Program Comprehension Report of Group 3

Shiqi Wu, Wenkang Gong, Yanqiu Mei, Zihan Kuang, Ruixuan Li
Department of Computer Science and Engineering
Chalmers University of Technology, Gothenburg, Sweden
{shiqiw, wenkangg, yanqiu, zihank, ruixuanl}@student.chalmers.se

I. Selected Techniques and Rationales

Four main techniques were selected for conducting code comprehension: SonarQube, PyDeps, Js-Analyzer, and document mining.

For static code analysis, SonarQube was chosen since it is being widely used for static code analysis in industry and has comprehensive quality assessment abilities for large-scale projects and supports various programming languages.

For dependency analysis, since the primary programming language of Core and Frontend are different, PyDeps was used to analyze the Core repository, and Js-Analyzer was used for Frontend analysis. Both of them are flexible, where Js-Analyzer supports dynamic switching to different files at different analysis stages, and PyDeps could generate the dependency graph based on the Python imports. Before switching to PyDeps, our group tried PyReveng3 and PyCg as well, but found them impractical in our use case due to complex setups or compatibility issues.

Document mining was also useful for understanding the design of both overall Home Assistant structure and Hue integration. Home Assistant has detailed documents providing key insights such as the architecture of the entity and integration and the interactions between its modules. The Hue developer's guide also demonstrated Hue APIs, the state-management mindset of Hue system, as well as some best practices.

II. High-level Analysis of Home Assistant Core

A. Static Code Analysis

Our group identified three main characteristics of Home Assistant Core: its modular architecture, controlled technical debt, and balanced complexity distribution.

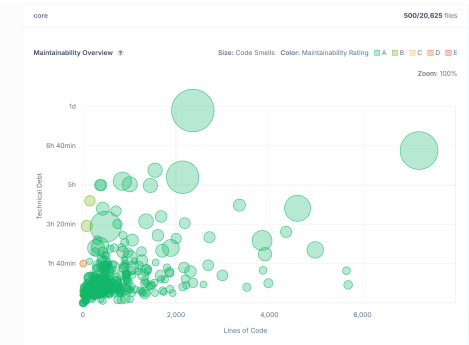


Fig. 1: SonarQube Maintainability Overview for Home Assistant Core

First, the system demonstrates a clear modular architecture. The maintainability visualization shows that quality issues are distributed across many small files rather than concentrated in large modules, with most files below 2,000 lines and favorable maintainability scores (Figure 1).

Second, the project maintains very low technical debt relative to its size. Despite exceeding two million lines of code, SonarQube reports only a 0.1% technical debt ratio and an A-rating for maintainability, indicating consistent quality practices throughout development.

Third, system complexity is well distributed across components. With 95,826 functions across 11,905 classes in 20,625 files and cyclomatic complexity of 167,183, individual components remain manageable. This reflects the plugin-based design where complexity is contained within specific integration modules rather than centralized.

The analysis also identified 1,246 reliability issues, 43 security vulnerabilities, and 4.5% code duplications, and they are typical findings for a system of this scale that potentially pointed at the areas for targeted improvements.

B. Dependency Graph Analysis

To further understand the architecture of Home Assistant Core, we analyzed the dependency graph using the PyDeps tool. The resulting visualization indicates that Home Assistant is not a monolithic application but rather a multi-layered platform. Its structure is defined by several central hub modules rather than a single linear backbone, showing characteristics comparable to those of the operating systems, as shown in Figure 2.

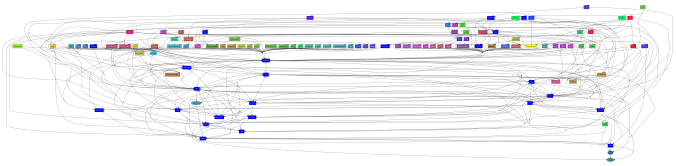


Fig. 2: Dependency graph of Home Assistant Core. It involved a lot of modules, but a clear multi-layered structure is visible.

The analysis of the dependency graph reveals a layered architectural model with three notable characteristics. At the foundation, a set of modules functions as the system's architectural hubs. These include `homeassistant.core`, `homeassistant.helpers`, and `homeassistant.components`, which collectively act as a kernel and service layer. They provide fundamental

functionality such as the state machine, event bus, and stable APIs, and serve as the gravitational centers of the graph with particularly high fan-in.

Second, this kernel is surrounded by a vast, decoupled ecosystem of integrations, the components. These integrations depend on the kernel but remain independent of one another, which is the foundation of the system's modularity and extensibility.

