Façade Pattern

Shin-Jie Lee (李信杰)
Associate Professor
Computer and Network Center
Department of CSIE
National Cheng Kung University

Design Aspect of Facade

Interface to a subsystem

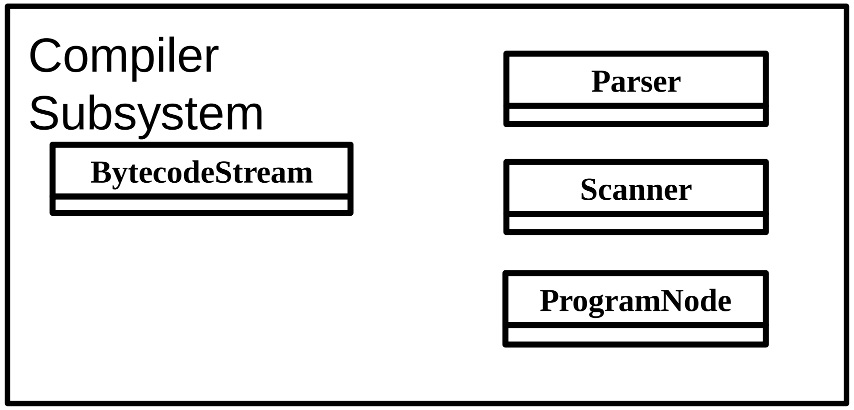
Outline

- Requirements Statement
- Initial Design and Its Problems
- Design Process
- Refactored Design after Design Process
- Another Example
- Recurrent Problems
- Intent
- Façade Pattern Structure
- Homework

A Programming Environment

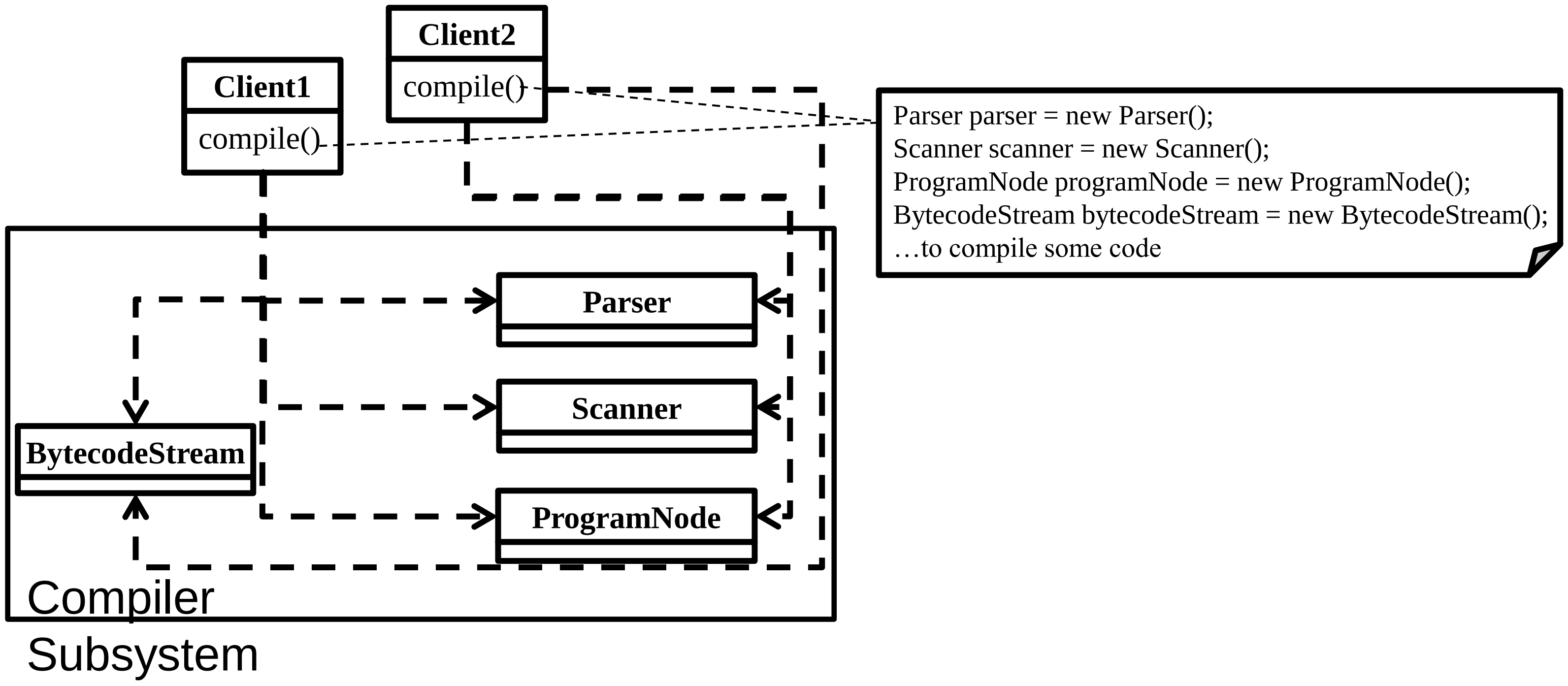
Requirements Statement$_1$

- A compiler subsystem contains classes such as Scanner, Parser, ProgramNode, and BytecodeStream.



