Chapter 1

How to Make a Monkey Do Something Smart

1.1 Build Your Own Monkey

This is a tutorial on designing and constructing the behavior for an artificial intelligent agent. There are other ways to make monkeys do intelligent things, but this is more interesting and doesn't involve issues of animal welfare.

Designing the intelligence for an artificial agent requires determining **when** to do **what**, and **how** to do it.

1.1.1 When

When can mean "at what time", but it is more powerful if one can be more general and say "in what circumstances." **When** is the problem of *action selection*. At any time, we need to be able to say what the monkey should do *right now*.

1.1.2 What

What is a problem of terminology and abstraction — at what level of granularity do we have to determine **when** the monkey will act? How much detail do we have to give? Assume that right now we want the monkey to stay out of trouble while we decide what to do next. Should we tell the monkey to 'wait'? To 'sit'? To 'put your legs under your body, put your hands on your knees, look around the room and at approximately 2 minute intervals (using a normal distribution with a standard deviation of 30 seconds around the interval) randomly select one of three situation-appropriate screeches and deliver it with gesticulations'?

This problem is reciprocal and analogous to the problem of segmentation in perception. In agent research it is often called *behavior decomposition*.

1.1.3 How

If we are building a monkey from scratch, then at some level we have to take care of what every individual component is doing at every instant in time. Generally speaking, it is not a good idea to try to specify incremental movements of every component at every millisecond time slice.

On the other hand, we can make the question of **when** easier by separating the **what** from the **how**. **What** is the primitive actions we refer to when specifying **when**. **How** is the way we individual components move for any **what**.

Making **when** decisions easier can make describing **how** harder. For example, if we decide that one **what** should be 'create world peace,' or even 'go buy a banana,' programming the **how** becomes complicated. On the other hand, if we make **how** something easy like 'move your little finger up an inch', we will have to do a lot of work on **when**.

1.1.4 Behavior Oriented Design

How do we choose our **what**s to strike the right balance between **how** and **when** to make the whole problem of our monkey's intelligence as simple as possible?

The answer is to take educated guesses at **what**, and then develop **how** and **when** iteratively.

If a **how** becomes too complicated, we decompose it into simpler pieces (new **what**s). For example, if 'wait' turns out to be too complicated a thing to build, we might split it into 'sit and scratch', 'snooze', 'look around', and 'play banjo.' We then need to recombine the **what**s using some **when**s. For example we might want to say 'snooze if you're tired enough', 'look around every 2 minutes', 'play the banjo when no one is listening' and 'scratch if you aren't doing anything else.'

If a **when** becomes too complicated, we develop new **how**s to support and simplify the decision process. For example, we may want to build a new **how** for the monkey so she can tell whether she's tired, or whether anybody's listening.

BOD exploits the traditional software engineering tools such as hierarchy and modularity to make things as simple as possible. It heavily exploits the advances of object oriented design and corkscrew development methodologies. It also uses new representations and understandings of intelligent processes from artificial intelligence (AI).

1.2 Say How

The way we say **how** under Behavior Oriented Design (BOD — see above) is using object oriented programming methodologies. The particular language isn't that important, except that development in general and our corkscrew methodology in particular, goes much faster in untyped languages like lisp, perl or smalltalk than in typed ones like C++ or Java. Of course, typed languages can *run* relatively quickly, but in general, the time spent *developing* an agent is significantly more important than the speed at which it can, in the best case, execute commands. *Finding* that best case is harder than making the agent run fast.

1.2.1 The Simplest Behavior

In BOD we decompose **how** into modules called *behaviors*, which we code as objects. Behaviors are responsible for perception and action. Perception is the interpretation of sensory input into information useful for controlling effectors. Effectors are anything that affects the external world. They might include motors on a robot, nodes in a model for a virtual reality character, the speaker or screen of a personal computer, or a teletype for an agent trying to pass the Turing test. Since behaviors are responsible for governing effectors, they must also perform any learning necessary for perception and control. Thus, like *objects* in software engineering, they consist of program code built around the variable state that informs it.