Finally, the graph confirms a strict, uni-directional dependency flow: specific integrations rely on the generic kernel, but not vice versa. This enforces architectural stability by preventing feature-level changes from impacting the core.

Overall, the dependency graph provides insights that complement static code analysis. It shows that Home Assistant Core is organized as a robust and extensible platform, comparable to a micro-operating system. This layered design, with a stable kernel and a decoupled plugin architecture [1, p. 178], underpins the system's ability to support a large and diverse community-driven ecosystem.

C. Document Mining

The developer documentation provided the essential design rationale for the system. As illustrated in Figure 3, the core architecture consists of central registries and a service hub. This stable foundation is extended via a modular integration process, where external devices are abstracted into standardized 'Entity Platforms'.

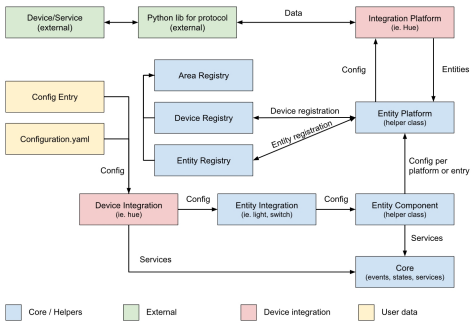


Fig. 3: Home Assistant's documented architecture.

III. High-level Analysis of Home Assistant Frontend

A. Static Code Analysis

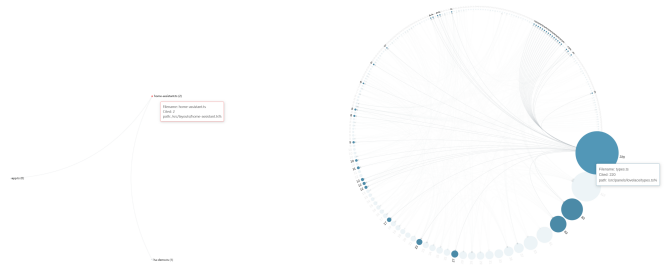
SonarQube analysis shows that the Home Assistant Frontend is a large-scale application with 329,493 lines of code across 2,120 files, comprising 17,219 functions and 1,396 classes. Most of the logic is concentrated in the `src` directory (313,295 lines, 95% of the total), while supporting infrastructure resides in `hassio` (10,418 lines) and `build-scripts` (3,003 lines). A summary of quality metrics across project components is shown in Table I. The abbreviations in Table I: LoC denotes lines of code; Sec, security issues; Rel, reliability issues; Main, maintainability issues; and Hotspots, security hotspots.

TABLE I: SonarQube code quality metrics for the home-assistant-frontend project components

Component	LoC	Sec	Rel	Main	Hotspots
frontend	329,493	1	135	6,714	68
.devcontainer	5	0	0	0	1
build-scripts	3,003	0	0	8	3
cast	1,462	0	0	40	0
hassio	10,418	0	4	175	0
landing-page	784	0	0	14	9
public	15	0	0	0	0
script	67	0	0	0	0
src	313,295	1	131	6,476	54
test	220	0	0	1	1

The project maintains a technical debt of 79 days with a debt ratio of 0.4%, indicating that development practices still keeps the codebase under control despite its size. Complexity measures (cyclomatic complexity of 44,525 and cognitive complexity of 34,460) confirm non-trivial control flow but remain within manageable limits considering its scale. Reliability and security issues are relatively few (135 and 1, respectively), and although 68 security hotspots are present, these are distributed rather than concentrated in critical files as shown in Table I. Documentation, however, is limited: comment coverage is only 1.3%. Overall, the analysis suggests a mature codebase with clear boundaries between infrastructure and business logic, but with scope for improving maintainability through better documentation.

B. Dependency Graph Analysis



(a) (b)

Fig. 4: (a) Dependency view of `home-assistant.ts`. Dependency view of `home-assistant.ts`. The number in parentheses indicates the citation count. (b) Folder relationship map for `src/panels`. Node size represents the relative number of times a file is referenced — the larger the node, the more frequently it is cited. The large node in the bottom is `types.ts`, which is referenced by many other nodes.

Dependency graphs generated by the Js-Analyzer tool provided further insights into the frontend architecture. Figure 4a shows the dependencies around `home-assistant.ts`, while Figure 4b illustrates the relationships among panels in the `src/panels` directory. The frontend is organised around a small kernel of high-fan-in modules, complemented by a large set of loosely coupled panels and dialogs.

