Closed-Source Development Deliverable 1

Justin Abrokwah
Yi Shiun Chiu
Spencer McCoubrey
Parth Thakkar

Architecture Commentary

The architecture of matplotlib seems nearly entirely based on the principles of agnosticism, encapsulation and decoupling. Between the major layers we cover in our UML (PyPlot, Artist, Backend), each layer is only responsible for a key component of the library. PyPlot is the state-based user interaction model, allowing users to write scripts to interact and control Artist and Figure classes. PyPlot exposes only clean, concise APIs in order to allow the user to fully implement any functionality they want, without needing to care about graphical context, rendering, operating system or environment. This is largely due to the fact the PyPlot doesn't even know these implementation details. The backend layer of Matplotlib successfully encapsulates all of the system heavy logic, allowing PyPlot to call clearly defined base class level methods without needing to worry about any system changes.

This style of encapsulation and agnosticism continues even within the backend layer itself, where every actionable portion of the service layer is abstracted neatly into base level classes. This includes the Renderer, FigureCanvas, and the other classes shown in the UML provided. Each class has one single major responsibility, and delegates tasks to other dependencies. A common design pattern in the backend layer is dependency injection, where the system determines which _Backend implementation is correct based on context, which in turn decides which implementations of the core classes need to be instantiated and passes them into the construction of other classes whose logic depends on them. This allows for all of the core backend classes to run agnostically to one another, assuming they will be given the correct dependency, again encapsulating away different system requirements to each actor in the system (eg. Renderer knows how to draw path in SVG; FigureCanvas knows how to display an SVG; etc.). Inheritance plays a large role in this dependency injection framework, as every layer assumes inherited functionality from a base class that each context correctly implements.

The Artist layer is largely based on inheritance (shown in the UML), where every renderable/drawable object inherits from the base classes. This allows users to edit configurations and interact with Figures, which in turn manage and edit Artists. Artists appropriately implement methods using dependency injection given a Renderer to render themselves within a context. Another pattern used in the implementation of the Figure object is the observer pattern. Each figure can contain multiple axes, which have states. Observers (functions) are added to an Axes so that whenever the state of Axes changes, all the observers will be updated with the new Figure to keep all components of the system on the same page. In Figure.py, the add_axobserver(self, func) function is used to add an observer, and in functions like sca() and delaxes(), the pattern is executed as the observers are updated.

Overall, Matplotlib does an excellent job decoupling the major layers of the system, creating a maintainable architecture. One improvement that could be made is the use of a

dependency-injection framework. Currently it is done internally, using an in memory Python dictionary object with package-paths specifying the correct dependency based on file type. We believe that using a well maintained and tested dependency injection framework would clean up this code, and make it more configurable.

Software Development Process

For our project, we decided to use the Kanban software development process due to its overall flexibility, enabling us to work productively. As university students, we have differing schedules and responsibilities both in and outside of school; thus we need a process that fits our lives. Kanban is designed to be flexible, to not completely tear down the schedule and plans that already exist; but instead is built to fit the already existing environment and team.

Kanban has many benefits for this type of project including the ease of adapting to changing requirements. Kanban is flexible in nature, so if things change from a requirements or even a logistics point of view, the process will easily accommodate our needs. Inherently in Kanban is the idea of visualizing the tasks on a "board". Our group decided to use trello and modify it slightly to fit the process and our needs. Trello allows us to make clean, simple boards that effectively visualize the state and size of tasks. We can easily see the assignee, expected completion date, size, relevant links and descriptions; and it allows us to compile our discussions and notes into the board itself. Trello is designed with user stories and tasks in mind without restricting us to mandatory columns or issues like Scrum would.

The idea of Kanban's work in progress limit was very appealing as it assures the team that we will not become overwhelmed with work at any point. By artificially restricting the number of tasks per column, we ensure that we focus on current tasks and prioritize accordingly without creating a backlog of non-relating tasks. Since trello is also accessible and easy to use, this allows each team member to manage their own issues on the board, only pulling and assigning tasks to themselves when they are ready. This empowers each member to feel responsible for their task, and to see it all the way to completion.

