Lecture 16

ECE 1145: Software Construction and Evolution

Design Pattern Catalog
(CH 9, 19 - 23, 25, 27)

Announcements

- Iteration 5: Test Stubs and More Patterns due Oct. 31
- Relevant Exercises: 9.1, 19.6, 20.4 21.2

Questions for Today

What problems can design patterns address (and how)?

Recall: Design Patterns

A **design pattern** is a solution to a problem in a context.

“Pattern” concept originally applied to houses, urban planning
→ Patterns apply at different scales, can be used recursively

The collection of patterns (**pattern catalog**) is a design tool.

Also a **communication** tool – “pattern language”.
→ Facilitate design discussions
→ Represent knowledge gained through design experience

Recall: Design Patterns

So far: Strategy, State, Abstract Factory

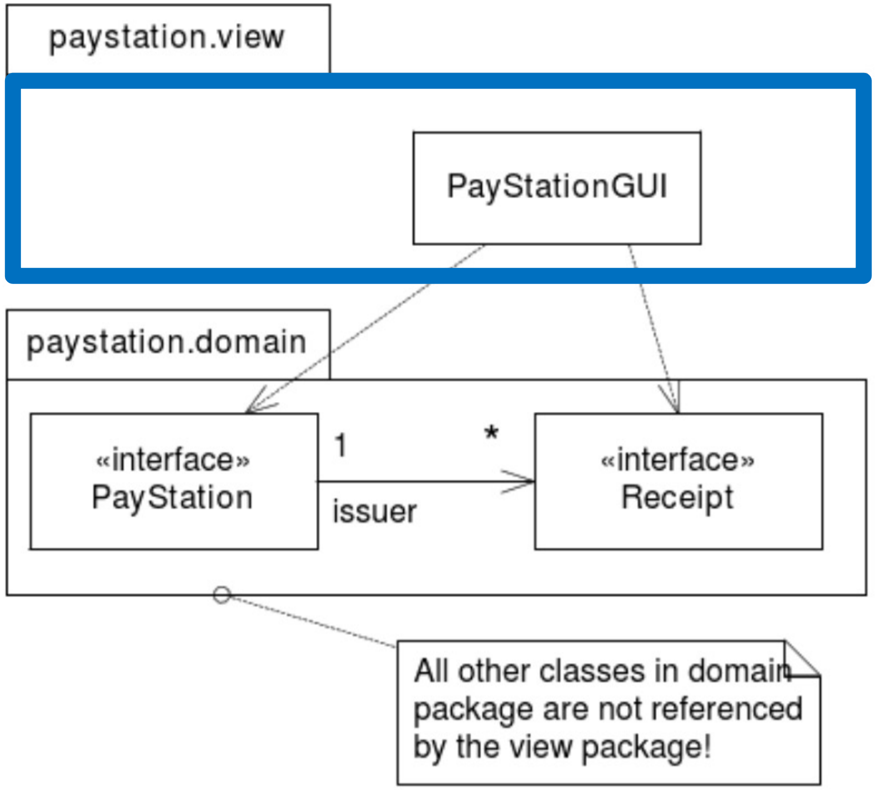
Today:
Façade, Decorator, Adapter, Builder, Command, Proxy, Null Object

- ③ **Encapsulate behavior that varies**
- ① **Program to an interface**
- ② **Favor object composition**

Façade Pattern

Suppose we are adding a graphical user interface (GUI) to the pay station:

paystation.view - only coupled with PayStation and Receipt

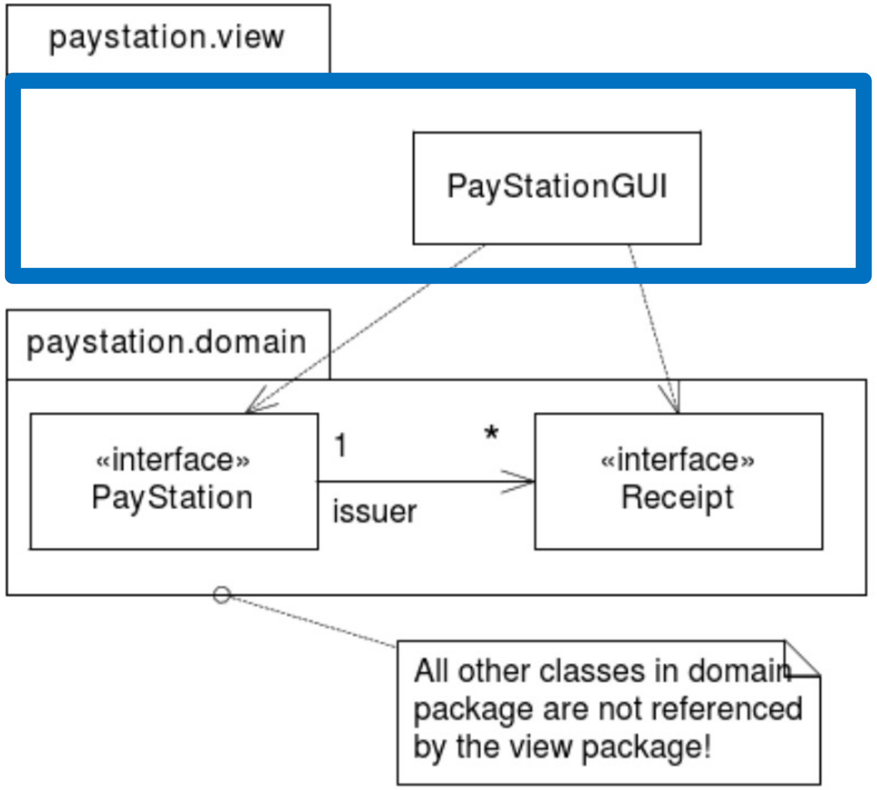


Façade Pattern

Suppose we are adding a graphical user interface (GUI) to the pay station:

paystation.view - only coupled with PayStation and Receipt

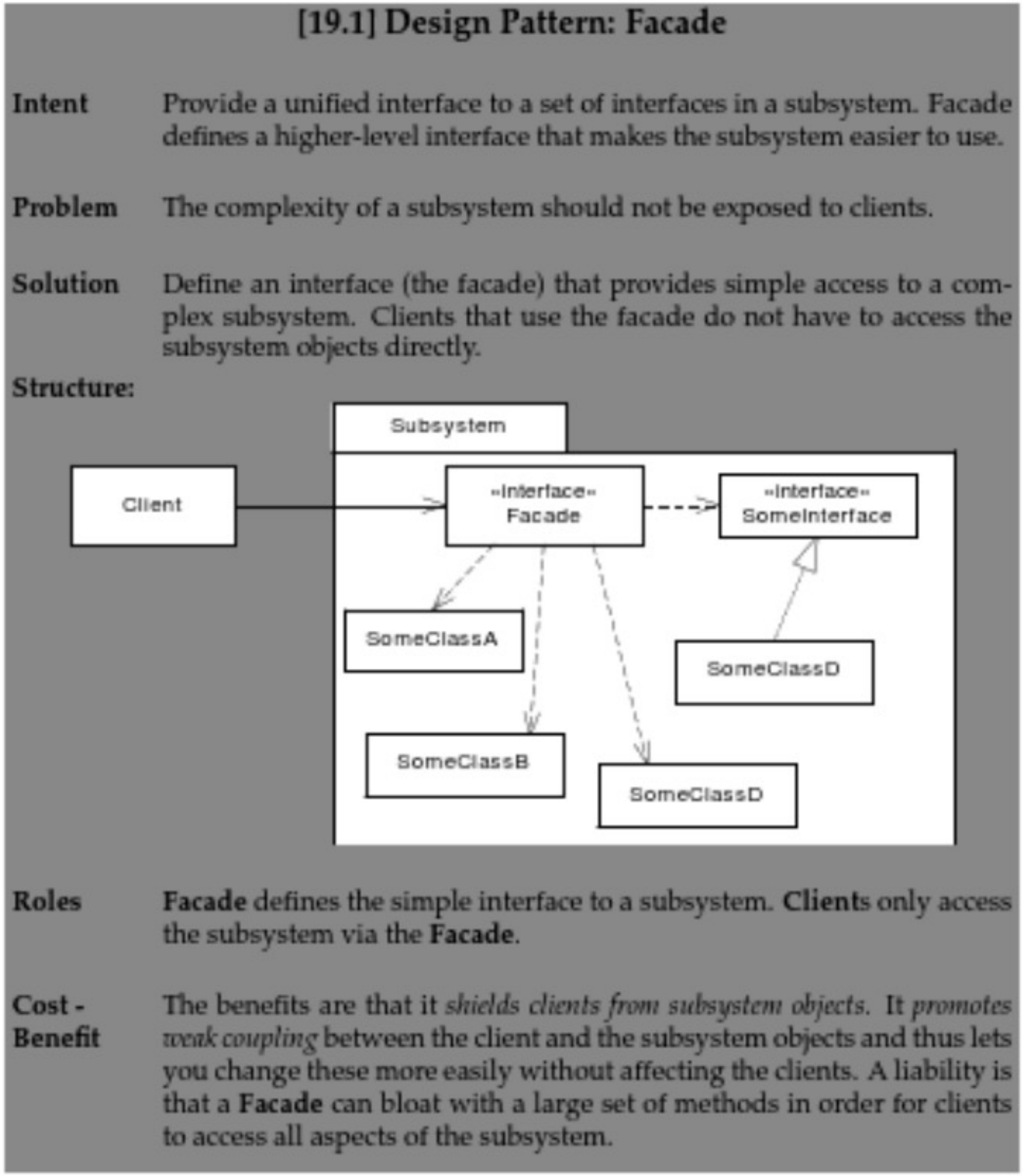
Façade: Provide a unified interface to a set of interfaces in a subsystem.



Façade Pattern

Problem: The complexity of a subsystem should not be exposed to clients

Solution: The **façade** shields the client from the subsystem complexity by presenting a simple interface



[19.1] Design 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.

Problem The complexity of a subsystem should not be exposed to clients.

Solution Define an interface (the facade) that provides simple access to a complex subsystem. Clients that use the facade do not have to access the subsystem objects directly.

Structure:

Roles **Facade** defines the simple interface to a subsystem. **Clients** only access the subsystem via the **Facade**.

Cost - Benefit The benefits are that it *shields clients from subsystem objects*. It *promotes weak coupling* between the client and the subsystem objects and thus lets you change these more easily without affecting the clients. A liability is that a **Facade** can bloat with a large set of methods in order for clients to access all aspects of the subsystem.

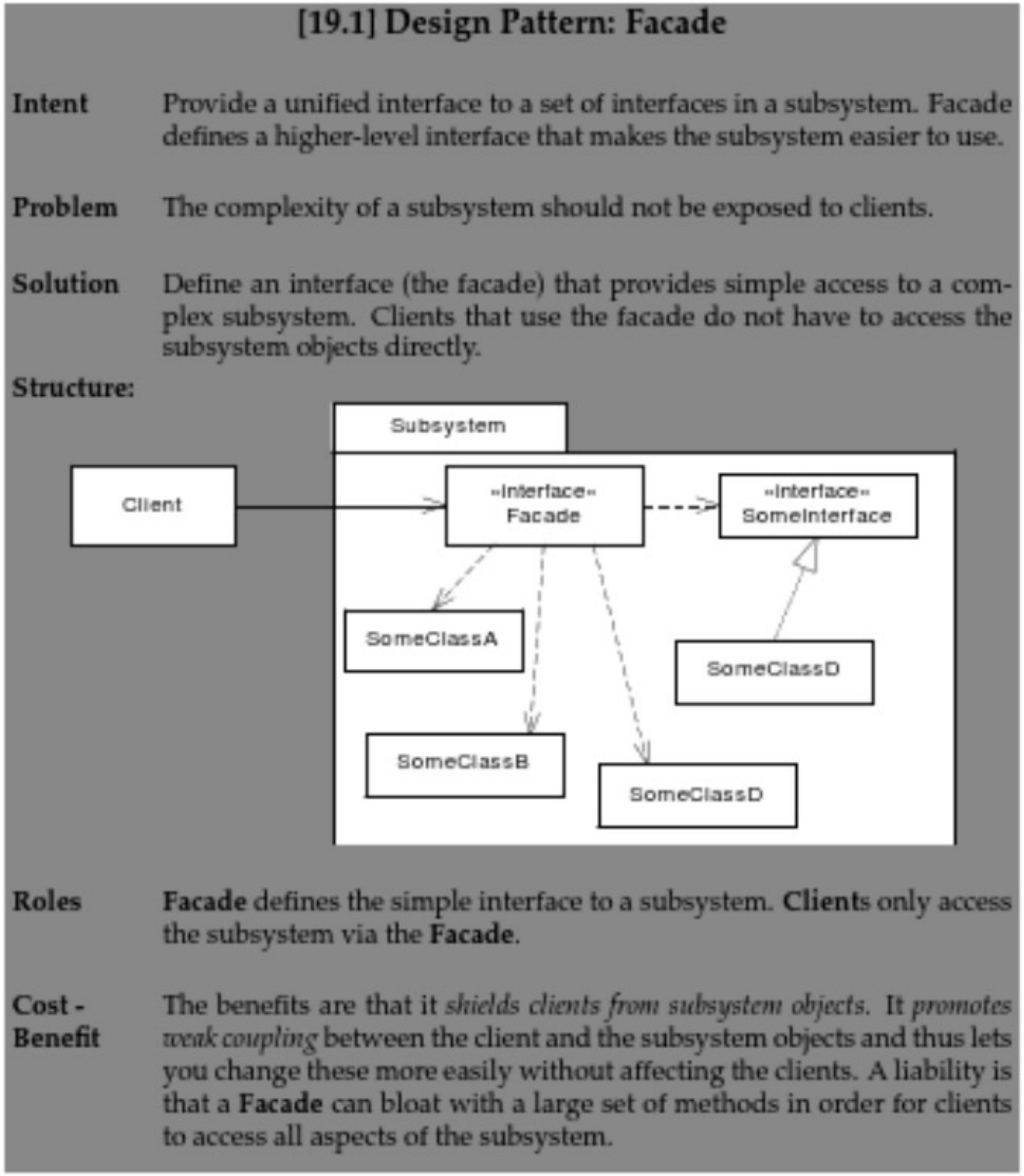
Façade Pattern

Problem: The complexity of a subsystem should not be exposed to clients

Solution: The **façade** shields the client from the subsystem complexity by presenting a simple interface

The pay station has embodied the façade pattern from the beginning!