The kernel is formed by files such as `home-assistant.ts`, `home-assistant-main.ts`, and `src/entrypoints/core.ts`. Together with the base class `HassElement`, they manage global state, routing, authentication, and service coordination. Surrounding this kernel, the `src/panels` directory contains dozens of self-contained panels (e.g., Lovelace, Map, and Media Browser) implemented as custom elements. These panels depend on the kernel but not on each other and are loaded lazily through a resolver mechanism. The large node `lovelace/types.ts` highlights the common reuse of type definitions across panels. A similar organisation applies to dialogs, which are modular overlays triggered from panels.

The data flow follows a uni-directional pattern where panels and dialogs invoke backend services through the `hass` API (`callService()`, `callWS()`, `callApi()`), while the kernel receives updates via WebSocket subscriptions and propagates changes through the `hass` object. This architecture ensures that global state is only updated centrally, preserving separation of concerns and maintainability.

C. Document Mining

The developer documentation clarifies and confirms several aspects of the frontend design. It describes the architecture as consisting of a bootstrap script (`core.ts`), an app shell (`app.ts`), a set of panels, and a set of dialogs. The bootstrap handles authentication and establishes the WebSocket connection, while the app shell provides routing and the sidebar. Panels implement main views, and dialogs provide modal flows.

The documentation emphasises the central role of the `hass` object, which stores global state and exposes API helpers. Properties such as `hass.states` and `hass.user` provide access to current data, while methods like `hass.callService()`, `hass.callWS()`, and `hass.callApi()` support backend interaction. API calls are gradually being replaced by WebSocket communication. Updates flow uni-directionally through the system, with the root component updating the `hass` object in response to backend changes, and this updated object is then passed down to all components.

These findings match the results of static and dependency analysis, showing that the Frontend, like the Core, uses a layered design. It has a small kernel with a modular application layer around it, which keeps extensions separate and allows the community to develop them in parallel without affecting each other.

IV. Detailed Analysis of Hue Integration

The file structure of the Hue integration is clear. It has the required `manifest.json`, `__init__.py`, `config_flow.py`, `bridge.py`, as well as dedicated modules for services and device triggers, and two separate subdirectories `v1` and `v2` that supports legacy and current bridge version.

The purpose of this integration is to allow Home Assistant to control and monitor Philips Hue devices connected via a Hue Bridge (both legacy v1 and the newer v2), exposing lights, motion sensors, remotes, and scenes as entities within the Home Assistant ecosystem.

It interacts with the Philips Hue Bridge through the Hue APIs, the `aiohue` library for device models and asynchronous communication, and the Home Assistant Core framework via entities (e.g., `LightEntity`), the device registry, configuration flows, services (e.g., scene activation), and the event/state system.

In the rest of this section we will demonstrate our reflections together with code examples. There are many metrics and data for reviewing during the analysis, but here we will only present those we found important and interesting.

A. Document Mining

Because the Hue integration was contributed by the Home Assistant community, the integration code base itself lacks comprehensive documentation; however, two sources were particularly helpful in understanding it: the official Philips Hue developer documentation, which provides the original SDK and API reference as well as the Hue system design, and the Home Assistant Core development guide, which explains how to implement new integrations.

The Hue documentation introduces the system's core concepts and APIs. The Hue system is state-based: every device is managed by the Hue bridge and assigned a unique URL. Developers interact with devices by sending requests to the bridge, for example, `/resource/scene/{id}`, which supports GET, PUT, and DELETE for saving or retrieving light group settings. These resources offered a clear overview of Hue's structure before implementation and remain useful for future development.

The Home Assistant Core documentation clarified how Hue entities integrate into the system. For example, the `HueLight` integration extends both `HueBaseEntity` and `LightEntity`. Entities serve as interfaces that connect new integrations with Home Assistant's flow, as illustrated in Snippet 1.

```
class HueLight(HueBaseEntity, LightEntity):
    _fixed_color_mode: ColorMode | None = None
    entity_description = LightEntityDescription(
        key="hue_light", translation_key="hue_light", has_entity_name=True, name=
            None)
    def __init__(self, bridge: HueBridge, controller: LightsControlle, resource:
          Light) -> None:
        """Initialize the light."""
        super().__init__(bridge, controller, resource)
```

Snippet 1: HueLight implementation

B. Static Code Analysis

SonarQube and PyTest was used to analyze the code quality of the Hue integration.

