Solar System Explorer

An Interactive C++ / Qt6 Educational Application

Project Overview

Solar System Explorer is a multi-feature, interactive desktop GUI application that visualizes our solar system in an animated 2D scene, allowing students, teachers, and enthusiasts to navigate celestial bodies, explore planetary data and satellite missions, and test their knowledge with integrated quizzes.The project is implemented in C++17 using the Qt6 Widgets framework and is built with CMake. Architecturally, the application is organized into six layers — models, factory, observers, strategies, data, and ui — and applies four classical design patterns (Factory Method, Observer, Singleton, and Strategy) together with the SOLID principles. This report describes the application's features, architecture, class design, and the specific design decisions taken, with UML diagrams illustrating the structural and behavioral aspects of the system.

Feature Summary

- Animated 2D solar system rendered with QGraphicsScene, including the Sun, eight planets, fifteen moons, and twenty-two real-world satellites and spacecraft.
- Custom 2D graphics: radial gradients, Saturn's ring system, Jupiter and Saturn cloud bands, Earth continents, the asteroid and Kuiper belts, four labelled galaxies, and six emission nebulae.
- Mouse-driven camera: scroll-wheel zoom, right-click pan, click-to-track, and an automatic zoom-to-body that adjusts magnification for stars, gas giants, and moons.
- Search-as-you-type bar that queries any of the 47 catalogued bodies by partial name and snaps the camera to the result.
- Hierarchical side menu organized as Sun → Planet → Moon and Spacecraft → Agency → Mission, with smooth slide-in animation.
- Information panel that updates reactively on selection and shows category, radius, distance, temperature, day length, year length, mass, narrative description, fun facts, and distance facts.
- Toolbar View Mode toggle that swaps the rendering strategy at runtime between a stylized look and an information-overlay mode that labels every body with its category and distance.
- Five-tab "Learn More" dialog (Overview, Fun Facts, Science, Distances, Related) per body.
- Space Encyclopedia: 14 expandable topics on eclipses, light-years, gravity, the Big Bang, dark matter, and more.
- Personal Notes system: in-memory CRUD with title, content, timestamp, bookmarking, and three views (per-body, all-notes, bookmarks-only).
- Space Quiz: 50-question, five-round game randomly drawn from a pool of 96 questions, with shuffled options and explanations after each answer.

Scope and Size

The codebase is approximately 2,423 lines spread across 45 source files (23 .h and 22 .cpp). It is organized into six logical packages and depends only on the C++17 standard library and Qt 6 Widgets, making it portable to macOS, Linux, and Windows.

How to Build and Run

1. Prerequisites

- A C++17 compiler (clang 12+, gcc 9+, or MSVC 2019+)
- CMake 3.16 or newer
- Qt 6 with the Widgets module (Qt 6.2 LTS or newer recommended)

2. Build

From the project root:

```
cd SolarExplorer
cmake -S . -B build -DCMAKE_PREFIX_PATH=$(brew --prefix qt)
cmake --build build -j
```

On Linux replace the prefix flag with the location of your Qt 6 install (for example /opt/Qt/6.6.0/gcc_64/lib/cmake). On Windows use the Qt-supplied developer prompt.

3. Run

```
./build/SolarExplorer
```

On macOS the binary may also be launched as build/SolarExplorer.app if Qt produces an app bundle.