Pay station hardware -> PayStation interface



[19.1] Design 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.

Problem The complexity of a subsystem should not be exposed to clients.

Solution Define an interface (the facade) that provides simple access to a complex subsystem. Clients that use the facade do not have to access the subsystem objects directly.

Structure:

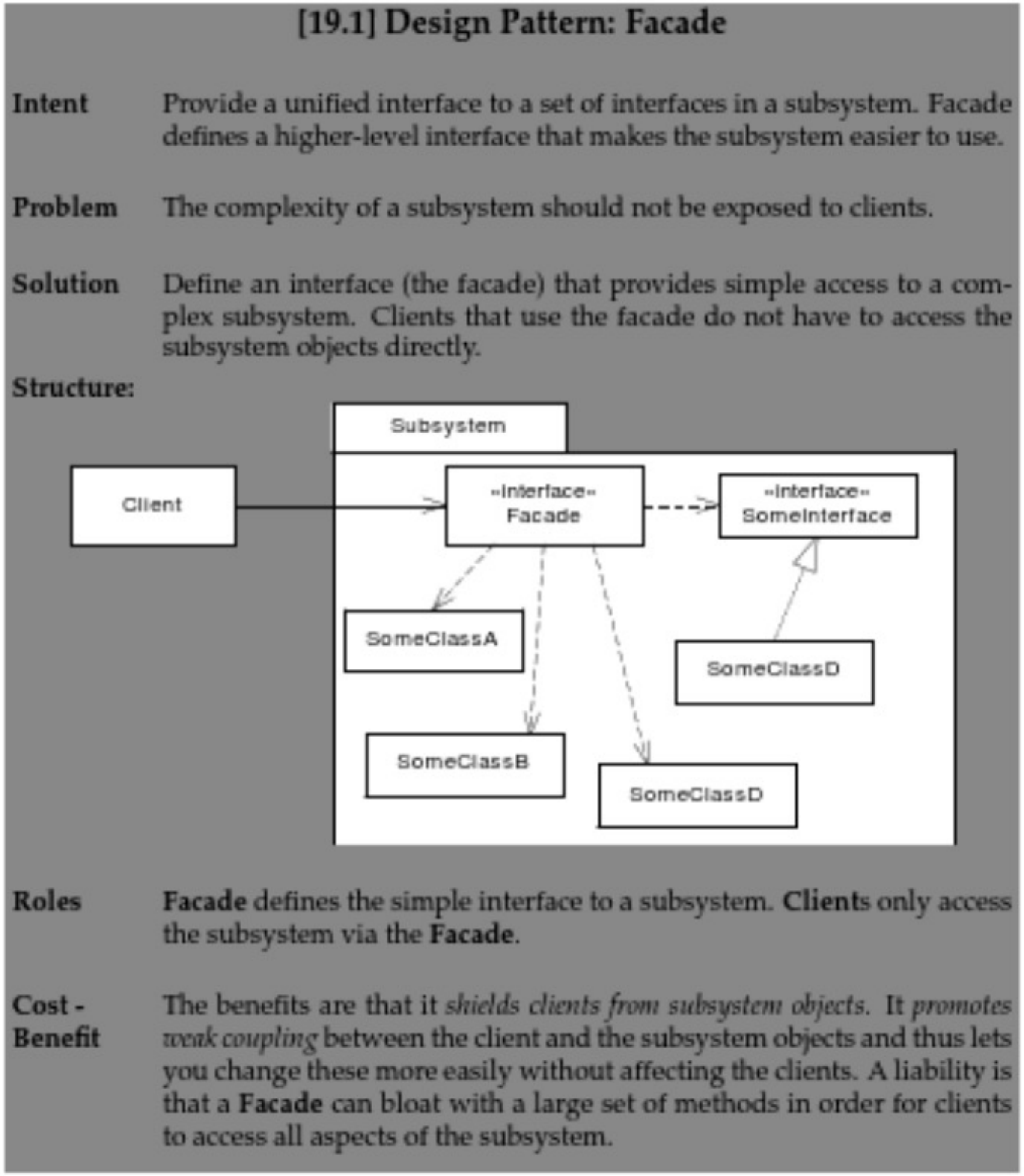
Roles **Facade** defines the simple interface to a subsystem. **Clients** only access the subsystem via the **Facade**.

Cost - Benefit The benefits are that it *shields clients from subsystem objects*. It *promotes weak coupling* between the client and the subsystem objects and thus lets you change these more easily without affecting the clients. A liability is that a **Facade** can bloat with a large set of methods in order for clients to access all aspects of the subsystem.

Façade Pattern

☺

- Subsystem is also shielded from changes in clients
  (e.g., GUI vs. hardware)
- **Weak coupling**



[19.1] Design 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.

Problem The complexity of a subsystem should not be exposed to clients.

Solution Define an interface (the facade) that provides simple access to a complex subsystem. Clients that use the facade do not have to access the subsystem objects directly.

Structure:

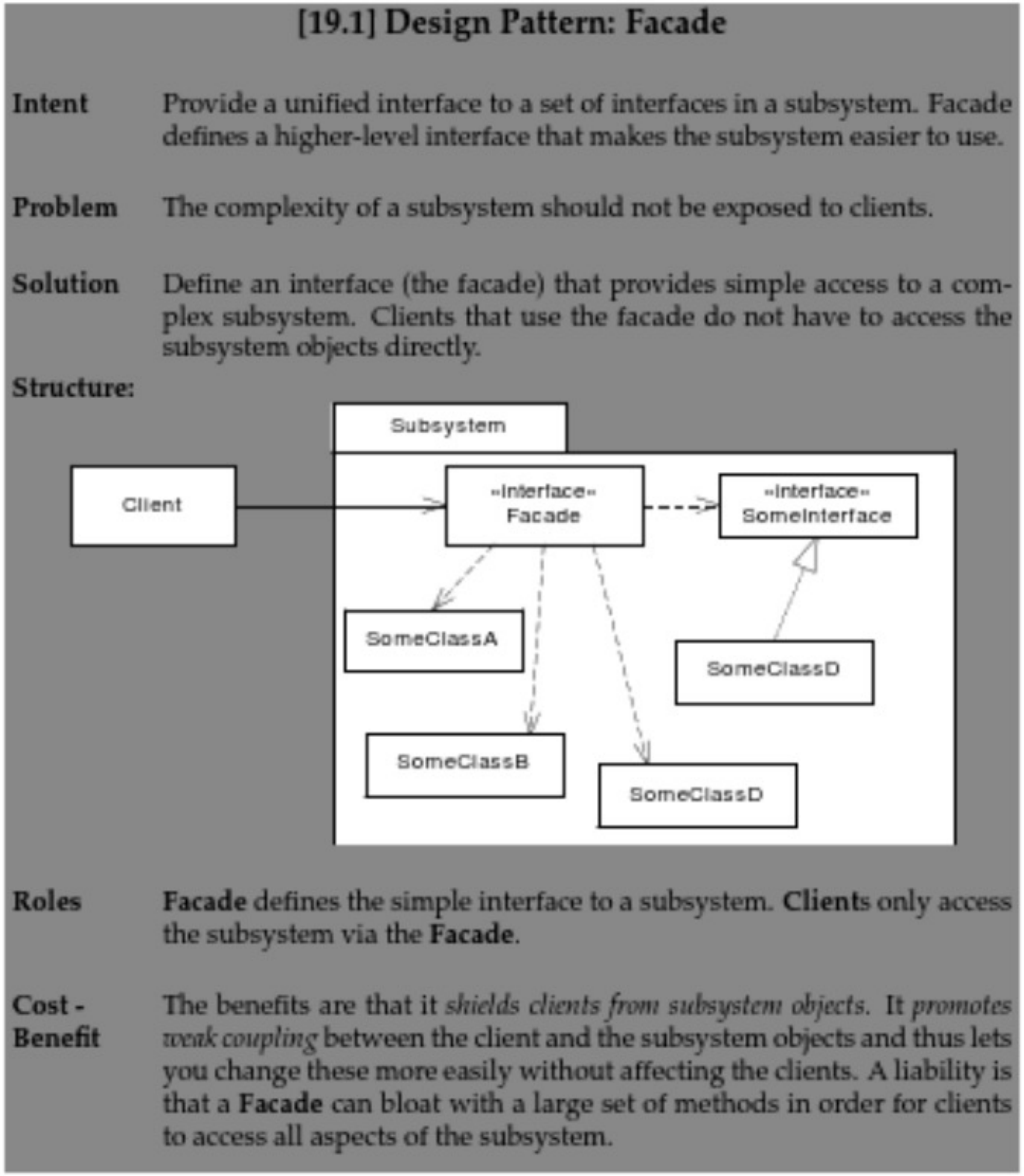
Roles **Facade** defines the simple interface to a subsystem. **Clients** only access the subsystem via the **Facade**.

Cost - Benefit The benefits are that it *shields clients from subsystem objects*. It *promotes weak coupling* between the client and the subsystem objects and thus lets you change these more easily without affecting the clients. A liability is that a **Facade** can bloat with a large set of methods in order for clients to access all aspects of the subsystem.

Façade Pattern

☹

- May have lots of methods
- Avoid access to inner objects



[19.1] Design 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.

Problem The complexity of a subsystem should not be exposed to clients.

Solution Define an interface (the facade) that provides simple access to a complex subsystem. Clients that use the facade do not have to access the subsystem objects directly.

Structure:

Roles **Facade** defines the simple interface to a subsystem. **Clients** only access the subsystem via the **Facade**.

Cost - Benefit The benefits are that it *shields clients from subsystem objects*. It *promotes weak coupling* between the client and the subsystem objects and thus lets you change these more easily without affecting the clients. A liability is that a **Facade** can bloat with a large set of methods in order for clients to access all aspects of the subsystem.

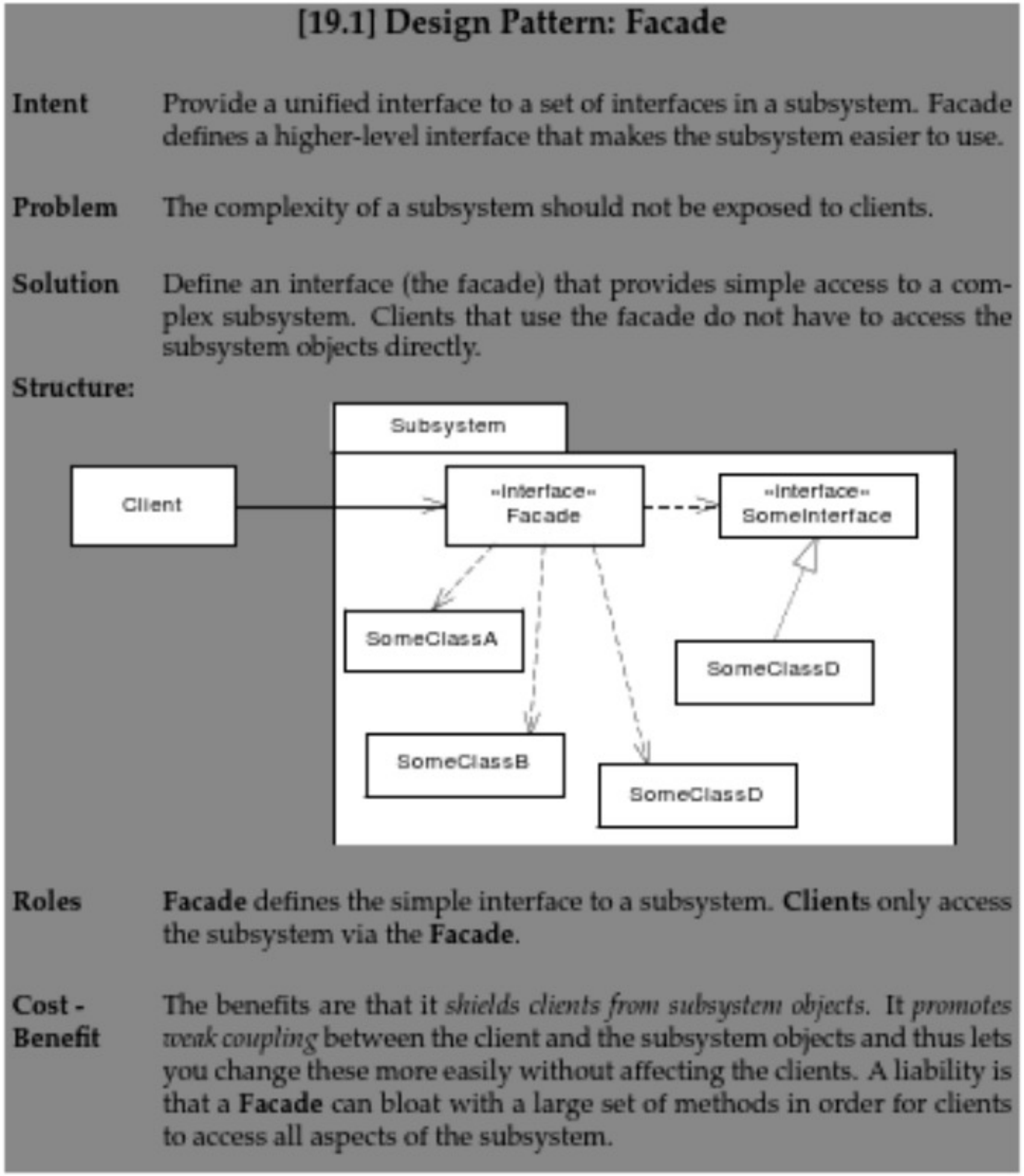
Façade Pattern

☹

- May have lots of methods
- Avoid access to inner objects

Options:

- Make the façade **opaque** – no references to objects within the subsystem are returned to clients
- Make the façade export **read-only objects** – façade can export references to objects created within the subsystem, but only via a read-only interface (immutable objects)



[19.1] Design 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.