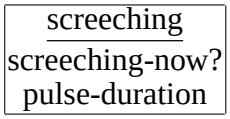
The simplest possible behavior is one that requires no perception at all, and no state. So, let's assume we've given our monkey a sound card attached to a speaker for one of its effectors. One ultimately simple behavior would be a screeching behavior that sends the sound card the instruction to go "EEEEEEEEEE..." all of the time. We'll call this the screeching behavior. We'll draw it like this:

1.2.2 Behaviors with State

Unfortunately, constant screeching does not have much æsthetic appeal, nor much communicative power. So we might want to make our screeching behavior a little more sophisticated. We can give it a bit of state, and only have the action communicated to the soundboard when that bit is high.

To make our screeching behavior even more interesting, we might want it to be pulsed, like "EEee EEee EEee". This requires some more state to keep track of where in the pulse we are. If we make our screeching sound a function of the pulse's current duration, we only need an accumulator to keep track of how long our monkey has been screeching.

Now the screeching behavior looks like this:



We draw the state inside the behavior's box.

1.2.3 Behaviors with Perception

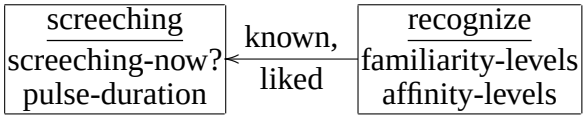
Relatively little behavior operates without regard to other events in the environment, or is controlled *open loop*, without feedback. For example, we might want our monkey to be able to modulate the volume of her screeching to be just loud enough to be heard over the other ambient noise in the room. The monkey should screech louder at a party than while she's sitting in a quiet house. This requires the monkey to have access to noise input, (a

microphone of some kind) and to be able to process that information to determine her own volume. We might include this all as part of our screeching behavior.

On the other hand, some perception might be useful for screeching, but require many states or processes that are generally unrelated to screeching. In that situation, it makes more sense for the additional perception to be handled by another behavior, or set of behaviors.

For example, real monkeys start screeching when they see someone enter the room. They also make *different* screeches depending on whether that person is a friend, an enemy, or a stranger. Visual recognition is a fairly complicated task, and is useful for more things than just determining screeching, so we might want to make it a part of another behavior. Replicating state is generally a bad idea in software engineering (it makes it possible the copies will become out of synch), so it is better if the screeching behavior uses the state of the visual recognition behavior to help it select the formants for a particular screech.

Here is a drawing of two behaviors:

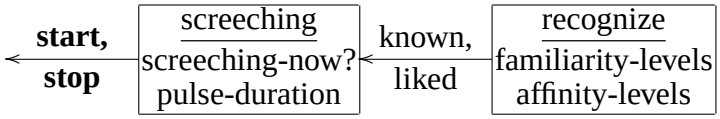


The method calls used to relate the two are put on an arrow between them. The direction of the arrow indicates the flow of information.

1.2.4 Behaviors with Triggers or Processes

Mentioning multiple behaviors brings up the possibility of conflicts between behaviors. For example, what if our monkey is at a surprise party, and sees the main guest walk into the room? The monkey should inhibit her screeching until someone gives the signal to start shouting. Similarly, if this is to be a very polite monkey, she shouldn't start screeching exactly when someone new comes up to her if she is eating a bite of cake! First she should swallow.

Under BOD, conflict resolution is handled by allowing an action selection mechanism to determine **when** things should be expressed. The interface between **when** and **how** is called **what**. A **what** is coded as a method on the object underlying a particular behavior. So for our screeching behavior, there might be two **what**s, a 'start' and a 'stop'. Deciding to do a **what** now can be viewed either as deciding to *release* or *trigger* an action of a particular behavior.



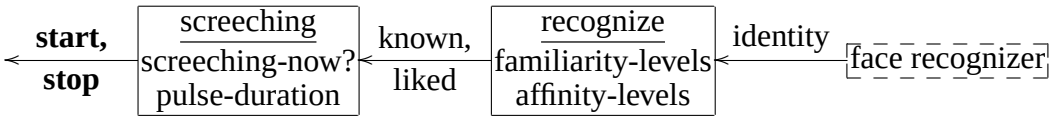
On the other hand, some actions of behaviors (such as learning or perceptual processing) may run continuously or spontaneously without interference from the **when** part of the intelligence. So long as they cannot interfere with other behaviors, there is no reason to coordinate them. For example, there's no reason (unless we build a duplicitous monkey)

to control the selection of formants from the **when** system. The screeching behavior could be continuously choosing the appropriate screech to make, regardless of whether it is currently screeching or not, by having a process constantly resetting its state on the basis of the identity (or lack of known identity) of any person the monkey is observing.

