

# Build System Facade Pattern

## 1 Pattern Name and Classification

**Pattern Name:** Build System Facade

**Classification:** Structural design pattern applied to build and deployment architecture.

The core idea is to introduce a simple, high-level build entry point (the facade) that hides the complexity of an underlying, more powerful build system and related tooling.

In the concrete case considered here:

- **Facade:** A top-level `Makefile` acting as a task runner and command alias layer.
- **Subsystems:** `CMakeLists.txt` (core build system configuration) and Docker (containerized build and runtime environments).

## 2 Intent

Provide a *single, convenient interface* to a *complex build and deployment subsystem*.

The Build System Facade pattern aims to:

- Offer developers simple, memorable commands (e.g., `make wasm-debug`, `make docker-build`).
- Keep the detailed compilation logic, platform detection, dependency management, and optimization flags encapsulated in CMake.
- Centralize container orchestration and workflow automation (e.g., Docker builds and runs) in the `Makefile` without duplicating build logic.

In short, it addresses the problem:

“Our build, toolchain, and environments are complex, but daily developer workflows should be simple, repeatable, and hard to mess up.”

## 3 Also Known As

Common alternative names and closely related labels include:

- Wrapper `Makefile`
- Task Runner over CMake
- Two-Level Build Pattern
- Build Orchestration Facade

## 4 Motivation

Consider a project with:

- Multiple target platforms (e.g., WebAssembly, desktop, mobile).
- Containerized build and runtime environments (e.g., Emscripten builder image, Nginx runtime image).
- A rich CMake configuration handling compiler flags, dependencies, and presets.

A new developer joining the project should *not* need to:

- Remember exact `cmake` invocations for each platform and configuration.
- Know all the details of Docker image tags, volume mounts, and port mappings.
- Manually copy long commands from documentation and risk subtle mistakes.

Instead, the project provides a top-level `Makefile`:

- Targets like `make wasm-debug` or `make native-release` call into CMake with the appropriate presets.
- Targets like `make docker-build` or `make docker-run` encapsulate the Docker workflows.

Behind this facade:

### CoreBuilder (CMake):

Encodes the full knowledge of sources, include paths, compiler standards, optimization levels, platform-specific options (such as Emscripten or native), and dependencies (e.g., SDL, OpenGL).

### ContainerRuntime (Docker):

Provides standard images for building and running the artifacts produced by CMake.

The developer experience becomes:

- Use a small set of Make targets.
- Rely on the facade to orchestrate CMake and Docker correctly.
- Benefit from consistency between local development and CI/CD pipelines.

## 5 Applicability

Use the Build System Facade pattern when:

- The build is non-trivial:
  - Multiple platforms or configurations (debug/release, native/WASM).
  - Cross-compilers or specialized toolchains.
  - Containerized build or runtime environments.
- You want:
  - New developers to get started with one or two simple commands.
  - A single, stable entry point for typical workflows.
  - Less risk of diverging build procedures across developers and CI.

The pattern helps repair or prevent poor designs such as:

- Copy-pasted build commands in documentation that quickly drift out of sync.
- Multiple scripts (one per platform or task) that duplicate file lists and flags.
- Direct use of raw `cmake` and `docker` commands everywhere, leading to inconsistent configurations.

You recognize the need for this pattern when:

- Documentation includes long, fragile command lines for routine tasks.
- Onboarding instructions require manual variation of command arguments depending on OS or configuration.
- Developers frequently break builds because they forget “magic” flags or environment details.

## 6 Structure

At a high level, the Build System Facade pattern organizes the build and deployment architecture into clients, a facade, and underlying subsystems.