Problem The complexity of a subsystem should not be exposed to clients.

Solution Define an interface (the facade) that provides simple access to a complex subsystem. Clients that use the facade do not have to access the subsystem objects directly.

Structure:

Roles **Facade** defines the simple interface to a subsystem. **Clients** only access the subsystem via the **Facade**.

Cost - Benefit The benefits are that it *shields clients from subsystem objects*. It *promotes weak coupling* between the client and the subsystem objects and thus lets you change these more easily without affecting the clients. A liability is that a **Facade** can bloat with a large set of methods in order for clients to access all aspects of the subsystem.

Decorator Pattern

One of AlphaTown's pay stations often overflows with 5-cent coins, so they want a **log file** of coin types with time stamps.

Decorator Pattern

One of AlphaTown’s pay stations often overflows with 5-cent coins, so they want a **log file** of coin types with time stamps.

(3) What behavior varies?

Decorator Pattern

One of AlphaTown’s pay stations often overflows with 5-cent coins, so they want a **log file** of coin types with time stamps.

(3) What behavior varies?

→ Accept payment (additional behavior)

Decorator Pattern

One of AlphaTown’s pay stations often overflows with 5-cent coins, so they want a **log file** of coin types with time stamps.

(3) What behavior varies?

→ Accept payment (additional behavior)

(1) Program to an interface:

→ PaymentAcceptor? PayStation?

Decorator Pattern

One of AlphaTown’s pay stations often overflows with 5-cent coins, so they want a **log file** of coin types with time stamps.

(3) What behavior varies?

→ Accept payment (additional behavior)

(1) Program to an interface:

→ PaymentAcceptor? PayStation?

(2) We will **compose** the required behavior by putting an **intermediate object** with the same interface in front of the pay station object

Decorator Pattern

One of AlphaTown's pay stations often overflows with 5-cent coins, so they want a **log file** of coin types with time stamps.

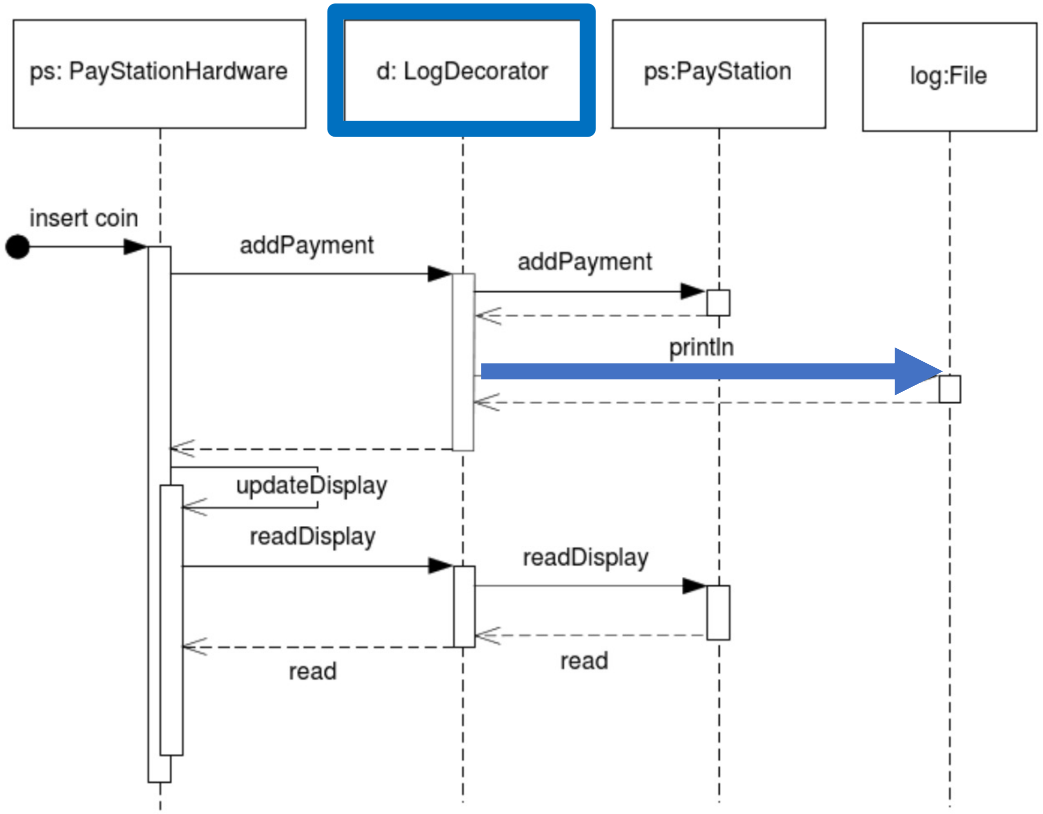
(3) What behavior varies?

→ Accept payment (additional behavior)

(1) Program to an interface:

→ PaymentAcceptor? PayStation?

(2) We will **compose** the required behavior by putting an **intermediate object** with the same interface in front of the pay station object



Decorator Pattern

One of AlphaTown's pay stations often overflows with 5-cent coins, so they want a **log file** of coin types with time stamps.

(3) What behavior varies?

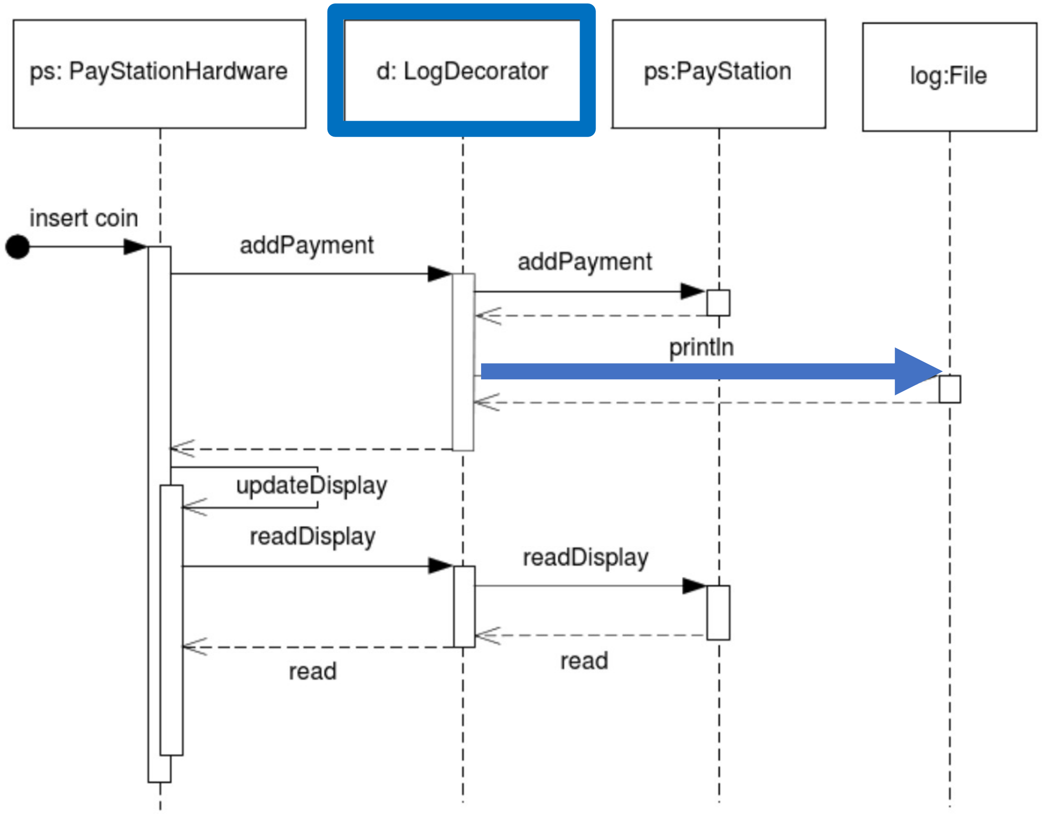
→ Accept payment (additional behavior)

(1) Program to an interface:

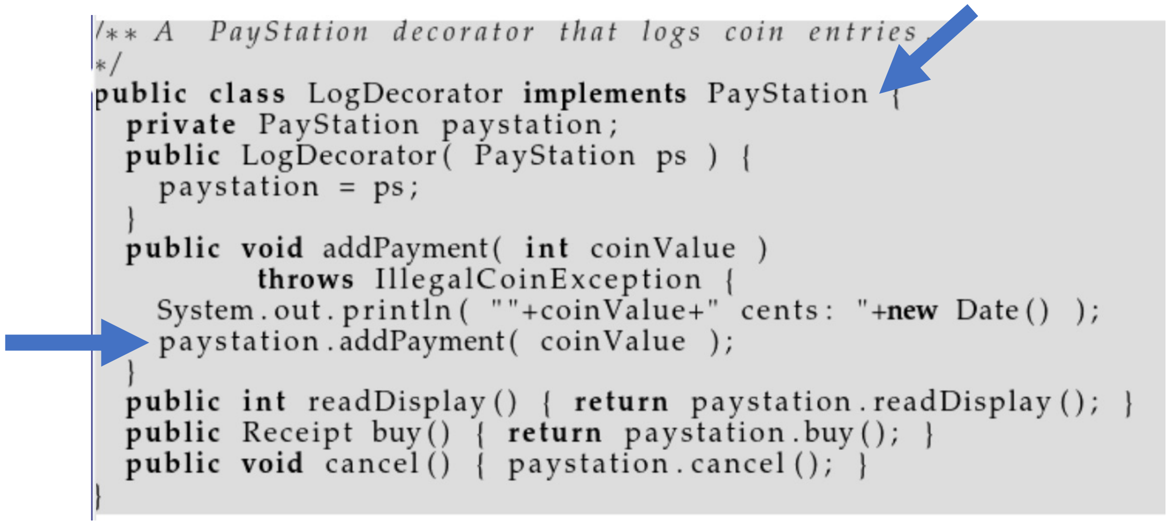
→ PaymentAcceptor? PayStation?

(2) We will **compose** the required behavior by putting an **intermediate object** with the same interface in front of the pay station object

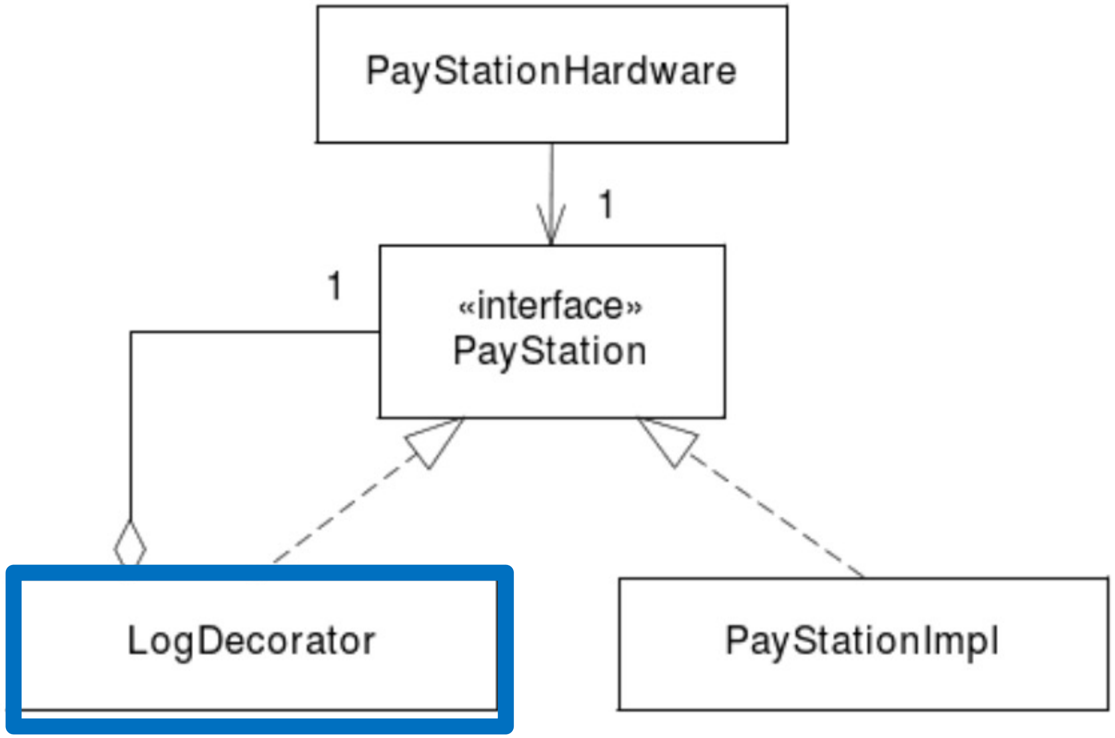
We've "decorated" the pay station's behavior with the additional behavior of logging coins



