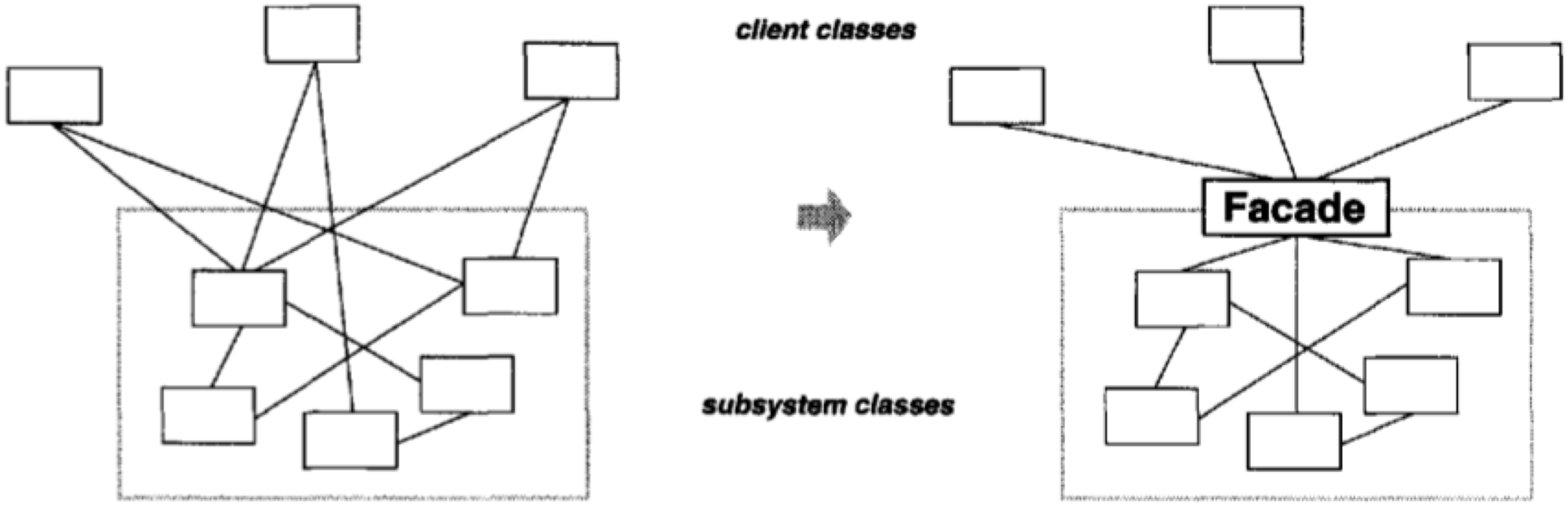
Facade Pattern

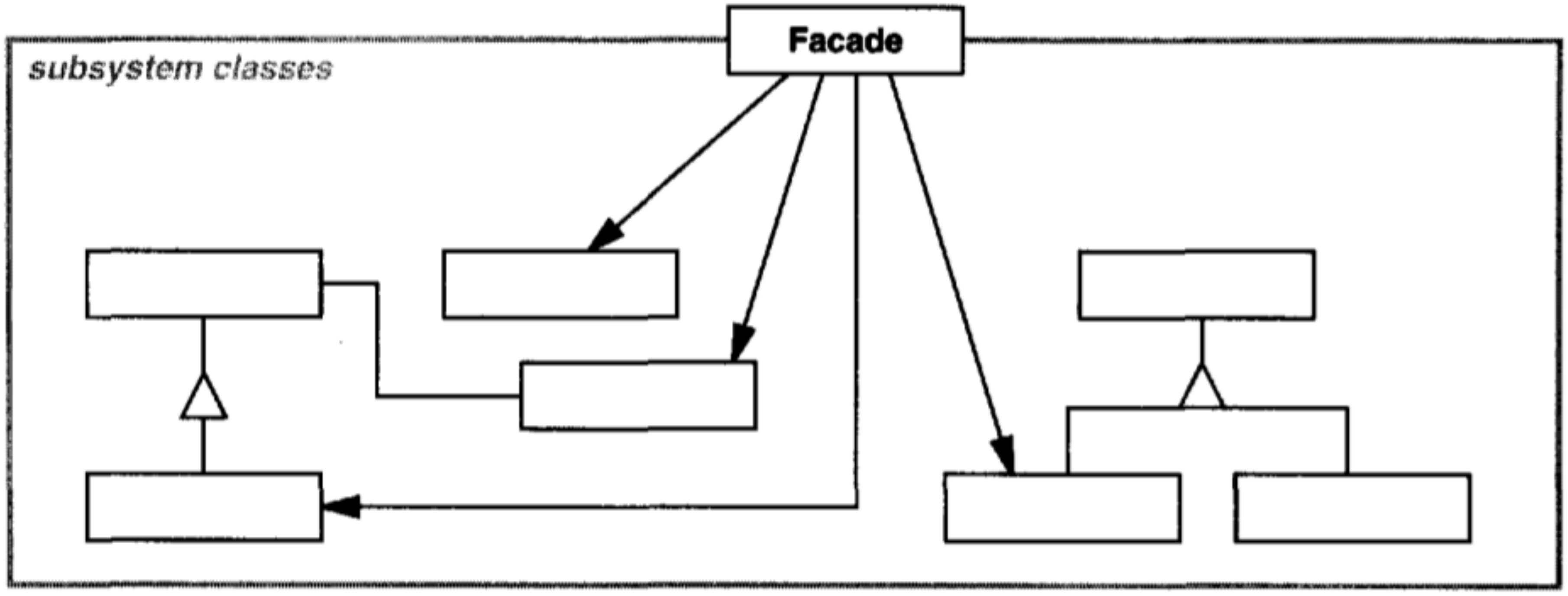
Problem – Façade

- Façade provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

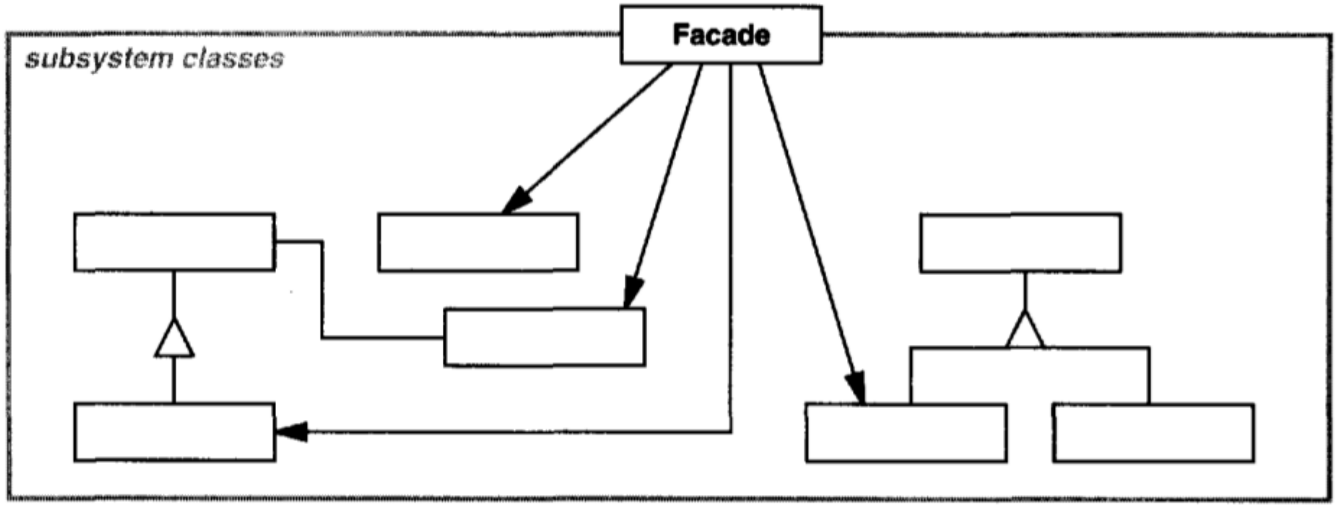


Facade Pattern

- *“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. ”*



Participants



- **Facade**
  - knows which subsystem classes are responsible for a request.
  - delegates client requests to appropriate subsystem objects.
- **Subsystem Classes**
  - implement subsystem functionality.
  - handle work assigned by the Facade object.
  - have no knowledge of the facade; that is, they keep no references to it.

Mostly we need only one Facade object is required.