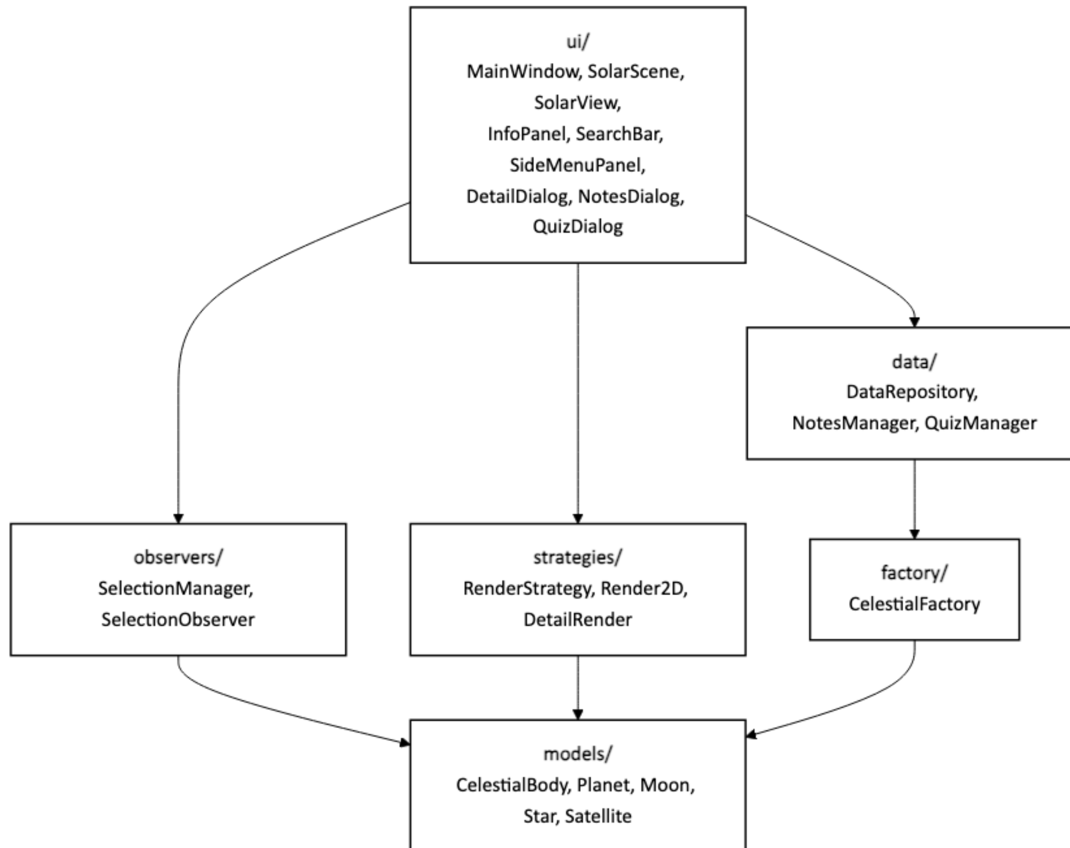
4. Quick Tour

1. Click any planet, moon, or satellite to select it; the right-hand panel updates and the camera tracks it.
2. Press the ||| button (top-left) to open the hierarchical side menu.
3. Type in the search box to filter all bodies as you type.
4. Open Space Encyclopedia, My Notes, or Space Quiz from the toolbar.
5. Use the scroll wheel or the +/- buttons to zoom; right-drag to pan; press Reset View to return to the overview.

System Architecture

The application is structured as a layered architecture in which dependencies flow strictly downward. The UI layer depends on domain models and on data managers; the domain models depend on nothing project-internal. This keeps the model code testable and free of Qt-specific details where possible (only QColor leaks into CelestialBody for convenience).

1. Package Diagram



Arrows denote “depends on.” Lower layers do not know about higher layers.

2. Responsibilities by Package

Package	Responsibility
models/	Pure-domain classes for celestial bodies. Contains the abstract base CelestialBody and four concrete subclasses (Planet, Moon, Star, Satellite).
factory/	Centralized object construction. CelestialFactory::create maps a BodyType enum to the correct concrete subclass.
observers/	Notification mechanism. SelectionManager broadcasts selection changes to any class implementing SelectionObserver.
strategies/	Pluggable rendering algorithms. RenderStrategy interface plus Render2D and DetailRender implementations.

Package	Responsibility
data/	Data and service singletons. DataRepository holds the body catalogue; NotesManager persists user notes; QuizManager builds randomized question sets.
ui/	Qt widgets, dialogs, and the QGraphicsScene. Talks to data and observers but never creates models directly.

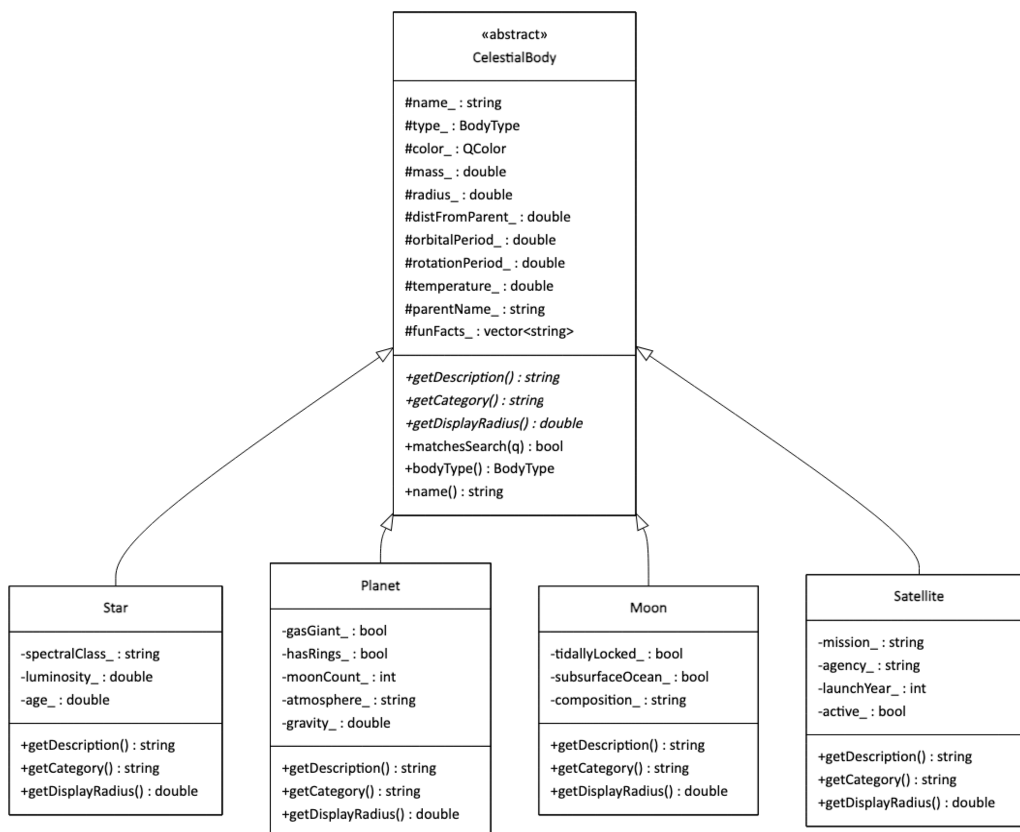
3. Build System

CMakeLists.txt enables AUTOMOC, AUTORCC, and AUTOUIC so Qt's code generators run automatically for any class with the Q_OBJECT macro. The project links only against Qt6::Widgets; no third-party libraries are required.

Domain Model and Class Design

The heart of the application is the CelestialBody hierarchy. All UI consumers (the info panel, the detail dialog, the search bar, the side menu) depend only on the abstract base class, which lets the system handle a Star, a Planet, a Moon, and a Satellite uniformly.