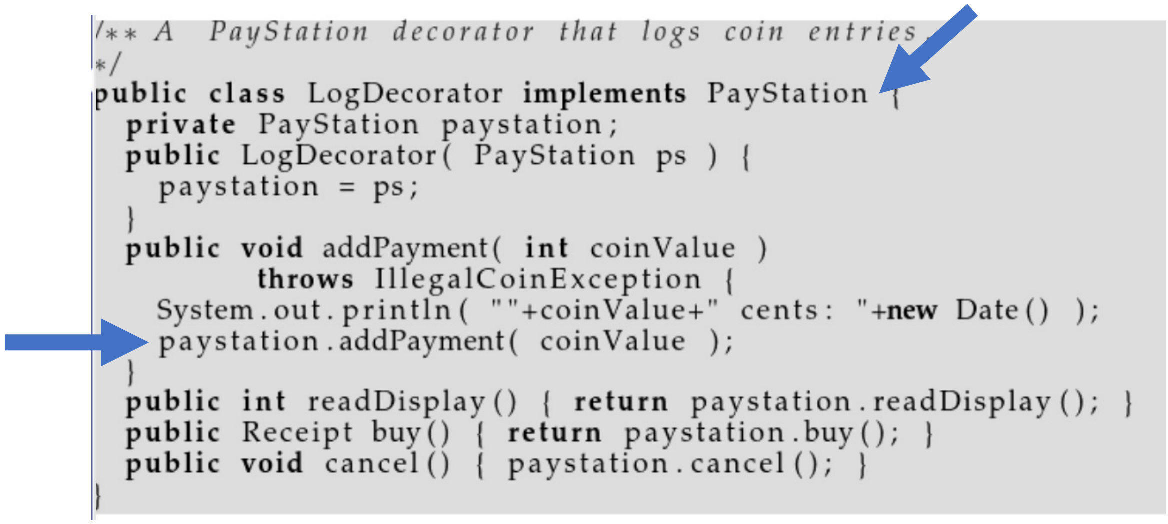
Decorator Pattern



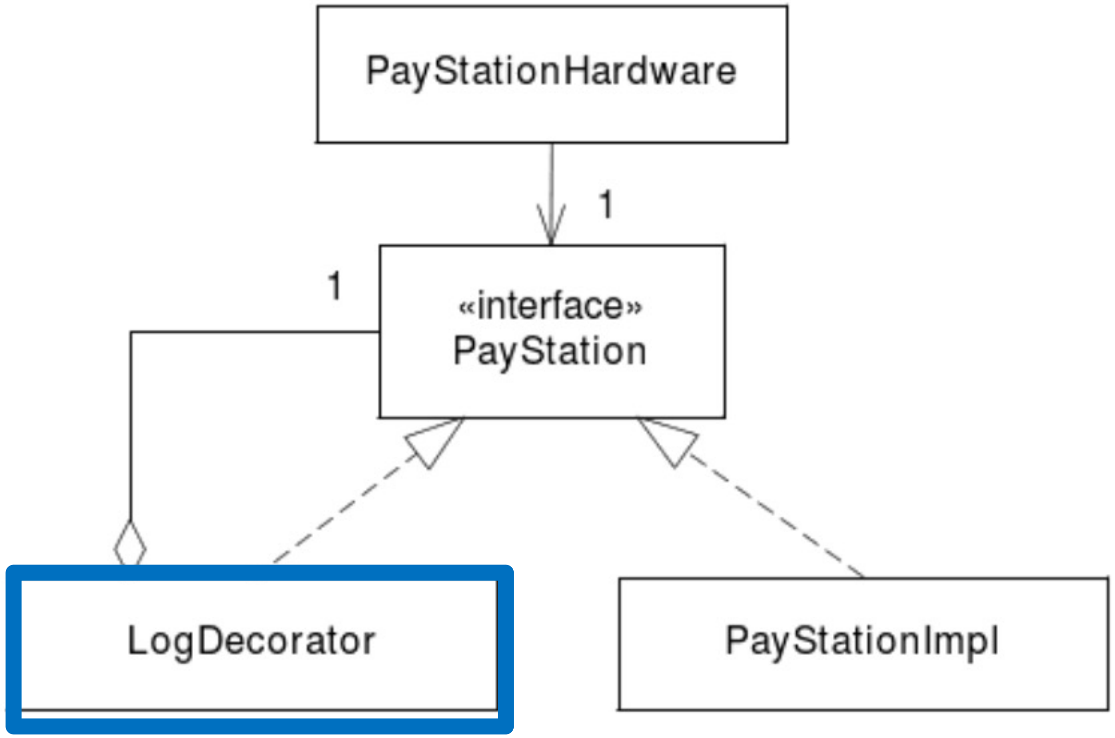
```
/** A  PayStation  decorator  that  logs  coin  entries.
*/
public class LogDecorator implements PayStation {
  private PayStation paystation;
  public LogDecorator( PayStation ps ) {
    paystation = ps;
  }
  public void addPayment( int coinValue )
          throws IllegalCoinException {
    System.out.println( ""+coinValue+" cents: "+new Date() );
    paystation.addPayment( coinValue );
  }
  public int readDisplay() { return paystation.readDisplay(); }
  public Receipt buy() { return paystation.buy(); }
  public void cancel() { paystation.cancel(); }
}
```



Decorator Pattern



```
/** A  PayStation decorator that logs coin entries.
*/
public class LogDecorator implements PayStation {
  private PayStation paystation;
  public LogDecorator( PayStation ps ) {
    paystation = ps;
  }
  public void addPayment( int coinValue )
          throws IllegalCoinException {
    System.out.println( ""+coinValue+" cents: "+new Date() );
    paystation.addPayment( coinValue );
  }
  public int readDisplay() { return paystation.readDisplay(); }
  public Receipt buy() { return paystation.buy(); }
  public void cancel() { paystation.cancel(); }
}
```

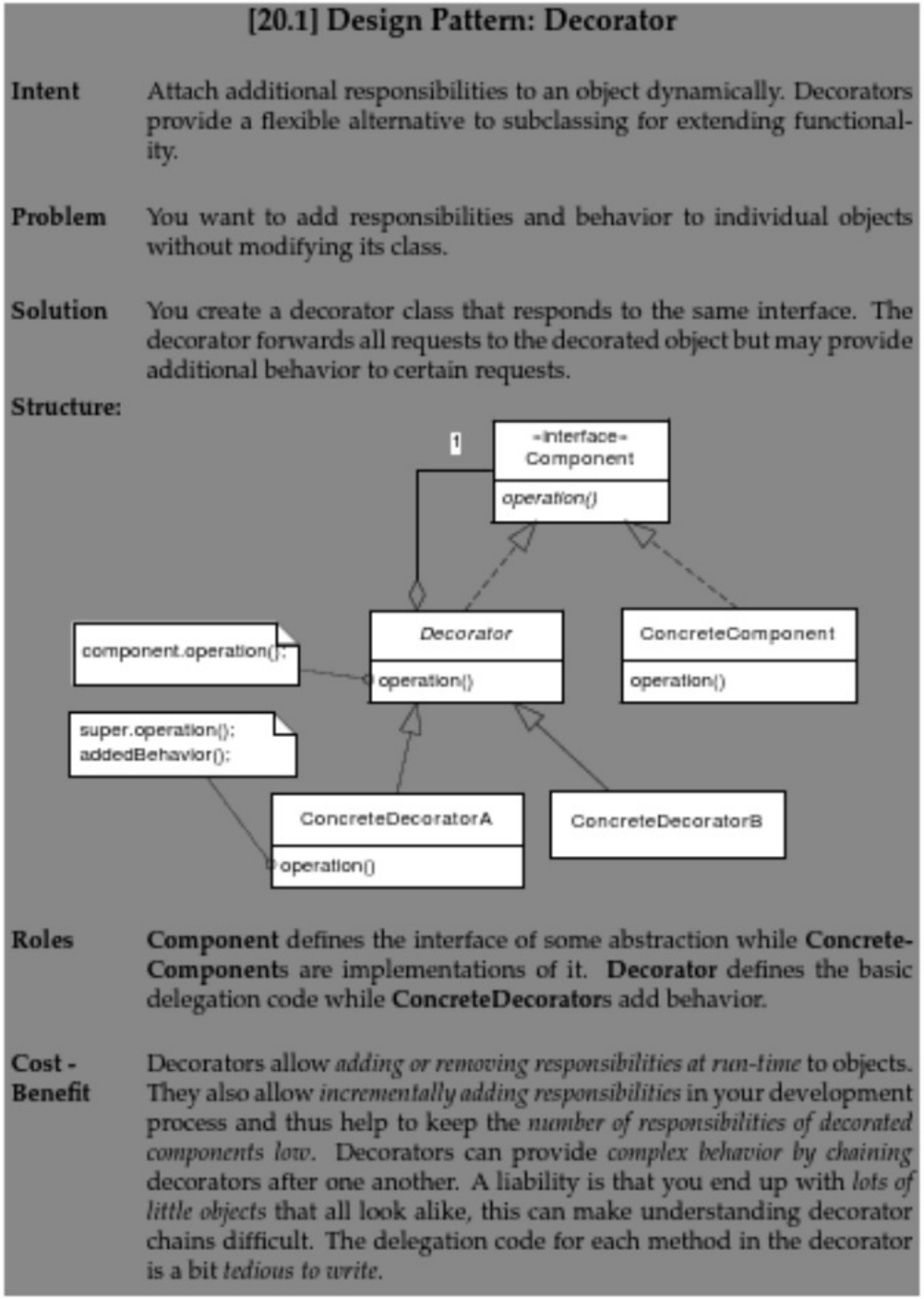


Decorator: Attach additional responsibilities to an object dynamically

Decorator Pattern

Problem: We want to add responsibilities/behavior to an object without modifying it

Solution: Create a decorator with the same interface as the object; the decorator forwards requests to the decorated object, but may provide additional behavior for some requests



[20.1] Design Pattern: Decorator

Intent Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Problem You want to add responsibilities and behavior to individual objects without modifying its class.

Solution You create a decorator class that responds to the same interface. The decorator forwards all requests to the decorated object but may provide additional behavior to certain requests.

Structure:

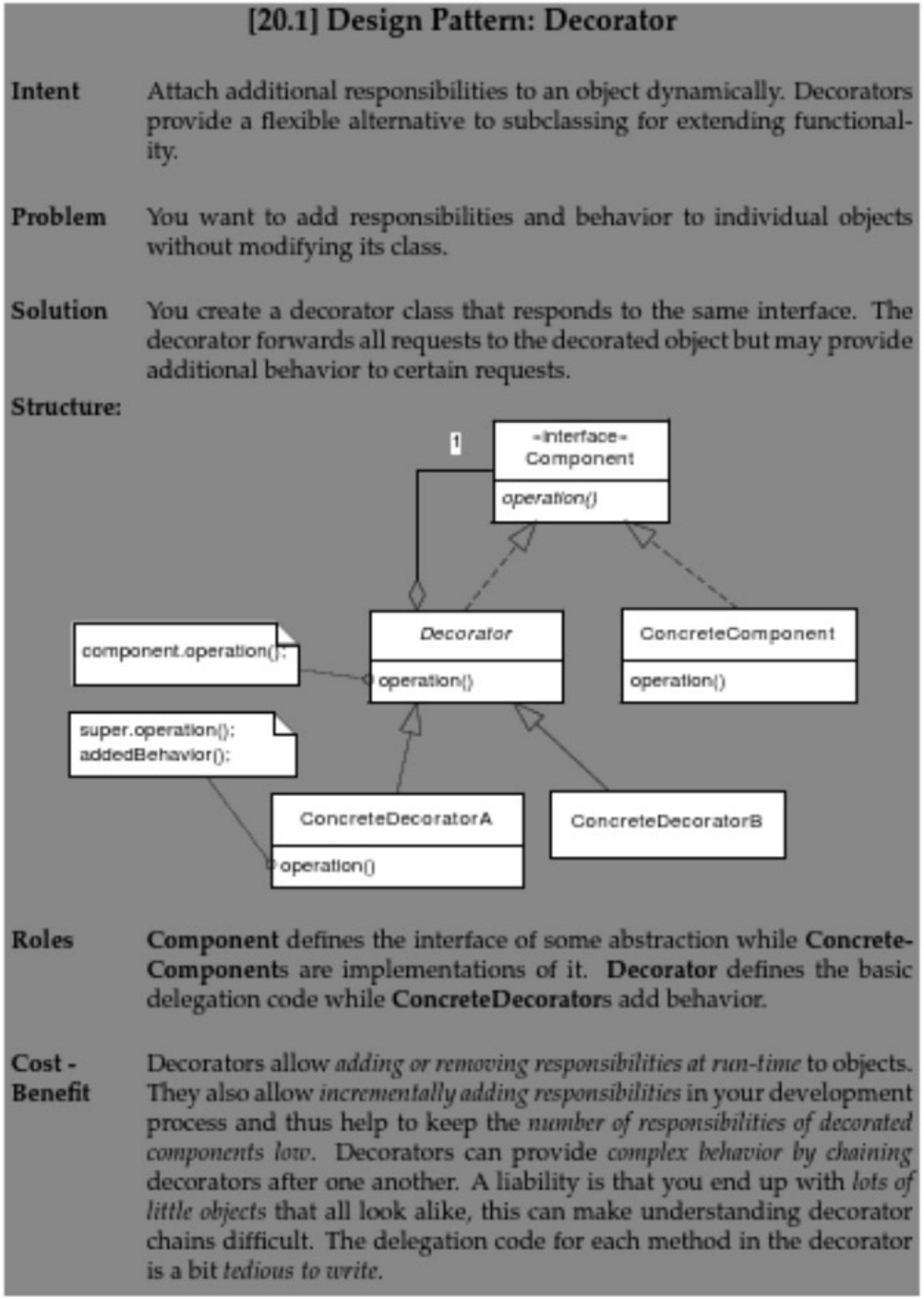
Roles **Component** defines the interface of some abstraction while **Concrete-Components** are implementations of it. **Decorator** defines the basic delegation code while **ConcreteDecorator**s add behavior.