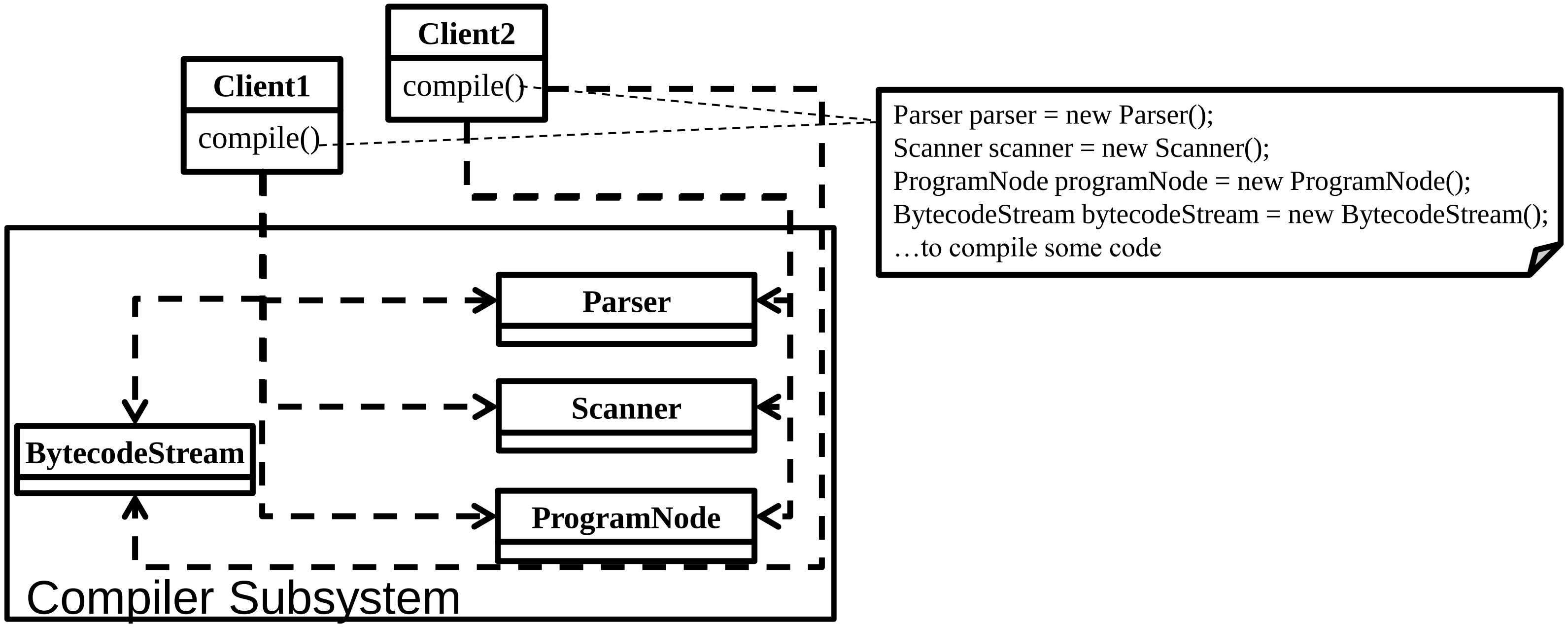
Requirements Statement$_2$

- The client classes need to use Scanner, Parser, ProgramNode, and BytecodeStream to compile some code.



```
Parser parser = new Parser();
Scanner scanner = new Scanner();
ProgramNode programNode = new ProgramNode();
BytecodeStream bytecodeStream = new BytecodeStream();
…to compile some code
```

Initial Design

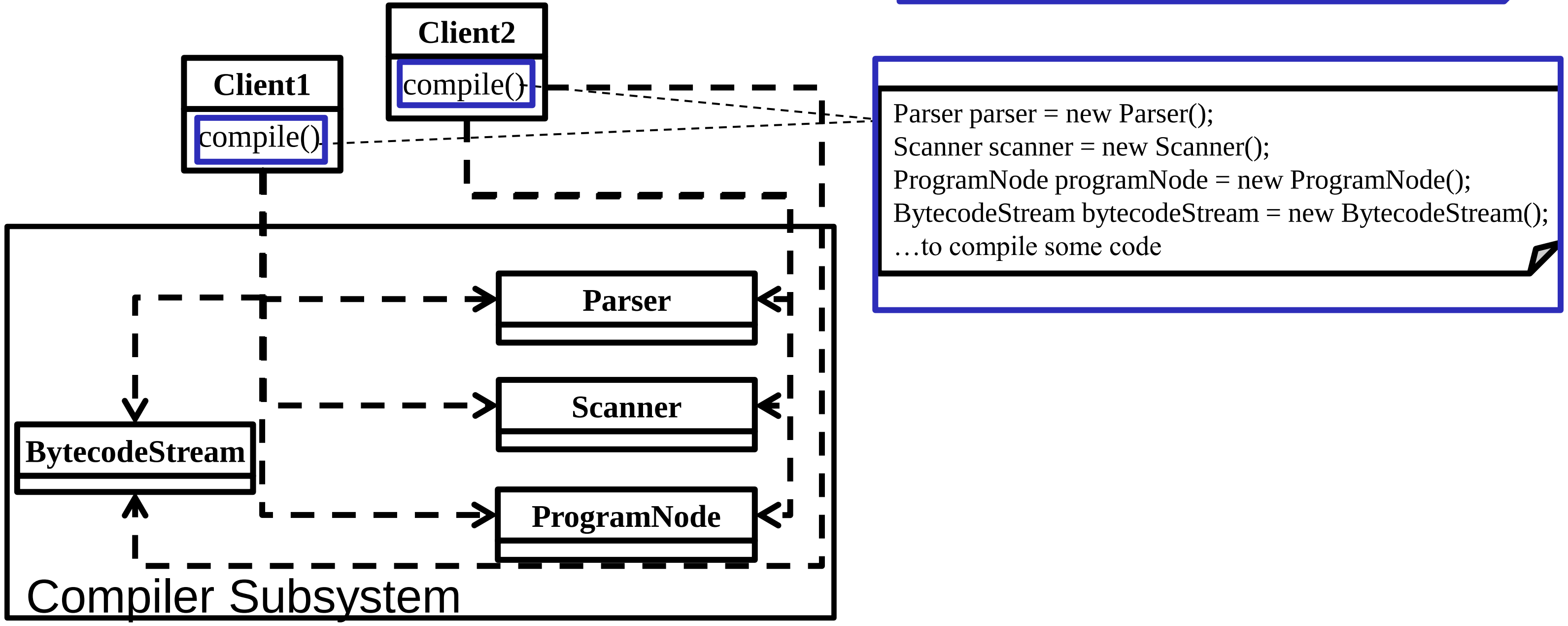


Parser parser = new Parser();
Scanner scanner = new Scanner();
ProgramNode programNode = new ProgramNode();
BytecodeStream bytecodeStream = new BytecodeStream();
…to compile some code