1. Class Diagram — Domain Model



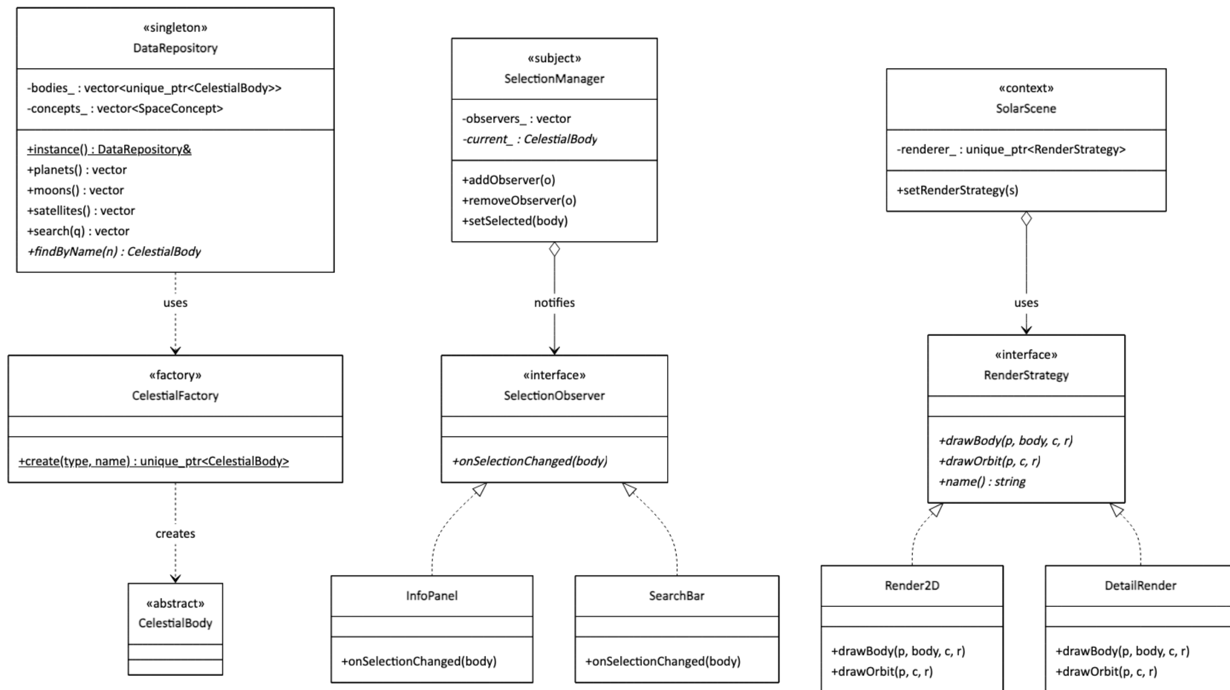
2. Why an Abstract Base Class?

Three behaviors differ per body type but every consumer needs them:

- getDescription() produces a human-readable narrative. A Star describes spectral class and luminosity; a Satellite describes mission and agency. The text is too different to express with shared fields, but every UI consumer needs “some description.”
- getCategory() returns a short, displayable label such as “Gas Giant,” “Natural Satellite,” or “Active Satellite.” It varies with both type and runtime state.
- getDisplayRadius() returns a hand-tuned visual size (in scene units) so that Jupiter dominates its inner cousins without making Mercury invisible. This decouples scientific radius from display scale.

These three methods are pure virtual; this is genuine polymorphism, not a tagged union with an enum dispatch.

3. Class Diagram — Patterns and Services



Singleton, Factory, Observer, and Strategy collaborate to decouple construction, notification, and rendering.

Design Principles (SOLID)

Each SOLID principle is examined below, with concrete code references and an honest assessment of how well it is realized.

1. Single Responsibility Principle (SRP)

Each class has one well-defined reason to change.

- NotesManager (data/NotesManager.cpp) manages a single concern — in-memory note storage and CRUD operations. It exposes addNote, updateNote, deleteNote, toggleBookmark, and three query methods.
- QuizManager (data/QuizManager.cpp) is responsible only for assembling shuffled question sets out of a fixed pool of 96 questions.
- SearchBar (ui/SearchBar.cpp) owns a single concern: text-driven filtering of body names and emitting a bodySelected signal.
- InfoPanel (ui/InfoPanel.cpp) is responsible only for displaying whatever the SelectionManager points to.

MainWindow is the largest class and orchestrates the top-level layout. While its primary job is composition, it does build the Encyclopedia dialog inline. In a future iteration this would be extracted to a dedicated ConceptsDialog class to fully respect SRP.

2. Open/Closed Principle (OCP)

The system is open for extension and closed for modification along its main extension axis: adding a new kind of celestial body. Adding a hypothetical Comet type requires:

1. Adding COMET to the BodyType enum (one line).
2. Writing a Comet : public CelestialBody subclass with the three virtual methods.
3. Adding one case to CelestialFactory::create.

None of the existing UI consumers — InfoPanel, SearchBar, DetailDialog, SideMenuPanel — needs to be modified, because they all consume CelestialBody* polymorphically.

3. Liskov Substitution Principle (LSP)

Every CelestialBody subtype honors the contract of the base. The three pure-virtual methods always return a non-empty string or a positive radius, and no subclass throws or has hidden preconditions. As proof, DataRepository and InfoPanel manipulate every body through the same CelestialBody* pointer; substituting a Star for a Planet at runtime never produces undefined behavior.

4. Interface Segregation Principle (ISP)

SelectionObserver (observers/SelectionObserver.h) is intentionally minimal — a single onSelectionChanged(CelestialBody*) method. Implementers are not forced to provide stubs for

unrelated callbacks. This is the smallest useful observer interface for the application's notification needs.

5. Dependency Inversion Principle (DIP)

High-level UI policy depends on abstractions where it matters most:

- SolarScene holds a std::unique_ptr<RenderStrategy> rather than a concrete renderer.
- SelectionManager works against the SelectionObserver interface, never knowing the concrete subscribers.
- DataRepository::create paths invoke CelestialFactory rather than calling new on Planet/Moon/Star/Satellite directly.

The data managers (DataRepository, NotesManager, QuizManager) are concrete singletons, which couples consumers to a specific implementation. For a single-user desktop app this is an acceptable trade-off; the report acknowledges the testability cost in the Trade Off section.

