a comma-separated value record of information regarding the specific event, includes:
    - time, event type, from entity, to entity, extra information
  + More events can be added at run-time, as needed, via AddEvent.
  + Implements the Observer design pattern, which decouples event propagation (subscription, notification, un-subscription), from event handling.

The EventManager serves as a communications hub for the simulation. It comes with a pre-set event vocabulary that defines domain-specific activities that happen throughout the simulation (i.e. deployments, unit movement, entitiy updated, etc.). The EventManager implements and wraps the Observer design pattern to decouple event propagation from event handling. Any object that needs to listen to the event stream can trivially subscribe to the EventManager via the AddListener/RemoveListener, and can dispatch events to other subscribers. This provides a highly flexible way for communicating important facts to other entities, without entities knowing about eachother (no coupling, no direct method calls). Dispatched events are actually time-stamped (relative to the TimeManager) packets with simple messages (for logging), the event type, a from entity, a to entity, and optional data (can be anything). This allows the event system to provide highly sophisticated functionality for the simulation, without “breaking” or modifying existing parts. The event vocabulary can be extended trivially via the AddEvent method, so that users can define run-time events and attach listeners (particularly useful for statistics and logging, or experimenting/debugging).

* BehaviorManager
  + Initializes, stores, and **directs** the updating of state data for behaviors (UnitBehavior objects).
  + Primarily, the BehaviorManager serves as a repository for behavior references so that behaviors (detailed below) can be shared amongst many units.
  + It will be able to schedule global behavior changes, if necessary.

Unit behavior is covered in more detail under the concrete implementation. The behavior manager centralizes access to behavior, and helps load defaults.

* UpdateManager
  + Provides abstraction for synchronizes entity updates.
  + Maintains last update, next update, and handles update requests.

The update manager works in conjunction with the TimeManager and the other managers to tag entities as having requested updates. This keeps the event traffic off of the TimeManager’s queue, and allows efficient bulk-update requests. As managers process their updates, they only have to query the Update manager for entities requesting an update on the current day. The update manager also maintains the last known update of the entity, so that duplicate-updates (particularly with time-dependent functions) are idempotent.

* OutputManager
  + Initializes, stores, and **directs** the updating of state data for output
    - Trends objects
    - Tables of GenericRecords (dictionaries of GenericRecord objects)
      * Deployment History
      * Sand Trends / Pool Charts
      * Summary Statistics
      * State transition history.
      * Event logging (for verbose debugging).
    - Visualization
      * PoolCharts
      * DotPlots
    - Other
  + Provides a centralized interface for recording trend data, or accumulating averages, etc.
  + Serves as the interface between Marathon and its output media.
    - Handles streaming of output to Excel (efficient).
    - Handles streaming of output to CSV or XML files.
    - Handles any other serialization tasks.

Parameters…..

Major Muscle Movements

**Fill = Shortest Path**

If we represent our fill context as an abstract graph, then we can utilize a number of well-know algorithms developed specifically to efficiently search graphs.[[57]](#footnote-57)



This is currently called the **fillgraph**. A **fillgraph** is a Directed Acyclic Graph [[58]](#footnote-58)that has a few other properties:

1. There is always a “**Filled**” node, which has no outbound edges. This is used for generating paths.
2. There exists a set of **Source** nodes, all of which contain a single outbound zero-cost edge leading to the “Filled” node. These represent sources of supply. They are labeled with the name of a source.
3. There exists a set of **Sink** nodes, all of which contain no inbound edges (they are terminal nodes). These represent unique **fillrules**, or sinks of demand. They are labeled with the name of a sink.
4. There are any number of intermediate nodes. These nodes fall into one of two categories:
   1. **Equivalencies** are associated with the = operator. They contain any number of zero-cost outbound edges to nodes that are equivalent.
      1. Ex. “= (SRC1, Source 1)” would indicate that the node “SRC1” has a zero-cost outbound arc to the Source node “Source1”.
   2. **Substitutions** are associated with the |> operator. Substitutions contain positive-cost edges that relate a Donor node to a Recipient node that can receive substitutions via an intermediate donation node labeled “|> ([Recepient], [Donor], [cost])”. The recipient contains a single outbound arc of cost [cost] to the donation node. The donation node contains outbound edges to all of the Donor’s connected outbound nodes, with costs identical to donor’s outbound edge costs.
      1. Ex. “|> (SRC2, SRC1, 2) ” indicates that the node “SRC2” has a outbound edge of cost 2 to an donation node “|> (SRC2, SRC1, 2)”. The donation node has outbound edges with destination nodes and costs identical to node “SRC1”’s outbound edges.
5. Sinks have outbound edges to one or more intermediate nodes (usually zero cost, but can vary as needed).



Given a **fillgraph** meeting the above criteria, computing the result of the fill algorithm on a particular demand is equivalent to finding the shortest path from the demand’s fill rule (a sink), to the singular **Filled** node. Thanks to efficient algorithms, we can perform this quickly.[[59]](#footnote-59) The resulting path is also a full accounting of the logic used to source said demand. It can be as complicated, or as simple, as circumstances require. By design, if a path is found, the last node will be the **Filled** node, and the actual **Source** will precede the last node.



**Paths for Fill Rule 1. Shorter paths have thinner Line widths. The substitution path costs more.**



**Paths for Fill Rule 2. Shorter paths have thinner Line widths. There are more paths due to the equivalence relationship, which composes with the substitution relationship for SRC1.**



**Paths for Fill Rule 3. There are is only one path.**

The **shortest path fill** supports many additional features that enable analysts to derive significantly more mileage from the graph. We can trigger events to dynamically update the graph, temporarily disabling valid paths, or creating entirely new paths. Similarly, we can encode the result of highly sophisticated cost functions (demand prioritization, supply prioritization, etc.) into the edge weights of the graph itself. Keep in mind, with the ability to draw equivalencies and substitutions, much of the content generation burden is lifted from the analyst. Analysts only have to define partial Fillrulles, and can compose new Fillrules from other Fillrules. The entire fill context can be programmatically or manually changed. [[60]](#footnote-60) **Any information that contributes to the decision process for fills can be codified and represented in the graph structure, without ever needing to change the logic behind the algorithm.** [[61]](#footnote-61)



**Representing a disruption in a Source. The edge weight from Source2 to Filled is made infinite, making the edge impossible to cross. There is now no way to travel from FillRule3 to Filled.**