1.2.5 Behaviors that Aren't Objects

Some **how**s may be easier to build using other means than coding them from scratch. For example, they may be available in external packages, or they may be easier to learn than to program. That's OK too: in that case, both **what**s and inter-behavior methods are just an interface to those other programs or packages.

Often even in these cases it is useful to have another more conventional behavior that maintains state determined by the external behavior. For example, for our monkey's face recognition, we might use a commercial package that returns the identity of the individual as a vector. We might also have a coded object behavior that learns from experience to categorize these vectors into friend, enemy, familiar neutral or unfamiliar.



1.3 Say When

In BOD, **when** is controlled using structures that are read by a special behavior for action selection. In AI, structures that control action selection are generally called *plans*. BOD uses hand-coded, flexible plan structures. Such plans are often called *reactive plans*, because with them the agent can react immediately (without thinking) to any given situation.

1.3.1 The Simplest Plan

The simplest plan is just a list of instructions, for example:

$$\langle \text{get a banana} \rightarrow \text{peel a banana} \rightarrow \text{eat a banana} \rangle \tag{1.1}$$

Such a list is called a simple sequence, or sometimes an *action pattern*.

1.3.2 Conditionality

Of course, specifying the complete behavior for the entire lifetime of your monkey in a sequence would be tedious. (Besides, it's provably impossible.) A more common way to specify **when** is to associate a particular *context* which the agent can perceive with a **what.** Such a pairing is often called a *production rule*. The context is called the rule's *precondition* and the **what** is called its *action*.

For example, the plan 1.1 could be changed into a set of rules:

$$\begin{aligned} \text{(have hunger)} &\Rightarrow \text{get a banana} \\ \text{(have a banana)} &\Rightarrow \text{peel a banana} \\ \text{(have a peeled banana)} &\Rightarrow \text{eat a banana} \end{aligned} \tag{1.2}$$

I have put the contents of the precondition in parenthesis to indicate they are really a question. If the question is answered 'yes', then the rule should fire — the action should be executed.

It might look like our new plan is as good as or better than our old plan. For one thing, we've specified something new and critical — **when** to execute the plan itself. For another, if somebody hands our monkey a peeled banana, she will be able to execute the rule 'eat a banana' without executing the whole sequence in plan 1.1.

Unfortunately, it's not that easy. What if we had another sequence we wanted our monkey to know how to do. Let's say that we intend to have our monkey to a dinner party, and we want her to be able to pass bananas to other guests[1]. Here's the original sequence:

$$\langle \text{get a banana from left} \rightarrow \text{pass a banana to right} \rangle \tag{1.3}$$

But if we translate that into rules:

$$\begin{aligned} \text{(left neighbor offers banana)} &\Rightarrow \text{get a banana from left} \\ \text{(have a banana)} &\Rightarrow \text{pass a banana to right} \end{aligned} \tag{1.4}$$

Now we have two rules that operate in the same context, 'have a banana'. What should our monkey do?

We could try to help the monkey by adding another piece of context, or *precondition*, to each of the rules. For example, *all* of the rules in plan 1.2 could include the precondition 'have hunger', and all the rules in the plan 1.4 could have the condition 'at a party'. But what if our monkey is at a party *and* she's hungry? Poor monkey!

The problem for the programmer is worse than for the monkey. If we want to determine what the monkey will do, we might have to add an exception to rules we've already written. Assuming we think being polite is more important than eating, when we begin writing our party rules, we'll have to go back and fix plan 1.2 to include 'not at a party'. Or, we might have to fix the behavior that runs the monkey's rules to know that party rules have higher priority than eating rules. But what if we want our monkey to eventually eat at the party?

Overall, although the production rule structure is powerful and useful, so is the sequence. Our monkey should keep the steps of plan 1.3 together in a sequence, so that if she takes a banana from the left, she knows to try to pass it to the right. In any other circumstance, if she has a banana, she never needs to think of passing it.

1.3.3 Basic Reactive Plans

To summarize the previous section:

[1] Don't try this with real monkeys.

- Production rules are useful because they facilitate flexible behavior and the tying of action to context. However, they rapidly become difficult to manage because of the amount of context needed to differentiate rule firing.