# Build System Facade Pattern - Structural View

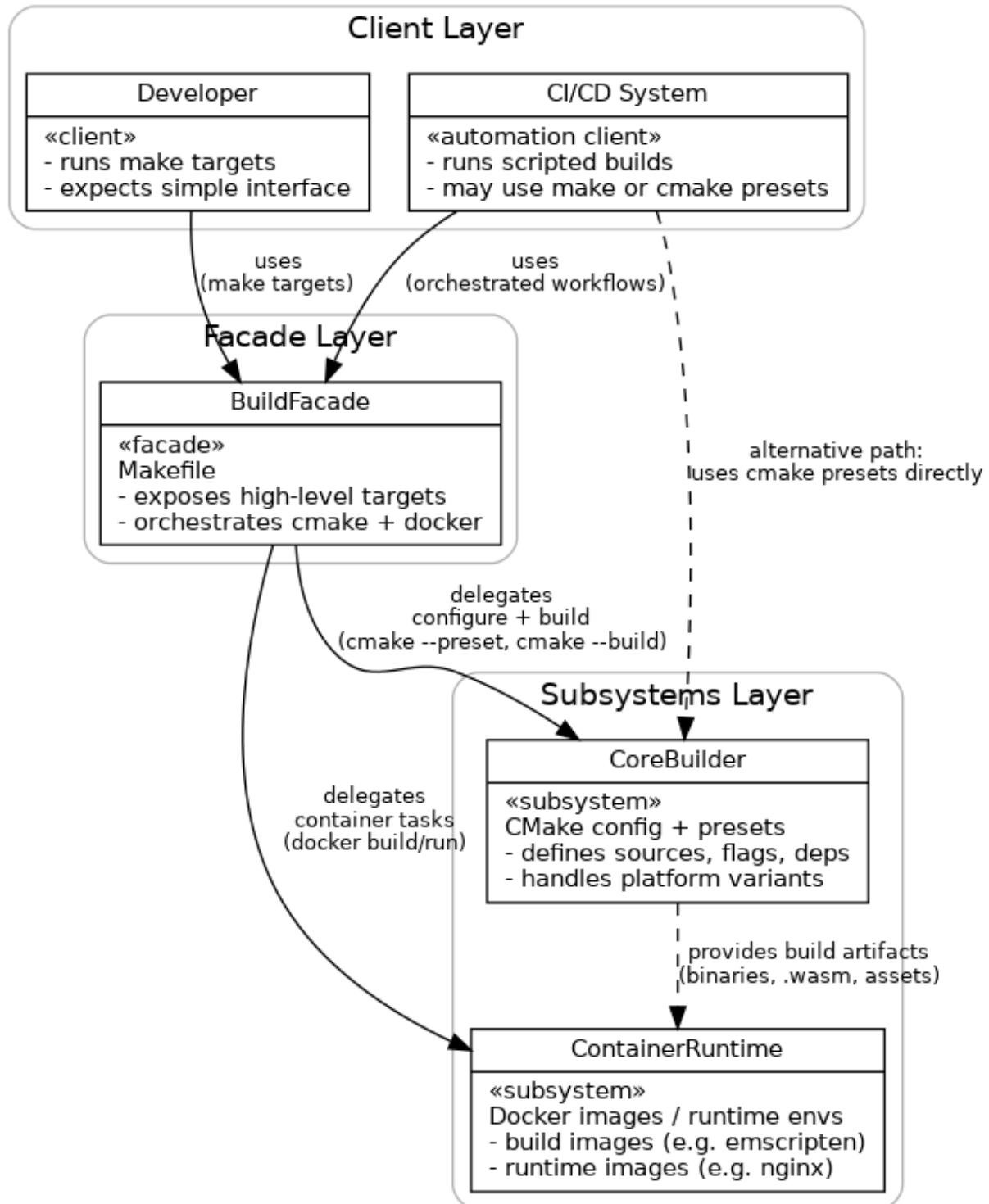


Figure 1: Structural view of the Build System Facade pattern showing clients, facade, and subsystems.

Conceptually, the structure consists of:

- **Developer** — interacts only with the facade.
- **BuildFacade** — a Makefile providing high-level targets.
- **CoreBuilder** — CMake configuration and presets that perform the actual build.
- **ContainerRuntime** — Docker configuration for build and runtime environments.
- **CI/CD System** (optional) — may call the same facade targets or directly use CMake presets.

Typical interaction sequence:

1. Developer runs a Make target, e.g., `make wasm-debug`.
2. The Makefile (BuildFacade) invokes `cmake -preset wasm-debug` and then `cmake -build -preset wasm-debug`.
3. For container workflows, the Makefile invokes `docker build` and `docker run`, referencing the artifacts produced by CMake.

## 7 Participants

### Developer

Triggers builds and workflows via simple commands (e.g., `make wasm-debug`, `make docker-run`). Does not need to know internal arguments for CMake or Docker.

### BuildFacade (Makefile)

A thin, high-level interface exposing targets such as:

- `wasm-debug`, `wasm-release`
- `native-debug`, `native-release`
- `docker-build`, `docker-run`

Delegates to CMake for compilation and to Docker for container tasks, and avoids reimplementing build logic.

### CoreBuilder (CMake configuration)

- Defines source files, include directories, and targets.

- Configures compiler standards and flags (e.g., C23, optimization levels, warnings).
- Encodes platform-specific settings (WASM vs. native, debug vs. release).
- Manages discovery and configuration of dependencies (e.g., SDL, OpenGL).

### ContainerRuntime (Docker configuration)

Provides Dockerfiles and associated configuration for:

- Build environments (e.g., an emscripten builder container).
- Runtime containers (e.g., Nginx-based image serving the WebAssembly application).

Assumes CMake has already produced the appropriate artifacts.

### CI/CD System

- May call the same Make targets, or invoke `cmake -preset` directly.
- Ensures that CI builds follow the same flows as local development.

## 8 Collaborations

- **Developer → BuildFacade:**
  - Developer uses Make targets as the primary interface.
  - The Makefile abstracts away CMake and Docker details.
- **BuildFacade → CoreBuilder:**
  - For configuration: runs `cmake -preset <preset>`.
  - For compilation: runs `cmake -build -preset <preset>`.
- **BuildFacade → ContainerRuntime:**
  - For container workflows: runs `docker build` and `docker run`.
  - Uses known output directories (e.g., `build-wasm/`) defined by CMake.
- **CI/CD System → BuildFacade/CoreBuilder:**
  - CI jobs can reuse the same Make targets or CMake presets.
  - This maintains consistency between local development and CI builds.