Problems with Initial Design

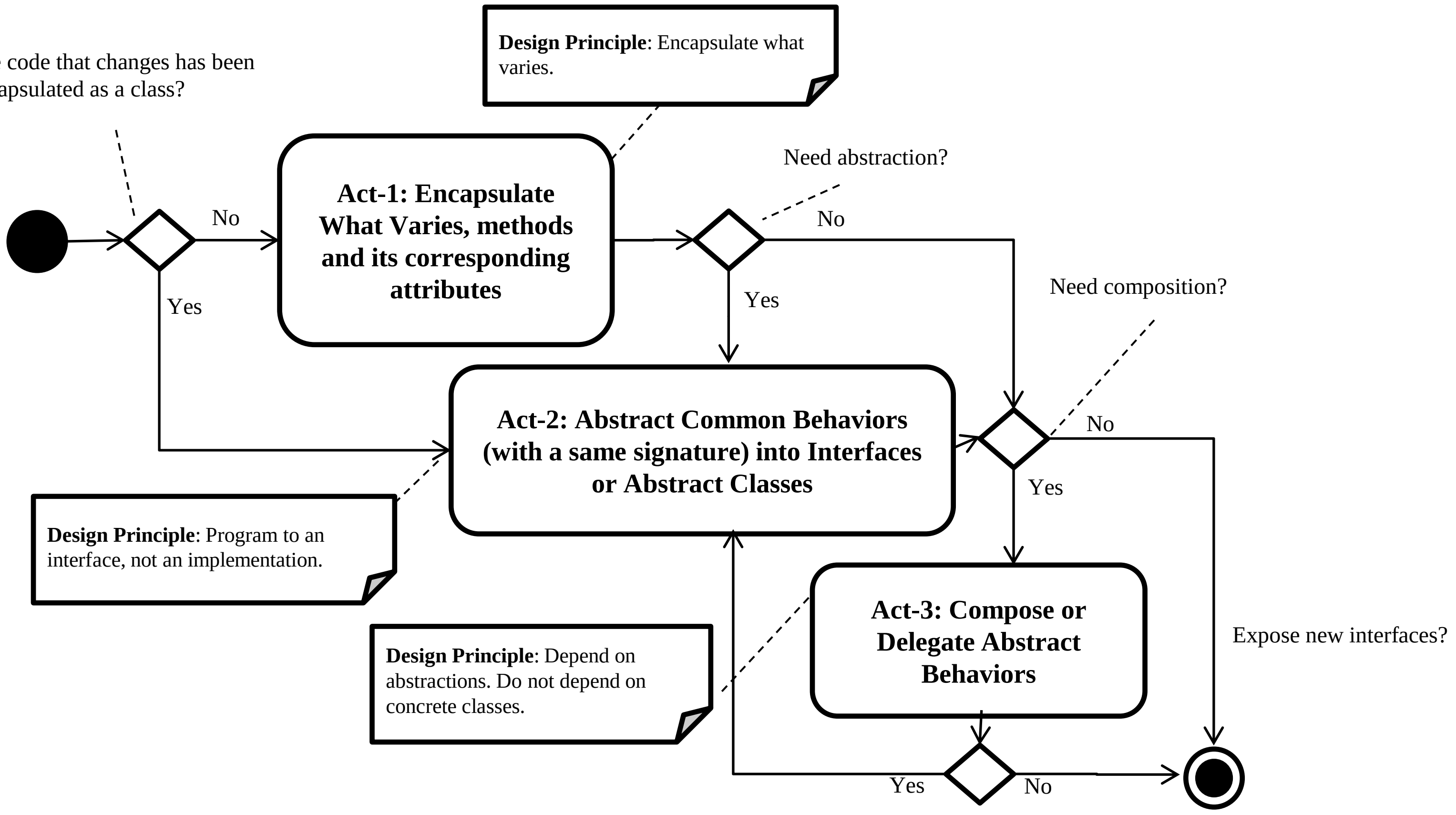
Problem1: When more and more clients want to use this compiler subsystem, it will cause a lot of duplicate code.

Problem2: If compiler subsystem changes, all the clients will be modified.



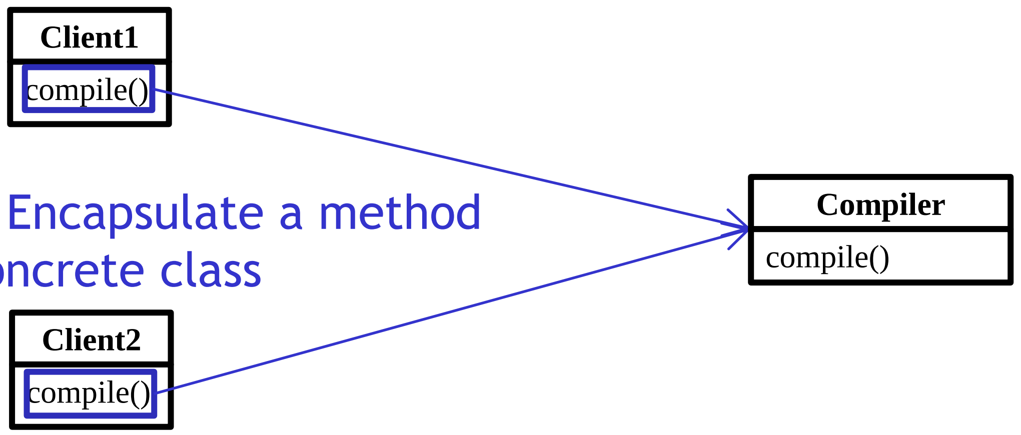
```
Parser parser = new Parser();
Scanner scanner = new Scanner();
ProgramNode programNode = new ProgramNode();
BytecodeStream bytecodeStream = new BytecodeStream();
…to compile some code
```

Design Process for Change



Design Principle: Encapsulate what varies.
The code that changes has been encapsulated as a class?
No
Yes
Act-1: Encapsulate What Varies, methods and its corresponding attributes
Need abstraction?
No
Yes
Need composition?
Act-2: Abstract Common Behaviors (with a same signature) into Interfaces or Abstract Classes
No
Yes
Design Principle: Program to an interface, not an implementation.
Act-3: Compose or Delegate Abstract Behaviors
Expose new interfaces?
Design Principle: Depend on abstractions. Do not depend on concrete classes.
Yes
No