- Sequences work well because they carry that context with them. A **what** embedded in a sequence carries disambiguating information about things that occurred just before or just after. The sequence itself represents an implicit *decision* made by the monkey which disambiguates the monkey's policy for some time.

We can combine many of the attributes of these two features using another structure, the *Basic Reactive Plan* or BRP. Let's try to rewrite plan 1.1/1.2 again:

$$\text{(have hunger)} \Rightarrow \uparrow \left\langle \begin{array}{r} \text{(full)} \Rightarrow \textit{goal} \\ \text{(have a peeled banana)} \Rightarrow \text{eat a banana} \\ \text{(have a banana)} \Rightarrow \text{peel a banana} \\ \Rightarrow \text{get a banana} \end{array} \right\rangle \tag{1.5}$$

What this notation indicates is that rules relevant to a particular activity have been clustered into a BRP. The BRP, like a sequence, limits attention to a small, fixed set of behaviors. It also encodes an ordering, but this time not a strict temporal one. Instead, it records a *prioritization*. Priority increases in the direction of the arrow. If the monkey already has a peeled banana, she'll eat it. If she has a whole banana, she'll peel it. Otherwise, she'll try to get a banana.

Notice the last rule doesn't need a precondition: instead, it's guarded by its low priority. It will only fire when the monkey's action selection attention is in the context of this BRP, and none of the other rules can fire. Also, notice that the the plan has a special rule called *goal* for detecting when its task is finished. A BRP ends if either none of its rules can fire, or if it achieves its goal (if it has one.)

This is a much more powerful structure than a simple sequence. If the monkey eats her banana, and she still isn't full, she'll get another one!

1.4 Say What

Now that we have at least a rough idea of how to encode **how** and **when**, we can get back to the critical question of saying **what**. Remember, **what** is the question that determines how hard it is to program **how** and **when**. A good configuration minimizes the complexity of both.

Determining **what** is determining the *interface* between **how** and **when**. We need a list of primitive concepts which will occur in the reactive plans, and we need to know how to express them as methods on the behavior objects.

1.4.1 Initial Decomposition

The process of Behavior Oriented Design (BOD) is split into two parts: the initial decomposition, and the development cycle. The initial decomposition doesn't have to be done

very well, because the development cycle will revise decomposition. However, getting off to a good start always helps make things come together faster.

Here are the steps of the initial decomposition:

1. Come up with a list in gross terms of things you want your monkey to do.
2. Describe your monkey's likely behavior as sequences of actions. These sequences will be turned into the reactive plans.
3. From the sequences, make a list of the different actions. You may also want to think of preconditions (sensing primitives) that might help with this list.
4. Figure out what your monkey needs to know and remember in order for the actions and preconditions to work. Group the action and sensing primitives you have listed together if they will be using the same state (knowledge and memories). This is the basis of the behavior library.
5. Choose a first plan and a first behavior to start implementing.

1.4.2 The Development Cycle

From a high level, there are just four things involved in really making your monkey smart:

- coding behaviors,
- coding reactive plans,
- testing and debugging code, and
- revising the specifications made in the initial phase.

These should be iterated over repeatedly and frequently. When you get a working reactive plan, comment it with the date, the author, what it does, and the version of the libraries it works with. Then save it. If you make significant changes to your behavior library, then rerun all the old reactive plans as a test suite to see if things still work.

Of course, coding is never really easy. That's why it's important to make it as easy to debug as possible. For debugging the behaviors, I strongly recommend doing your coding in an environment with an excellent debugger. I've also designed tools to help you execute and debug the reactive plans.

1.5 Revising the Specification

The item I'll go into some detail on is *revising the specification*. There are some tricks to this that are specific to developing things under BOD. This section just covers the basics. There are more details in section 1.7.4

The main design principle of BOD is *when in doubt, favor simplicity.* All other things being equal:

- It's better to use a sequence than to use a BRP.
- It's better to use a single primitive than to use a sequence.
- It's better to use control state than variable state in a behavior.

Now, if that's the guiding principle, the question is, when do you violate it?

- Use a BRP when some elements of your sequence either often have to be repeated, or often can be skipped.
- Use a sequence instead of one primitive if you want to reuse part, but not all, of a primitive in another plan.
- Add variables to a behavior if control state is unnecessarily redundant, or has too complicated of triggers.

