The Inverse Monkey Rule

This is an excerpt from the book Humans vs Computers, with a summary of all the key heuristics. Check out the full book at https://gojko.net/books/humans-vs-computers/

Émile Borel proposed the famous infinite monkeys theorem in 1913, suggesting that given infinite time and attempts, monkeys would come up with the works of Shakespeare. Borel's theorem is a nice illustration of statistics and calculus, but in practice the probability is infinitely small. On the other hand, inverting the subjects and the outcome gives us something a lot more practical, in much less time: *the Inverse Monkey Rule*.

Smart people, hitting keys intentionally on a computer keyboard, given just a few months, will almost surely produce some kind of monkey crap.

An important consequence of the Inverse Monkey Rule is that perfect software is impossible. Without an infinite amount of time and infinite knowledge, people will always make mistakes. But we don't have to repeat errors that are predictable.

This part will hopefully help you avoid the mistakes that other people made to deserve a mention in this book. It contains a set of ideas on how to avoid similar problems, with heuristics for analysing, developing and testing computer systems. Please note that this isn't a comprehensive list of test cases or a full testing strategy, just a quick digest of the stories in this book. Use it as a check-list along with your other tools, and as an inspiration when looking for more ideas.

Personal names

Personal identifiers such as names are a crucial piece of our identities. Many software problems result from a fundamental conflict between the two key aspects of names. On one hand, names are personal, so they carry a lot of cultural and family heritage, tracing back to a time long before computers. Specific spellings, accents and parts of names have meaning and can't just be simplified or changed to make processing easier. On the other hand, useful computer identifiers need to be standardised and easy to store and process. In many cases, various software systems need to agree on someone's identity. All those systems were designed by different people, working under specific constraints, making their own assumptions. That's why small inconsistencies and bugs in handling names in one component can easily create a mess in collaborating systems.

Here are some often-overlooked oddities of personal names, which you should remember when designing software:

Single-letter names aren't always initials, so it's bad practice to use length checks to prevent people from entering initials (*Stephen O*, *A Martinez*, *O Rissei*).

There's no universally acceptable standard for maximum name length. The International Civil Aviation Organization (ICAO) allows up to 64 characters per name. Many governments today limit registered baby names to those that fit on a passport, which may be up to 40 letters. Of course, different governments have different standards. Also, people born before machine readable passports weren't subject to that restriction. Some names can get very long (e.g. *Christodoulopoulos*, *Srinivasaraghavan*, *StopFortnumAndMasonFoieGras*, *Wolfeschlegelsteinhausenbergerdorff*, *Rhoshandiatellyneshiaunneveshenk*, *Keihanaikukauakahihulihe'ekahaunaele*. Double-barrelled surnames can also get quite long (e.g. *Plunkett-Ernle-Erle-Drax*).

Names aren't permanent, and in most countries people can easily change their names as many times as they want.

Names don't just consist of letters. They can contain accents and apostrophes (*O'Stephen*, *Keihanaikukauakahihulihe'ekahaunaele*), dots (*GoVeg.com*), dashes (*Thurman-Busson*), numbers (*Number 16 Bus Shelter*, *Jon Blake Cusack 2.0*) and probably some other classes of symbols. It's best not to assume any specific character set for validity checks.

People don't always have a given name and a surname. Some people are mononymic – they have only a single name (e.g. *They*, *Teller*, *Naqibullah*). It's best not to ask for first and last name separately. When communicating with external systems, make sure you can handle cases in which one of those two fields is missing.

There's no universally accepted standard for working with mononymic names. Many government systems require first and last names to be recorded separately, and some will set mononymic names as the given name, some as the surname. Some use the mononymic name for both fields (*Neli Neli*). When matching names against external sources, consider that the sources might be using different schemes for single names. Some countries use markers such as FNU, LNU or XXX for the other name when recording mononymic people. Detect those markers and consider them when matching or validating external records, so you don't end up interpreting them as given names (*No Name Given Sandhya*). But don't assume these are always markers (someone can theoretically change their name to *XXX*).

People don't always have just one or two given names and surnames. *Tracy Nelson* has 138 middle names. A nice example is *Rosalind Arusha Arkadina Altalune Florence Thurman-Busson*. For a good edge case, remember James Dr No From Russia with Love Goldfinger Thunderball You Only Live Twice On Her Majesty's Secret Service Diamonds Are Forever Live and Let Die The Man with the Golden Gun The Spy Who Loved Me Moonraker For Your Eyes Only Octopussy A View to a Kill The Living Daylights Licence to Kill Golden Eye Tomorrow Never Dies The World Is Not Enough Die Another Day Casino Royale Bond.

Null isn't just a computer kill word, it's also a perfectly valid name.