Another significant feature is the ability to examine the topology of the graph to determine “equivalence classes”. For purposes of runtime performance, and potentially distributed simulations, we can utilize the same fillgraph, derived entirely from analyst content, to automatically determine which sets of Sources and Sinks are connected. To demonstrate the utility of this technique, a modified version of our sample fillgraph follows:



**A new graph with added nodes and edges. Fillrule4 is a newsink, with a path to Source 3 through SRC3. Fillrule5 is a new sink, with no paths. Source4 is a new source with no incoming edges.**The decomposition algorithm is relatively straightforward:

1. Calculate **H{V,E}**, a reduced graph that only contains Sources, Sinks, and edges from Sinks to Sources.
   1. This is done by using depth first search on each Sink to determine which sources it touches. These pairs become edges in E.
2. Add complementary edges to transform **H** into an Undirected Graph.
3. Pick an arbitrary **Start** node/vertex from V in **H**{V,E}.
4. Perform a union-find on H, starting from the **Start**.
   1. Union-find is a simple algorithm that explores the edges of the nodes of every node connected to Start, collecting all unique nodes.
5. Result of the union-find(Start) is a dependent set of Sinks and Sources. Group these together, store, and remove all nodes /vertices in the group from V in **H**{V,E}
6. Pick the next node from remaining nodes in V, union find again, storing the results, until no nodes remain in V.
7. Each of the stored union-find results is an equivalence class, or an independent set.
   1. Any of these results that contain only 1 entry indicate either a source or a sink that cannot be utilized (**islands**).
   2. We can tell the analysts this as an error check, and prune them from the analysis.
      1. Don’t waste resources on **islands**  in the simulation.



**Calculate** **H, a simplified version of the original fillgraph that eliminates intermediate nodes.**



**Convert H into an undirected graph (no arrows).**



**Calculate Equivalence Classes from H. Prune the islands from the data, then divide and conquer the simulation runs.**

The algorithm allows Marathon to decompose an arbitrarily large simulation into a finite number of smaller simulations that are provably independent, and to find gaps[[62]](#footnote-62) in the content in the process. This is a form of a Divide and Conquer algorithm, in which a large problem is decomposed into several substantially smaller independent problems to make efficient use of computational resources and memory to drastically reduce overall runtime. Combined with distributed computing, in which each independent simulation is run on a separate computing platform, Marathon would be entering into a significantly more industrialized computing realm. This opens the potential for a slew of possibilities, including much larger experimental design, unsupervised learning, the application of population-based metahueristics (i.e. genetic algorithms)[[63]](#footnote-63), and the potential for GPGPU.[[64]](#footnote-64)

**Low Level Objects**

The object descriptions from the previous section provided an abstract overview of the functionality required by each “manager”, and effectively delegated the tasks of a Marathon simulation into component pieces. Unfortunately, the implementation details of “how” UnitData is updated, “how” demands are filled, etc. still linger.

The detail-oriented reader might have noticed the emphasis given to the words “direct” and “update” in the bullet-list descriptions. The reason for the distinction is fundamental to our abstraction process. The direction of a task is much different than the execution of a task. Execution implies access, authority, and ability to make fundamental changes to a thing (in this case some state in an object). Direction of a task is akin to giving orders. Hopefully, the following example will reinforce the difference:

A company commander directs the platoon leaders to take a hill. Each platoon leader directs the squad leaders to take objectives necessary to take the hill. Each squad leader directs the squad members to move towards said objectives and shoot the enemy. Each squad member executes the directed orders to move and shoot. Moving and shooting are concrete actions that result in changes. One can visualize the abstraction layers as a sort of “conceptual transform” operator that turns a higher-level input into one or more lower-level outputs.

The beauty of stratified design, is that, like an onion, we can peel back layers of abstraction until we have gone deep enough. In our notional example, the squad member was “low enough”. The logical depth of abstraction, then, is the extent to which our problem needs to be concretely described. This is the next layer of the onion, so to speak. If we ever need to go lower in our design, we peel back another layer and implement.

Where managers served as logical routing systems, and databases for related information, our low-level objects implement more of the details and do the “grunt” work of making the system go. They are the privates receiving orders from their squad leader. The breadth of their understanding is thus limited, but the depth of their understanding is very great. In essence, these objects have much less abstract notions of “update” or “move” or “fill”.

Still, the abstraction layer holds. Even our low-level objects will be composed of “lower level” objects, just as the nucleus is composed of Protons and Neutrons, who are themselves composed of smaller bits. This is evidenced by the relation between UnitData and UnitBehavior, which we now discuss.

**Units and Behaviors**

Assuming that some higher level infrastructure is in place to coordinate the simulation, what would we actually “need” to calculate and change to provide data for results? At this point in our stratified design, we have to at least start to “twiddle the bits”. The answer to the initial question can be found in another question: “What kind of data do we want to collect?” Again, abstraction helps us to triangulate where to build concrete implementations. In the case of Marathon, we have an institutionalized set of standard outputs and metrics. For instance, we know that we need to capture the individual deployment and movement histories of large populations of units. From this information, we can derive many output metrics for the system, including the number of deployments, the time spent deployed, the time spent not deployed, missed deployments, etc. Thus, we “know” that we must provide enough state data to capture individual unit movement throughout the simulated system. We know that we must somehow record the history of these movements through the simulated time horizon.

While we now know a few more things about our problem, more dependencies snuck in during our evaluation of “what kind of data?”. One obvious dependency is, “how do units ‘move’?” Another is “what do units do after they move?” Finally, “what are the appropriate sequence(s) of moves?”

**Previous Implementation**

Examining previous answers to these questions will be enlightening on more than one front. The prototype implementation answered these questions through some straightforward, if inefficient means. The beast of burden in the previous implementation was the multidimensional array, or an array of heterogeneous containers (“structs” in C/C++, Records in F#, Structures in Clojure/Lisp) . The data representation of entities reflected the design method in place at the time. Rather than operating from abstract form to concrete implementation in a stratified approach, the previous designer built “from the ground up”, starting with concrete data structures for the most concrete pieces of the simulation. Unfortunately, pouring concrete can lead to inflexible, or unmanageable architectural consequences that are nearly irreversible. Additionally, the data structure dominated the design to the point that it crossed multiple abstraction layers, effectively preventing stratified design.