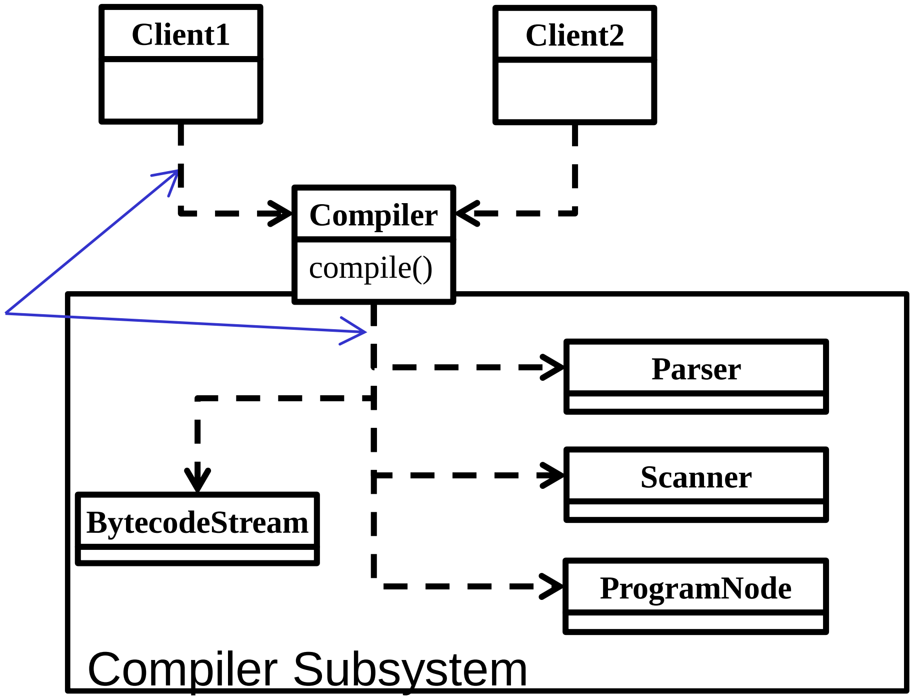
Act-1: Encapsulate What Varies



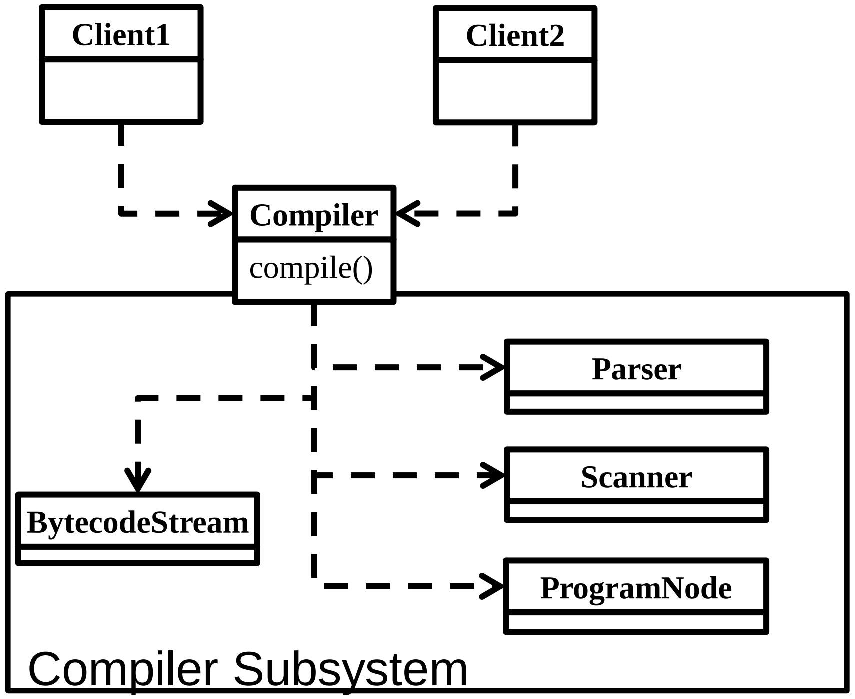
Act-1.2: Encapsulate a method into a concrete class

Act-3: Compose Abstract Behaviors

Act-3.4: Delegate behavior to a method of a concrete class



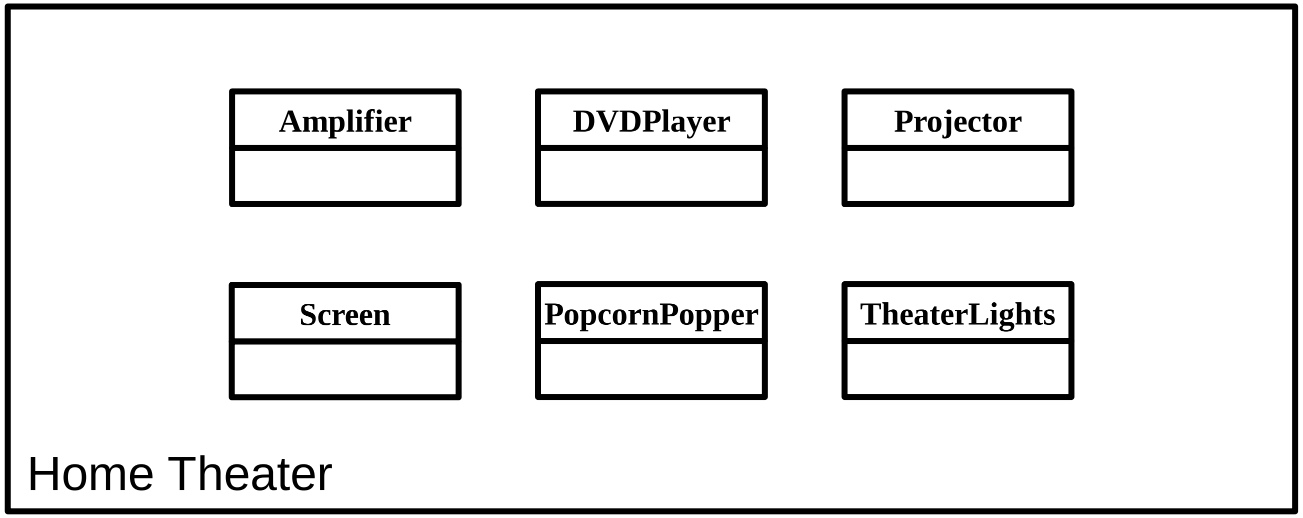
Refactored Design after Design Process



Home Theater

Requirements Statement$_1$

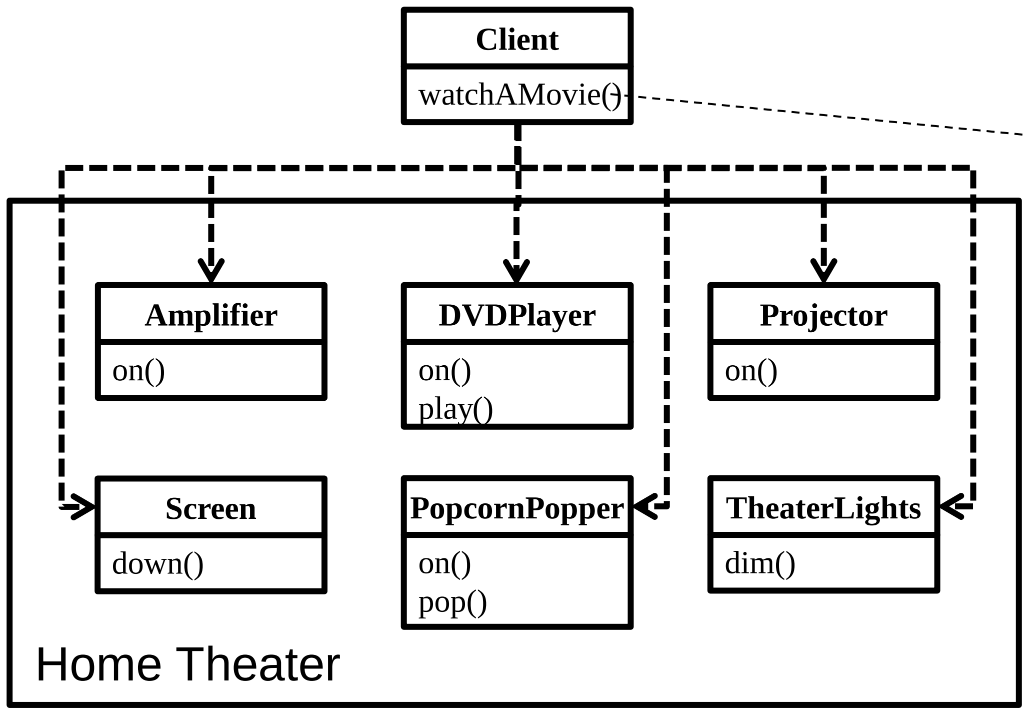
- A Home Theater consists of an amplifier, a DVD player, a projector, a screen, a popcorn popper, and theater lights.



