GNUstep Architecture: From Conceptual to Concrete

Group 10

Group 10 Video

https://youtu.be/t4O--A5dBbA

Group 10

- Chuka Uwefoh – Team lead
- Lore Chiepe - Presenter
- Henry Chen
- James Wang - Presenter
- Ryan Jokhu
- Zachary Stephens

Preview

1. Introduction and Overview
2. Derivation Process
3. Conceptual Architecture
4. Concrete Architecture
    - i. High-Level Architecture
    - ii. Chosen 2nd Level Subsystem – libs-back
5. Key Interaction Diagrams (Use Cases)
6. Reflexion Analysis and Divergences
7. Conclusion

GNU step recap

Objective: GNUstep aims to provide a cross-platform Objective-C framework compatible with OpenStep.

Key Goals:

- Modularity and extensibility.
- Cross-platform support (Linux, Windows, macOS).
- Compatibility with OpenStep and Cocoa APIs.

Audience: Developers, architects, and software engineers.

Derivation Process

- **Structured Process**:
  - Maintain OpenStep compatibility.
  - Ensure modularity and extensibility.
  - Support modern system environments.
- **Key Decisions**:
  - Layered architecture for separation of concerns.
  - Abstraction boundaries for portability.
  - Iterative development for API refinements and optimizations.

Conceptual Architecture

Core Components:

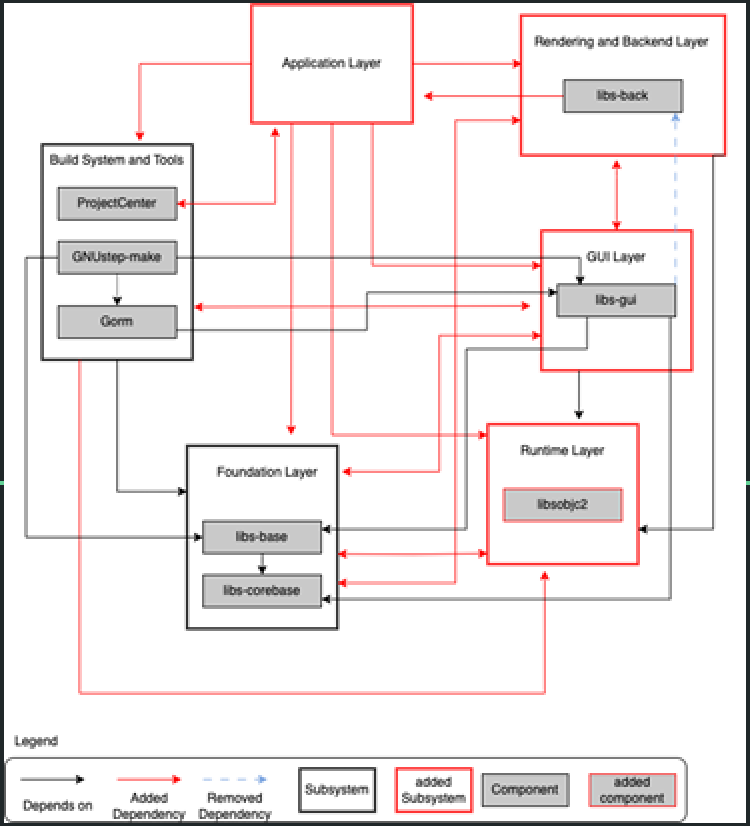
- libs-base and libs-corebase: Fundamental data structures and utilities.
- libs-gui: Graphical elements and user interaction.
- libs-back: Platform-specific rendering.
- Development Tools: Gorm, GNUstep-make, ProjectCenter.

Cross-Platform Compatibility: Ensures applications run smoothly on multiple OSes.

Conceptual vs Concrete

- **Conceptual**:
  - High-level overview of roles and relationships.
  - Focuses on cross-platform compatibility as a distinct layer.
- **Concrete**:
  - Detailed implementation with specific libraries and dependencies.
  - Embeds platform independence within libs-back and libs-base.