Test, Sample and many other common words are also valid names. Just because a user's surname is Test doesn't mean that it's actually a test account. When testing, avoid using specific names to mark example data, because real users might get caught by this as well.

Fictional character names aren't necessarily always fake (*Superman Wheaton*, *Buzz Lightyear*, *Darth Vader*). Names that are also those of popular brands aren't always fake either (*Facebook Jamal Ibrahim*, *Google Kai*). Common English (or any other language) words or phrases in a name don't necessarily make it fake (*Elaine Yellow Horse*, *Above Znoneofthe*).

Time

Time is a quite a tricky subject for software. Most people have an established intuitive perception about time, so it's easy to oversimplify and overlook edge cases. In addition, although the concept of time is simple, there are at least three distinct versions of it, and they aren't always synchronised.

Cosmic time is passing in the real world without any care for humans. It's governed by the laws of physics. Although Einstein famously declared it to be relative, for most purposes it's the same everywhere on planet Earth. It's also continuous and, except in bad science fiction, always moves in a single direction.

Elapsed time deals with periods between two reference points in cosmic time. This is the time we can measure, and deals with periods such as seconds. Elapsed time doesn't have any notion of midnight, summer or Tuesday next week. This type of time is a human invention, but apart from someone choosing the two reference points for measurement, it doesn't depend on humans. Instead, it depends on the recording machinery. For computers today, this mostly means that elapsed time isn't continuous, but discrete, in increments of milliseconds. Theoretically, it should be the same everywhere on Earth, but practically it's not. Measuring devices use different precision and accuracy. Your computer and your mobile phone may measure the same period differently, and they both diverge slightly from NIST-F1, the atomic clock that controls the official time in the USA. Lastly, elapsed time isn't infinite. It's subject to the capacity of measurement equipment, which is why many older computers can't see beyond 2038.

Clock time is the one used for calendars, to guide our daily lives, schedule meetings and keep society synchronised. Clock time deals with concepts such as 14:45pm, wake-up alarms, and beer o'clock. It's different in different places of the world, driven by solar cycles, the Earth's rotation, and the needs of the communities living in a particular area. It's a uniquely human thing, subject to politics, government conventions and manipulation. It can jump ahead, move backwards, stall or stretch.

Most of the time, excuse the pun, the three types are the same for all practical purposes. But the problems start when they suddenly diverge, even if only for a moment. Here are some commonly overlooked quirks of time that cause problems:

Days aren't always exactly 86,400 seconds long. Leap seconds can make a day one second longer. (*31 December 2016* had a leap second.)

Coordinated Universal Time (UTC) and Greenwich Mean Time (GMT) time zones aren't always the same. They can drift by up to 0.9 seconds. That's why leap seconds exist.

Leap seconds don’t necessarily have to be positive. If the Earth’s rotation required it, it would theoretically be possible to introduce a negative leap second. This has never happened so far – but once it does, better stay home that day.

Years aren’t always 365 days long. Remember the occasional 29 February.

Leap years don’t happen every four years. Three out of four end-of-century years don’t have a leap day.

Clock time doesn’t always go forwards. Daylight saving can make it jump backwards.

Computers couldn’t care less about clock time, they only deal with elapsed time. Applying clock time arithmetic to elapsed time often leads to problems. For example, adding one month to the current time doesn’t produce a period of exactly one month in clock time. Time zones, different numbers of days in a month and other exceptions can cause wrong calculations.

Scheduling future events by using elapsed time is dangerous. For example, ‘same time tomorrow’ isn’t always 24 hours away. Daylight saving can shift the clock. People might travel into a different time zone. The longer the period, the more chance of a mess.

Clock time and elapsed time don’t always move by the same amount. Daylight saving can create big gaps.

Daylight saving time isn’t applied consistently across all countries, or even within a single country. For example, most of the USA observes daylight saving time, but Hawaii does not. In Australia, New South Wales observes daylight saving time, but Queensland does not.

The daylight saving time schedule isn’t fixed. It’s a political agreement, subject to change. For example, Israel synchronised time zone changes with the east of Europe in 2013, moving the end of summer time from early September to late October.

Elapsed time isn’t always positive. Most computers represent dates before 1970 as negative numbers.

Unlike cosmic and clock time, elapsed time isn’t infinite. Check the numeric limits of your date records, and test around those. For example, for typical 32-bit dates, test for dates before 1970 and after February 2038.

Missing or invalid time might mask itself as 1 January 1970 or 31 December 1969, especially if you’re using third-party components outside your control.

Addresses

Postal addresses are an important link between the virtual realm of the Internet and the physical world. Apart from the obvious role in shipping the stuff people buy online, knowing users’ addresses is also critical for calculating delivery prices, correctly accounting for tax, and applying territory-specific limitations.