Requirements Statement$_2$

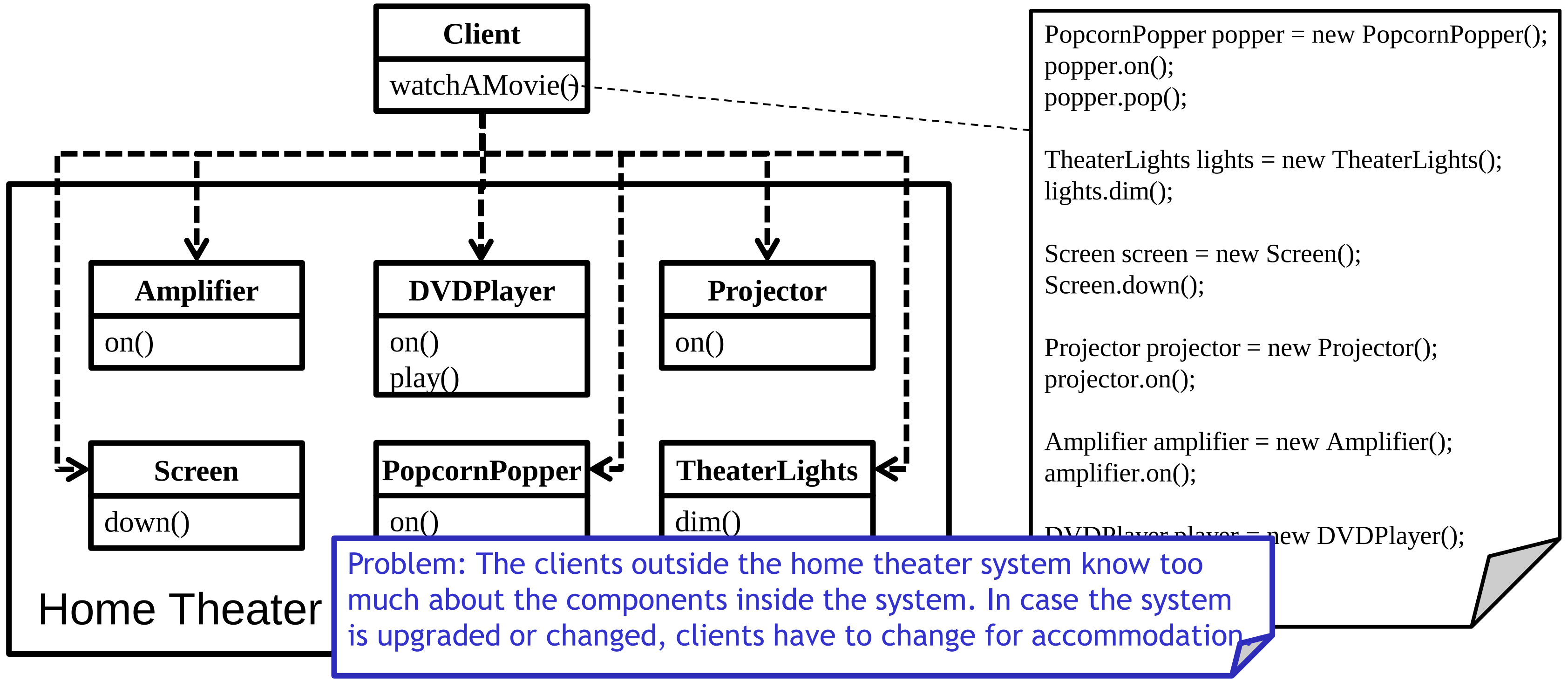
- A user can watch a movie through the following process:
  1. Turn on the popcorn popper
  2. Start the popper popping
  3. Dim the lights
  4. Put the screen down
  5. Turn the projector on
  6. Turn the sound amplifier on
  7. Turn the DVD player on
  8. Start the DVD player playing

Initial Design



```
PopcornPopper popper = new PopcornPopper();
popper.on();
popper.pop();

TheaterLights lights = new TheaterLights();
lights.dim();

Screen screen = new Screen();
Screen.down();

Projector projector = new Projector();
projector.on();

Amplifier amplifier = new Amplifier();
amplifier.on();

DVDPlayer player = new DVDPlayer();
player.on();
player.play();
```

