Notes - Organizational Development - Testing and Quality - Part 2

- Dr Nick Hayward

A brief introduction to testing and quality with example usage for software projects.

Contents

Intro

Testing, and quality assurance in general, is often closely associated with efficient and correct use of version control.

Initially, we might consider testing as a simple matter of ensuring that developed software, and its underlying code, executes and performs as expected producing the defined output and results.

However, testing and quality assurance may also encompass many disparate parts to achieve this initial simple overarching goal.

For example, we might consider different specific testing options, including manual and automated testing, integration with CI/CD pipelines, its usage with version control, and its role in a product's final release and subsequent maintenance.

We might also consider testing from the perspective of UI and UX requirements, playtesting, and other various options.

UI/UX testing

A good example of a commonly manual form of testing is initial UI and UX testing, in particular relative to actions and tasks for achieving a set goal within a given project and UI based application.

As designers, developers, and builders we are, of course, interested in how we can quantify the cognitive load involved and required for performing a given task. A better understanding of some of the load issues within our application and interface helps guide us in apportioning emphasis and control throughout our design.

So, for a particular task, we could compile a list of the required actions, operations, or steps that one of our users might require for that task under normal circumstances. We might then estimate or assign a percentage, or some other arbitrary value, which represents our understanding of the effort involved for an individual action. Then, we'd total all of the action scores to assign an overall score for the effort for the selected task. We could then evaluate different design options and alternatives by comparing these collated scores.

The *Keystroke-Level Model for the Goals, Operators, Methods, and Selection Rules* analysis approach, or KLM-GOMS model, (*Card et al, 1983*), is one example of an analysis technique based on this concept. However, instead of assigning scores representing effort, an estimate of the time required for each action is assigned instead. The time required to complete a task is considered a good proxy for physical effort. However, one notable issue is that it doesn't accurately measure the intensity of expended mental effort.

KLM-GOMS model KLM-GOMS (Keystroke-Level Model) is a quantitative modeling tool used in UI and UX testing to predict how long it will take disparate sets of users to complete a specific task with no errors. It predicts task times based on a simple set of physical and mental operators including keystrokes, button clicks, pointer movement, keyboard to mouse movement, and thinking time.

So, a user will accomplish a goal by dividing it into a series of tasks. For each task grouping, our user will take a moment to build their mental representation and then choose an appropriate strategy for accomplishing the task at hand. This preparation is called the *task acquisition time*. Naturally, it can vary from very short periods for simple, routine tasks, to much longer periods, perhaps a few minutes, for more creative, original or thought provoking tasks.

Following this *task acquisition*, our user will then continue with their chosen task using a sequence of actions or operations. The total required time to complete the actions is called the *task execution time*. Therefore, the total time required by our user to complete a chosen task will be the sum of the initial *task acquisition time* and the *task execution time*.

n.b. There are more recent examples of modified models for mobile devices, such as phones, but they are often derived examples as parallels to the original Keystroke-level model.

KLM-GOMS usage To reach an approximate estimation for the task execution time, KLM-GOMS defines basic operations as follows.

n.b. we'll stick with a simple assumption of our system using a keyboard and mouse option.

Code	Operation	Time (in seconds)
K	Key press & release (keyboard)	Best Typist (135wpm) = 0.08
		Good Typist (90 wpm) = 0.12
		Avg. Skilled Typist (55 wpm) = 0.20
		Poor Typist (40 wpm) = 0.28
		Typing Random Letters = 0.50
		Typing Complex Codes = 0.75
		Worst Typist = 1.20
P	Point mouse to an object on screen	1.10
B	Button press or release (mouse)	0.10
H	Hand from keyboard to mouse & vice-versa	0.40
M	Mental preparation (operation)	1.20
T(n)	Type string of characters	n x K seconds

wpm = words per minute

Source: Kieras, D. 1993. Wikipedia

So, let us consider an example implementation for the KLM-GOMS model. If we asked a user to find all of the *v* characters in a text document, we might might consider it as follows.

For example, let us conceptually open this document within a basic text editor.