6. Other Principles Applied

- Separation of concerns. The folder hierarchy mirrors the conceptual layering. Domain code does not import any Qt UI header except QColor.
- Encapsulation. State is held in private/protected fields and accessed via getters.
- DRY. Body construction is funneled through the factory so no fragment of the codebase repeats the "new Planet, then set fields, then push_back" boilerplate.
- Composition over inheritance for services. Managers are aggregated by reference into UI widgets rather than inherited.

Design Patterns

Four classical Gang-of-Four patterns are present in the project. Each is described with the role it plays, the participating classes, and a representative code snippet.

1. Factory (Creational)

Intent: Centralize the construction of CelestialBody subtypes so that callers can request "a planet named Earth" without depending on the Planet constructor or coupling to header files for every concrete type.

Participants: CelestialFactory (creator), CelestialBody (product), Planet/Moon/Star/Satellite (concrete products), DataRepository (client).

```
// factory/CelestialFactory.cpp
std::unique_ptr<CelestialBody>
CelestialFactory::create(BodyType type, const std::string &name) {
    switch (type) {
        case BodyType::STAR:      return std::make_unique<Star>(name);
        case BodyType::PLANET:    return std::make_unique<Planet>(name);
```

```
        case BodyType::MOON:      return std::make_unique<Moon>(name);
        case BodyType::SATELLITE: return std::make_unique<Satellite>(name);
        default: return nullptr;
    }
}
```

Benefits realized: A new body type does not require changes outside the factory and the new subclass; ownership is expressed through std::unique_ptr (RAII); and DataRepository never includes Star.h, Moon.h, etc. in its header.

2. Observer (Behavioral)

Intent: Allow several UI components to react to a change in the currently selected celestial body without each component knowing about the others.

Participants: SelectionManager (subject), SelectionObserver (interface), InfoPanel and SearchBar (concrete observers).

```
// observers/SelectionManager.cpp
void SelectionManager::setSelected(CelestialBody *body) {
    current_ = body;
    for (auto *o : observers_) o->onSelectionChanged(body);
}

// ui/InfoPanel.cpp
InfoPanel::InfoPanel(SelectionManager &mgr, QWidget *parent)
    : QWidget(parent), selMgr_(mgr) {
    buildUI();
    selMgr_.addObserver(this);   // <-- subscribe
}
```

Benefits realized: When the user clicks a planet in SolarScene, calls selMgr_.setSelected(body) trigger the panel update, the side-menu close, and the camera zoom. All without SolarScene knowing those receivers exist. New observers can be added without changing the publisher.

3. Singleton (Creational)

Intent: Provide a single, lazily initialized point of access to long-lived service objects whose state must be shared across the entire UI.

Participants: DataRepository (catalogue of all bodies), NotesManager (user notes), QuizManager (question pool).

```
// data/DataRepository.cpp
DataRepository &DataRepository::instance() {
    static DataRepository r;   // Meyers's singleton: thread-safe in C++11+
    return r;
}
```

Justification: Each manager is genuinely application-global, that means the catalogue of bodies and the user's notes have one canonical instance, and the application is a single-process desktop GUI without unit-test infrastructure where the cost of singletons would dominate. The Meyers form (function-local static) is thread-safe per the C++11 standard and avoids the classical double-checked-locking pitfalls.

4. Strategy (Behavioral)

Intent: Encapsulate the rendering algorithm for celestial bodies so that the scene can swap between an overview style (Render2D) and a close-up style (DetailRender) without conditional logic spread across the painter.

Participants: RenderStrategy (interface), Render2D and DetailRender (concrete strategies), SolarScene (context).

```
// strategies/RenderStrategy.h
class RenderStrategy {
public:
    virtual ~RenderStrategy() = default;
    virtual void drawBody(QPainter &p, CelestialBody *b,
                          const QPointF &c, double r) = 0;
    virtual void drawOrbit(QPainter &p, const QPointF &c, double r) = 0;
    virtual std::string name() const = 0;
};

// ui/SolarScene.h (excerpt)
std::unique_ptr<RenderStrategy> renderer_;
void setRenderStrategy(std::unique_ptr<RenderStrategy>);
```

Wiring at runtime: SolarScene owns the active strategy via *std::unique_ptr<RenderStrategy>*. PlanetItem::paint resolves its scene with a dynamic_cast and delegates every repaint to *scene->renderer()->drawBody(...)*. The toolbar exposes a View Mode combo box; selecting an item invokes *scene_->setRenderStrategy(std::make_unique<...>())*, which swaps the renderer and triggers a redraw. The two concrete strategies produce visibly different scenes:

1. **Render2D** — the default stylized look. Each body keeps its hand-tuned decorations: Saturn's ring system, Uranus's vertical ring, Jupiter and Saturn cloud bands, Jupiter's Great Red Spot, Earth continents and clouds, the Mars polar ice cap, and the Sun's glow and corona.
2. **DetailRender** — an information-overlay mode aimed at learning. Each body is drawn as a simple shaded sphere but is annotated with a category badge above (e.g. "GAS GIANT," "TERRESTRIAL PLANET," "NATURAL SATELLITE"), a larger bolded name, and a distance line below ("57.9 M km from Sun," "384 K km from Earth," "547 km altitude") so that the visual itself teaches the user about every body without requiring them to click.

This is a textbook application of Strategy: the algorithm that turns a CelestialBody into pixels varies, the rest of the system does not, and the swap happens at runtime by replacing one object.

5. Template Method (Behavioral)

Intent: Define a fixed set of operations that every celestial body must support, declared as pure virtual methods in the abstract base class, so that all UI consumers can interact with any body type through a single uniform interface without knowing the concrete type.