We've already talked about the first rule a bit, in sections 1.3.2 and 1.3.3. The second rule is a basic principle of software engineering: *Never code the same thing twice — Make a generic function instead.* There's a simple reason for this. It's hard enough to get code written correctly and fully debugged once. Doing it again is asking for trouble.

The third rule is the principle of reactive intelligence, and the third heuristic helps explain why you don't want a fully reactive agent.

The important thing is that BOD specifications are made to be changed, as the programmer learns more about the problem, and as the agent gets more complex. Let's do another example. Consider the primitive 'get a banana' we used in plan 1.5. **How** does the monkey get a banana? Lets suppose we've coded the monkey to go to the kitchen, climb on the counter, and look in the fruit bowl. If there's a bunch, she should break one off; if there's a loose one, she should take it; if there's none, she should throw a fit.

Clearly, a good deal of this could be coded either as a plan, or as a behavior. The principle of BOD is that it should be a behavior, *until or unless* you (as the programmer) could be using some of those same pieces again. So, for example, if you next decide you want your monkey to be able to get you a glass of water, you now have a motivation to write two plans:

$$\text{'get a banana'} \Rightarrow \langle \text{go to the kitchen} \rightarrow \text{take a banana} \rangle \tag{1.6}$$

$$\text{'get glass of water'} \Rightarrow \langle \text{go to the kitchen} \rightarrow \text{pour glass of water} \rangle \tag{1.7}$$

Notice that these plans are now guarded with not a question, but an element of a plan, a **what**. We have changed a particular **what** (get a banana) from being a simple method on a behavior to being a sequence. But we *don't* need to change our old plan (1.5).

1.6 Making a Complete Agent

Plans that contain elements that are themselves plans are called *hierarchical*. The natural questions about a hierarchy are "where does it start?" and "where does it end?" We already know that the plan hierarchies end in behaviors, in the specification of **how**.

The start could in principle be any plan. Once that plan ends, the agent's intelligence is just over. This makes sense for certain kinds of software agents that might be called into existence just to perform a certain job. But what if we are interested in making something like a monkey? Something that lasts for some time, that decides what to do based on a set of motivations and principles?

We call this sort of agent a *complete agent*. And at the start (or *root*) of its plan hierarchy, we put a BRP specially designed never to end. This special designing just involves blocking the two ways BRPs can end. First, the BRP has no goal, so it never 'succeeds' and completes. Second, the BRP must have at least one element that can always run, so it never fails.

Here's a BRP that might govern the monkey we've been building:

$$\text{'life'} \Rightarrow \uparrow \left\langle\left\langle \begin{array}{r} \text{(at a party)(obligations exist)} \Rightarrow \text{be polite} \\ \text{(hungry)} \Rightarrow \text{eat a banana} \\ \text{(friends around)} \Rightarrow \text{make friends comfortable} \\ \Rightarrow \text{wait} \end{array} \right\rangle\right\rangle \tag{1.8}$$

Notice I added a sense for noticing obligations. That way, our monkey can eat even when she's at a party, so long as she's not aware of any social obligations. I didn't specify a goal, and I included a low-priority behavior that can always run, so the act 'life' should never end.

Describing and ordering the motivations of a complete agent is actually part of the specification process that is integral to BOD. I skipped over it before. But here's the complete list of things that need to be specified and revised over the course of building a complete agent:

1. A high level description of what your agent does.
2. Collections of actions (in sequences and BRPs) that perform the functions the agent needs to be able to do. *The Reactive Plans*
3. The list of **what**s (including questions / senses) that occur in reactive plan. *The Primitive Interface*.
4. The objects collecting state needed by the primitives, and the program code for acquiring and using that state. *The Behavior Library*.
5. A prioritized list of high-level goals the agent will need to attend to. *The Drive Collection*.

Drive collection is a special name for this top / root BRP.

1.7 Things I've Brushed Over

1.7.1 How do BRPs Really Work?

BRPs begin when they've been triggered, and end when either no element can execute, or if a special element called a 'goal' meets its precondition. While they are operating, on any

cycle the highest priority element that can execute, will execute.

Under this arrangement, the plan might run forever. Maybe this is what you want. If it isn't, there are two ways to stop it. First, you can specify a maximum number of times any particular element can execute. Second, you can specify a time-out for the entire BRP.

Can a BRP call itself, or another plan that calls it? — Yes. This is because no stack is saved when a BRP passes control to one of its children. This is called a Slip-Stack Hierarchy, and it allows for cycles in reactive plans.

