Solar System Explorer

An Interactive C++ / Qt6 Educational Application

C++17 | QT6 WIDGETS | QGRAPHICSSCENE | CMAKE

2,423 LINES | 45 FILES | 6 PACKAGES | 5 DESIGN PATTERNS

What is Solar System Explorer?

A multi-feature desktop GUI visualizing our solar system in an animated 2D scene. Navigate celestial bodies, explore planetary data, and test knowledge with integrated quizzes. Built in C++17 using Qt6 Widgets, organized into six architectural layers.

Animated 2D Solar System

Sun, 8 planets, 15 moons, 22 satellites. Custom radial gradients, Saturn rings, Jupiter cloud bands, Earth continents.

Birds Eye View

Scroll-wheel zoom, right-click pan, click-to-track, automatic zoom-to-body.

Explorer Side Menu

Hierarchical Sun→Planet→Moon and Spacecraft→Agency→Mission with smooth slide-in animation.

Info Panel

Radius, distance, temperature, mass, orbital period, description, fun facts — updates reactively on selection.

Space Encyclopedia

14 educational topics: eclipses, gravity, Big Bang, dark matter, light-years and more.

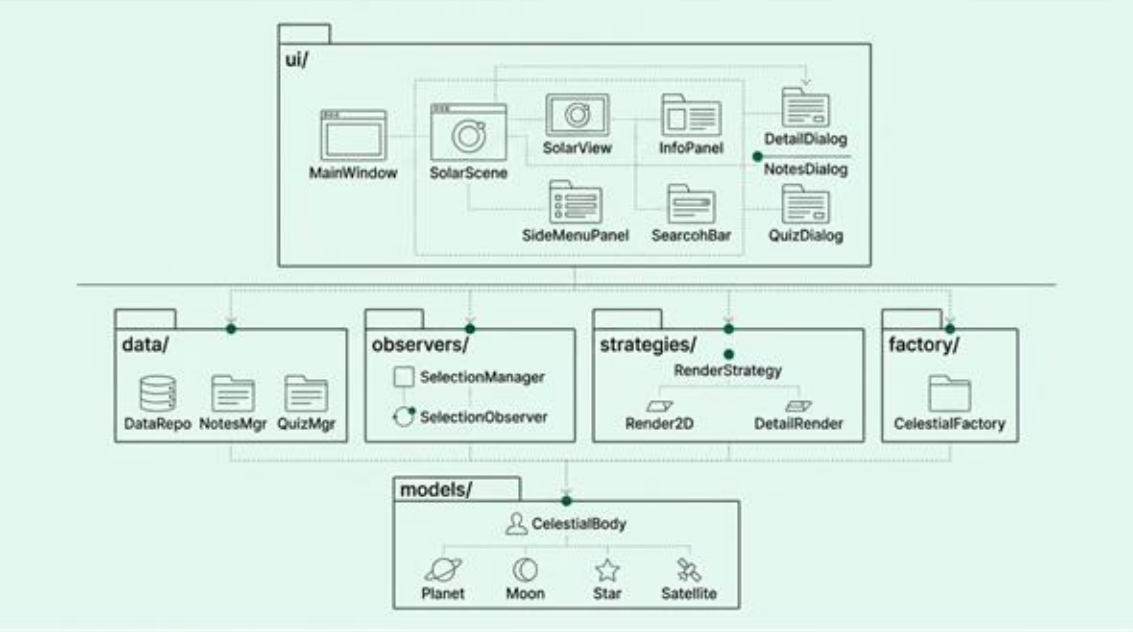
Notes & Quiz

Personal notes with CRUD and bookmarking. 50-question quiz from a pool of 96, shuffled every session using Mersenne-Twister.

ARCHITECTURE

Layered Architecture — Package Diagram

A strict layered architecture where dependencies flow only downward. Upper layers depend on lower layers; lower layers never import upper ones.



- Dependencies flow only downward
- `ui/` layer sits at the top
- `models/` layer sits at the bottom
- Six packages total across the architecture
- Clear separation of concerns between layers
- Lower layers stay independent of upper layers

DEMO

Gang-of-Five Patterns

Each pattern solves a specific architectural challenge — from centralised construction to runtime algorithm swapping.

Factory Method

Creational — Centralise construction of `CelestialBody` subtypes. Callers never depend on concrete constructors.

Observer

Behavioral — Notify UI components when a body is selected. Zero direct references between scene and panels.

Singleton

Creational — Single global instances for `DataRepository`, `NotesManager`, `QuizManager`.

Strategy

Behavioral — Swap rendering algorithm at runtime between **Stylized** and **Detail** view.