Cost - Benefit Decorators allow *adding or removing responsibilities at run-time* to objects. They also allow *incrementally adding responsibilities* in your development process and thus help to keep the *number of responsibilities of decorated components low*. Decorators can provide *complex behavior by chaining* decorators after one another. A liability is that you end up with *lots of little objects* that all look alike, this can make understanding decorator chains difficult. The delegation code for each method in the decorator is a bit *tedious to write*.

Decorator Pattern

☺

- Composition is key: Add responsibilities to a role without affecting the underlying object
- General solution: Can decorate other implementations besides AlphaTown
- Decorators can be **chained**



[20.1] Design Pattern: Decorator

Intent Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Problem You want to add responsibilities and behavior to individual objects without modifying its class.

Solution You create a decorator class that responds to the same interface. The decorator forwards all requests to the decorated object but may provide additional behavior to certain requests.

Structure:

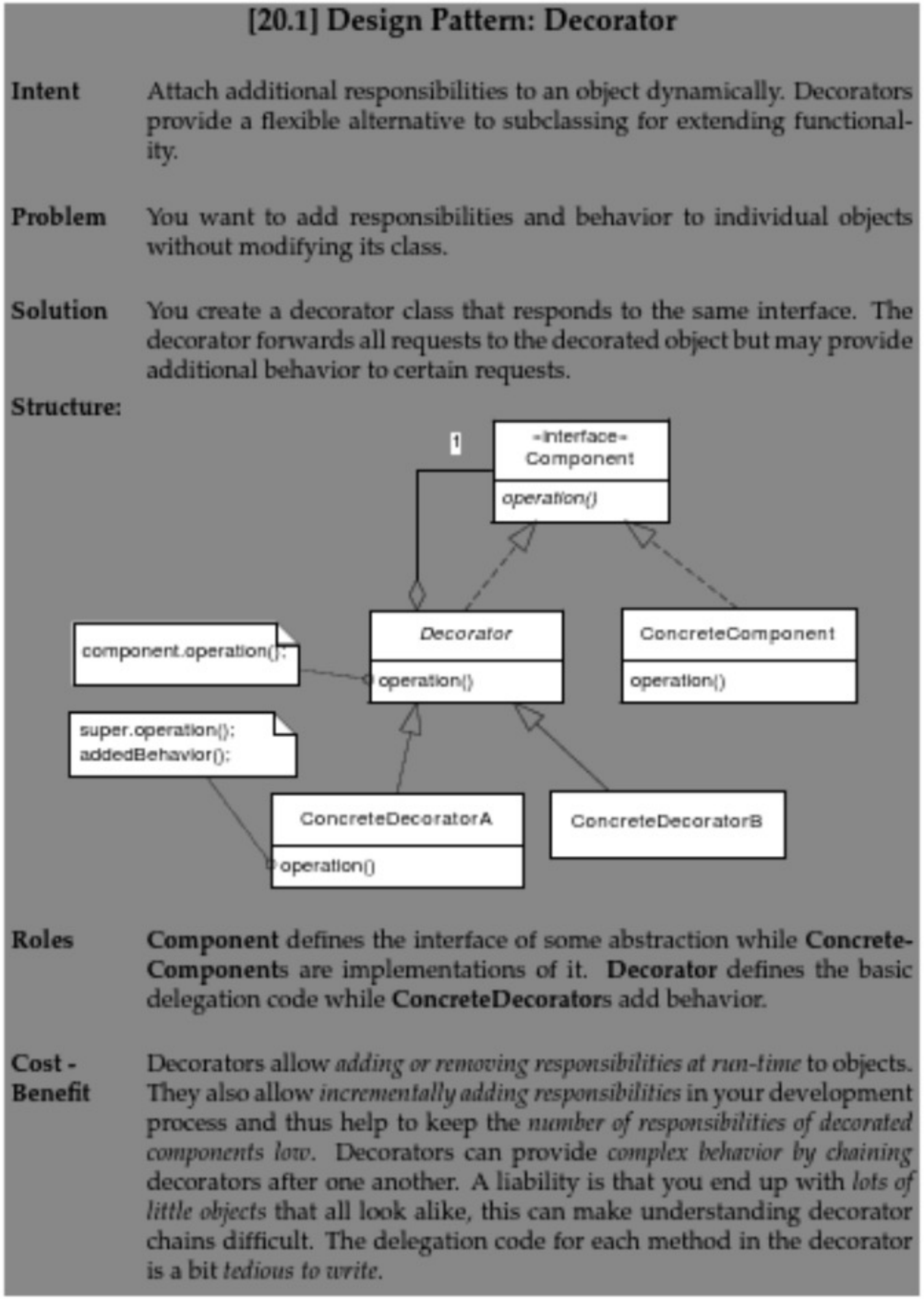
Roles **Component** defines the interface of some abstraction while **Concrete-Components** are implementations of it. **Decorator** defines the basic delegation code while **ConcreteDecorators** add behavior.

Cost - Benefit Decorators allow *adding or removing responsibilities at run-time* to objects. They also allow *incrementally adding responsibilities* in your development process and thus help to keep the *number of responsibilities of decorated components low*. Decorators can provide *complex behavior by chaining* decorators after one another. A liability is that you end up with *lots of little objects* that all look alike, this can make understanding decorator chains difficult. The delegation code for each method in the decorator is a bit *tedious to write*.

Decorator Pattern

☹

- Analyzability may suffer due to distribution of behavior across separate classes
- Delegation code in the decorator can be tedious to write



[20.1] Design Pattern: Decorator

Intent Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Problem You want to add responsibilities and behavior to individual objects without modifying its class.

Solution You create a decorator class that responds to the same interface. The decorator forwards all requests to the decorated object but may provide additional behavior to certain requests.

Structure:

Roles **Component** defines the interface of some abstraction while **ConcreteComponents** are implementations of it. **Decorator** defines the basic delegation code while **ConcreteDecorators** add behavior.

Cost - Benefit Decorators allow *adding or removing responsibilities at run-time* to objects. They also allow *incrementally adding responsibilities* in your development process and thus help to keep the *number of responsibilities of decorated components low*. Decorators can provide *complex behavior by chaining* decorators after one another. A liability is that you end up with *lots of little objects* that all look alike, this can make understanding decorator chains difficult. The delegation code for each method in the decorator is a bit *tedious to write*.

Adapter Pattern

LunaTown wants to use our pay station, but they want rate correlated with **phases of the moon** (?!)

Adapter Pattern

LunaTown wants to use our pay station, but they want rate correlated with **phases of the moon** (?!)

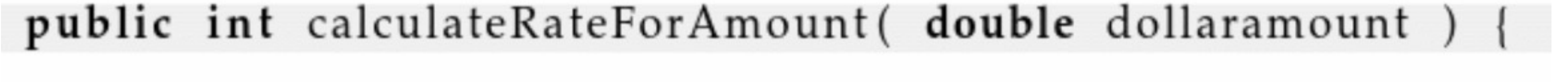
- At full moon, rates should equal Alphatown
- At new moon, rates should be doubled
- In between (waxing and waning), rates should proportionally vary between the extremes

Adapter Pattern

LunaTown wants to use our pay station, but they want rate correlated with **phases of the moon** (?!)

- At full moon, rates should equal Alphatown
- At new moon, rates should be doubled
- In between (waxing and waning), rates should proportionally vary between the extremes

LunaTown provides a class to do the calculation:



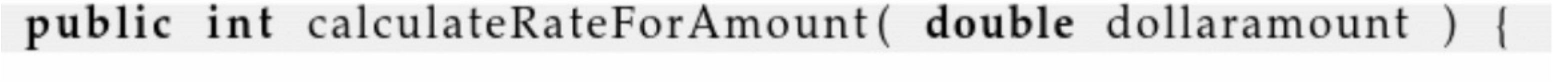
```
public int calculateRateForAmount( double dollaramount ) {
```

Adapter Pattern

LunaTown wants to use our pay station, but they want rate correlated with **phases of the moon** (?!)

- At full moon, rates should equal Alphatown
- At new moon, rates should be doubled
- In between (waxing and waning), rates should proportionally vary between the extremes

LunaTown provides a class to do the calculation:



```
public int calculateRateForAmount( double dollaramount ) {
```

but the interface does not follow our production code conventions, and we don’t have access to the source code… ☹

Adapter Pattern

(3) Variability: Rate calculation

(1) Interface: RateStrategy

(2) Composition:

- Provided calculator does not implement RateStrategy

→Use an **intermediate object** between the pay station and LunaTown calculator

Adapter Pattern

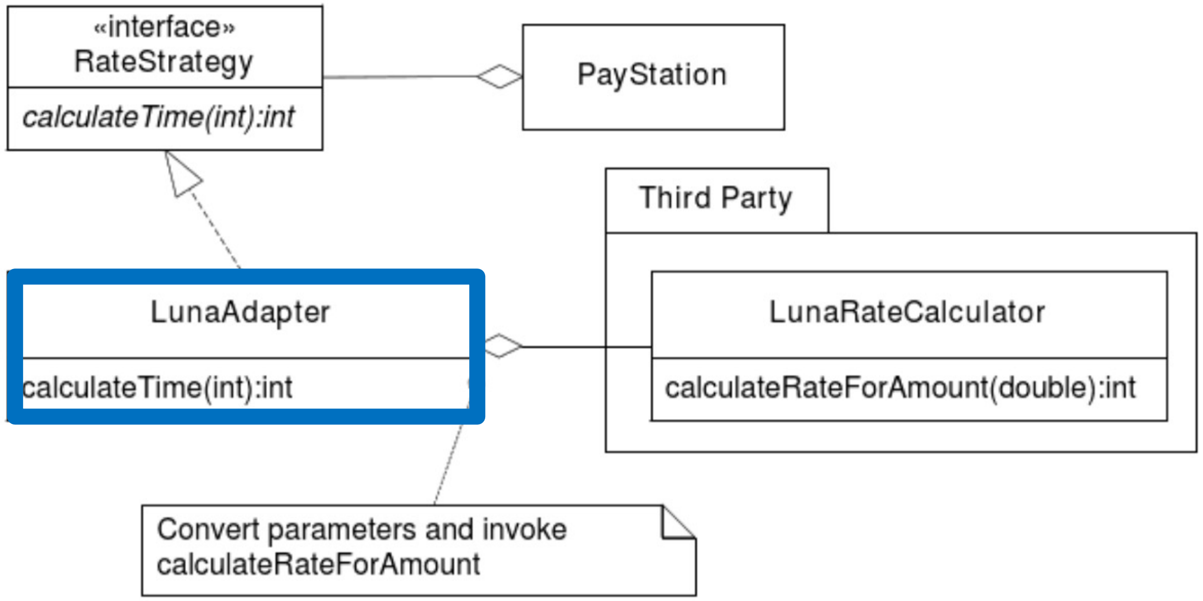
(3) Variability: Rate calculation

(1) Interface: RateStrategy

(2) Composition:

- Provided calculator does not implement RateStrategy

→Use an **intermediate object** between the pay station and LunaTown calculator



Adapter Pattern

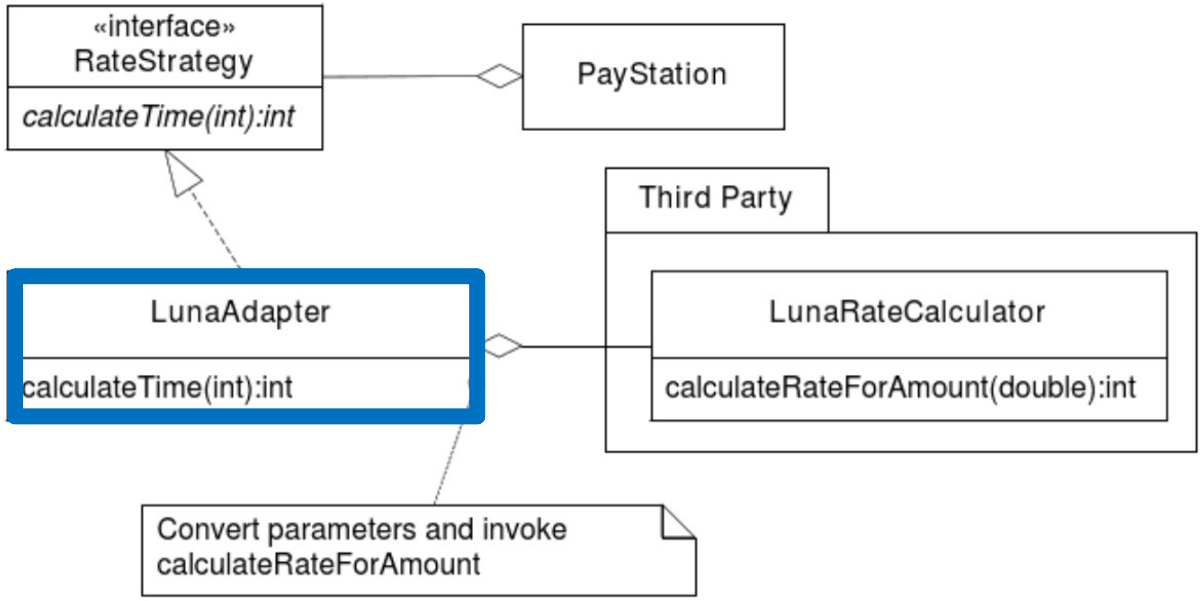
(3) Variability: Rate calculation

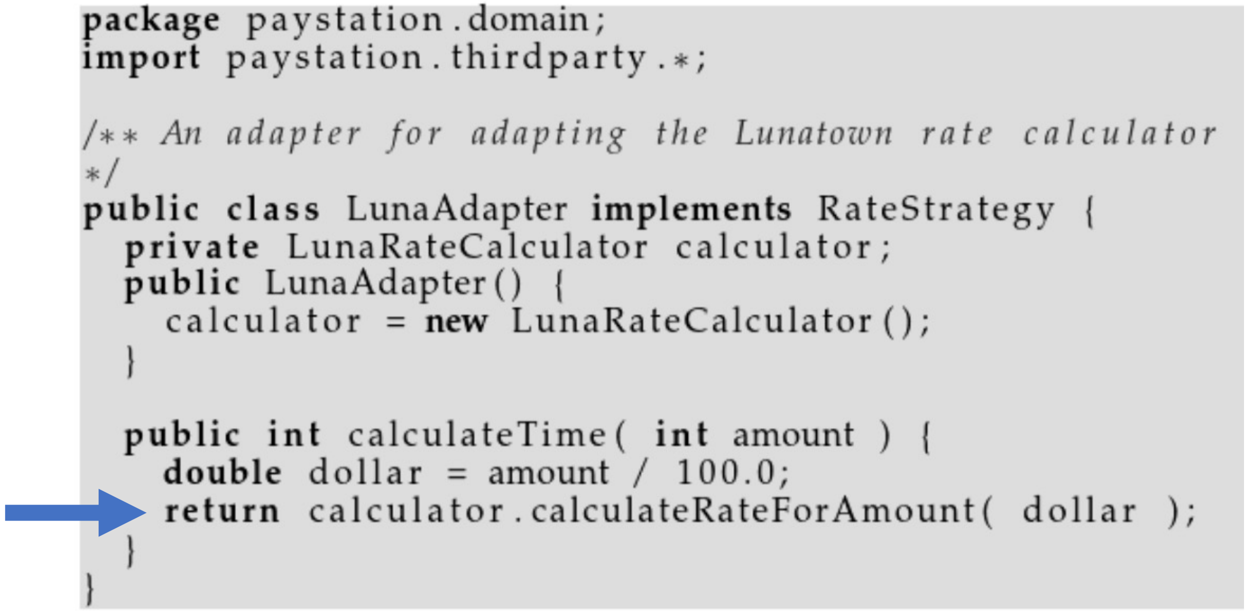
(1) Interface: RateStrategy

(2) Composition:

- Provided calculator does not implement RateStrategy

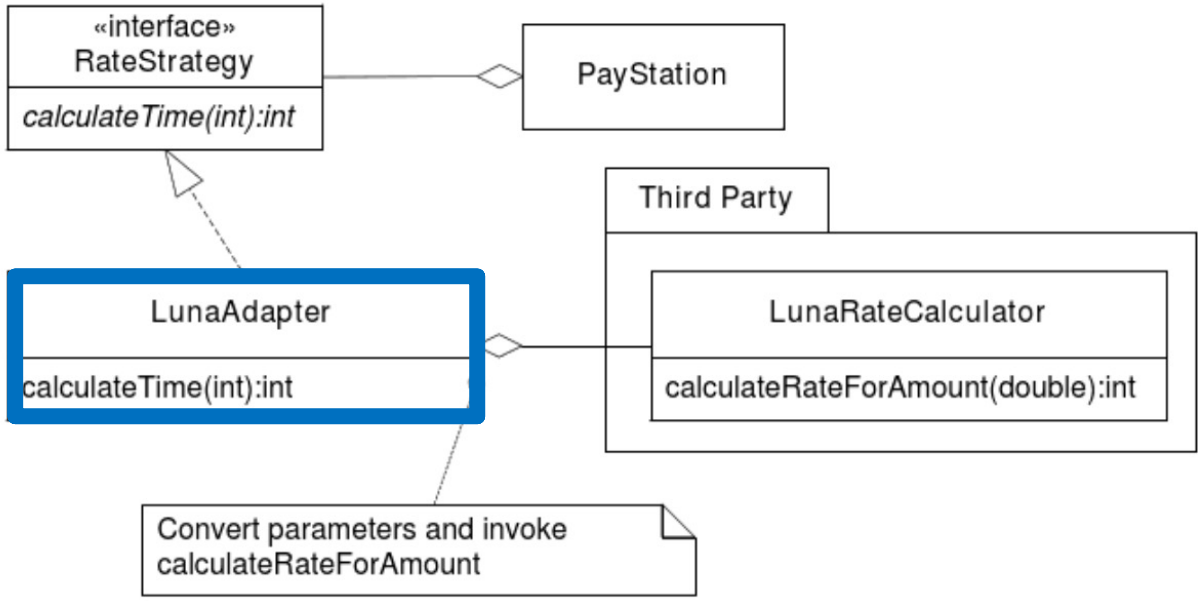
→Use an **intermediate object** between the pay station and LunaTown calculator





```
package paystation.domain;
import paystation.thirdparty.*;

/** An adapter for adapting the Lunatown rate calculator
*/
public class LunaAdapter implements RateStrategy {
  private LunaRateCalculator calculator;
  public LunaAdapter() {
    calculator = new LunaRateCalculator();
  }

  public int calculateTime( int amount ) {
    double dollar = amount / 100.0;
    return calculator.calculateRateForAmount( dollar );
  }
}
```

Adapter Pattern



(3) Variability: Rate calculation

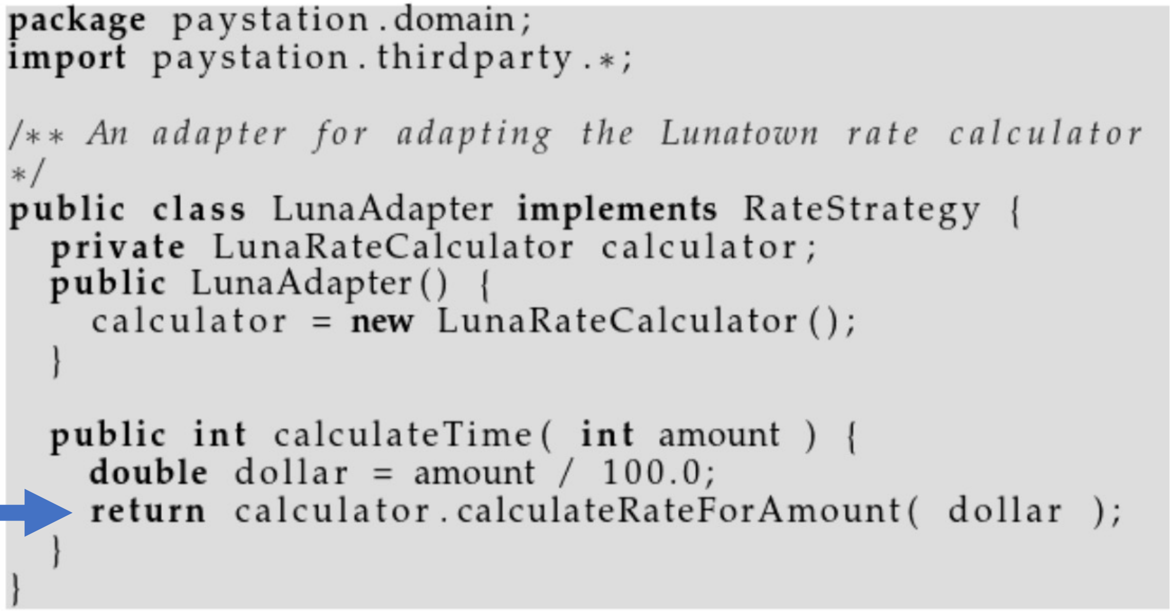
(1) Interface: RateStrategy

(2) Composition:

- Provided calculator does not implement RateStrategy

→Use an **intermediate object** between the pay station and LunaTown calculator

Adapter: Convert the interface of a class into another interface that clients expect.

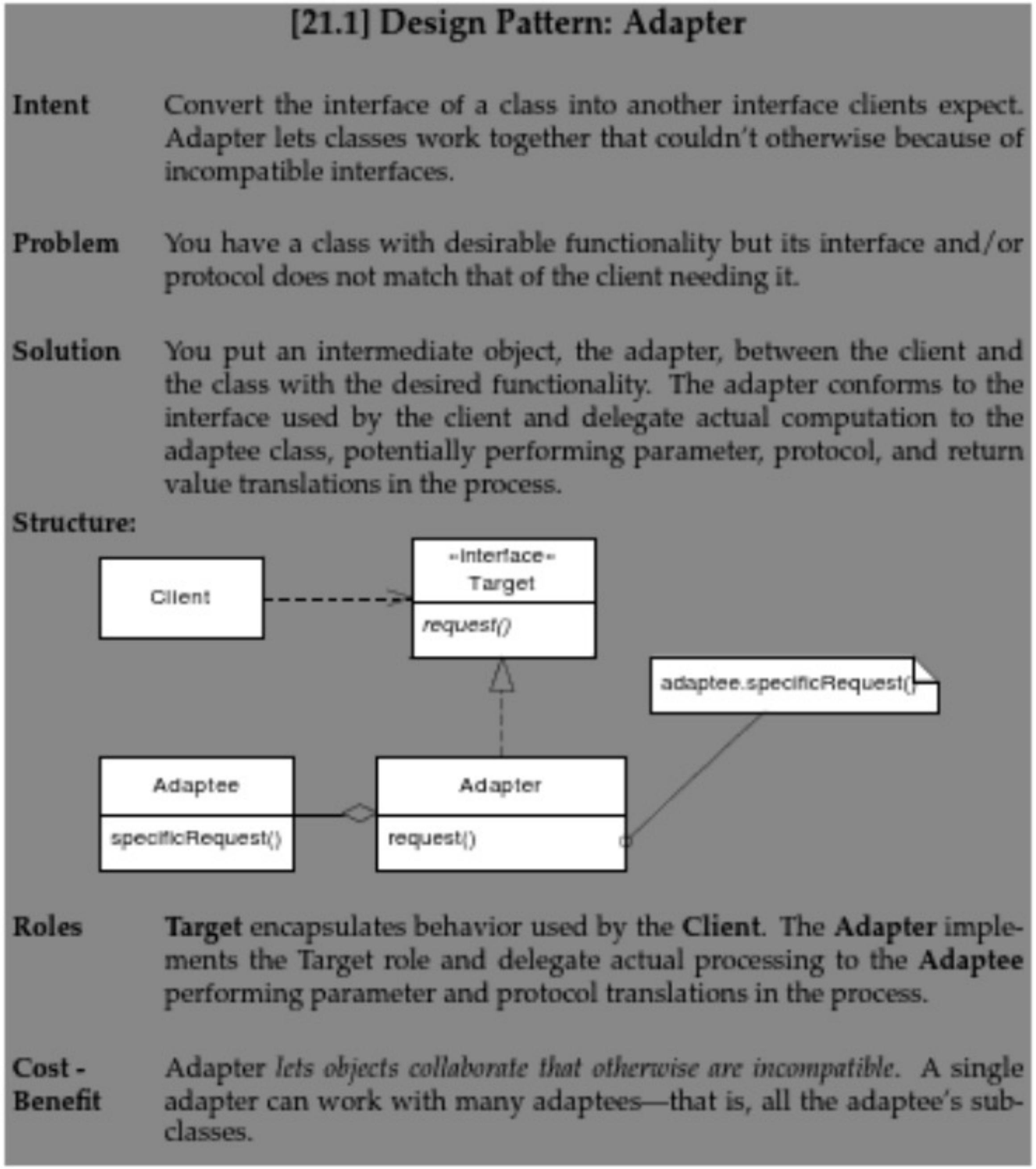


```
package paystation.domain;
import paystation.thirdparty.*;

/** An adapter for adapting the Lunatown rate calculator
*/
public class LunaAdapter implements RateStrategy {
  private LunaRateCalculator calculator;
  public LunaAdapter() {
    calculator = new LunaRateCalculator();
  }

  public int calculateTime( int amount ) {
    double dollar = amount / 100.0;
    return calculator.calculateRateForAmount( dollar );
  }
}
```

Adapter Pattern

Problem: We have a class with desirable functionality but its interface/protocol do not match the client that needs it

Solution: Put an **adapter** object between the client and the class with the necessary functionality (the adaptee); the adapter conforms to the client's expected interface but delegates to the adaptee



[21.1] Design Pattern: Adapter

Intent Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Problem You have a class with desirable functionality but its interface and/or protocol does not match that of the client needing it.

Solution You put an intermediate object, the adapter, between the client and the class with the desired functionality. The adapter conforms to the interface used by the client and delegate actual computation to the adaptee class, potentially performing parameter, protocol, and return value translations in the process.

Structure:

Roles **Target** encapsulates behavior used by the **Client**. The **Adapter** implements the Target role and delegate actual processing to the **Adaptee** performing parameter and protocol translations in the process.

Cost - Benefit Adapter *lets objects collaborate that otherwise are incompatible*. A single adapter can work with many adaptees—that is, all the adaptee's subclasses.