At the atomic level, entity data was contained in a 2D array of UnitData records. UnitData contained state information for a specific entity, such as BOG, Dwell, last date to Reset, current location, etc. Since the previous design chose to explicitly handle time via a constant interval (a time-step simulation), the straightforward approach rested in dimensioning the array to be #units x #Intervals. Thus, each cell in the array pointed to a unit’s history sampled at a constant interval. The resulting algorithm simply iterated over this container, copying state from the previous time interval, and updating state for the current time interval. The theme of maintaining large (identically dimensioned) 2D arrays of #things x #timeintervals quickly dominated the rest of the design. In short, the concrete nature of the 2D, time-indexed array provided a straightforward (to some) means of containing time-variable data. Due to the concrete (read simple) nature of indexing into arrays, the task of fetching a unit’s state in time meant an index into the array to modify a record of unit data. On the surface, this would appear to be favorable.

Unfortunately, this particular implementation is very inefficient. It relies on explicit manipulation of unit records as they exist in the array, requires redundant copying of large sets of unit data over time intervals, and it requires n\*timeinterval space to allocate the array as the application size grows. Additionally, the design methodology led to very concrete forms of accessing and modifying unitdata in the array, which created large chunks of redundant, nested iterative loops and conditionals. The unintended consequences of initiating development with concrete, low-level design manifested in a bloated, obfuscated codebase with extremely limited capacity for extension, change, and even error-checking.

**Alternative Implementation**

One (current) alternative is to take the “good” from the array implementation and incorporate it into a stratified design. The sole “good” element of the array implementation rested in the concrete data encapsulated in the UnitData records. This serves as a useful template for the “kinds” of information we will require in our simulation. In fact, the information included in the UnitData records stemmed from years of iterative improvements in institutional knowledge of rotational modeling. It makes perfect sense to utilize it as a basis for our primitive unit objects.

In our stratified design, we are not concerned with the concrete management of time at the unit level of detail (just as the squad member is not concerned with the strategic value of the location he’s moving to). Thus our unit data can be divorced from a structure that explicitly encodes time (the 2d Array). Rather, we can treat each unit as an independent set of state data (not dissimilar to the record), but with additional functionality and utility provided by the VBA object model (or the wonderful world of functional programming in Clojure and F#). We can then write many many processes that consume unit data and provide functionality useful to the simulation. It should be noted that if we had an explicit need to encode time into our data structure, we could choose it as an alternative implementation option in our stratified design WITHOUT BREAKING the higher layers of abstraction. The converse (as demonstrated by the failed transformation of the array-based prototype) does not hold. Again, thinking in layered, top-down abstractions, provides the ability to choose the appropriate implementation for the job at hand.

**Units Are Simple Objects**

By wrapping our unit data in an object, we gain multiple benefits, the greatest of which is the access to higher-level functionality of objects (this is more important in VBA; in our functional languages, this problem does not exist). Like records/structs/userdefinedtypes, classes (objects) allow us to provide a clean interface to a chunk of heterogeneous data (unit data). They also allow us to attach additional functionality to an object, including class-specific methods and properties that greatly improve the fluency of code, wipe out droves of boiler-plate indexing calls, and allow for instant, modular extension of functionality. In essence, we can add an explosive amount of information to a “dumb” container of data, like a UnitData record, to make it far more pleasant to work with. By exchanging the concreteness of an array-structure, we gain a vast amount of clarity, expressiveness, and extensibility in the object representation of units. Furthermore, units fit nicely into our higher-level design: they are simply objects owned by the SupplyManager. All the SupplyManager has to do is instruct its objects to update themselves. The unit object’s interpretation of what “update” means can vary drastically, depending on our implementation of update, which is something that we will exploit over and over again.

The general strategy then, is to wrap the concrete data we need for simulating units inside of an object called UnitData. UnitData links the supply-manager abstraction layer with the concrete state inherent in unitdata. Units can now be modeled as individuals, referenced individually, included in collections easily, etc. Also, since units are objects, they can be accessed by reference, rather than by value. This means that we can pipe object references (integer “pointers”) to any sub or function, rather than copying the array or the entire struct (which VBA forces us to) when operating on datastructures by value.

* UnitData
  + Stores state data for a unit
    - Primitive data: Name, SRC, CycleTime, Location, etc.
    - *Behavior reference* (to be explained)
    - *Policy reference* (to be explained)
  + **Directs** the updating of state data for a unit
    - *Update method.*

**Units Have Behaviors…..Behaviors Have Units**

While we have an answer to the question of atomic unit data, what about the questions surrounding “supply physics”? The attentive reader will probably think that we still have many unanswered, non-trivial questions, such as “What makes a unit move?”, “What are the sequences of moves a unit takes”, etc. The reader is right to criticize this gaping hole, but should not be surprised by the delayed evaluation of these questions. Abstraction, stratified design, and delayed evaluation are all parts of our philosophy. We deal with the phenomena of behaviors because now is the appropriate point to do so.

The treatment of behaviors will, not surprisingly, focus on the abstract and quickly hone in on possible implementations. Our goal is to provide a mechanism for combining unit state with some motivating force that causes units to “do stuff.” The nature of unit behaviors should be dependent on some things: some sort of policy that describes the nature of unit behaviors in supply and demand, and an interpreter of policy that can evaluate policy in the conjunction with the current unit state to apply changes necessary for simulation. A brief physics analogy follows:

In a physics simulation, we could combine a particle’s positional data with the equations of motion, and an integration mechanism. The behavior (integration mechanism) would evaluate the policy (equations of motion) in the context of entity data (particle position) to apply force over time to create movement (the simulation).

All we need to do is extend our analogy to the UnitData, UnitBehavior, and rotational Policy. We can generate our “supply” physics by relating these three components to help the simulation (any simulation actually) to derive a unit’s state (akin to a particle’s position) as a result of its rotational policy (the equations of motion) interpreted by its behavior (the integrator) into an applied result (a change in position).

In the current implementation of Marathon, we exploit the newfound properties of UnitData objects by including references to a behavior and a rotational policy as data. This allows units to maintain enough information to simulate themselves individually. This also has the powerful implication that units ARE individuals. We could have an infinite variety of behaviors and policies if we wanted to, which creates vast opportunities for custom-made scenarios that require unique unit behavior. Even better, we are not REQUIRED to generate vast amounts of behaviors and policies, because the link between units, behaviors, and policies is based on reference. In essence, we can make things as unique as necessary, or as generic as necessary. All units can follow the same rotational policy using the same behavior (via identical references), or categories of units can have drastically different behaviors. Furthermore, we can easily modify our behaviors and policies as functions of time or events, merely by updating references. Subscribing units will automatically modify themselves to suit. Again, we are presented with a situation where we have definitive answers to our questions in one layer of abstraction, but to further realize what policies and behaviors are, we must extend our design deeper.