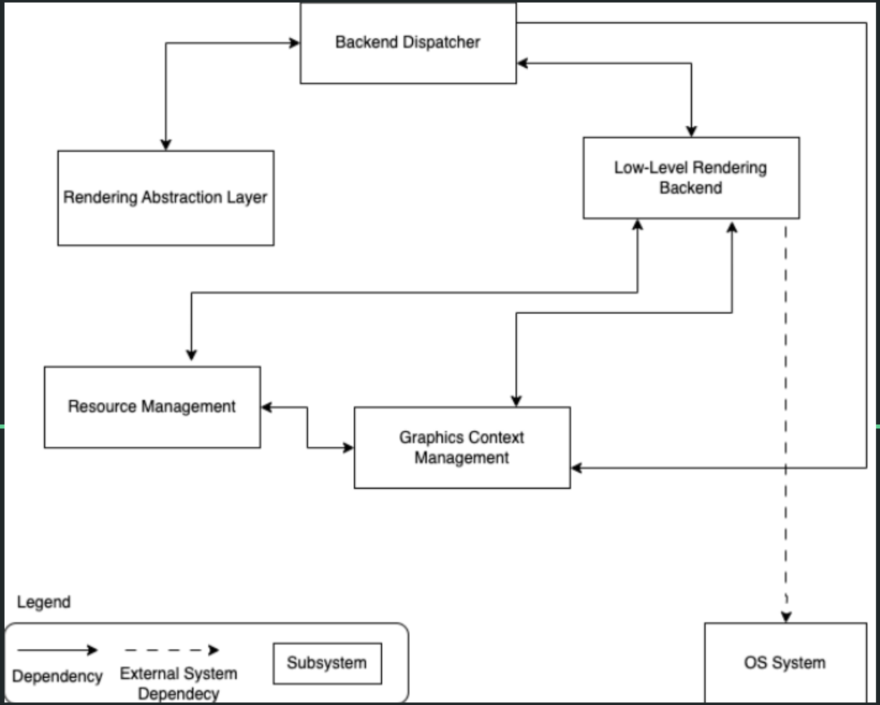
High-Level Concrete Architecture



Layered Structure:

- **Application Layer**: User-developed apps.
- **GUI Layer (libs-gui)**: Manages UI and events.
- **Foundation Layer (libs-base & libs-corebase)**: Core Objective-C classes.
- **Rendering and Backend Layer (libs-back)**: Abstracts platform-specific rendering.
- **Build System and Tools**: Compilation and development support.
- **Runtime Layer (libobjc2)**: Objective-C runtime features.

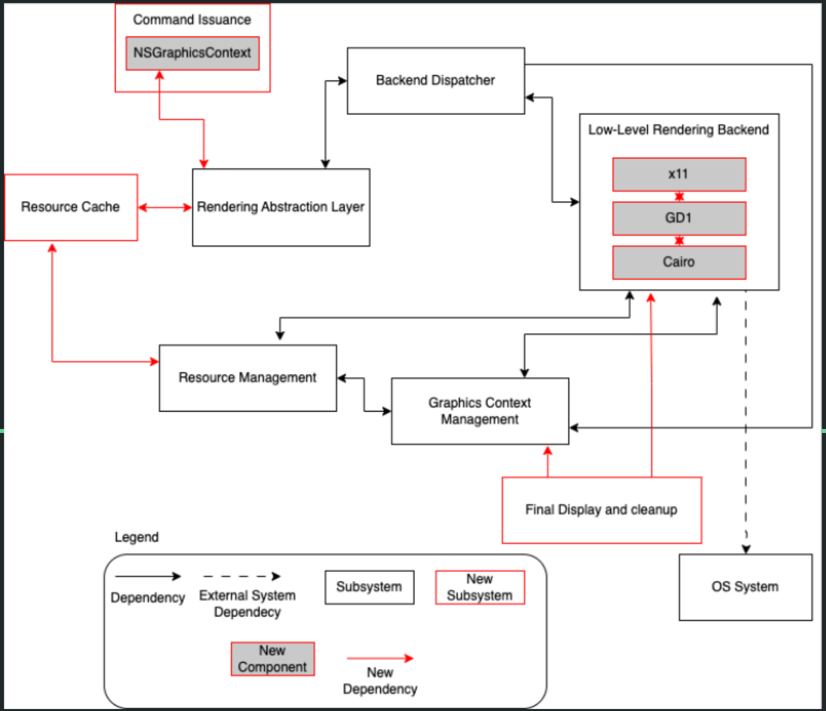
2nd Level Subsystem – libs-back – Conceptual Architecture