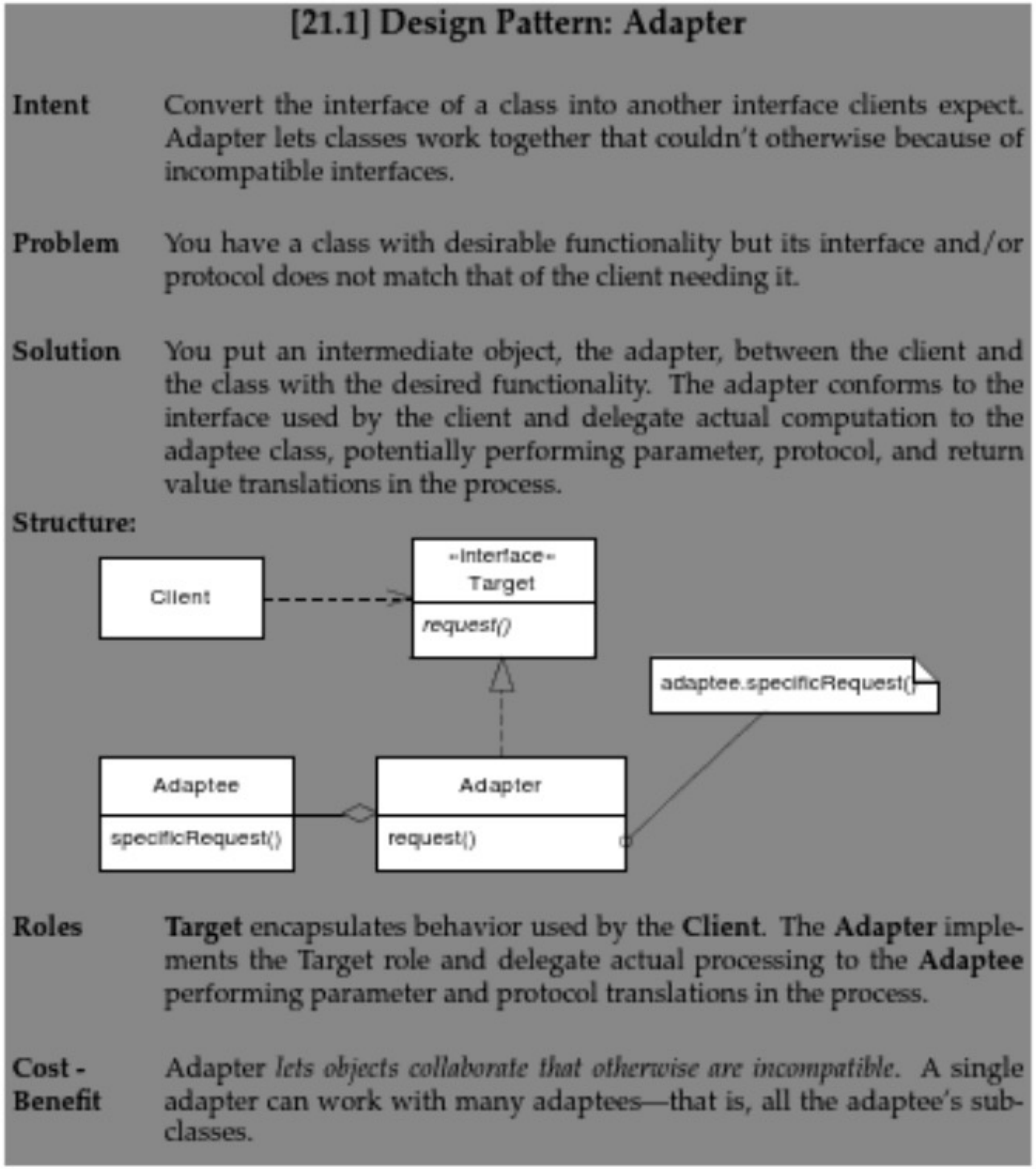
Adapter Pattern

☺

- Adapter can adapt all implementations/subclasses of the adaptee

☹

- Adapter can't be reused for other adaptee classes



[21.1] Design Pattern: Adapter

Intent Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Problem You have a class with desirable functionality but its interface and/or protocol does not match that of the client needing it.

Solution You put an intermediate object, the adapter, between the client and the class with the desired functionality. The adapter conforms to the interface used by the client and delegate actual computation to the adaptee class, potentially performing parameter, protocol, and return value translations in the process.

Structure:

Roles **Target** encapsulates behavior used by the **Client**. The **Adapter** implements the Target role and delegate actual processing to the **Adaptee** performing parameter and protocol translations in the process.

Cost - Benefit Adapter *lets objects collaborate that otherwise are incompatible*. A single adapter can work with many adaptees—that is, all the adaptee's subclasses.

Builder Pattern

Suppose we have a word processor that stores documents with some internal data structure, and we want to export documents in different formats (e.g., XML, HTML, ASCII)

Builder Pattern

Suppose we have a word processor that stores documents with some internal data structure, and we want to export documents in different formats (e.g., XML, HTML, ASCII)

- Code to iterate through the internal data structure will be identical
- Shared components of data structure: sections, subsections, paragraphs, …
- Only the parts that build the **output representation** will differ

Builder Pattern

Suppose we have a word processor that stores documents with some internal data structure, and we want to export documents in different formats (e.g., XML, HTML, ASCII)

- Code to iterate through the internal data structure will be identical
- Shared components of data structure: sections, subsections, paragraphs, …
- Only the parts that build the **output representation** will differ

How can we avoid the **multiple maintenance problem?**

Builder Pattern

(3) Variability: Construction of output of parts like section, subsection, etc. (but the parts are common)

Builder Pattern

(3) Variability: Construction of output of parts like section, subsection, etc. (but the parts are common)

(1) Interface: Encapsulate the construction of parts in a **builder** interface, with methods to build each part (e.g., buildSection)

Builder Pattern

(3) Variability: Construction of output of parts like section, subsection, etc. (but the parts are common)

(1) Interface: Encapsulate the construction of parts in a **builder** interface, with methods to build each part (e.g., buildSection)

(2) Composition: Write the data structure iterator once (**director**) and have it request a delegate **builder** to construct the parts

Builder Pattern

(3) Variability: Construction of output of parts like section, subsection, etc. (but the parts are common)

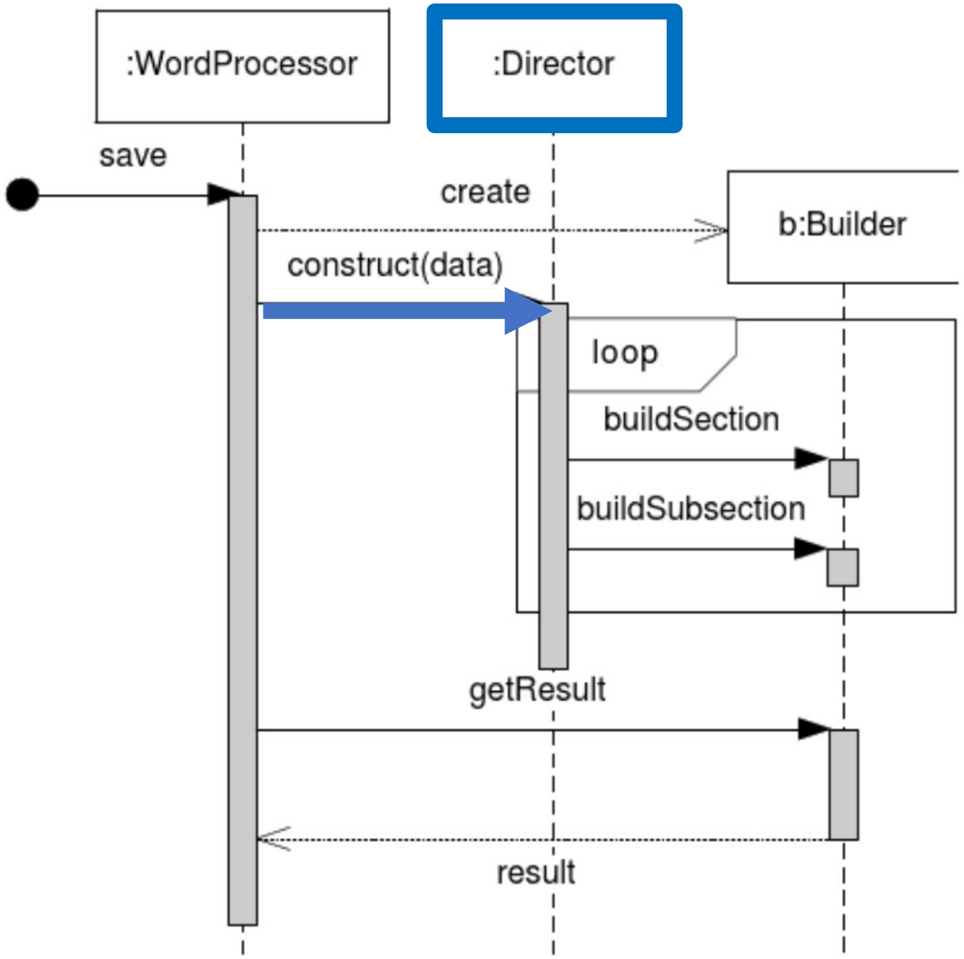
(1) Interface: Encapsulate the construction of parts in a **builder** interface, with methods to build each part (e.g., buildSection)

(2) Composition: Write the data structure iterator once (**director**) and have it request a delegate **builder** to construct the parts

Builder: Separate the construction of a complex object from its representation so that the same construction process can create different representations

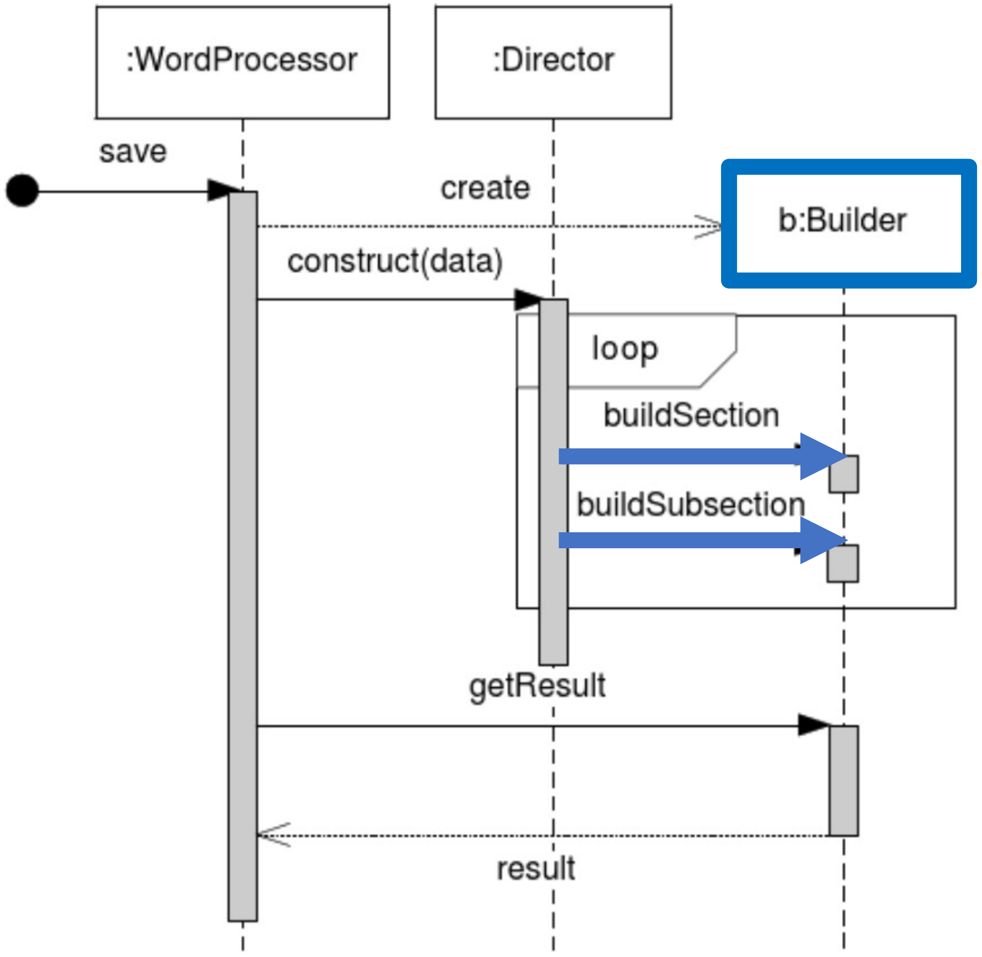
Builder Pattern

- The **director** contains the construct() method for iterating over the data structure



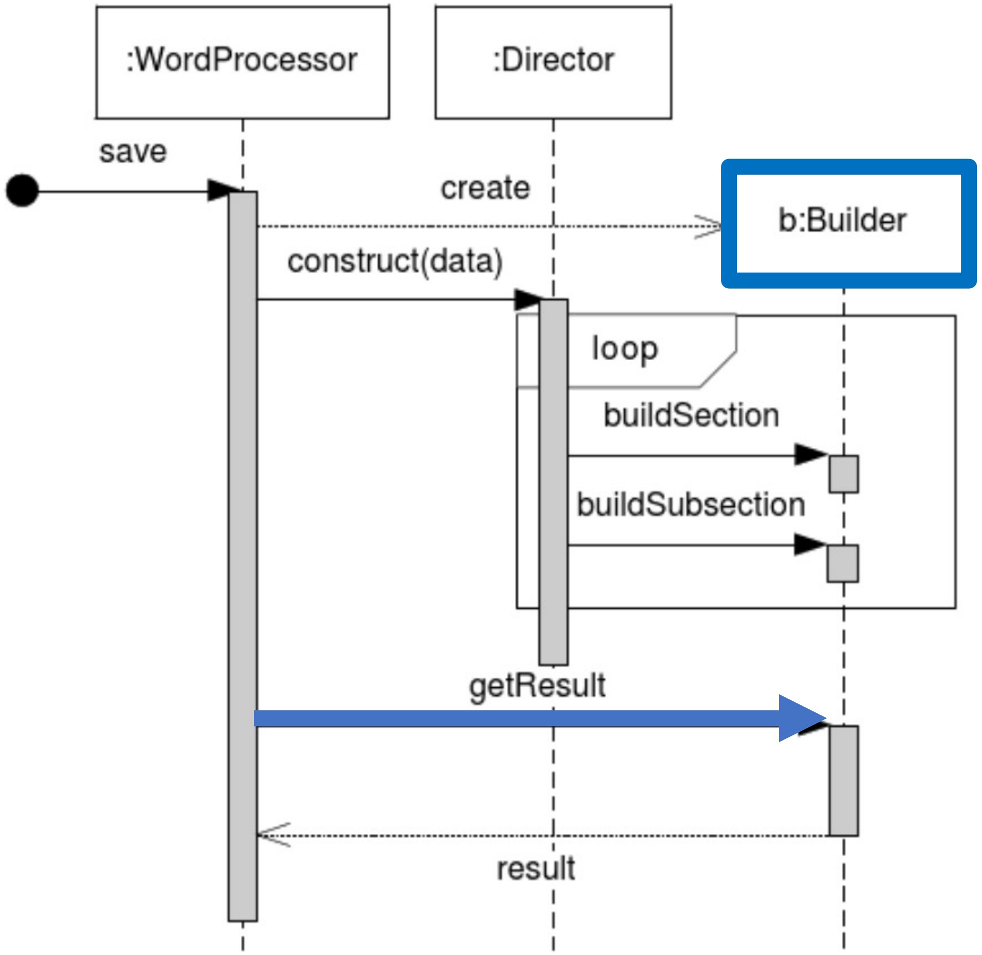
Builder Pattern

- The **director** contains the construct() method for iterating over the data structure
- The **builder** contains methods to build each part based on output format