**Unit Behaviors Implemented as Finite State Machines**

One plausible implementation of unit behaviors is to hard code single functions, and combine possible behaviors into a massive Select (or Switch or Match) statement to handle all the logical branches and possibilities. In fact, this is more or less the previous implementation. The upside is that, at least early on, behaviors are easy to model. You simply hard-code in the unit-changing logic inside of a supply update loop, and process the loop every day of the simulation. The downsides begin to sprout when units need to have unique, possibly stochastic, or event-driven behaviors. If even one unit is nominally different, then the ENTIRE behavioral logic inside of the supply update loop must be modified to account for the new case. As time drags on, the logic becomes a nightmarish monolith of entangled conditional statements, incremental state changes, and obfuscated causality.

Another plausible implementation is to factor out the redundant bits of the behavior logic, and to only encode behaviors that are “truly” unique. This approach allows flexibility of design (add new behavior cases, or even behavior systems as needed, and subscribe a unit to them), code re-use (multiple units can now reference your new and improved behavior, while the rest use the old or default behavior), and scoped design. Things are much easier to handle when we break them up into components that can be composed and referenced, rather than dealt with as monolithic blocks of logic.

The most common approach (not without its own flaws), is the Finite State Machine (FSM). For ease of understanding, the author will refer to possible nodes in the Finite State Machine as States, with the capital “S”, vs. state as in state data. Finite State Machines tend to be incredibly useful for a wide range of problems, particularly modeling simple behaviors. A Finite State Machine is literally a virtual “machine” that can exist in multiple “**S**tates”. Each state has a set of paths, or transition possibilities, that indicate which other states can be reached from the current State. Depending on the needs of the process or behavior being modeled, there may only be single paths emanating from each state, which indicates a very simple, directed flow of State change. Transitions can be probabilistic, where many States can be reached from the current State, according to some probably from 0.0 to 1.0. Certain States may be “absorbing States”, from which there is no escape. The FSM uses its States as a sort of roadmap to guide execution of some process. At the primitive level, the FSM is a representation of “where to go next, given the current State.” A simple analogy using a cat follows:

Assume we have a fat, tired, and fastidiously clean cat named Stella. The cat’s behavior can be accurately modeled by a FSM. After days of observation, we realize that the Stella’s behavior at any given point in time will be one of three States: Sleeping, Eating, or Cleaning. Stella eats after she sleeps. Stella cleans after she eats. Stella sleeps after cleaning. The result is a simple directed graph:

In the simple case, the FSM tells what the next State will be, given any initial state. In the cat example, *Sleeping* leads to *Eating*, which leads to *Cleaning*, which leads to *Sleeping*…forever (we don’t have *Death* state in our graph). The beautiful thing about FSMs is that they fit into our natural abstraction and stratified design paradigm exceedingly well. Humans, with limited perception, tend to categorize behaviors in this manner very often, and they provide a very declarative narrative of the underlying behavior. All we need are the possible States, and the transitions between them. Depending on our needs, the FSM can be very simple (this is preferred), or highly complex (this can be detrimental to design), or a blend of the two using abstraction. Also, the description of the FSM does not stray into the concrete category. Only the information necessary for describing the relations between states is exposed. What “Sleep”, “Clean”, and “Eat” mean, in the concrete sense of how Stella changes or the specific actions performed during these States, is an independent concern.

Our focus is on how units behave, not cats. So what would a simple UnitBehavior FSM look like? The following graph describes a unit’s a behavior very simply.

Without knowing what “Dwelling”, “Bogging”, or “Start New Cycle” mean, we can still derive highly useful information from the graph. If this graph describes the unit’s behavior, then we know that a unit can transition from *Dwelling* to (*Dwelling* **OR** *Bogging)*, from *Bogging* to *Start New Cycle*, and from *Start New Cycle* to *Dwelling*.

What if we need additional detail about the behavior? We can always simply update the existing FSM description. For instance, if we need to add another state, perhaps to End Current Cycle, we draw it in the graph:

Our new behavior indicates that units changing State from *Bogging* will transition to *End Current Cycle*. The naïve solution to building complex behaviors with FSMs consists of adding new nodes to the stategraph. This works fine for small graphs, but as the number of states grows, our easily-readable chart begins to converge on spaghetti.

Fortunately, we can always rely on our earlier mechanism for dealing with complexity: abstraction. What if we starting and ending cycles are really very similar States that belong to a greater State of behavior (a SuperState)? At the Bogging, Dwelling level of understanding, we can collapse Start New Cycle and End Current Cycle into a third state, a Super State, which “contains” the underlying states and helps us abstract them away without losing them entirely.

Our new graph allows us to think of unit behavior in terms of three general states, *Dwelling*, *Bogging*, and *Admin*. If we needed to drill down to a lower level of detail, in the Admin state, we would see two states contained within *Admin*, *Start New Cycle* and *End Current Cycle*. With this mechanism in place, we can mitigate the complexity of our behaviors, while employing the utility of the Finite State Machine. This is referred to as a Hierarchical Finite State Machine (HFSM), which provides many improvements over the FSM. Mainly, it enables us a choice in how we compose the description of our behavior. If we decide to add many additional States, we can willingly create a complicated-looking FSM. However, if we wish to abstract details away across various layers, we can potentially have unlimited detail, while keeping our notion of abstract States manageable.