Key Concepts:

- Abstraction of platform-specific details
- Focus on efficient rendering

2nd Level Subsystem – libs-back – Concrete Architecture



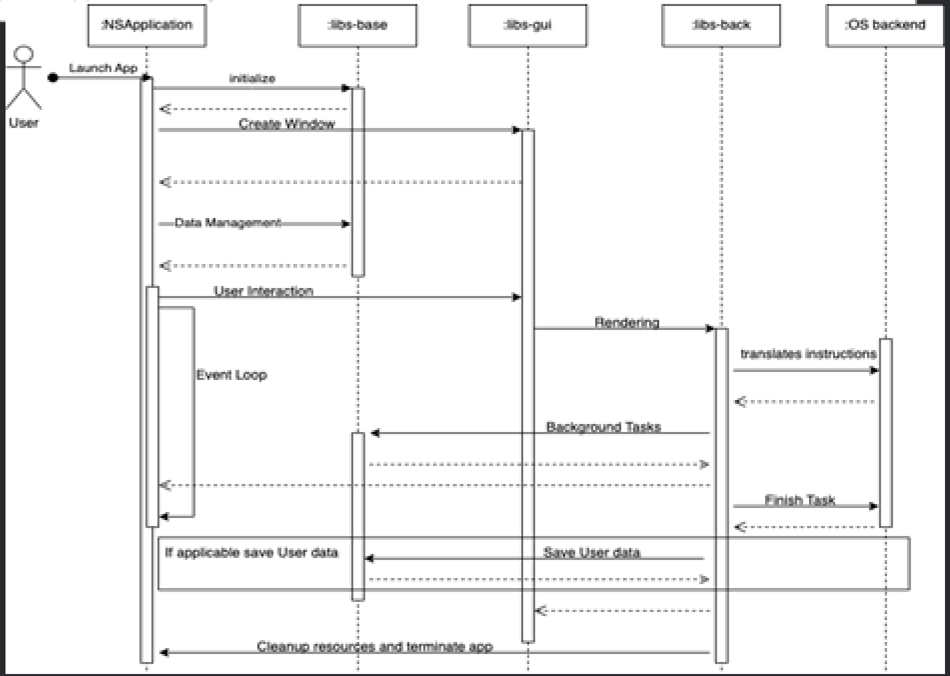
Core Components:

- NSGraphicsContext - Initiates rendering requests
- Rendering Abstraction Layer (RAL) - Standardizes commands
- Backend Dispatcher - Routes to appropriate platform
- Low-Level Rendering Backend - Executes platform-specific rendering

- **Supporting Systems:**
  - Graphics Context Management
  - Resource Management & Cache
  - Final Display & Cleanup