## 9 Consequences

### Benefits

- **Simplified developer experience:**
  - Developers interact with a small, stable set of commands.
  - Onboarding is faster and less error-prone.
- **Single source of truth for compilation:**
  - CMake owns source lists, flags, and platform-specific logic.
  - The Makefile does not duplicate compile-time configuration.
- **Separation of concerns:**
  - CMake handles compilation and configuration.
  - Makefile handles orchestration and task-level workflows.
  - Docker handles environment isolation and deployment/runtime concerns.
- **Improved CI/local symmetry:**
  - CI pipelines can call the same presets or Make targets.
  - Reduces “works on my machine” discrepancies.

### Trade-offs

- Two layers must be maintained:
  - CMake configuration (the core build logic).
  - Makefile facade (the user-facing command surface).
- If not carefully managed, the Makefile can accumulate logic that belongs in CMake:
  - Duplicate compiler flags.
  - Diverging platform conditionals.
  - Repeated file lists or build rules.

- Requires discipline to keep the facade thin and declarative, mostly delegating to CMake and Docker rather than re-implementing logic.

The pattern enables the project to vary build toolchain specifics and container details independently of the developer interface. CMake presets, Docker images, or target platforms can change without affecting the top-level commands that developers use day-to-day.

## 10 Implementation

Key guidelines for implementing the Build System Facade pattern:

### 1. Keep the Facade Thin

- The Makefile should primarily:
  - Call `cmake -preset` and `cmake -build -preset`.
  - Invoke `docker build` and `docker run` as needed.
- Avoid embedding complex logic, conditionals, or duplicated knowledge that belongs inside CMake.

### 2. Treat CMake as the Authority

- Place all source file lists, include directories, compiler flags, and definitions in `CMakeLists.txt` or related CMake modules.
- Configure platform-specific options and toolchain files through CMake presets.
- The Makefile should reference only the names of these presets.

### 3. Use Stable, Intuitive Targets

- Define clear targets such as:
  - `setup`, `build-wasm`, `build-native`
  - `docker-build`, `docker-run`
- Document these commands in the project's `README.md`.

### 4. Integrate Docker Carefully

- Keep Docker-specific steps in dedicated Make targets.
- Reference the build outputs (e.g., executables or `.wasm` files) produced by CMake, using known build directories such as `build-wasm/`.

### 5. Language- and Toolchain-Specific Considerations

- For C/C++ projects:
  - Set language standards (e.g., C23 or C++23) in CMake.
  - Configure optimization and warning levels centrally.
- For CI:
  - Prefer calling the same presets or Make targets used locally.
  - This ensures a unified build pipeline from local machines to CI.

## 11 Sample Code

### Makefile (Facade)

```
.PHONY: wasm-debug native-debug docker-build docker-run

wasm-debug:
    cmake --preset wasm-debug
    cmake --build --preset wasm-debug

native-debug:
    cmake --preset native-debug
    cmake --build --preset native-debug

docker-build:
    docker build -t wasm-game .

docker-run:
    docker run --rm -p 8080:80 wasm-game
```

### CMake Presets (CoreBuilder)

```
{
  "version": 3,
  "configurePresets": [
    {
      "name": "wasm-debug",
      "generator": "Ninja",
      "binaryDir": "build-wasm",
      "cacheVariables": {
        "CMAKE_BUILD_TYPE": "Debug",
        "CMAKE_TOOLCHAIN_FILE": "/path/to/Emscripten.cmake"
      }
    },
    {
      "name": "native-debug",
      "generator": "Ninja",
      "binaryDir": "build-native",
      "cacheVariables": {
        "CMAKE_BUILD_TYPE": "Debug"
      }
    }
  ]
}
```

In practice, developers only interact with the Make targets; the underlying CMake presets and Docker configurations remain encapsulated behind the facade.

## 12 Known Uses

The pattern is common in:

- Projects that:

- Standardize on CMake for cross-platform builds.
- Provide a small top-level `Makefile` for convenience.
- Game engines and graphics frameworks that:
  - Need both WebAssembly and native builds.
  - Use containerized toolchains (e.g., an Emscripten Docker image).
- Infrastructure tools and libraries that:
  - Require consistent build and test flows across many environments.
  - Want one “developer command surface” but multiple internal pipelines.

Any project using a `Makefile` layer over CMake and Docker in this manner is effectively applying the Build System Facade pattern.

## 13 Related Patterns

### **Facade Pattern (GoF)**

The direct conceptual inspiration: the `Makefile` serves as a facade over the complex subsystems of CMake and Docker, providing a simplified, unified interface.

### **Adapter Pattern**

When the `Makefile` translates generic developer-oriented targets into tool-specific commands and arguments, it also behaves as an adapter between developer intent and concrete build tools.

### **Builder Pattern**

CMake itself acts as a builder that assembles the final build products (executables, libraries, WebAssembly modules) from configuration and inputs.

### **Superbuild Pattern**

In some ecosystems, a top-level CMake project orchestrates multiple subprojects and external dependencies (a “superbuild”). Here, the `Makefile` plays a similar orchestration role, but at a higher level, coordinating CMake and Docker instead of multiple CMake subprojects.