As an aside, the FSM and HFSM can also be viewed in somewhat of a tree-like fashion, which reinforces the concept. If we take our HFSM of unit behavior, we could view it like so:

This depicts a Behavior Tree, which has risen as an efficient alternative to HFSM. The same states still exist, but they are processed from Left to Right. *UnitBehavior* initiates in *Dwelling*, until *Dwelling* is done, then changes to *Bogging*, until *Bogging* is done, then proceeds to *Admin*. *Admin* behavior begins in *Start New Cycle*, then proceeds to *End Current Cycle*. *End Current Cycle* is the last behavior in the group, so the behavior moves back to the original *Admin* behavior to see what the next behavior should be. *Admin* behavior is the last behavior in the group, so again we step back to *UnitBehavior*. Behavior Trees are another way of encoding an entity’s behavior, specifically how certain behaviors lead to other, or the fact that certain behaviors like *Admin*, are composed of subordinate Behaviors (*Start New Cycle* and *End Current Cycle*). The utility of Behavior Trees lies in the fact that they are easily visualized (like FSMs), and in the fact that behaviors can very easily be composed. As you make simple behaviors, they can easily serve as primitive components in sequences of complicated behaviors. As an aside, Behavior Trees are the intended future of UnitBehavior, which is the reason for their inclusion. Currently, UnitBehavior is implemented closer to a simple FSM, which works surprisingly well. Again, implementation is not as much of a concern, thanks to a thoughtful abstraction process.

**States Are What They Eat**

Since UnitBehavior is currently implemented in an FSM, we will return to referring to individual elements of behavior as States (otherwise we would have to call them Finite Behavior Machines). Until now, States have served only as a nice, abstract organizational tool for characterizing the possible behaviors of “something”. Our current focus is on Units. Somehow, States must be “do things” to or with UnitData to accomplish changes in state, so that we get a simulation. The FSM description describes the possible States and how they relate (just like the graph), but the States themselves are actual concrete consumers (and producers) of UnitData. States are where the grunt work happens, in the form of concrete code. As a result, the abstract unit FSM, uses States as the tools for modifying any unit’s state (we say any unit, because multiple units may share behavior, and thus become operated on by a single referenced implementation of FSM). States then receive Units (UnitData) as input, and provide update units (UnitData) as output. They are merely functions of UnitData that return UnitData. States chew on unit data to produce changes in the simulation.

The two reasons for this design are composition and a separation of concerns. Individual States are not dependent on other states. They perform their prescribed task, taking UnitData as input, and returning the result of the State’s operating on the input. The returned output is still UnitData. We only have to perform operations necessary for the specific State, which allows developers to craft specific, but limited functionality into each State. Yet each State is guaranteed to take and return UnitData. Thus, we gain the ability to compose simple states into complicated states. This is a wonderful bonus (actually a cornerstone of functional programming).

There are many wonderful side-effects of this design as well. Our human minds only have to focus on smaller bits of truly different State handling code (less boiler plate, less redundancy), while we have the freedom to utilize our previous work (via composition) for future solutions. Oh, we also get to add an infinite number of states as we see fit. Also, if we need to add new data, we only have to add new bits to the UnitData class, which all states work on implicitly. Code is updated automatically via reference, and the compiler even gets to help us prevent errors. FSM for the win, all around![[65]](#footnote-65)

**UnitBehavior States Are Informed By Policy**

One last, useful element of UnitBehavior, is the notion of a Policy. The idea for Policies rose from the domain of rotational analysis. We tend to think of our rotational simulation as a function of Supply, Demand, and Policy, in which policy is applied to supply to fill demand (sounds familiar….). The result is a simulated history of units moving through supply, filling demands, and anything else we care to know about. Policy ends up being very important (and very variable). For instance, rotational policy can be made to change during a simulation, so that more (or less) units are generated from supply. The bottom line is that policy is enough of an integral piece of simulation, that it got its own Manager and a dedicated object implementation.

The elevation of policy has provided great dividends in terms of UnitBehavior, because Behavior is highly dependent on Policy. By packaging a policy reference within UnitData, we provide UnitBehavior States with all the information they need to make the appropriate changes to a unit. We also drastically increase our degrees of freedom, because Policy is now a first class data type with its own “subscribing” units. Each individual unit’s behavior is now a function of unit data, and policy. We can use our imaginations to combine changes in policy, and/or changes in behavior to account for unique scenarios that may arise.[[66]](#footnote-66) Again, the result is an efficient way to extend functionality without ripping up concrete implementations. If our policy implementation needs “more stuff”, we can extend it without breaking behaviors, and vice versa. Furthermore, as we add information to policy, or UnitData, we can build more complex and information-driven behaviors.

**Policy Implementation**

The behavior samples from a few pages back provided a clear, understandable depiction of related states using a graph (circles and arrows). We tend to think of Policies in much the same way, so it only makes sense to build them using the same tools. Policies are currently implemented as objects, who base data is a graph data structure that stores information about the policy. Unlike our StateGraph abstraction for behaviors, we store plenty of reference information in a policy’s graph, and derive different meaning from the relationships.

We will start with a canonical example:

A typical ARFORGEN rotational policy consists of a few key phases: *Reset*, *Train*, *Ready*, *Available*. An ARFORGEN lifecycle is the cycle of movement in the policy where a unit leaves *Reset*, and ends upon its return to *Reset*. The graph above depicts the general logic behind this flow. From the PolicyGraph, we know that A unit’s ARFORGEN Cycle, starting from *Reset*, will be *Reset|>Train|> Ready|> Available*, returning to *Reset*. This is called a cycle because, absent an external disruption, units in this path will continue cycling forever.[[67]](#footnote-67) Note that the PolicyGraph does not indicate “how” a unit, through phase transitions, could get to its *Deployed* phase.[[68]](#footnote-68) The graph does tell us what policy to follow if a unit finds itself *Deployed*: *Deployed |> Overlap |> Reset*. After these transitions, the policy continues on the original cycle forever. We will visit how *Deployed* could happen later.

Graphs are useful things, since they map easily to our mental picture of abstract concepts. We have already used them to describe behaviors, and now policy. Primarily, we described the relationship between States in UnitBehavior, and the transitional flow of Phases in Policy. We can add more information to the graph to increase its utility, fairly easily. The numbers in the next PolicyGraph communicate the cost[[69]](#footnote-69) (in days) that is required to facilitate traversing from one phase to another. The English description matches the mental model exactly : “It will take 182 days for a unit to move from *Reset* to *Train*.” Following our previous discovery of the cycle *Reset|>Train|>Ready|>Available* , we can say with confidence that “According to this policy, a unit’s ARFORGEN cycle is 1095 days in length.” We merely summed all of the costs as we moved along the path (182 + 183 + 365 + 365 = 1095). Similarly, if a unit finds itself deployed under this policy, it will take (320 + 45) = 365 days before it returns to *Reset*. The colloquial description of this policy is a “1:2 BOG/Dwell” policy. The description stems from the fact that the amount of time spent deployed (BOGGING) is one half the time spent Dwelling, under ideal conditions. That result cannot be derived from the information in the Policy Graph alone (purposefully).