SonarQube reported 3,555 lines of code with only four security hotspots. The two versions are similar in size (v1: 1,023 LOC; v2: 1,162 LOC), and despite supporting both for legacy compatibility, the overall scale remains manageable. No code duplications were found, indicating a well-maintained

structure. However, the integration shows high complexity (cognitive: 531, cyclomatic: 562), with 8 of 34 files exceeding a cyclomatic complexity of 20, suggesting some parts may still benefit from refactoring.

According to the PyTest code coverage report, the Hue integration achieves an overall coverage rate of 90%, which showed the well-designed test structure. This coverage level reduces the likelihood of introducing defects or breaking existing functionality when adding new features, and it added more confidence for us to choose to refactor and potentially add features. The files with lowest coverage are `/hue/services.py` (65%) and `/hue/device_trigger.py` (68%), hence one of the future improvements could be that increase the coverage for these scripts. Core components maintain higher coverage rates, with `config_flow.py` at 99%, `scene.py` at 90%, and `light.py` at 92%. Both API versions show consistent coverage patterns, with v1 and v2 implementations tested at comparable levels. The coverage analysis reveals 145 missing statements and 45 partial branches out of 2,115 total statements, indicating that most code paths are covered by the test suite. Table II showed the key detailed coverage data.

TABLE II: Selected PyTest coverage metrics for the Hue integration (paths under `homeassistant/components/hue/`). Most of the files have coverage greater than 80%, and some of the key modules have higher coverage than 90%.

Selected File	Stmts	Miss	Branch	BrPart	Cover
services.py	55	16	16	1	65%
device_trigger.py	54	13	28	13	68%
config_flow.py	159	2	46	1	99%
scene.py	102	3	32	10	90%
light.py	14	0	2	0	100%
TOTAL (all files)	**2,115**	**145**	**584**	**98**	**90%**

C. Dependency Analysis

The dependency analysis was conducted using the PyDeps tool in combination with manual inspection of the source code. The dependency structure for the Hue integration is relatively clear. For example, in the code snippet below, the `homeassistant` library supplies the core entities and modules required to connect the Hue integration to the broader Home Assistant workflow (Snippet 2), while the `aiohue` library provides the modules, models, and methods needed for Hue device representation, setup, asynchronous communication, and general class encapsulation (Snippet 3).

```
from homeassistant.core import Home
       Assistant, callback
from homeassistant.helpers.
       entity_platform import
       AddConfigEntryEntitiesCallback
```

Snippet 2: Imports from the Home Assistant Core module

```
from aiohue import HueBridgeV2
from aiohue.v2.controllers.lights
    import LightsController
from aiohue.v2.models.light
    import Light
```

Snippet 3: Imports from the aiohue module

In conclusion, the overall code comprehension results look promising and Hue appears to be a good choice for the target integration that the group will be working on.

V. Reflections

A. Reflections on the Home Assistant Project

Among all the code comprehension results, one of the aspects that left our group the most impression was its design for extending integrations. The entity-integration structure reflects the methodology behind Dependency Inversion principle[2], where every layer of the program depends on the interfaces, which made the repository scalable. This design is tactical and useful to take into consideration when the group members need to design a new system.

B. Reflections on the Code Comprehension Process

The code comprehension process is a educational step before the team starts to touch the real code. During this process, the team could setup a general picture about the architecture, implementation strategies, as well as cautions in development. Also, code comprehension not only could help the team find potential defects or improvements, but also could facilitate learning some good practices in the software design and development.

Extensive information will be retrieved and perceived during code comprehension, so our group suggests that instead of trying to remember everything, it is also important to try to setup an "indexing system" so that the group could know where to find useful information later in the refactoring and development phase.

C. Reflections on the Tool Selection

The code comprehension stage has an intrinsic requirements of getting things onboard in a limited time. Hence, instead of sticking to one pre-defined tools, our group switched from PyReveng3 to PyDeps after we found that it takes some time to install and run PyReveng3 and PyDeps has similar functionalities but can be used faster. Tools are meant to serve for developers, so one tool is appreciated as long as it helped the team gain information.

VI. Contribution Table

The total workload was evenly distributed (20% for each person) and all the group members worked dedicatedly. Here is our division of work for group assignment 1.

- HA-Frontend Analysis: Wenkang Gong, Yanqiu Mei
- HA-Core Analysis: Zihan Kuang, Ruixuan Li
- Hue Integration Analysis: Shiqi Wu
- Presentation: Shiqi Wu, Zihan Kuang, Yanqiu Mei

References

[1] Robert C. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall, 2018.

[2] Robert C. Martin. "The Dependency Inversion Principle". In: *C++ Report* (May 1996).