Participants: CelestialBody (abstract class defining the template), Planet / Moon / Star / Satellite (concrete subclasses implementing the steps).

```
// models/CelestialBody.h
class CelestialBody {
public:
    virtual std::string getDescription()   const = 0;
    virtual std::string getCategory()      const = 0;
    virtual double getDisplayRadius() const = 0;
};

// models/Star.cpp — fills in its own version
std::string Star::getDescription() const {
    return name_ + " is a " + spectralClass_ + " star with luminosity "
           + std::to_string(luminosity_) + " L☉.";
}
std::string Star::getCategory()      const { return "Star"; }
double Star::getDisplayRadius() const { return 55.0; }

// models/Satellite.cpp — completely different answer, same interface
std::string Satellite::getDescription() const {
    return mission_ + " — operated by " + agency_;
}
std::string Satellite::getCategory()      const {
    return active_ ? "Active Satellite" : "Inactive Satellite";
}
double Satellite::getDisplayRadius() const { return 3.5; }
```

Benefits realized: Every UI consumer — InfoPanel, DetailDialog, SideMenuPanel, SearchBar — calls the same three methods on a CelestialBody* and gets the correct answer regardless of the concrete type underneath. There are no if (type == STAR) chains anywhere in the UI layer. Adding a new body type means implementing these three methods in the new subclass; the entire UI works with it immediately without any modifications. This is genuine polymorphism — the base class defines what must be answered, each subclass decides how to answer it.

5. Other Architectural Idioms

- **Model–View separation**. The models package contains pure-domain classes; the ui package contains all Qt-specific code. This is a recognized architectural style (the M and V of MVC) and is an explicit consequence of the SRP analysis above.
- **Repository pattern**. DataRepository hides the underlying std::vector<unique_ptr<CelestialBody>> and exposes domain-specific queries: planets(), moons(), moonsOf(parent), satellites(), search(query), findByName(name).
- **Signals and Slots (Qt idiom)**. Used pervasively for inter-widget communication. Conceptually a typed, decoupled Observer; pragmatically the language Qt expects for GUI work.

Key Subsystems Walkthrough

1. Animated Solar Scene

SolarScene (ui/SolarScene.cpp) extends QGraphicsScene. On construction it builds a 11000×11000 scene populated with:

- 600 near-field stars and 1000 far-field stars at randomized positions and brightnesses, drawn behind everything.
- Six emission nebulae composed of four stacked radial gradients each, tagged with names such as Orion and Eagle.
- Four background galaxies, each rendered as a tilted glow plus three logarithmic spiral arms drawn as Bezier paths.
- Eight orbit ellipses, then the Sun, then the eight planets, then their moons, then Earth-orbiting satellites with their own dotted orbit lines.

A QTimer fires every 33 ms and SolarScene::advanceAnimation increments each planet's orbital angle proportionally to its physically motivated speed array. Moons are positioned relative to whichever planet currently owns them, so when Earth moves, the Moon and the ISS follow in real time.

2. Quiz Subsystem

QuizManager generates a fresh quiz on every invocation. The pool of 96 questions is shuffled with a Mersenne-Twister seeded from the steady clock, the first 50 are taken, and then for each question the four options are independently shuffled while the correctIndex is rebound to the new position of the original answer. This guarantees each session is novel and resists answer-pattern memorization.

```
// data/QuizManager.cpp (excerpt)
std::shuffle(questions.begin(), questions.end(), rng);
```

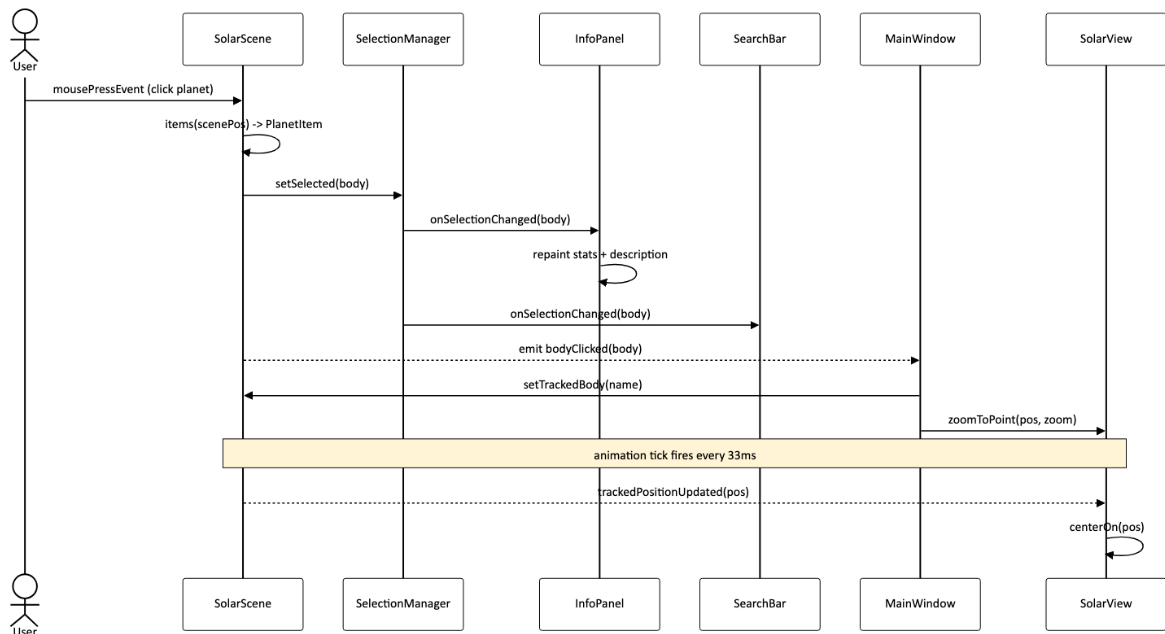
```
if (questions.size() > 50) questions.resize(50);
for (auto &q : questions) {
    std::string correct = q.options[q.correctIndex];
    int indices[] = {0,1,2,3};
    std::shuffle(std::begin(indices), std::end(indices), rng);
    /* permute options in place, then locate "correct" again */
}
```

3. Notes Subsystem

NotesDialog presents three tabs (Per-Body, All Notes, Bookmarks) backed by the NotesManager singleton. Each note carries a monotonically increasing id, a body reference, title, content, an automatically captured timestamp (“%Y-%m-%d %H:%M”), and a bookmark flag. The dialog supports add, edit, delete (with confirmation), and bookmark toggling.

