Facade Pattern

Structural pattern
Facade: Intent

- Provide a unified interface to a set of interfaces in a subsystem.
- Facade defines a higher-level interface that makes the subsystem easier to use.
Facade: Motivation (1)

Structuring a system into subsystems.

client classes

Facade

subsystem classes
Facade: Motivation (2)

Structuring a system into subsystems.
Facade: Motivation (3)

Structuring a system into subsystems.

- Compiler
  - Compile()
- Stream
- BytecodeStream
- CodeGenerator
- RiscCodeGenerator
- Scanner
- Token
- Symbol
- Parser
- ProgramNodeBuilder
- ProgramNode
- ExpressionNode
- VariableNode
Facade: Motivation (4)

Structuring a system into subsystems.

- Client
- Simulator
  - Transition
  - Places
  - RegularTransition
Facade: Applicability

- Use the Facade pattern when
  - you want to provide a simple interface to a complex subsystem.
  - there are many dependencies between clients and the implementation classes → promoting subsystem independence and portability.
  - you want to layer your subsystems → simplify the dependencies by using facades.
Facade: Participants

- Facade (Compiler)
  - knows which subsystem classes are responsible for the request.
  - delegates client requests to appropriate subsystem objects.

- subsystem classes (Scanner, Parser, etc.)
  - implement subsystem functionality.
  - handle work assigned by the Facade object.
  - have no knowledge of the facade; that is they keep no references to it.
Facade: Collaboration

- Client communicate with the subsystem by sending requests to Facade.
  - Facade forward requests of clients to the appropriate subsystem objects(s).
  - Facade may have to translate its interface to subsystem interfaces.
- Clients that use the facade don’t have to access its subsystem objects directly.
Facade: Consequences

*Benefits*

- It shields clients from subsystem components
  - making the subsystem easier to use
- It promotes weak coupling between the subsystem and its clients
  - let you vary the components of the subsystem without affecting its clients
  - layer a system and the dependencies between objects
  - reducing compilation dependencies
  - simplify porting systems to other platforms
- It doesn't prevent applications from using subsystem classes
  - you can choose between ease of use and generality
Facade: Implementation

- Reducing client-system coupling
  - if there are different implementation of subsystems
    - make Facade an abstract class, or
    - configure a Facade object with different subsystem objects.

- Public versus private subsystem classes
  - a subsystem is analogues to a class
    - a class encapsulates states and operations
    - a subsystem encapsulates classes
      - the public interface to a subsystem consists of classes that all clients can access
      - the private interface is just for subsystem extenders
      - the Facade class is part of the public interface, but it’s not the only part
  - C++ namespace let you expose just the public subsystem classes
Facade: Related Patterns

- **Abstract Factory**
  - can be used with Facade to provide an interface for creating subsystem objects in a subsystem-independent way.
  - can also be used as an alternative to Facade to hide platform-specific classes.

- **Mediator**
  - is similar to Facade in that it abstracts functionality of existing classes.

- **Singleton**
  - usually only one Facade object is required
  - Facade objects are often singletons.
Mediator Pattern

Behavioral pattern
Define an object that encapsulates how a set of objects interact.

Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
Mediator: Motivation (1)

- OO design
  - distribution of behavior among objects
  - object structure with many connections between objects
- A mediator is responsible for
  - controlling and coordinating the interactions for a group from referring to each other explicitly.
Mediator: Motivation (2)
Mediator: Motivation (3)

Mediator

aClient  aFontDialogDirector

Colleagues

aListBox  anEntryField

ShowDialog()

WidgetChanged()

GetSelection()

SetText()
Mediator: Motivation (4)

DialogDirector
- ShowDialog()
- CreateWidget()
- WidgetChanged(widget)

FontDialogDirector
- CreateWidget()
- WidgetChanged(Widget)

Widget
- Changed()

ListBox
- GetSelection()

EntryField
- SetText()

director->WidgetChanged(this)
Mediator: Applicability

- Use the Mediator pattern when
  - a set of objects communicate in well-defined but complex ways. The resulting interdependencies are unstructured and difficult to understand.
  - reuse an object is difficult because it refers to and communicates with many other objects.
  - a behavior that’s distributed between several classes should be customizable without a lot of subclassing.
Mediator: Structure (1)
Mediator: Structure (2)

A typical object structure
Mediator: Participants

- **Mediator (DialogDirector)**
  - defines an interface for communicating with Colleague objects.

- **ConcreteMediator (FontDialogDirector)**
  - implements cooperative behavior by coordinating Colleague objects.
  - knows and maintains its colleagues.

- **Colleague classes (ListBox, EntryField)**
  - each Colleague class knows its Mediator object.
  - each Colleague communicates with its mediator whenever it would have otherwise communicated with another colleague.
Mediator: Collaboration

- Colleagues send and receive requests from a Mediator object. The mediator implements the cooperative behavior by routing requests between the appropriate colleague(s).
Mediator: Consequences (1)

**Benefits**
- It limits subclassing
  - A mediator localizes behavior that otherwise would be distributed among several objects → changing this behavior requires subclassing Mediator only.
- It decouples colleagues
  - Colleague and Mediator can be varied and reused independently.
- It simplifies object protocols
  - Replace many-to-many interactions into one-to-many interactions.
  - One-to-many relationships are easier to understand, maintain, and extend.
Mediator: Consequences (2)

- **Benefits**
  - It abstracts how objects cooperate
    - Mediator helps clarify how objects interact in a system

- **Drawbacks**
  - It centralizes control
    - The Mediator pattern trades complexity of interaction for complexity in the mediator.
    - Mediator can become more complex than any individual colleague → making the mediator itself hard to maintain.
Mediator: Implementation

● Omitting the abstract Mediator class
  – if there is only one mediator

● Colleague-Mediator communication
  – Colleagues communicate with mediator when an event occurs.
    ● Implement the Mediator as an Observer using Observer pattern.
      – Colleague classes act as Subjects → sending notifications to mediator whenever they change state.
      – Mediator responds by propagating the effect of the change to other colleagues.
    ● When communicating with the mediator, a colleague passes itself as an argument → allowing the mediator to identify the sender (C++: RTTI).
Mediator: Related Patterns

- Facade differs from the Mediator in that it abstracts a subsystem of objects to provide a more convenient interface.
  - Facade: unidirectional protocol
    - Facade objects make requests of the subsystem classes but not vice versa.
  - Mediator: multidirectional protocol
    - Mediator enables cooperative behavior between colleague objects.

- Colleagues can communicate with the mediator using the Observer pattern.