- move mouse to **search** menu (requires hand to mouse, mental preparation, and point mouse to an object on the screen)
- select the **search** menu from the editor's main menu (requires button press and release of the mouse)
- click on the **find text** link in the **search** menu (requires mental preparation and point mouse to an object on the screen, button press and release of the mouse, and hand from mouse to keyboard)
- enter the required search term, for example **v** (requires key press and release for two characters…)
- click the **OK** or **search** button (requires hand from keyboard to mouse, mental preparation and point mouse to an object on the screen, and a click and release of the mouse)

This sequence can be encoded and expressed using the KLM-GOMS model, thereby allowing us to estimate the required average time for such a task.

Our result might resemble the following table.

Action	KLM-GOMS Code	Time (in seconds)
move mouse to **search** menu	H (hand to mouse)	0.40
	M + P (search menu)	1.20 + 1.10
select **search** menu…	BB (select search menu)	2 * 0.10
click on —find text— link…	M + P (find text menu item)	1.20 + 1.10
	BB (select menu item)	2 * 0.10
	H (hand from mouse to keyboard)	0.40
enter search term —et—	KK (type —et— characters)	2 * 0.20 (avg. typist)
click the —OK— button	H (hand from keyboard to mouse)	0.40
	M + P (—OK— button)	1.20 + 1.10
	BB (click button)	2 * 0.10
Total		9.10

BB = double button press to simulate mouse click and release (0.20 seconds)

You can see that the results are noticeably influenced by the type of user involved, and we would expect a more skilled or experienced user to complete such tasks in less time. This might be achieved by greater familiarity with the application or perhaps by employing a series of keyboard shortcuts. What is interesting is the comparison of these different paths and options to determine which is quickest, and which is best for the majority of our users. In effect, it becomes our default path.

For example, we might consider a comparable example for a desktop based email application, and a common task such as *sending an email.* Such usage might be defined with the following *keystroke-level* actions,

- pointing - user points to the *New Email* button
- clicking - user clicks on the *New Email* button
- typing - user types the recipient's email address, subject, and email body
- pointing - user points to the *Send* button
- clicking - user clicks on the *Send* button
- …

As with the earlier example, we might define times per action, and parts thereof such as hand to mouse &c., and then calculate performance per user to gauge options and designs. Such tests help us, as designers, create more efficient and user-friendly interfaces. By clearly understanding the cost of various actions, designers may optimise the UI to minimise time and effort required for users to complete their tasks.

There are also obvious limitations to such methodologies. It provides only a general rough estimate, and it naturally assumes that users know the correct sequences of actions in order to complete a given task. Also, it does not take into account user errors, application issues, processing considerations, and so on.

However, as designers, models such as KLM-GOMS give us a repeatable way to compare the efficiency of different interface and application alternatives. All other things being equal, the alternative that can be done in the least amount of time will tend to be the most convenient for our users, and thereby create the lowest cognitive load.

KLM-GOMS and touchscreens For touchscreen based devices, traditional KLM-GOMS needs to be modified as the interaction dynamics are different from keyboard and mouse-based interfaces. For such use cases and examples, researchers have proposed various updated models, including the *Fingerstroke Level Model* (FLM) and the *Touch Level Model* (TLM).

For example,

- FLM - proposed for predicting task performance in touch-sensitive user interfaces
- TLM - introduces new operators and modifications to KLM-GOMS to accommodate modern touch-screen interfaces

These models may be used to help model human task performance on a touchscreen device and, with proper benchmarking, accurately predict actual user performance. They allow us to compare predicted performance across different variations of a user interface. Such methods for touchscreen based devices do not require users or a working prototype, only a description of the path through the required software. In this respect, they may be considered mutations, or variations on a theme, of the standard KLM-GOMS model and testing example.

In summary, KLM-GOMS and its variants like FLM and TLM can be effectively used for UI and UX testing of touchscreen devices by providing quantitative predictions of user performance, thereby aiding in the design and evaluation of user interfaces for such devices.

FLM example usage We may consider a hypothetical example of how Fingerstroke Level Model (FLM) could be applied in a user interface design for a touchscreen device.

For example, we might be currently designing a mobile application for online music sales. The task we want to analyse might include *Adding an item to the shopping cart*.