Although it is very useful, we still need to add more information to the graph. One bit of critical information would be whether or not Policy has any bearing on a unit’s behavior. We already indicated that behavior interpreted policy to make changes to a unit, but there is no obvious roadmap for this in the examples so far. The simple solution is to store State information in tandem with the ARFORGEN phase information on the policy map. Just as we enhanced our graph with useful time information, we can detail the State that corresponds to a unit’s behavior under a policy:

We can theoretically add as much information as necessary to any graph structure using the implemented data structure in Marathon. Using graphs to handle our data allows us to store and utilize our data in a format similar to the abstract representation. We get the added benefit of tying different domains (Policy and Behavior) together using a familiar structure.

For a unit operating using a reference to this policy in its UnitData, the unit’s behavior will poll the policy for information so it can make decisions: “what’s the next location, according to your policy?”, “What is the State associated with your current location?”, “How long should you wait in your current location?” This means that we can implement the logic for behaviors in a general sense, and efficiently model excursions just by changing the structure of the policy (or changing policy dynamically). Thus, the behavior effectively parses a unit’s unique policy to determine the “how” that we have avoided discussing. Once the appropriate State is determined, the unit’s behavior modifies it according to the appropriate Unit State handling function.

**GenericStateData Contains Information Useful for All States**

Our behavior FSM will need to retain some intermediate information which, in the general sense, creates a useful set of information for ALL FSMs to know, whether they are for UnitBehavior or not. This universally-useful data is codified in a helper object called GenericStateData. It primarily serves as a universal book-keeping mechanism for FSMs. Specifically, it contains primitive data elements useful for any FSM, such as the current state, the previous state, the expected duration of the state, the absolute start time of the state, the time elapsed in the current state, etc. These are all necessary, useful, and portable chunks of data that “any” FSM can use. Rather than copy the data around, we encapsulate it in a class that is attached to the underlying data of the object with simulated behavior. In a “proper” object-oriented language, we could implement this as a required interface, or contract, that any class with behaviors must provide an implementation for. Due to the scope of our design, and the limits of VBA, we simply wrap the functionality inside of a component class, and attach the data to any class with behavior.[[70]](#footnote-70)

**Current Implementation of UnitStates Within the UnitBehavior Object**

Where previous sections built the abstract reasoning for structure of UnitBehaviors, we proceed with an existing VBA implementation of UnitBehaviors, namely the code structure behind UnitStates and the objects required. Typical FSM implementations (in C++, C#, or Java for example), will utilize some base notion of a state, and then extend that implementation for each special-use case (each implemented State). In this scenario, all UnitStates would be derived from a base State class, fundamentally sharing the same functionality (such as the capacity for updating UnitData). The concrete State implementations would be special classes that inherited this functionality, but would exist as formal, separate, instantiable objects.

For practical reasons, we will not follow the standard FSM implementation scheme. One concern with the VBA design is the potential for a rapid explosion of classes. VBA has no faculty for Namespaces (an organizing feature within modern languages), so that every piece of code must reside in either a primitive module, or a class module. Class modules, as one may infer, are actually specifications for instantiable objects. Normal modules are “primitive” objects, in the sense that they are collections of functions, procedures, and global variables. Modules may communicate with each other, but the vastly more effective method is to use classes to instantiate formal objects. VBA classes provide many (but not all) of the benefits of typical object-oriented languages, including polymorphism, encapsulation, and inheritance. Still, without higher-order language constructs like namespaces, class templating (to design generic objects), or Lambda functions (an anonymous function and the cornerstone of functional programming), we must find a happy medium for implementing potentially many States in VBA without bloating our code base with too many classes.

As a result, the current design calls for organizing related states within a behavior as functions associated with the behavior object. The actual state implementation is simply a function (a method in object oriented speak), that resides within the behavior object. This provides a clean separation of concerns, and prevents the proliferation of excess classes. Thus, a Behavior simply consults its library of State Handler functions when updates are requested. By definition, all State Handlers for a given type of data look identical, in that their input is the same “type” as their output. It makes sense to use behaviors as function routers (or dispatchers if you think in the Scheme language).

Thus, Encapsulation using objects enforces an auditable flow of control and data, allowing for adding to or removing depth from our abstraction layers[[71]](#footnote-71).

The current implementation wraps unit data and functionality into a UnitData object.

Again, the UnitData object:

* Stores state data for a unit
  + Primitive data: Name, SRC, CycleTime, Location, etc.
  + *Behavior reference*
  + *Policy reference*
* **Directs** the updating of state data for a unit
  + *Update method.*
  + *ChangeState method.*

When simulating, each UnitData object has a reference to the policy and behavior that determine the context of its state changes. To update the unit, all an external caller really has to do, is make a call to the Update method for UnitData, providing the time as an update argument. The UnitData object then routes this call to its internally referenced behavior. Essentially, the unit passes itself to the target behavior. The mechanics of the behavior then decide what to do with the unit just passed in. Since the entire unit is referenced, the behavior has access to the particular policy associated with said unit. All the information required for the behavior to do its job is neatly encapsulated within the UnitData object.

UnitBehavior determines the current state that the unit is in, and then routes the UnitData to an identically labeled **State Handler Function** inside of the UnitBehavior class module. This is accomplished via the UnitBehavior’s own Update function, which takes the UnitData and the change in time as an argument. Once the current State of the unit is derived – such as *Bogging*, *Dwelling*, etc. – the behavior routes the UnitData and the time delta to the appropriate function by way of a simple Select Case statement.[[72]](#footnote-72) The State Handler Function then determines what changes to make to the unit to represent the passage of time in the state (maybe accumulating BOG time, or Dwell time, or forcing a move to another location). After the updates are complete, the reference to the updated unit is returned to the original calling function: the UnitData object’s update method. As stated in previous sections, the net effect of a unit update is to return a reference to the unit, after undergoing simulated behavior for the specified time.