Template Method

Behavioral — CelestialBody skeleton defines getDescription(), getCategory(), getDisplayRadius() as pure virtual; subclasses fill in the steps.

SINGLETON · CREATIONAL

Singleton Pattern

Provide a single, lazily-initialized global point of access to long-lived service objects shared across the entire UI.

DataRepository

Catalog of all 47 celestial bodies. Exposes `planets()`, `moons()`, `satellites()`, `search()`, `findByName()`.

NotesManager

User notes CRUD — `addNote`, `updateNote`, `deleteNote`, `toggleBookmark`, 3 view modes.

QuizManager

Assembles shuffled 50-question sets from a pool of 96 using Mersenne-Twister seeded from `steady_clock`.

```
// data/DataRepository.cpp
// Meyers Singleton — thread-safe C++11
DataRepository &DataRepository::instance() {
  static DataRepository r; // init once
  return r;
}

// Usage anywhere in the codebase:
auto &repo = DataRepository::instance();
for (auto* p : repo.planets()) { /* ... */ }
```

Singletons reduce testability (harder to mock). Meyers form avoids double-checked-locking pitfalls and is thread-safe per C++11.

OBSERVER · BEHAVIORAL

Observer Pattern

Allow multiple UI components to react to a body selection change without any component knowing about the others.

Participants

- **SelectionManager** — subject, maintains list of observers
- **SelectionObserver** — abstract interface with single method
- **InfoPanel** — concrete observer (updates stats panel)
- **SearchBar** — concrete observer (clears search)

Benefits

- SolarScene never references InfoPanel or SearchBar directly — fully decoupled
- Adding a new UI component = implement the 1-method interface + call `addObserver()`
- Zero changes to existing classes
- Wiring = just 2 lines of glue per observer

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

// ui/InfoPanel.cpp — subscribe on construction
InfoPanel::InfoPanel(SelectionManager &mgr,
 QWidget *parent)
 : QWidget(parent) {
 buildUI();
 mgr.addObserver(this); // register
}
```

FACTORY METHOD · CREATIONAL

Factory Method Pattern

Centralize construction of CelestialBody subtypes so callers never depend on concrete constructors or include concrete headers.

Participants

- **CelestialFactory** — creator
- **CelestialBody** — abstract product
- **Planet, Moon, Star, Satellite** — concrete products
- **DataRepository** — client

Benefits

- Adding a new body type = 1 new class + 1 switch case — zero changes to any UI code
- Ownership via `std::unique_ptr` (RAII — no memory leaks)
- DataRepository never `#includes` concrete headers
- Directly supports the Open/Closed Principle

```
// 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;
 }
}
```

Template Method Pattern — Behavioral

Define the algorithm skeleton in the base class; subclasses fill in the steps without changing the overall flow.

Participants

- **CelestialBody** — abstract base class; defines the template
- **Planet, Moon, Star, Satellite** — concrete subclasses; implement the steps

How it works here

All UI consumers call the same three methods on any `CelestialBody*`: `getDescription()`, `getCategory()`, and `getDisplayRadius()`.

The base class enforces the contract; the UI stays ignorant of the concrete type.

Benefits

- Type-agnostic UI across all celestial bodies
- Add a new body type by implementing the 3 methods

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

// subclass examples:
// Star -> "G-type main sequence star..."
// Planet -> "Gas giant orbiting Sun..."
// Satellite -> "Active spacecraft..."
```

STRATEGY · BEHAVIORAL

Strategy Pattern

Encapsulate rendering algorithms so SolarScene can swap between Stylized and Detail view at runtime without conditional logic spread through the painter.