We might break down this task into the following fingerstroke-level actions,

- pointing - user points to the item, such as a song or album, they want to add to the cart
- tapping - user taps on the item to view its details
- scrolling - user scrolls down to read the item details
- tapping - user taps on the *Add to Cart* button
- …

Each of these actions has an associated time cost in the FLM, following a similar pattern to the standard KLM-GOMS model.

For example, pointing might take `1.1 seconds` , tapping might take `0.2 seconds` , and scrolling might take `0.8 seconds` per screen. Adding these up, we can estimate that this task will take approximately `2.3 seconds` for an *expert user* to complete without errors.

This is, of course, a tad simplified example. Actual usage might involve more complex interactions, and thus more fingerstroke-level actions. However, it illustrates how FLM can be used to analyse and predict user performance in touchscreen interfaces.

The goal of using models like FLM is to help us, as designers, create more efficient and user-friendly interfaces. By understanding the cost of different actions, designers may optimise the interface to minimise the time and effort required for users to complete their tasks.

Playtesting

Another area of testing, which may commonly be seen with the development of UI based applications and games, is *playtesting*.

So, what is playtesting?

The general concept is pretty straightforward. Playtesters are chosen to play your game or use your application, for example, and then answer questions, describe their experience, and generally provide feedback about the game or application itself. In effect, we monitor testers playing the game, for example, to see what does and does not work, what they like, dislike, and so on.

One of the great benefits of playtesting, beyond feedback itself, is that it reminds us to see the end result of the development process, such as a game or application, through the eyes of a user. We're able to observe any particular concepts, objects, or patterns that a tester may focus upon.

For example, where do they focus within the application's screen, which components do they touch, how do they navigate a particular task or challenge, and, of course, where do they get stuck or delayed.

We observe, record, and note anything of interest. This forms our record of the playtest session, and each playtester.

designers and playtesting Whilst this may sound tedious in some respects, from a designer's perspective we may also gamify this regular task to some extent. The playtesters can become the guides to a particular task, challenge, and level within our application. They will highlight and demonstrate strengths and weaknesses within designs, mechanics, and general usage, such as gameplay.

However, some designers and developers may still choose to only involve playtesters at the end of a project's production or even not at all. The perception of such groups may be clouded or influenced by a tight schedule, lack of funds and resources, or a sense of anxiety with feedback itself. Perhaps the playtesters will dislike or hate the product being developed, or the developers may be pressured or forced in to modifying or removing certain cherished aspects of their application or game.

However, such a mindset is inherently counterproductive for the design and development of games, and applications in general.

The nature of games, and many UI centric applications, inherently discourages such individual, one way communication of concepts, thoughts, and ideas.

playtesting scenarios and sessions As we conceive and organise a playtesting session, we are effectively inviting players or users to simply come and play our games, or test a given application. Regardless of a game or application's current state, we're listening to what they say about their experience. This becomes an invaluable opportunity to gauge relative successes and failures in the game or application itself.

We can use this session both directly and indirectly. We may gauge user or player reactions as they interact with our game or application, carefully discern if silence is indicative of focus, boredom, or perhaps a combination of both. With practice, we learn to associate types of feedback with the current testing context, and use such patterns to help grow and develop our games and applications.

This feedback is a crucial aspect of the iterative design and development process. Each game and application will transform and grow as a result.

This style of testing is also reflective of a flexible approach and consideration of guidelines, design rules, and patterns. Designing a game or application needs to be a fluid process, which will evolve and grow over the many cycles of a project's development.

A detailed, sophisticated game, for example, is unlikely to be conceived from scratch, and then developed to successful completion and release, without going through this important and valuable process.

player experience goals and aims As we introduce playtesting and players, we need to consider goals for a player's experience with our game.

These are commonly known as **player experience goals**, and, as you might expect, these are goals that we may define for a player whilst testing and playing our game.

However, these are not defined features of the game (specific gameplay, mechanics &c.). Instead, we may consider them descriptions of interesting, useful, unique situations or scenarios, which a player may discover.