**Generating a Supply**

**Scripting UnitBehavior Changes**

**Scripting Policy Changes**

**Event-Driven UnitBehaviors**

**Representing Demands**

**Controlling Demands Via Activation and De-Activation**

**Filling Demands**

**Priority of Demand**

**Suitability of Supply**

**Capturing Output**

**Generic Output (GenericRecords)**

**Time-Series Output (Trends)**

**Visualizations**

**(more to come)**

1. Rather than the individual soldier. [↑](#footnote-ref-1)
2. According to Rand (citation needed). [↑](#footnote-ref-2)
3. We assume constantly, but the increase in readiness is more closely related to a step function of varying step width, which is validly approximated linearly. [↑](#footnote-ref-3)
4. ARFORGEN is the institutional model for force generation, although hybrids and alternative models are being examined. [↑](#footnote-ref-4)
5. As of Jan 2012. [↑](#footnote-ref-5)
6. One consequence is that ARFORGEN is dynamically unstable, and will not tend toward an ideal state without control. [↑](#footnote-ref-6)
7. Temporal questions are incredibly common, if not mandatory, in most force generation analysis. [↑](#footnote-ref-7)
8. Optimization performed iteratively in response to events within the simulation, usually used for dynamic control. [↑](#footnote-ref-8)
9. For purposes of abstraction, the three simulations are viewed as completely independent, although the current implementation subordinates them to a single, overarching simulation engine. [↑](#footnote-ref-9)
10. Perhaps millions in the very near future….if for some reason it’s needed. This is currently a limitation of the VBA host platform. [↑](#footnote-ref-10)
11. Policies are intended to look structurally similar to the ARFORGEN policy depicted earlier, but they can be entirely different as well. For some units, the structure of the state transitions intentionally looks nothing like an ARFORGEN cycle, and may not even be cyclical in nature. The policy system is truly general. [↑](#footnote-ref-11)
12. Corner cases and unanticipated changes imposed by sponsor requirements invalidated Marathon 2 and the initial VBA port. Unable to adapt, the designs had to be abandoned. [↑](#footnote-ref-12)
13. This is somewhat circular, since time is technically an event… [↑](#footnote-ref-13)
14. Aside from deep philosophical questions of relative complexity, Marathon tends to be something most folks find too complicated to understand at first glance. [↑](#footnote-ref-14)
15. “an underlying scheme that governs functioning, developing, or unfolding.” [↑](#footnote-ref-15)
16. “to create, fashion, execute, or construct according to plan.” [↑](#footnote-ref-16)
17. The noun and verb forms of “design” are really two sides of the same coin, in that they embody the aforementioned design philosophy underlying Marathon. [↑](#footnote-ref-17)
18. No matter the complexities of a contemporary implementation, one should always be able to rely on some familiar design philosophy as a guiding light to orient. [↑](#footnote-ref-18)
19. This is due primarily to data constraints and the lack of a firm understanding of the relations between various systems. We could easily construct an incredibly sophisticated, and quantitatively complex, representation, but its validity would likely suffer. Even if we do attain the required data in the future, the incidental complexity for managing, parsing, and verifying the data will still inject complications. [↑](#footnote-ref-19)
20. Input validation alone accounts for a significant blend of human and machine heuristics to verify the integrity of source data. “Matching” the basic unit types (SRCs) between supply and demand is an arduous task, even for subject matter experts. [↑](#footnote-ref-20)
21. The process of applying Marathon during a study introduces practical concerns that have nothing to do with force generation: input data validation, experimental design, discrete-event simulation, optimization, run-time scripting, database management, analysis of algorithms, interactive visualization, exploratory data analysis, and some statistics. [↑](#footnote-ref-21)
22. Ease of use is relative. Ease of use is often balanced against flexibility. [↑](#footnote-ref-22)
23. In reality, there is an infinite combination of the two approaches, but deriving a useful example [↑](#footnote-ref-23)
24. The human mind only has a finite capacity, so our ability to comprehend massive, intricate designs is fundamentally limited. We tend to become more conservative in our design process, when large, intricate, and interconnected processes are involved, because of the fear of “breaking something” due to change. The end result is an ever-shrinking comprehension of the total design, and a fear of change. [↑](#footnote-ref-24)
25. Per Polya. [↑](#footnote-ref-25)
26. It is vitally important to highlight both the successes of previous designs, and the context for long-term “failure” of the design philosophy. In this case, I assert that the design philosophy failed due to the inability to adapt to emerging features, and the requirement for generally drastic and widespread changes to Marathon’s infrastructure. This is not to say the developers and analysts using Marathon were incompetent or unsuccessful; the opposite is true. They were very successful in demonstrating a useful capability to the Army, so successful that their reward was an ever-expanding feature-set due to increasingly difficult studies. [↑](#footnote-ref-26)
27. Courtesy of the Merriam-Webster Dictionary; <http://www.merriam-webster.com/dictionary/chimera> . [↑](#footnote-ref-27)
28. Input validation alone accounts for a significant blend of human and machine heuristics to verify the integrity of source data. “Matching” the basic unit types (SRCs) between supply and demand is an arduous task, even for subject matter experts. [↑](#footnote-ref-28)
29. Or any complex process or phenomena. [↑](#footnote-ref-29)
30. We will not throw out the “how”. We merely delay its discussion until needed. Too often, we worry about “how” before understanding “what” we need. As the adage goes, we put the cart before the horse. [↑](#footnote-ref-30)
31. Indeed, many of these questions arose directly from sponsors. Marathon is, in a sense, a collected embodiment of responses. [↑](#footnote-ref-31)
32. In some cases, the answer might be “maybe.” Sometimes incidental design can arise from artistic use of known techniques, which implicitly results in a design (form following function), but this is arguably rare. [↑](#footnote-ref-32)
33. This has actually happened, hence the need for alternate modes of communication. [↑](#footnote-ref-33)
34. This is a broad generalization (indeed a fallacy). A more appropriate statement would be “people want to know enough about Marathon”, which includes various levels of “what” and “how”, depending on the audience. [↑](#footnote-ref-34)
35. Abstraction is our bridge to analogy, and analogy is our tool for cutting out the noise. [↑](#footnote-ref-35)
36. Computer Scientists like to call this Pseudo Code. I like to call it English. [↑](#footnote-ref-36)
37. This is a running theme in Mathematics, Operations Research, Computer Science, and any attempt at approximating reality. [↑](#footnote-ref-37)
38. This can be a double-edged sword if the complexity manifests in user-interaction or data requirements. Assume that the system is well designed so as to allow potentially complex, or potentially simple representations. [↑](#footnote-ref-38)