Sequence Diagram (User Interaction Flow)

The following sequence captures what happens when the user clicks a planet in the scene. It illustrates how the Observer pattern and the Qt signal-slot mechanism collaborate to keep the UI consistent.



A click in the scene triggers Observer notifications first, then a bodyClicked signal whose slot drives camera tracking. UI components stay decoupled — SolarScene never references InfoPanel or SearchBar directly.

Reflection, Trade-offs, and Future Work

1. What Worked Well

- The folder-per-package layout made the code easy to navigate during the rapid iteration of UI features, and made the SOLID story straightforward to articulate.
- The Observer pattern proved decisive when adding the side menu and the search bar: each was wired in with two lines of glue (addObserver / emit) and zero changes to existing classes.
- Funnelling all object construction through a single factory eliminated a class of bugs in which a new field on Planet would be forgotten in a second creation site.

2. Trade-offs

- Three singletons were chosen over a dependency-injection container. The cost is testability where a future xUnit harness will need a way to substitute mocks. The benefit was speed of development for a single-user desktop application.
- Notes are kept in-memory only. Persistence to ~/.solarexplorer/notes.json is the natural next addition; the existing NotesManager API is already shaped for it.

3. Future Work

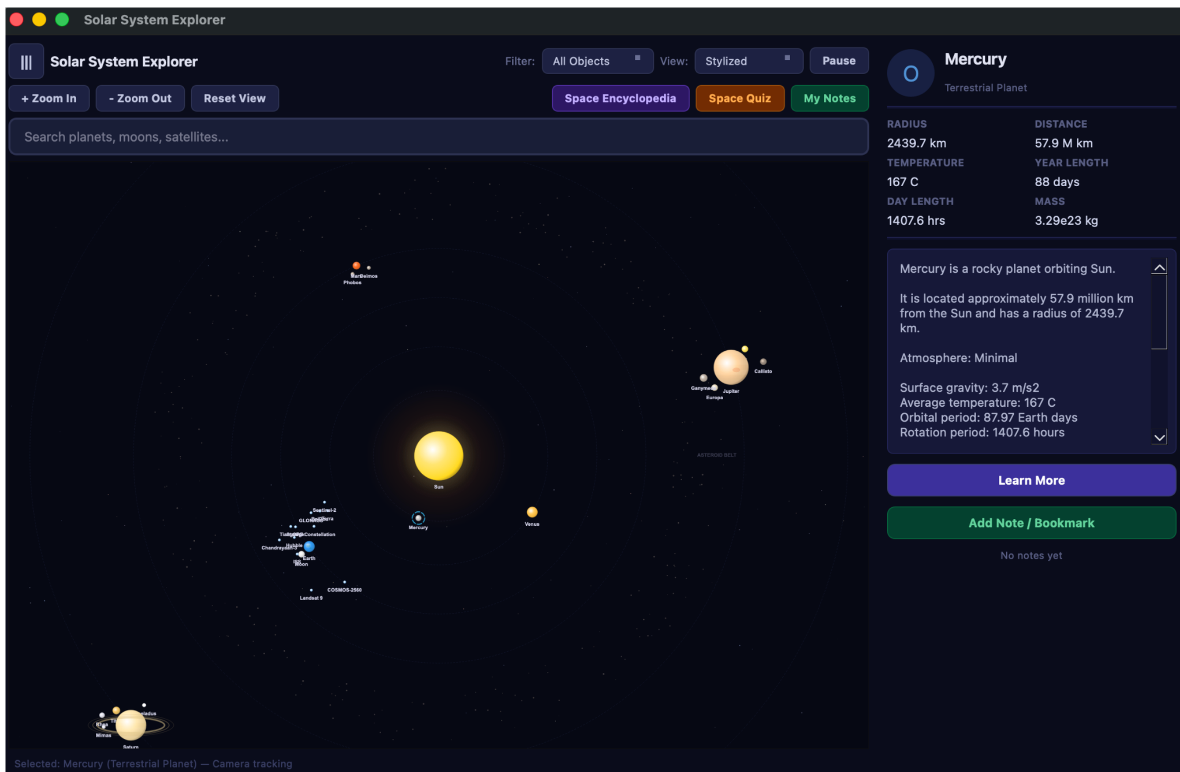
1. Persist NotesManager to JSON on shutdown and load on startup.
2. Extract MainWindow’s inline Encyclopedia dialog into ConceptsDialog for symmetry with NotesDialog and DetailDialog.
3. Move quiz questions and body data into JSON files under data/ so non-programmers can extend content.
4. Add a 3D rendering strategy that uses Qt 3D, taking advantage of the existing Strategy abstraction without other code changes.