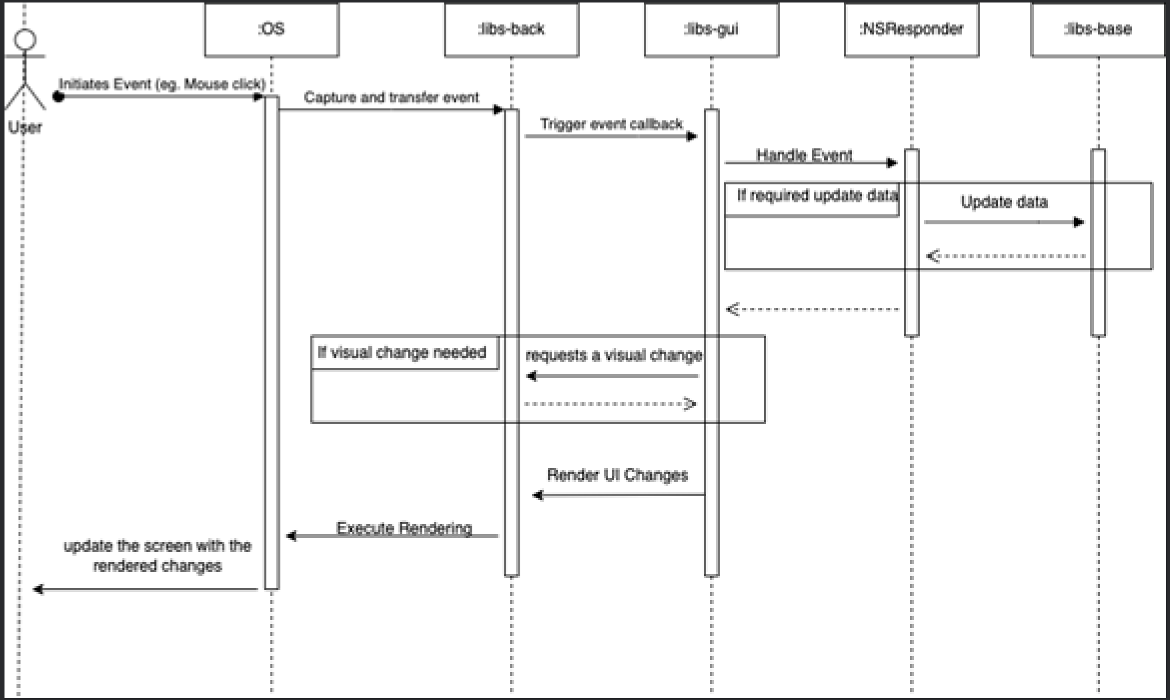
Sequence Diagrams

Figure 3- Application lifecycle use case



1. The user launches a GNUstep application.
2. NSApplication initializes, setting up the event loop.
3. The application interacts with libs-base for data management.
4. libs-gui handles user interactions and passes rendering tasks to libs-back.
5. libs-back translates rendering instructions to the underlying OS backend.

Sequence Diagrams



1. User input is captured by the OS and forwarded to libs-back.
2. libs-back notifies libs-gui components via event callbacks.
3. NSResponder processes the event and updates the interface.
4. If necessary, the application logic updates state via libs-base.
5. The UI is refreshed via libs-gui, using libs-back to render changes.

Reflexion Analysis and Divergences

High-Level Architecture

Expectation Vs. Reality	Rationale
Bidirectional Component Dependencies Expected: Unidirectional dependency (libs-gui → libs-back) Reality: Two-way communication between components	Necessary for event handling and rendering callbacks; makes strict layering impractical

Reflexion Analysis and Divergences

Subsystem - libs-back

Expectation Vs. Reality	Rationale
Bidirectional Dependencies with libs-gui Expected: One-way rendering requests Reality: Two-way communication for updates	Required for real-time updates, event handling, and efficiency on varying platforms
Direct System API Interaction Expected: Simple middleware forwarding Reality: Direct communication with X11, Windows GDI, Cairo	Performance optimization, reduced overhead
Platform-Specific Conditional Code Expected: Unified API with automatic platform handling Reality: Hardcoded platform checks (#ifdef statements)	Easier platform-specific maintenance, reduced code duplication
Decentralized Resource Management Expected: Single shared resource cache Reality: Distributed caching across subsystems	Prevents bottlenecks in multi-threaded rendering

Conclusion

- **Subsystems from conceptual architecture break down into multiple interdependent parts**

- **Maintains layered architectural style alongside:**
  - Split layers
  - Visible application layer
  - High connectivity between subsystems

- **libs-back analysis reveals:**
  - More interconnected structure
  - Addition of smaller subsystems