For example, a player may progress through a particular level. We may describe the expected emotions of this level as

"a player should begin rapidly, and encounter a sense of frustration as they tackle sets of problems. As they progress from problem to problem, this frustration is replaced with a sense of achievement. Ultimately, satisfaction results as they complete the level."

Another common example is a description of structure for a particular gaming experience, e.g.

"a player should be free to wander and experience the game at their own pace, and in their chosen order…"

Particularly useful for broader RPGs, for example.

In effect, we're trying to describe our game from the perspective of a player, and not as a designer and developer. For example, what should a player expect from aspects of the game, and the game overall.

Such goals also prove very useful as a way of describing aspects of a game as we plan its initial design and layout. It helps prevent an initial focus on the minutiae of a game's development, and instead plan the game as a player.

We may also use such goals later in each playtesting scenario to help correlate expected game design with the reality expressed by our playtesters.

initial prototypes and playtests As we begin designing our initial game concepts, we need to begin prototyping and testing.

We don't necessarily mean a digital, interactive prototype. Simply a playable version of the initial game idea. So, the first thing we can do is create a physical prototype of our game's core concepts, playable mechanics, and structure.

A physical prototype is a great way to perceive, test, and demonstrate these core concepts before we begin the job of painstakingly developing our game. We can use different mediums, including pen, paper, cards, cardboard, or even, perhaps, act out the game itself or important scenes and components.

With this technique, we are simply trying to ensure we perfect, as far as possible, our initial, simple model before artists, developers or producers are allowed to start work.

It may sound slightly bizarre, but we're using this stage of prototyping to ensure that the game's designer receives feedback on the core concepts as soon as possible.

In effect, we're checking that our play testers are able to achieve their player experience goals.

That the game is something worth playing.

Design and development patterns

As noted, we may consider this type of design and development as encouraging a known iterative pattern.

In effect, we're utilising this iteration to complete the following general steps:

- consider general ideas and concepts for your application or game project
  - discuss, read, watch, listen…anything to help inspire ideas and concepts
  - set player/user experience goals for the type of app/game you'd like to create
  - consider concepts and mechanics you want in your application or game
  - brainstorm initial top 3-5 ideas in your project group
- prototype - stage 1
  - create an initial physical prototype for your top 3 ideas (where applicable)
  - useful to help with selling your concept (e.g. to funders, other developers, testers…)
  - example artwork, character concepts, story themes and outlines…
  - act out gameplay examples for games…
- prototype - stage 2
  - start creating initial digital prototypes

    - interactive examples to test core functionality and usage
    - several prototypes will usually be created
        * each testing different concepts and examples
    - try to keep this quick, and easy to modify and update
        * do not get too preoccupied with the overall fidelity…
    - playtest these digital prototypes

It doesn't matter if you can't draw or design. Rough sketches and outlines are as useful for conveying ideas and concepts.

- document design and development requirements
    - use any notes, sketches, lists, &c. created during previous steps
        * these will help suggest structure and ideas for formal documentation
    - compile a full list of requirements, and development goals for your **actual** app/game
    - try to keep this documentation open to collaborative usage and editing
        * it will need to adapt and update as you develop the project
- build and produce your project
    - check each team member knows exactly what they need to do…
    - consider desired milestones for your project's development
        * check design and development at each milestone
        * evaluate current state of project as a group
    - start developing the final project…
- test, test, and test again
    - after you reach a given milestone, quality assurance is now possible
    - should highlight working, well considered functionality…
    - it will not resolve all issues
    - playtesting may continue to ensure quality and accessibility for players and users

As you start to iterate through production and testing steps, you should find that errors, updates, and general modifications get smaller (hopefully) with each iteration. This will often be a result of a carefully considered, planned, and prototyped project concept in the earlier steps of this iterative process.

benefits and usage You might be thinking that this seems a lot of work and preparation before you even reach the digital design and development phase.

In some respects, you'd be correct. However, you might be glad to hear that development, and the gaming industry, provides some examples and guidance for customising iterative patterns. As with most guidelines, recommendations, and systems, you can and should modify them to fit your game or app's specific requirements. As you gain greater experience of design and development, this initial planning and testing may be streamlined at certain stages.