Conclusion

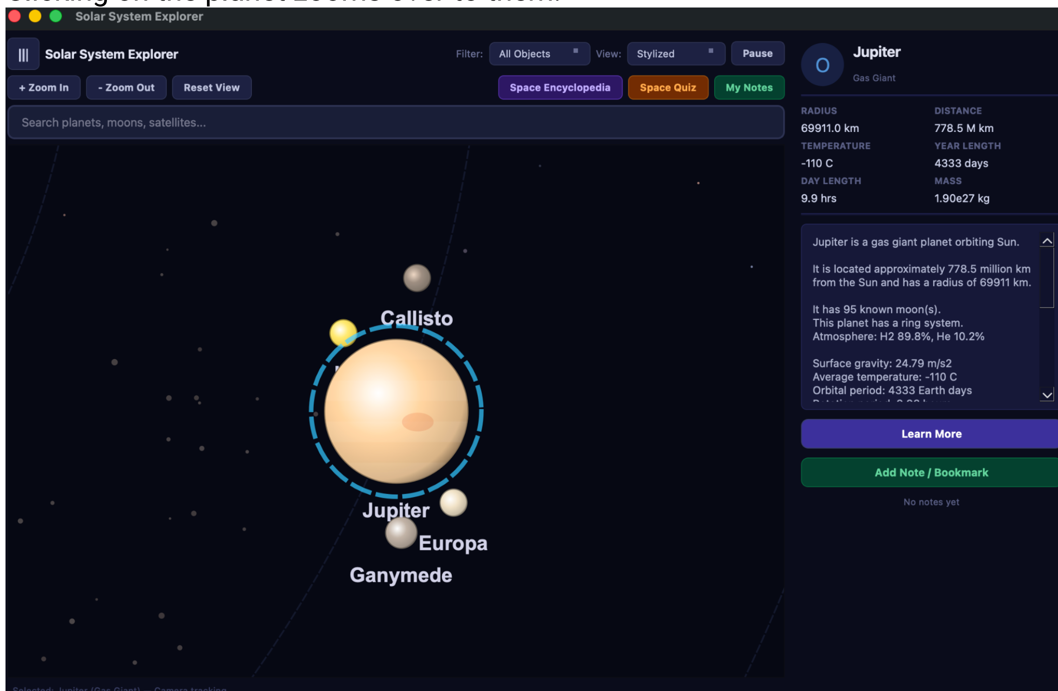
Solar System Explorer satisfies the four hard constraints of the assignment that is substantial scope, demonstrable use of design principles, demonstrable use of design patterns, and a Qt6/C++ GUI with deliberate choices at every layer. The model package is pure, the factory localizes construction, the observer wiring keeps UI components decoupled, and the strategy interface leaves room for a richer renderer without disturbing client code. The decisions taken were guided by SOLID, and the trade-offs are recorded honestly above so they can be revisited.

Screenshots

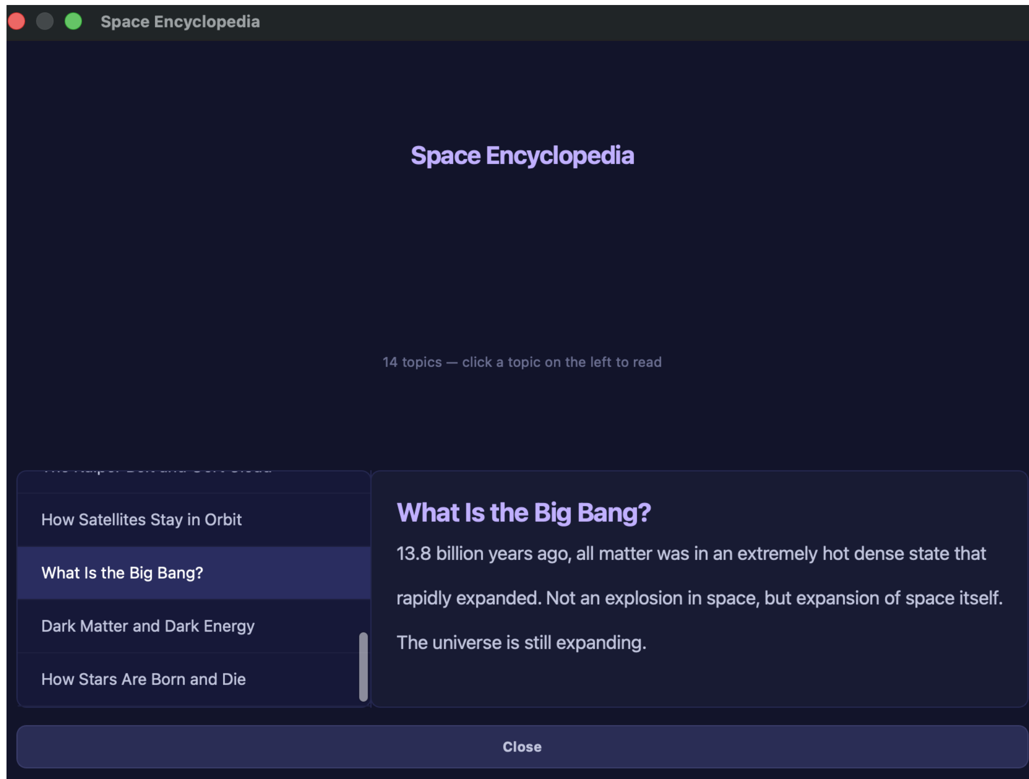
Main Home Screen where you can explore the planets by hovering over them. You can “pause” the time and space. Click on any specific object to see more details on the right panel!