On top of the work in progress limit, we as a team will moderate the amount of work each team member does. We want to ensure each member is doing their fair share of work and everyone's own timeline and schedules are being respected. To do this, we will design tasks in such a way to compartmentalize as much as possible, leading to smaller more attainable tasks. As well as monitoring the throughput of each member (not vigilantly of course but more on a high level), so that each member contributes as evenly as possible.

Kanban overall is flexible, forming to fit our needs as a team. This project is bound to change as are the requirements, so we need something that will change with us. Kanban provides a linear process, permitting us to visually see how we are progressing and how much we have left to accomplish. It is a transparent process that will also allow the TA's and course instructor to check in and easily verify and monitor the progress being made.

We decided against using any form of plan-driven development due to the fact that we expect requirements to change throughout the course. Also, we only receive requirements 1-2

weeks ahead of time, making plan-driven processes inefficient. We also decided against incremental processes as we have no client or business facing requirements, making the point of these processes mute. Along this vein, XP-programming was not chosen as we do not have a client by which to create a successful feedback loop for improvement, and since we have clearly defined deliverables, there is no need for priority management by a client in our process. Regarding the reuse development approach, since we aren't planning to reuse any components but to contribute to an existing product, we decided against this methodology as well.

Kanban Board

Our kanban board will be split into 7 key columns (WIP = work-in-progress limit):

1. **Explore:**
    - **a.** The explore column is for ideas and tasks that need flushing out. Any person can fill in details as needed until a task is sufficiently detailed enough to be placed into ready.
    - **b.** In theory there is no WIP for this column, but we will not be adding tasks too far in the future to reduce column size and make board cleaner.
2. **Ready:**
    - **a.** When a task is sufficiently researched and flushed out, it is placed in the ready column.
    - **b.** Any team member can pull from this column and assign the task to themselves.
    - **c.** The WIP for Ready will be about 3 tasks per team member (around 12) so that the column is orderly and we can visualize the priority of tasks easier.
    - **d.** A task in Ready can be pushed back into Explore if more research or planning is deemed to be necessary.
3. **In Progress:**
    - **a.** When a member takes a task from Ready and assigns it to themselves, it should be moved into In Progress. This stage is for tasks that are actively assigned and being worked on.
    - **b.** The WIP will be 6 total so that the team can focus on contained tasks, disallowing the scope of current work to span too far, but still allowing some members to actively work on more than 1 task.
    - **c.** A task in In Progress can be moved back into Ready if something else needs priority, or even Explore if the task was incorrectly designed.
4. **Testing:**
    - **a.** When a developer is done pure development, they move their task into testing to indicate they are currently self testing their unit of work.
    - **b.** The WIP will be 4, as at any given time a developer can only be testing 1 unit of work.
    - **c.** A task In Progress can be moved back to any column depending on the results of the testing.
5. **Waiting for Review:**

    a. When a task is completed from a work perspective, it is placed into this column to await other team members to approve the task.
    b. The WIP limit will be 6 to match the In Progress WIP to encourage the flow of tasks instead of allowing this column to become a bottleneck.
    c. Tasks can move backwards from this column if requirements change or additional work is required.
6. **In Review:**
    a. When a task is Waiting for Review and a team member wishes to conduct the appropriate review for the task, it is moved into this column and assigned to the reviewer. Review can include code review, grammar review, test approval, etc...
    b. The WIP will be 4 tasks as at most, each team member will be reviewing a different task at a given time.
    c. A task In Review can be moved essentially back into any column depending on the outcome of the review (i.e if tests fail, code review, fails to meet requirements, etc...).
    d. A task can be moved back into Waiting For Review if it is approved by one member, but requires another member's approval before moving on. The member will attach their approval label, unassign themselves and move the task accordingly accordingly.
7. **Done:**
    a. This column is for any and all tasks that have been completed fully and to specification. Any member moving a task into Done should fully read the task description to validate correctness and doneness.
    b. A task should never move from Done to any other column; if more work is required after completion, a new task should be made.
    c. There is no WIP for this column