If we consider physical prototypes, for a moment, we can see that their relevance and application may, in fact, be less useful for well established, tested mechanics and gameplay. Industry projects will often skip this step as part of their iterative design and development process.

This is simply because many companies produce games and apps with variations on standard, well tested game mechanics or app features. The designers and developers have a good idea how the game or app will work, and feel comfortable skipping ahead, so to speak. A lot of this is also due to industry pressures in general, including costs, timescales, resources, and player or user perceptions.

industry example However, such initial steps, including physical prototypes, become crucial if we are designing innovative mechanics and gameplay. If there are new examples and concepts, then it becomes crucial to plan and test thoroughly. In areas where we may lack experience, such testing and planning will be key to designing and developing a successful, playable game.

Electronic Arts (EA) is a company that has increasingly adopted such a design and development process for certain applicable games. They introduced internal training for pre-production methods in the mid-2000s, including workshops on physical prototyping and playtesting. Each of these have featured as part of their initial development.

Jeremy Townsend, who has worked at EAs Tiburon studio (best known for the Madden and Tiger Woods series of games), has used such **rapid prototyping** and pre-production methods to help inform game development. He notes, for example

> "Stay away from 3D prototyping if at all possible. Most game problems can be solved in 2D, even on paper," he said. "The Play's the thing - think of 3D prototyping as a big gun, you only want to use it as a last resort."

develop - EA at Grand Rapids

Whilst EA has also used Microsoft's XNA development tools for the XBox 360 console and Windows PCs, to help develop ideas quickly and efficiently, rapid prototyping still plays a key role,

As Townsend notes,

> "if something doesn't work you can correct away from it"

develop - EA at Grand Rapids

Spore, for example, was released by EA in 2008. It's an example of a **god** game, which became well known for its *procedural generation* and resulting *open ended* style of gameplay. This game used this type of pre-production testing and development. This included the creation of many different prototypes. For example,

- Spore - Prototypes

Linting

Linting is a process in software development that involves the use of tools called linters to analyse source code for potential errors, bugs, stylistic inconsistencies, and adherence to coding standards. The term *lint* originally comes from a Unix utility that scanned *C* code for errors and potential issues.

A linter programmatically scans a project's code with the goal of finding issues, which may lead to bugs or inconsistencies with code health and style. Some linters may even help fix such issues for developers. For example, if a developer declares the same *constant* twice in their JavaScript code, a linter may immediately flag such usage as an error.

Linting can help with a variety of things, including

- flagging bugs in a project's code from syntax errors
- giving warnings when code may not be intuitive
- providing suggestions for common best practices, depending on the current language
- keeping track of *TODO's* and other ongoing maintainence &c. comments and flags
- keeping a consistent code style
- ...

Linting is a key testing method each developer should know, and leverage for code development, sharing, and re-use. It may detect errors in code, including errors that can lead to initial or potential security vulnerabilities. Linting may detect formatting and stylistic issues, providing code that is commonly more readable and optimised for a given project or requirement.

Linting is particularly beneficial, as it may identify issues quickly and intuitively, preventing subsequent knock-on effects whilst running an app or during a code review. This immediate feedback may save developers from significant delays due to complex debugging and associated code reviews.

example linters There are many popular linters available for various programming languages.

For example,

- JavaScript - ESLint

- CSS - Stylelint
- Ruby - Rubocop
- C# - StyleCop, Sonar
- Python - Flake8
- C++ - IntelliSense Code Linter
- Rust - Clippy
- ...

Such linters may help developers analyse source code to uncover potential issues, including coding errors, stylistic inconsistencies, bugs, violations of coding standards, and potential security vulnerabilities. The choice of linter may often depend on specific needs of the project, and the chosen programming language.

Linters can detect a wide range of common errors in your code, including

- syntax errors - linters are noted for their common use in identifying syntax errors in interpreted languages such as JavaScript. Such mistakes in the code will, customarily, prevent the code from executing correctly.
- stylistic inconsistencies - linters can flag stylistic inconsistencies in your code, helping to maintain a consistent code style across your project
- unused variables, redundant code - linters may detect unused variables, unreachable or redundant code, which may commonly include parts of code that may never be executed
- type mismatches - some linters can catch mismatches in variable type
- potentially dangerous/unsafe data type combinations - linters may identify potentially dangerous data type combinations
- indexing issues - linters may prevent indexing beyond the end of an array
- ...