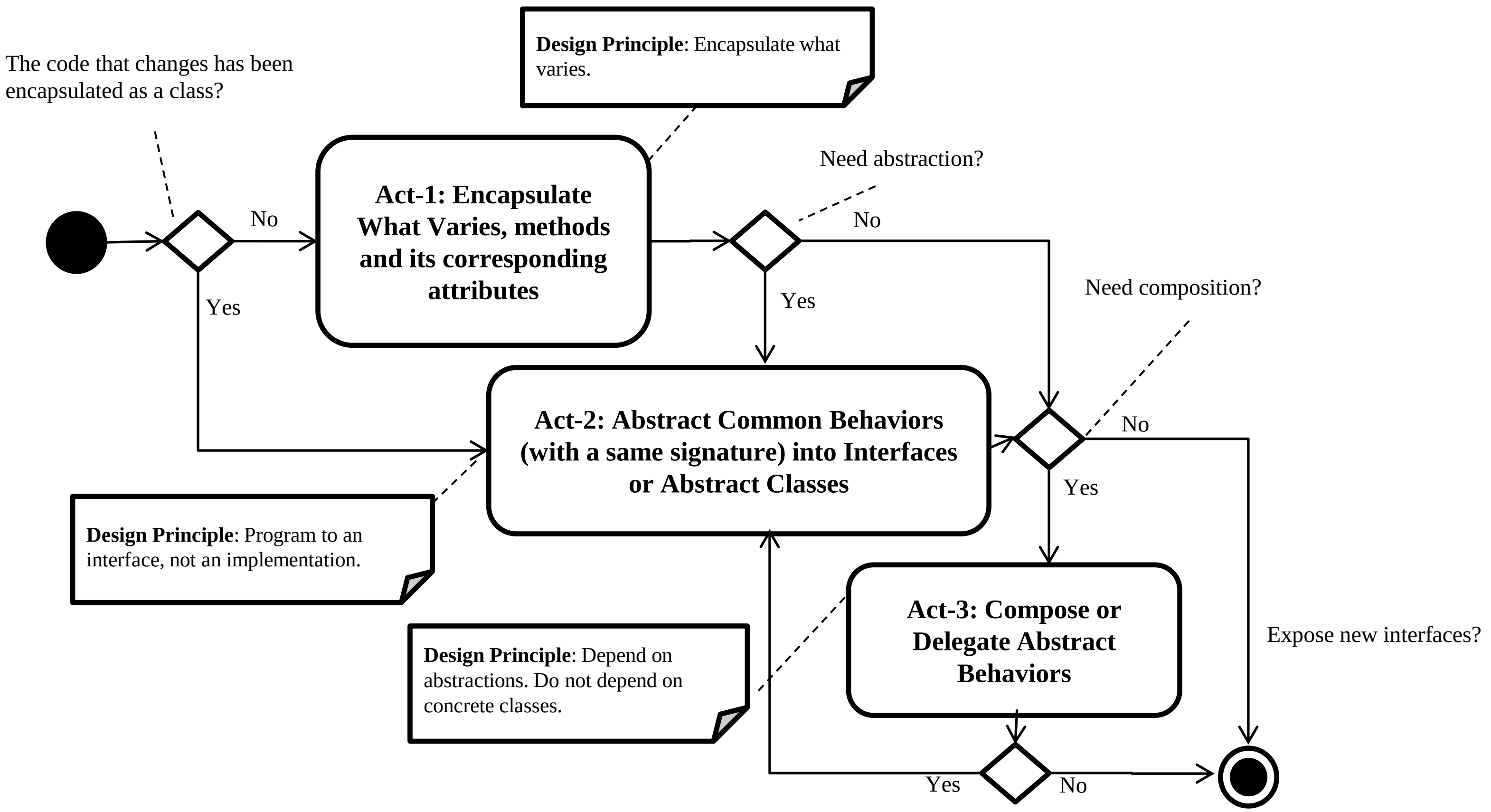
Problems with Initial Design



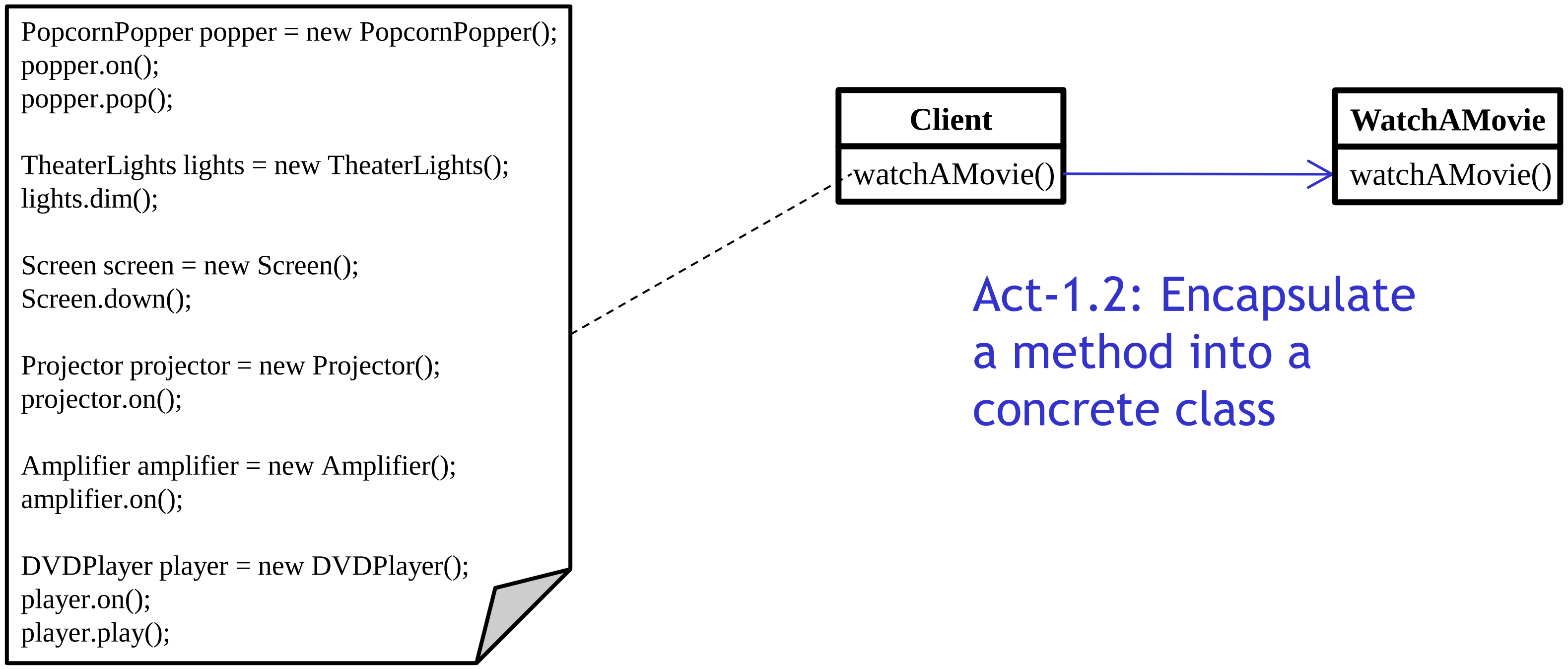
```
PopcornPopper popper = new PopcornPopper();
popper.on();
popper.pop();

TheaterLights lights = new TheaterLights();
lights.dim();

Screen screen = new Screen();
Screen.down();

Projector projector = new Projector();
projector.on();

Amplifier amplifier = new Amplifier();
amplifier.on();

DVDPlayer player = new DVDPlayer();
```

Problem: The clients outside the home theater system know too much about the components inside the system. In case the system is upgraded or changed, clients have to change for accommodation

Design Process for Change


The code that changes has been encapsulated as a class?
Design Principle: Encapsulate what varies.
No
Yes
Act-1: Encapsulate What Varies, methods and its corresponding attributes
Need abstraction?
No
Yes
Need composition?
No
Yes
Act-2: Abstract Common Behaviors (with a same signature) into Interfaces or Abstract Classes
Design Principle: Program to an interface, not an implementation.
Design Principle: Depend on abstractions. Do not depend on concrete classes.
Act-3: Compose or Delegate Abstract Behaviors
Expose new interfaces?
Yes
No