39. They are almost guaranteed to change. Note that things like rotating/updating supply, changing rotational policy, etc. all have levels of complexity and will be addressed. [↑](#footnote-ref-39)
40. Note, “ignored” may be too strong a word. In some cases, the host platform (Pro-Model) effectively prevented consideration of elegant solutions to certain forms of complexity and ensured the cycle of patch-bulldoze-rebuild. This continued until features were required that were beyond the realm of Pro-Model entirely, prompting a departure from the platform. [↑](#footnote-ref-40)
41. The History of Marathon illuminates further complicating factors that contributed to this design philosophy. [↑](#footnote-ref-41)
42. This may be overstated, but in the relative sense, the new architecture and design philosophy should allow a huge amount of flexibility. The goal is to extend and adapt the design purposefully, reflecting a sort of organic growth, where core components are inherently re-usable. This is orthogonal to the previous philosophy of patching, bulldozing, building new, patching, bulldozing, etc. [↑](#footnote-ref-42)
43. Extension allows us to “add more” information without corrupting older versions. This is desirable, as it retains compatibility with previous versions (especially data fields), and does not “force” the use of unnecessarily complex features. Marathon can retain the things it does well, and add new features as needed. The net effect is an accumulation of useful, composable functionality. [↑](#footnote-ref-43)
44. Later sections on implementation will go into further detail on stratified design, and its benefits. [↑](#footnote-ref-44)
45. The source of these strategies varies, but many come from the world of Functional Programming and classic texts such as *Structure and Interpretation of Computer Programs*, by Abelson and Sussman. [↑](#footnote-ref-45)
46. “We have also obtained a glimpse of another crucial idea about languages and program design. This is the approach of stratified design, the notion that a complex system should be structured as a sequence of levels that are described using a sequence of languages. Each level is constructed by combining parts that are regarded as primitive at that level, and the parts constructed at each level are used as primitives at the next level. The language used at each level of a stratified design has primitives, means of combination, and means of abstraction appropriate to that level of detail.” Abelson and Sussman, *Structure and Interpretation of Computer Programs, 140.* http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-15.html#%\_sec\_2.2 [↑](#footnote-ref-46)
47. Language is even an implementation detail. We can map the abstract Marathon algorithm to any language. In fact, by targeting a multilingual platform (like the .Net Common Language Runtime or the Java Virtual Machine), developers are free to utilize disparate host languages while successfully interacting with eachother. One example is C# interacting with VB.Net and Visual C++ libraries seamlessly. [↑](#footnote-ref-47)
48. This is why the functional programmers love what they do. In statically typed languages like Haskell and F#, this form of thinking has a great many side-effects, including a greater degree of verification and error-proofing. [↑](#footnote-ref-48)
49. You get modularity and backwards-compatibility for free. [↑](#footnote-ref-49)
50. We can generate an infinite variety of chart types from Marathon Data, without having to know anything about Marathon. Just the data abstraction. Similarly, Marathon can process any set of Marathon Data, without knowing where it came from. This allows us to implement interfaces to databases, worksheets, flat files, serialized objects, etc., which all provide Marathon Data. Furthermore, we can generate any number of custom Marathon Engines. Providing custom-built user interfaces for generating supply, experimental design, output analysis, etc. becomes feasible with a solid data abstraction. This also allows developers to distribute work more effectively. [↑](#footnote-ref-50)
51. This is not to disparage Plinko machines! In many cases, a static design is preferable, but for Marathon, static equated to brittle. [↑](#footnote-ref-51)
52. For a full discussion of the wonderful world of graphs, please see Google or Wikipedia. Note, these are mathematical structures, not Excel Graphs. [↑](#footnote-ref-52)
53. We can implement this in a number of ways inside the computer. For humans, we can draw them nicely on a whiteboard. They are visual and intuitive. [↑](#footnote-ref-53)
54. It can kill a lot of birds with one stone, essentially. [↑](#footnote-ref-54)
55. We can just as easily use eventful microseconds, but integer days tend to be preferable. The basis is arbitrarily changeable, however. [↑](#footnote-ref-55)
56. We generally introduce other stopping criterion as well. [↑](#footnote-ref-56)
57. Djikstra, Bellman-Ford, A\*, D\*, DepthFirstSearch, BreadthFirstSearch, etc. [↑](#footnote-ref-57)
58. Again, check Wikipedia. It’s basically a graph with no “cycles”, i.e. we cannot find a path through the graph’s nodes the circles back on itself (possibly leading to an infinite loop). [↑](#footnote-ref-58)
59. Quickly is a relative term. Efficiently in terms of computational cost is a better phrasing. [↑](#footnote-ref-59)
60. This is a fundamental tenet of many modern computer game designs, where content generation is vital and hard-coded logic often leads to failure. If you can completely change the behavior and actions of your entities without messing with the game engine, then work can be offloaded to designers and content generators. [↑](#footnote-ref-60)
61. We simultaneously gain expressiveness, efficiency, dynamism, and the ability to change. [↑](#footnote-ref-61)
62. Probably unintentional. [↑](#footnote-ref-62)
63. Marathon could be part of a fitness function in a Genetic Algorithm trying to optimize force structure, or a value function in a Dynamic Program, etc. The possibilities are endless. [↑](#footnote-ref-63)
64. General Purpose Computation on a Graphics Processing Unit. [↑](#footnote-ref-64)
65. The Wise Cynic would probably doubt these claims. There are diminishing returns, just as there is No Free Lunch. For the current application domain, however, these advantages are easily realized and translate into a highly useful, legible, and extendible design. Even in VBA! [↑](#footnote-ref-65)
66. Behaviors that change policy? Why not? Policies that change Behaviors? Sure! [↑](#footnote-ref-66)
67. This is exactly the intent of ARFORGEN. [↑](#footnote-ref-67)
68. There is no arrow (edge) leading into *Deployed*. [↑](#footnote-ref-68)
69. Sometimes referred to as the edge cost, or edge weight in Graph terminology. [↑](#footnote-ref-69)
70. This design should scale fairly well, and any changes to GenericStateData will percolate throughout dependent classes automatically. [↑](#footnote-ref-70)
71. We can mimic some functional programming design philosophy here as well, although we still deal with mutable objects. Another term for this is creating a “fluent interface”. [↑](#footnote-ref-71)
72. Another reason for utilizing hierarchical abstractions, or behavior trees, is to avoid growing this routing mechanism. It’s another point where our codebase could become convoluted. [↑](#footnote-ref-72)