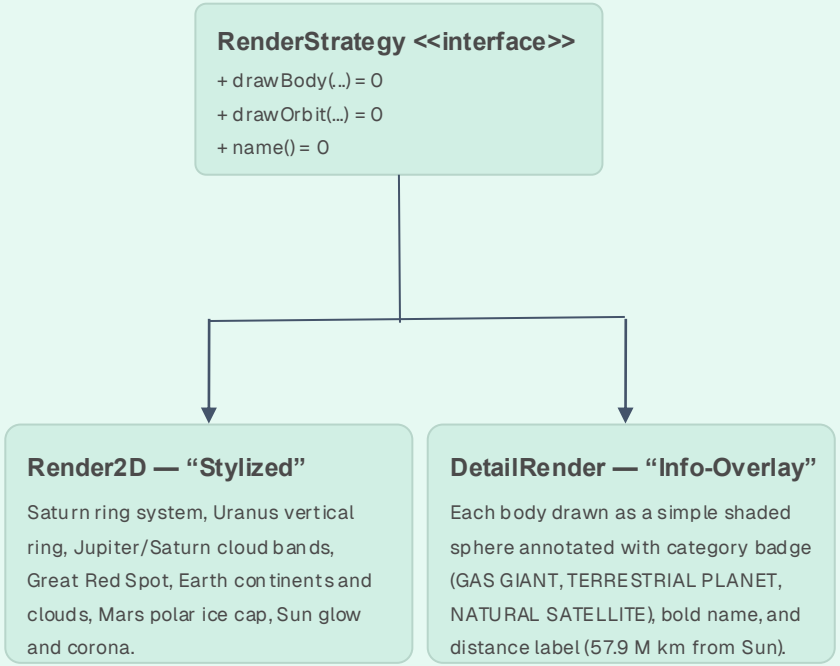
Participants

- **RenderStrategy** — abstract interface
- **Render2D** — stylized look
- **DetailRender** — info-overlay mode
- **SolarScene** — context holding `unique_ptr<RenderStrategy>`

```
// 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;
};
// Swap at runtime via toolbar toggle:
scene_->setRenderStrategy(std::make_unique<DetailRender>());
```

Defensive Programming

```
auto *s = dynamic_cast<SolarScene *>(scene());
    if (s && s->renderer())
        s->renderer()->drawBody(...)
```



SOLID PRINCIPLES

SOLID Design Principles Applied

S — Single Responsibility

Each class has one reason to change.

NotesManager = notes CRUD only.

QuizManager = question shuffling only.

InfoPanel = display selected body only.

RenderStrategy = defines how to render bodies..

O — Open / Closed

Open for extension, closed for modification.

RenderStrategy = we can add new rendering modes WITHOUT modifying existing code.

CelestialBody = InfoPanel, SearchBar, and renderers work with any CelestialBody subtype, no changes needed when adding new types

L — Liskov Substitution

Subtypes must be substitutable for their base types.

CelestialBody = Any CelestialBody subtype can be used wherever CelestialBody* is expected.

```
void PlanetItem::paint(QPainter *painter, ...) {
// body_ could be Planet*, Moon*,
Star*,Satellite*
  s->renderer()->drawBody(*painter, body_, ...);
}
```

I — Interface Segregation

Clients should not be forced to depend on interfaces they don't use.

SelectionObserver= Only selection events, no animation/network/data.

RenderStrategy = Only rendering, no UI/data/events.

D — Dependency Inversion

High-level modules depend on abstractions, not concrete implementations.

SolarScene holds unique_ptr<RenderStrategy> (not Render2D directly).

SelectionManager works against SelectionObserver interface.

DataRepository uses CelestialFactory (not new Planet directly).

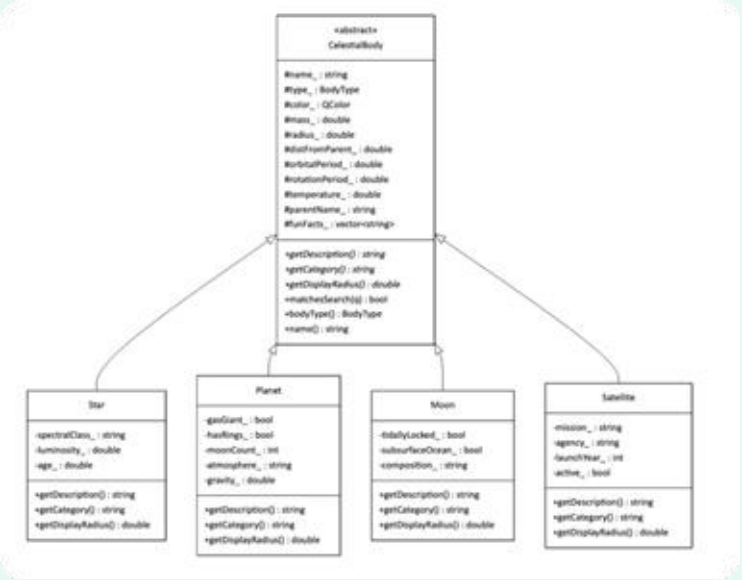
DOMAIN MODEL

UML — CelestialBody Hierarchy

The abstract `CelestialBody` base class defines the shared contract — name, mass, radius, orbital period, temperature, and three pure virtual methods. Four concrete subclasses extend it with domain-specific attributes.

- **Star** — spectrum, luminosity, age
- **Planet** — gasGiant, rings, moons, atmosphere, gravity
- **Moon** — tideLocked, ocean, composition
- **Satellite** — mission, agency, launchYear, active

All classes honor the Liskov Substitution Principle — `getDescription()` always returns a non-empty string, `getDisplayRadius()` always returns a positive double.



Thank You