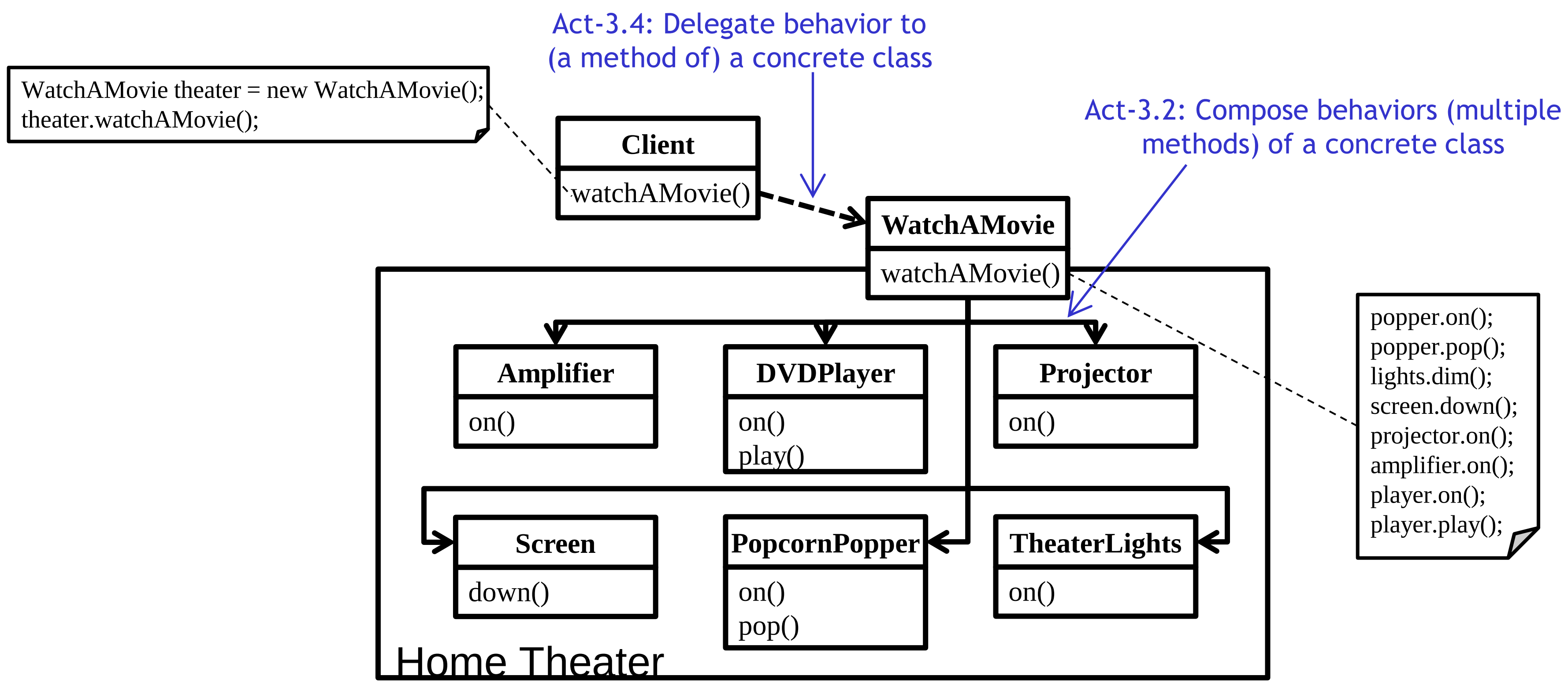
Act-1: Encapsulate What Varies



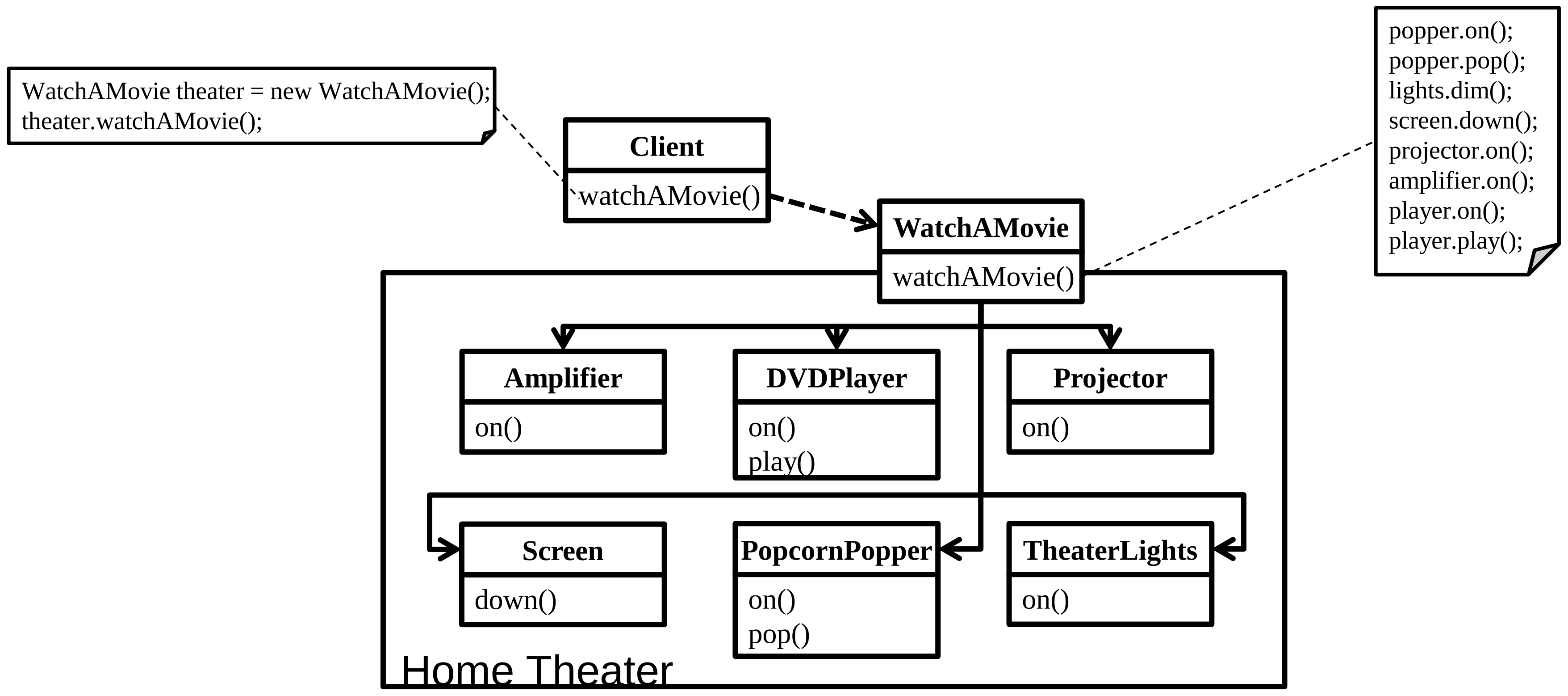
```
PopcornPopper popper = new PopcornPopper();
popper.on();
popper.pop();

TheaterLights lights = new TheaterLights();
lights.dim();

Screen screen = new Screen();
Screen.down();

Projector projector = new Projector();
projector.on();

Amplifier amplifier = new Amplifier();
amplifier.on();

DVDPlayer player = new DVDPlayer();
player.on();
player.play();
```