But postal addresses often play three more roles in software, which they were never intended for. With the lack of globally unique personal identifiers, addresses are also used to distinguish between two people with the same name, especially in countries that don’t have mandatory ID cards, such as the UK. Addresses also often serve as an additional piece of personal identification to match records from different systems, for example when banks check credit ratings. And parts of addresses, such as zip or post codes, are increasingly used as semi-secret information to prevent fraud, for example when verifying online credit card transactions.

For hundreds of years, postal delivery processes evolved to deal with inconsistent and incomplete addresses, but the new digital roles for address information require exact, precise and uniform data. Similar to names, the different conflicting roles of addresses create plenty of opportunities for software bugs.

Here are some often-overlooked facts that cause problems when handling addresses in software:

ZIP codes or post codes aren’t mandatory. Some countries don’t use post codes (*Fiji*, *UAE*).

Post code formats aren’t permanent, they change over time. For example, Singapore used two digits in the ’60s, four digits in the ’80s, and now uses six digits. Older records with post codes might use different formats from the current ones.

Some countries started using post codes relatively recently. For example, post codes were introduced in Ireland in 2014. For such cases, even though current addresses might have post codes, slightly older address records might not have that information.

Post codes aren’t always consistently used, even within a single country. For example, Jamaica doesn’t use post codes (the country tried to, but the system was suspended in 2007), but there are two-digit area codes for the capital, Kingston. China uses post codes, but Hong Kong does not.

There’s no universally agreed length for post codes. For example, Austria and Switzerland use four-digit codes. The Faroe Islands use three-digit codes. Iran uses up to ten.

Post codes aren’t just numeric; many countries use alphanumeric post codes. For example, *EC11AA* is a valid UK post code.

Post codes can contain spaces. For example, *EC1 1AA* is a common way of writing a post code in the UK.

Having the same or similar post codes doesn’t necessarily imply physical proximity. For example, rural codes in New Zealand can be far apart.

Post codes aren’t always the same in a city or area. In the UK, post codes are allocated to estates, blocks, buildings or even individual houses.

There’s no universally agreed minimum or maximum length for location names, including for street names or city names. For example, *Llanfairpwllgwyngyllgogerychwyrndrobwllllantysiliogogogoch* is a place in Wales, *Y* is a place in France. There are six villages called *Å* in Norway.

IP addresses aren't a reliable link to a physical location. Although many home broadband subscribers now effectively have an allocated IP address, there are too many exceptions and ways to spoof this information.

Out of all the categories of problems, real-world rules around addresses seem to change the most frequently at the moment. The examples and edge cases listed above were correct when I wrote this in 2017, but do check.

Numbers

Although modern software applications can appear to be processing text or displaying dates or emojis, in fact computers only deal with numbers. Bad assumptions about the mapping between numbers and contextual information such as shopping cart quantities often lead to trouble. Some values, such as a pack of cigarettes costing $23,148,855,308,184,500, are obviously wrong to humans, but to a computer that's just a number like any other. Some perfectly valid numerical values, such as 0, might make no sense when used as contextual information, such as characters on screen. Some contextual information requires a particular sequence of numbers, such as Unicode combinations, but computers will happily store and transmit invalid numerical sequences.

Here are some ideas around amounts and quantities to remember when developing software:

Don't assume you can apply trivial mathematical rounding to fractional amounts. There are specific rounding and truncation rules for financial information, and they vary by country.

Not all currencies use two decimals. For example, *Japanese yen* don't use decimal fractions. *Kuwaiti dinar* have three decimals.

Try amounts with and without decimal places, and with varying number of decimals.

There's no universally agreed standard for writing currency amounts (or if there is, normal people don't obey it). Expect users to input currency amounts inconsistently. To a person, 5000, $5,000, $5 000 and $5,000.00 mean the same thing. If you want consistent information, make sure to check for a particular format.

People in different countries use different separators for thousands and decimals. The US number 1,234.56 would be 1 234,56 in France. Don't just remove all the commas before turning a string into a number.

Try negative values where they're not expected (*–1 books*).

Don't assume quantities always need to be positive. In some business domains, it's perfectly acceptable to have a negative quantity (for example, to mark items returned by customers).

Avoid using special values to mark missing information (such as *0* or *No Plates*), as this can be interpreted as the actual value by someone else.

Avoid marking test data with special values (such as having a surname *Test*), as this can easily create false positives. It's best to have specially identified accounts for testing, which you can later clean up.

Explore rounding strategies, especially with things that accumulate over time. Small rounding errors can create a big mess.

With Unicode, memory length and screen length aren't necessarily related. Some Unicode symbols are very long ((*0xFDFD*), some are invisible (*0x200B*), and some combine with previous characters (*0x0597*).