1.7.2 How do Drive Collections Really Work?

Drive collections are a little more than just BRPs. They are used to monitor the environment for significant events that might change which reactive plan the agent is involved in. For example, our monkey could be in the middle of eating a banana, but if a surprise party were suddenly thrown, she'd immediately be polite (as soon as she realized she was in a party and had obligations!)

This isn't true of all plans — usually if the monkey is attending to one plan, she won't notice that she could be operating another one. This is an important characteristic of intelligence, *persistence.* It normally makes sense from an efficiency standpoint to finish something that's been started. Also, it's impossible to evaluate all possible plans at all possible times to choose the best next action: no agent has a brain big enough to do that.

Besides operating continuously, drive collections also keep track of the roots of the plan hierarchy. So if a part of a plan ends, the drive knows how to start it up again. For example, if the hunger drive is triggered, it will pass control to plan 1.5. But if the monkey doesn't have a banana, plan 1.5 passes control to plan 1.7. The drive collection is what keeps track of this passing of control, and then when 1.7 is done, the drive collection knows to start at the beginning of the hunger plan hierarchy again.

The drive collection can also be used to *schedule* drives that are important to be done occasionally, but should normally allow lower-priority drives to occur.

1.7.3 How Do 'Questions', Preconditions and Sequences Really Work?

Preconditions are really sequences (except they are executed atomically, so shouldn't contain any elements that take any time). The individual elements of preconditions are actually *perceptual primitives*, which like **what**s, are methods to behaviors. Remember, behaviors handle perception.

Precondition sequences can actually also include actions, if they are very quick. For example, if a monkey is looking for a red block on a blue block, its goal precondition might look like:

$$\langle(\text{see a blue block}) \rightarrow \text{look up a bit} \rightarrow (\text{see a red block})\rangle \tag{1.9}$$

Questions in preconditions can actually return more values than just true or false, and can actually be checked for different tests than just 'is true' or 'is equal'. But they hardly ever are. The precondition above would be more likely coded as:

$$\langle(\text{see-a-block 'blue}) \rightarrow \text{look up a bit} \rightarrow (\text{see-a-block 'red})\rangle \qquad (1.10)$$

The entire precondition object (inside the parenthesis) must return true or false. If it (or any action) returns false, a sequence aborts and returns 'false'.

1.7.4 Other Rules for Revising the Specification

- Use deictic representation — a little bit of state in a behavior replaces a lot of reactive plans. For example, you might have a piece of state called 'visual attention' that refers to what your monkey is looking at. Then you can follow either 'look at banana' or 'look at apple' with 'grasp what you see'.
- Any and all memory goes into a behavior. Memory might be deictic as above, symbolic like being able to remember names, or complex and statistical, like a Markov model learned to recognize speech or control limbs. It still belongs in a behavior.
- Behaviors are modular, but not encapsulated. **What**s are for communicating **when** something should happen, they communicate very little else. If, for example, a decision-making behavior needs reference to historical interactions stored in an episodic memory behavior, it should get them directly. See also the perception example in section 1.2.5.
- Building BRPs can be non-intuitive to conventional programmers. Here are some simple guidelines:
  - A BRP is essentially a backwards sequence — the highest priority event is recognizing that the goal is completed, the next highest priority thing is that which would get consummate the goal, and so on. Compare sequence 1.1 to BRP 1.5.
  - A BRP seldom needs more than five elements.
  - If a BRP is getting too confusing, then you probably want to create another BRP either as a sibling (two different solutions to the same problem) or a child (another BRP that can sometimes be called by this BRP).
  - You shouldn't need a long list of preconditions / 'questions' for any step of a BRP. If you find yourself using them, then either you want another BRP (as above) or you need to create a new perceptual primitive, and let a behavior handle the complexity. Behaviors control both action and perception.
- And finally, here's some suggestions for knowing when different kinds of memory go in different behaviors:
  - The original sensory signal is very different — like light vs. noise.
  - The memory needs to record events that occur at different rates — like words vs. conversations.

- The memory needs to decay (be forgotten) at different rates — like the exact pixel representation of a friend you just saw, vs. the fact that you saw them.
- Patterns from the memory emerge at different rates — like learning a new face vs. learning arithmetic.
- Different representations are most simple or minimal.