Act-1.2: Encapsulate a method into a concrete class

Act-3: Compose Behaviors



Refactored Design after Design Process



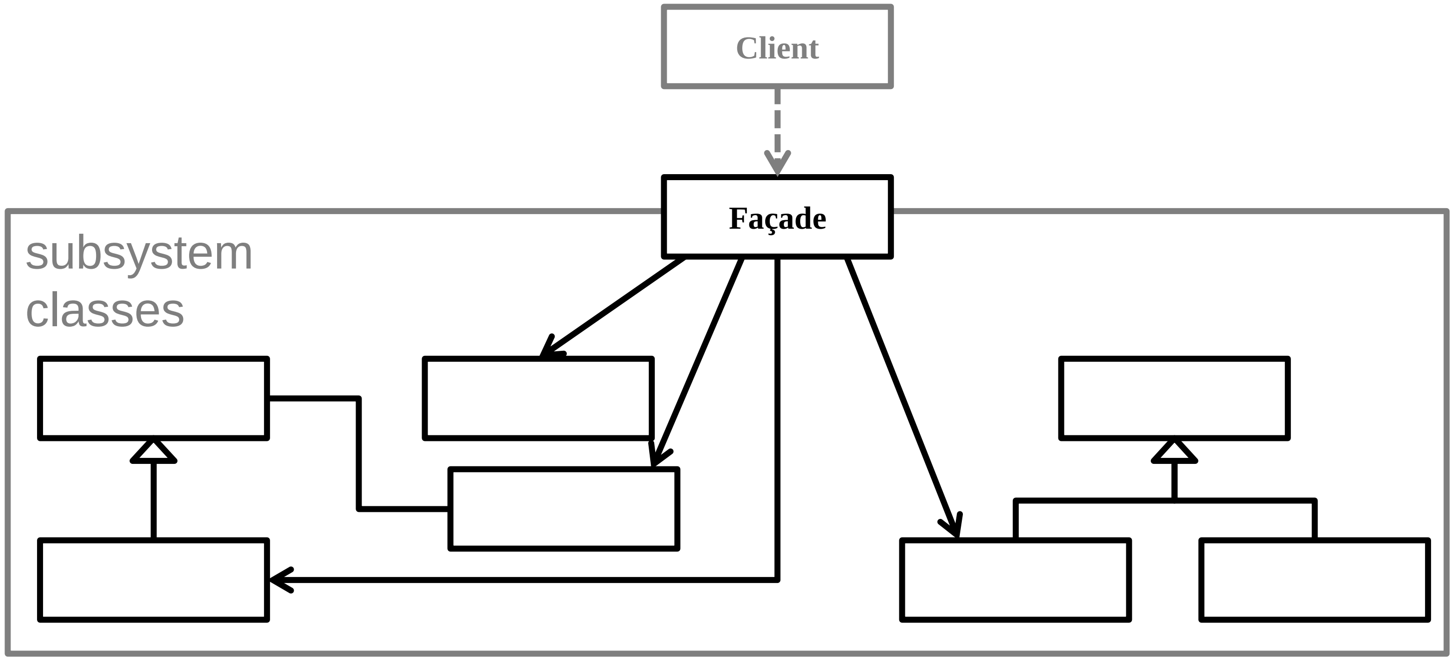
Recurrent Problem

- A common design goal is to minimize the communication and dependencies between subsystems.
  - One way to achieve this goal is to introduce a façade object that provides a single, simplified interface to the more general facilities of a subsystem.

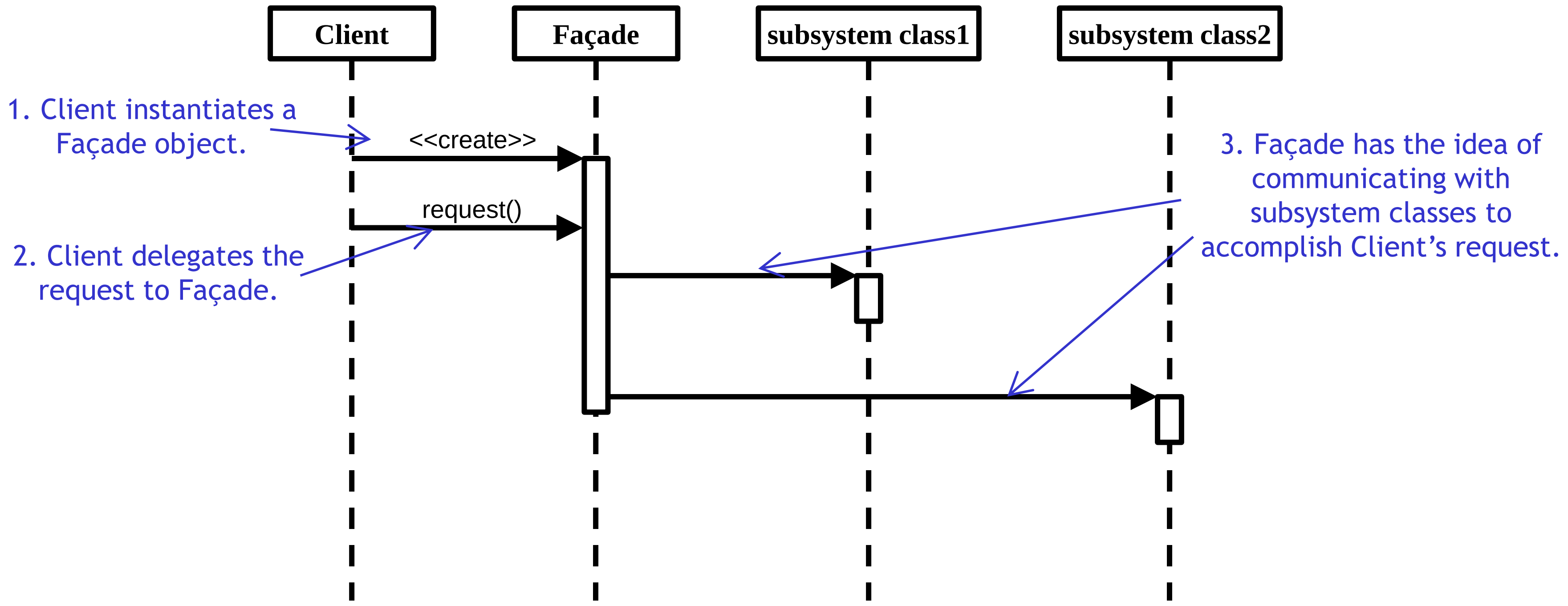
Intent

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

Façade Structure$_1$



Façade Structure$_2$


Client
Façade
subsystem class1
subsystem class2
<<create>>
request()
1. Client instantiates a Façade object.
2. Client delegates the request to Façade.
3. Façade has the idea of communicating with subsystem classes to accomplish Client's request.

Façade Structure$_3$

	Instantiation	Use	Termination
Façade	Client	Client	Don’t Care
subsystem classes	Don’t Care	Façade	Don’t Care