Check how the contextual data gets recorded and test around the corresponding numerical boundaries. For example, test what happens when the timer reaches the end of a 32-bit number range.

The number 0 is often interpreted as false information or missing data, or is just not expected in mapping to contextual values. Test what happens when your data maps to 0. Remember the 787 Dreamliner engines that would shut down when the control timer reached zero.

Process automation

Computers excel at doing things fast, but there's a general trend of trusting them too much to do their work well. Small errors can pass undetected for a long time, accumulate and build up a time-bomb. Perhaps even worse, pointing the computer in a wrong direction and letting it run off can cause small oversights to quickly escalate into a major blunder.

Lots of things can cause bad automation, even with the best intentions of people building the software. Third-party systems can send invalid, unexpected data. Migrating a legacy database may uncover lots of unforeseen edge cases. One part of the system can decide to go rogue and disrupt everything around it.

Apart from having a crystal ball that can see into the future, the best way to stop bad automation is to create an automated system of oversight. Build up monitoring and alerting mechanisms that can spot when something out of the ordinary is happening, and get people in to investigate before it's too late.

Here are some ideas on keeping automation in check:

Testing with small samples often doesn't uncover all the data-driven issues of large legacy databases. If you're converting a legacy database, run some basic characterisation statistics on the converted data and check with the domain experts whether things look all right. Remember the *Grand Rapids hospital* update that declared 5% of the population dead overnight.

If your system is automatically processing financial transactions, put monitors in place to check for trends. Good candidates are the expected volume of fraud or number of purchases per hour. If things fall too far outside the expected range, alert a person – even if things look as if they're in your favour. Remember the *610,000 Japanese yen* fat-finger error and *MiDAS* fiasco.

If your system is automatically changing some data, such as prices, put monitors in place to check that automated changes are inside a valid range. For example, alert a person if the price goes too low or too high. This will help you avoid cases such as the *28,639.14 Uber ride*, or Repricer Express *selling everything at $0.01*.

Put monitors in place to check whether one of your systems is behaving significantly differently from the rest. For example, if a single trading processor is running 90% of the volume, get someone to investigate why before it's too late. Remember how one of eight *Knight Capital SMARS* systems ran a previous version of the software and it almost bankrupted the company.

Consider that speeding up a single part of a process might create problems downstream. For example, increasing the capacity to send out customer notifications can overload your call centre and create more problems than it solves, such as in the *Centrelink robo-debt fiasco*.

If you're generating random outcomes and they need to fall within some expected business rules, make sure to check those rules before you publish the results. Random things are just that – random – and, in some cases, might be surprising. It's potentially better to alert a person, or even to crash the system, than to directly use such unexpected values. Remember the *Pepsi 349 lottery*.

If you ever use sample data to validate or monitor your software, make sure that your tests are clearly identified, isolated and don't end up matching any real-world cases. Remember *Jeff Sample* and the *50 police raids on Walter Martin's house*.

Biometric matching isn't magic, and biometry isn't necessarily unique. Unrelated people do look alike. Twins can trick smart photo algorithms or leave a similar voice signature; remember the *Kennedy sisters*.

Monitor whether third-party systems are sending you strange data. For example, check whether some values appear a lot more frequently than the others. This will help to identify special markers for missing or invalid records, in particular where blank values aren't allowed (remember the *NO PLATE* parking tickets). Make sure to check third-party data for more than one entity where you expect only one (remember *concurrent criminal sentences*). Check whether data is out of the usual range (for example, a *payment request for $23,148,855,308,184,500*).

If you're sending important messages through a third-party system, don't just trust that the notifications are dispatched. Build a mechanism requiring recipients to confirm that the messages have actually been delivered. Not everyone will confirm, of course, but you will at least be able to monitor trends and see if something unexpected happens, such as 50,000 people mistakenly dropping off the system. Remember the *Queensland OneSchool police e-mails*.

If you're working on a system that's supposed to work unattended and autonomously, leave it running for a long period of time and check whether gremlins appear. And do consider shutting the whole system down if it is mission critical and loses the ability to control itself.

Whenever you're using a slowly depleting but limited resource, make sure to build in monitoring, and send alerts when it starts getting dangerously low. For example, if you're using a count-down timer, notify someone to restart it before it gets to zero. Don't just rely on a published procedure for people to follow, because they might forget or have higher priorities at the time when things become critical.

If you're using any kind of hard-coded accounts for development and testing, make sure they don't somehow find their way into production software. Remember *five blanks granting Xbox access*.

As Porky Pig would say, 'That's all folks.' I hope these examples tickled your imagination, and that they'll inspire you to improve how you design, test and build software systems. If you'd like to dive into any of the stories mentioned in the book further, check out the articles and references on in following appendix.