As expected, these are just a few examples for linter usage and feature. The exact nature of issues a linter may detect will depend, of course, on the linter itself and the supported programming language. The goal of a linter is to help developers maintain high-quality, error-free code.

Linting in a sample context

We might consider an initial example usage for linting tools with various technologies suitable for a common web-based application.

Such technologies might include a consideration of a web stack including HTML5, CSS, and plain JavaScript (ECMAScript). As we begin designing our client-side web application, we may review the following options for initial testing with linting tools.

HTML linting As noted above, linting is the automated checking of defined source code for programmatic and stylistic errors. In the context of HTML, linting can be used to improve the readability of an app's code, making it clear where the various sections, headings, paragraphs, forms, buttons &c. may be structured to validate as well-formed markup.

HTML Lint, for example, is a popular tool for this purpose. It is a free online validator and tool to help promote well-formed, structured HTML markup. A developer may copy and paste or directly type in the online editor, allowing *HTML Lint* to format and validate the HTML markup.

We might also consider integration of various plugins for HTML linting, including extensions based on ESLint, as part of a larger build tool structure and chain. This might include use of such extensions with IDEs for day to day development usage.

Linting HTML markup not only improves readability for collaborative developers, it also helps to identify and highlight potential errors, security vulnerabilities, perhaps due to overlapping hierarchies, mismatched semantic markup &c., or coding style issues before they become an issue for application development. Such linting, of course, also helps to promote best practices in markup structure, semantics, and subsequent parsing and traversal.

Common linting rules for HTML markup, as noted, are designed to help improve the overall quality and structure of the underlying markup and semantics. Linting rules, therefore, may be defined to check and identify common errors, thereby enforcing a consistent markup style and structure for client-side applications.

Example linting rules might include some of the following options,

- tag names - ensures that all tag names match appropriate syntactic requirements

- attribute names - ensure that all attribute names are valid, matching defined syntactic requirements
- closing tags - all elements must be closed, unless void elements, with a matching closing tag
- self-closing elements - void elements like `<img>` , `<br>` , and `<input>` should be self-closed, the end tag is neither permitted nor required
- IDs and class usage - IDs and classes should be meaningful and not overly complex, IDs should be unique for current document
- indentation - consistent indentation should be used for readability
- aria role - checks that role attributes are valid, appropriate for provision of semantic meaning to content for accessibility usage
- html has lang - checks for the `lang` attribute
- image alt - checks for `alt` attribute in `<img>` tags
- meta viewport - checks for the meta `viewport` attribute
- tabindex no positive - checks that no element has a positive `tabindex`
- ...

The specific rules defined and applied will, commonly, depend on the requirements for the current HTML markup and project. The key requirement for such client-side web testing, for example, is well-structured, validated, and accessible markup.

We may also see similar linting usage and rules for use with CSS and JavaScript.

Resources

- Card, S.K., Moran, T.P. and Newell, A. *The psychology of human-computer interaction.* Lawrence Erlbaum Associates. 1983.
- EA at Grand Rapids - http://www.develop-online.net/tools-and-tech/grand-rapids/0116020
- Encouraging Design System Best Practices with ESLint Rules - Backlight - https://backlight.dev/blog/best-practices-w-eslint-part-3
- Holleis, P. et al. *Keystroke-level model for advanced mobile phone interaction.* CHI' 07. New York, USA. 2007.
- HTML Lint - https://html-lint.com/
- Kieras, D. *Using the Keystroke-Level Model to Estimate Execution Times.* 1993. http://courses.wccnet.edu/~jwithrow/docs/klm.pdf
- Nielsen, J. and Pernice, K. *Eyetracking web usability.* New Riders. 2009.
- Usability Body of Knowledge - KLM-GOMS - https://usabilitybok.org/klm-goms
- What Linting is and How to Use it - Web Developer - https://webdeveloper.com/tips-tricks/what-is-linting-is-and-how-to-use-it/