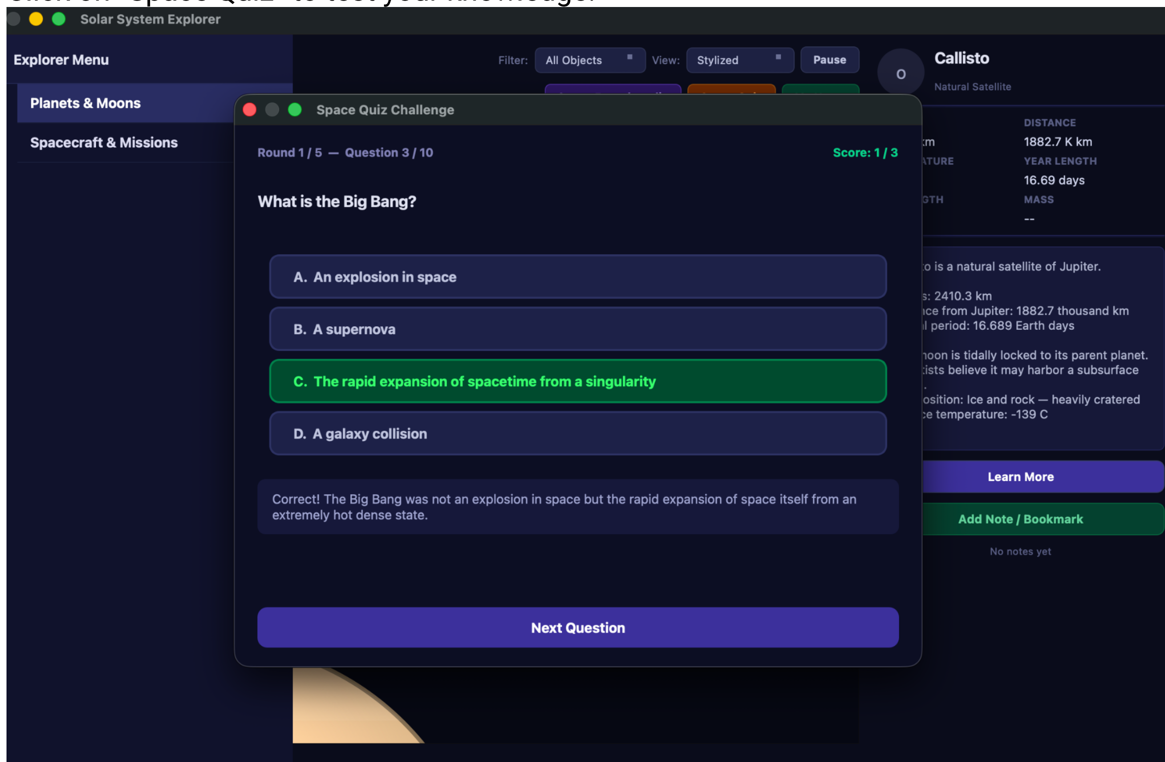
Clicking on the planet zooms over to them.



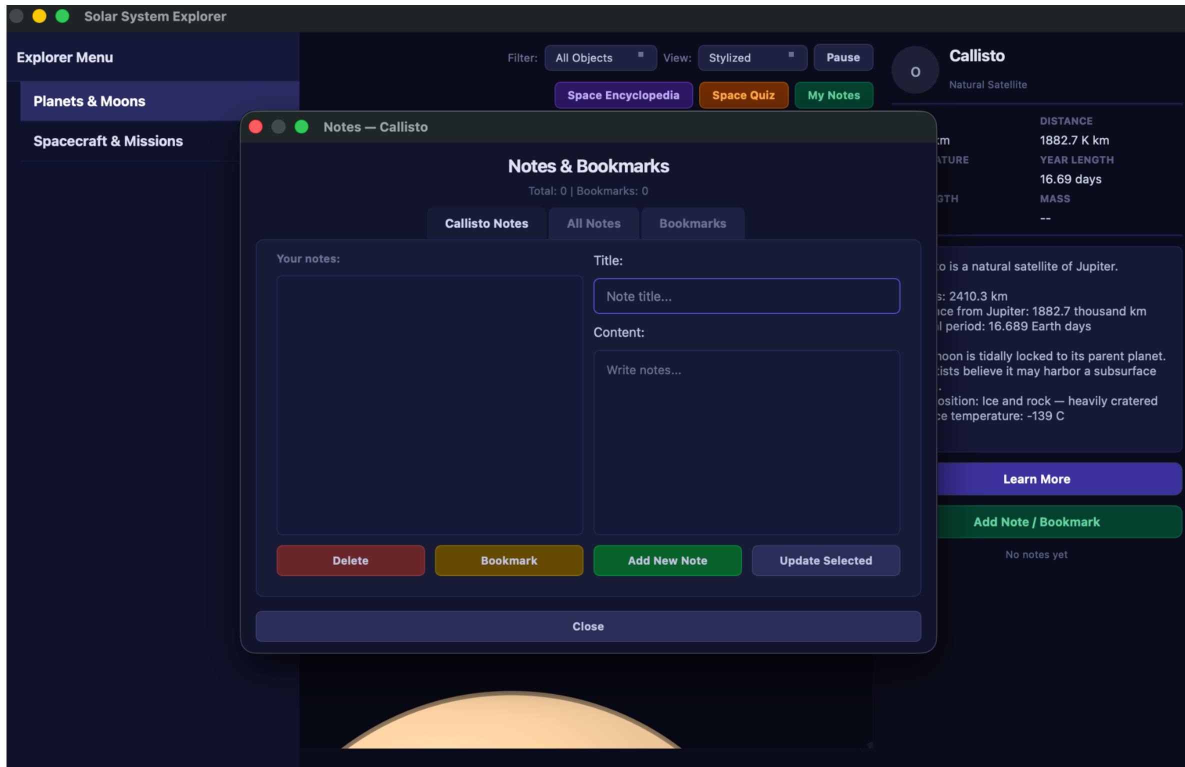
Click on “Space Encyclopedia” to learn more about our universe and its history (that we currently know about)



Click on “Space Quiz” to test your knowledge!



Add notes and bookmarks to the objects to refer to later. Maybe you found something that has not been uncovered yet. Who knows?


Solar System Explorer
Explorer Menu
Planets & Moons
Spacecraft & Missions
Filter: All Objects View: Stylized Pause
Space Encyclopedia Space Quiz My Notes
Callisto
Natural Satellite
Notes — Callisto
Notes & Bookmarks
Total: 0 | Bookmarks: 0
Callisto Notes All Notes Bookmarks
Your notes:
Title:
Note title...
Content:
Write notes...
Delete Bookmark Add New Note Update Selected
Close
DISTANCE
1882.7 K km
YEAR LENGTH
16.69 days
MASS
--
Learn More
Add Note / Bookmark
No notes yet