Thus Facade objects are often **Singletons**

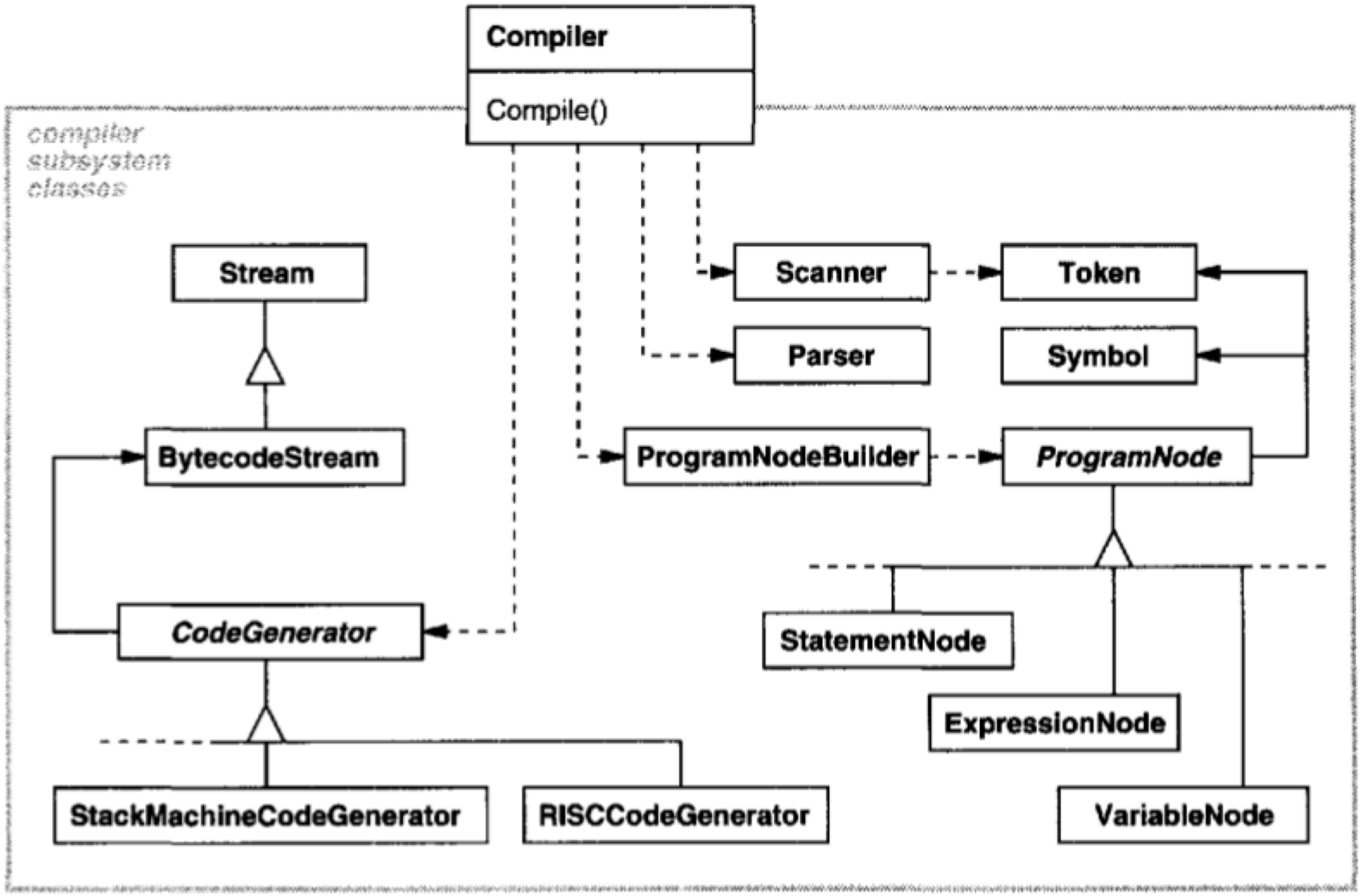
Example – Smart Home Facede

```
public class SmartHomeFacade {
          Amplifier amp;
         Tuner tuner;
          DvdPlayer dvd;
          CdPlayer cd;
          Lights lights;
          TV tv;


public SmartHomeFacade(Amplifier amp,   Tuner tuner,
                                                 DvdPlayer dvd,  CdPlayer cd, TV tv, ){

                    this.amp = amp;
                    this.tuner = tuner;
                    this.dvd = dvd;
                    this.cd = cd;
                    this.tv = tv;
                    this.lights = lights;
                    }
....
```

Example – HomeTheaterFacade

```
public class SmartHomeFacade {


....
        public void watchMovie(String movie) {
                System.out.println("Get ready to watch a movie...");
                popper.on();
                popper.pop();
                lights.dim(10);
                screen.down();
                projector.on();
                projector.wideScreenMode();
                amp.on();
                amp.setDvd(dvd);
                amp.setSurroundSound();
                amp.setVolume(5);
                dvd.on();
                dvd.play(movie);
        }. ... }
```

Example - Compiler



When use the Facade Pattern

Use Facade Patter when

- You want to provide **a simple interface to a complex subsystem.** A facade can provide a simple default view of the subsystem that is good enough for most clients.
- In your application, there are many dependencies between clients and the implementation classes of an abstraction. You can use a facade to **decouple the subsystem from clients** and other subsystems, thereby promoting subsystem independence and portability.
- You **want to layer your subsystems**. Use a facade to define an entry point to each subsystem level.

Consequences of Facade Pattern

- It **shields clients from subsystem components.** It reduces the number of objects that clients deal.
- It promotes **weak coupling between the subsystem and its clients (Decoupling)**.

  Weak coupling lets you vary the components of the subsystem without affecting its clients. Facades help layer a system and the dependencies between objects.
- **Reducing compilation dependencies** is vital in large software systems. You want to save time by minimizing recompilation when subsystem classes change.
- It **doesn't prevent applications from using subsystem classes** if they need to. Thus you can choose between ease of use and generality.

Summary – Facade Pattern

- Facade Pattern provides a unified interface to a set of interfaces in a subsystem.
- Facade Pattern is one of the **Structural Patterns**.

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