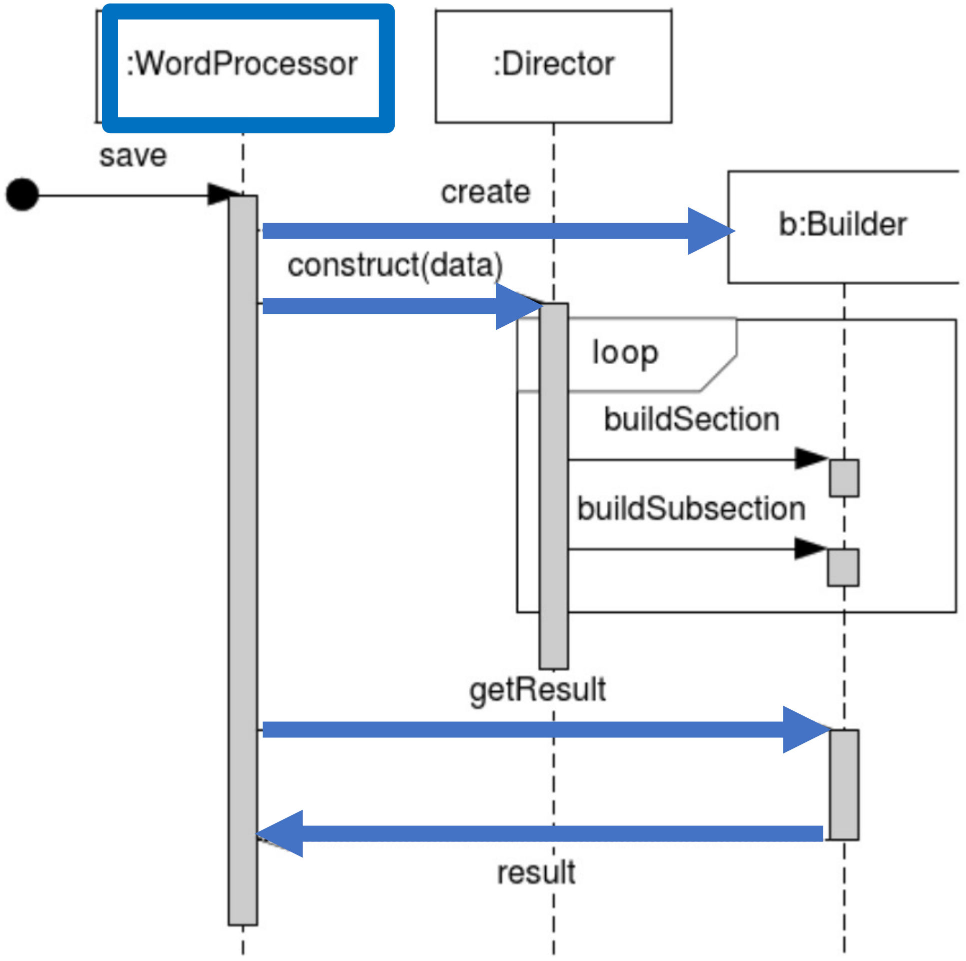
Builder Pattern

- The **director** contains the construct() method for iterating over the data structure
- The **builder** contains methods to build each part based on output format
- **ConcreteBuilders** implement each representation of the **product**, i.e., the output
    - getResult()



Builder Pattern

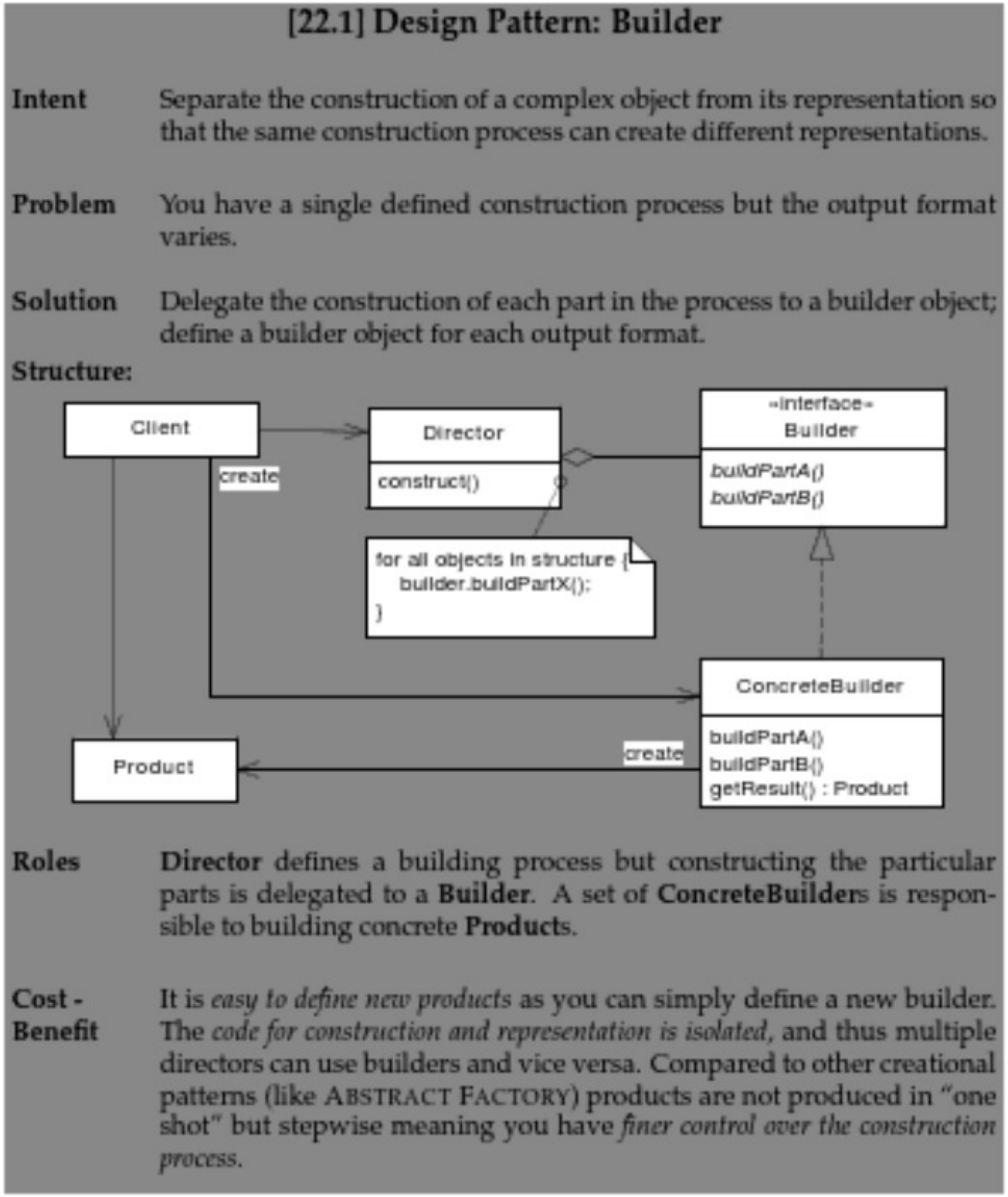
- The **director** contains the construct() method for iterating over the data structure
- The **builder** contains methods to build each part based on output format
- **ConcreteBuilders** implement each representation of the **product**, i.e., the output
  - getResult()
- The **client** creates the concrete builder, asks the director to construct, and retrieves the product from the builder



Builder Pattern

Problem: We have a single construction process but the output format varies

Solution: Delegate construction of each output part to a builder object, define a builder object for each output format



[22.1] Design Pattern: Builder

Intent Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Problem You have a single defined construction process but the output format varies.

Solution Delegate the construction of each part in the process to a builder object; define a builder object for each output format.

Structure:

Roles **Director** defines a building process but constructing the particular parts is delegated to a **Builder**. A set of **ConcreteBuilder**s is responsible to building concrete **Products**.

Cost - Benefit It is *easy to define new products* as you can simply define a new builder. The *code for construction and representation is isolated*, and thus multiple directors can use builders and vice versa. Compared to other creational patterns (like ABSTRACT FACTORY) products are not produced in "one shot" but stepwise meaning you have *finer control over the construction process*.

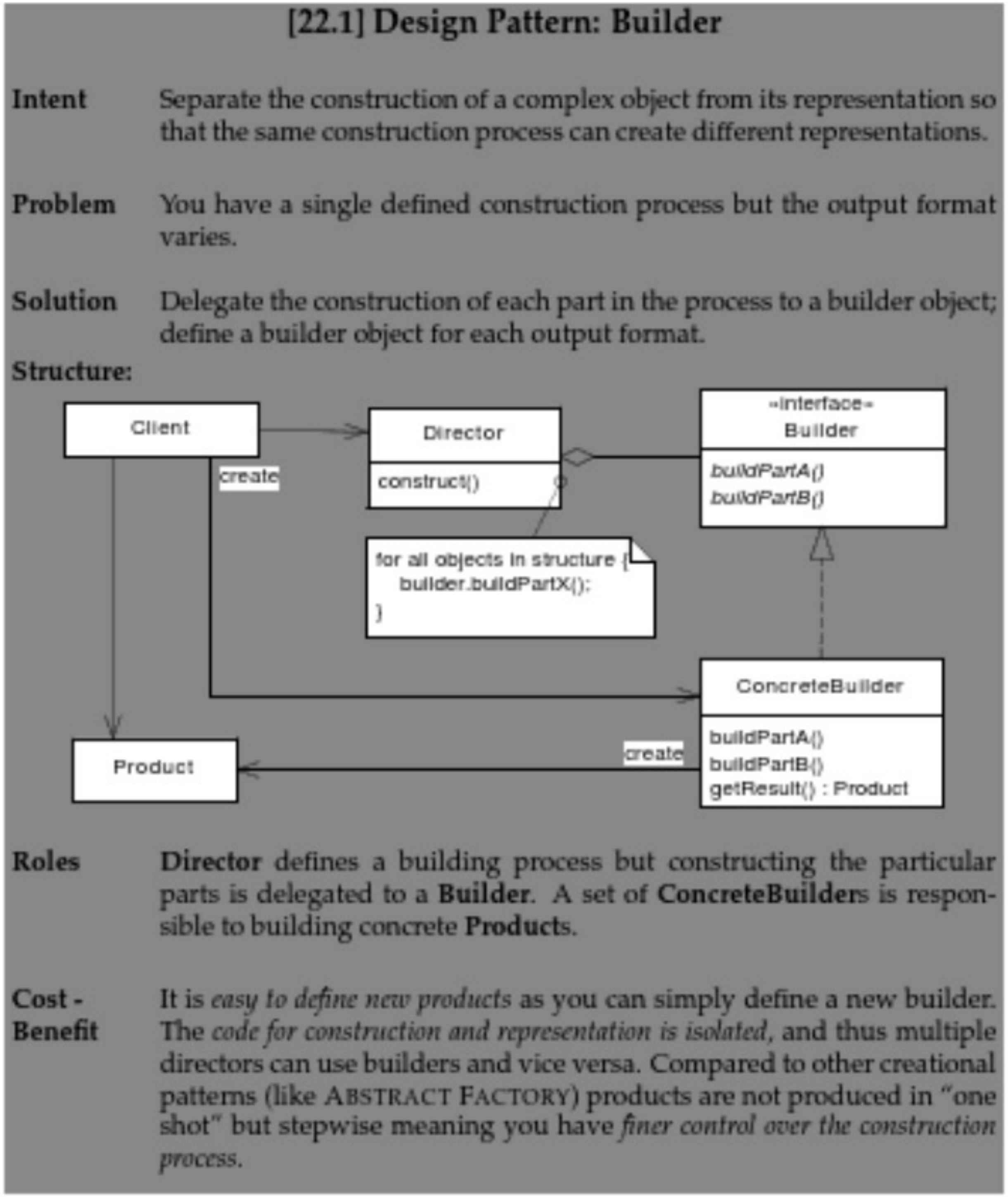
Builder Pattern

Problem: We have a single construction process but the output format varies

Solution: Delegate construction of each output part to a builder object, define a builder object for each output format

Builder is a **creational pattern**

→Similar to Abstract Factory, but provides more granular control over individual elements



[22.1] Design Pattern: Builder

Intent Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Problem You have a single defined construction process but the output format varies.

Solution Delegate the construction of each part in the process to a builder object; define a builder object for each output format.

Structure:

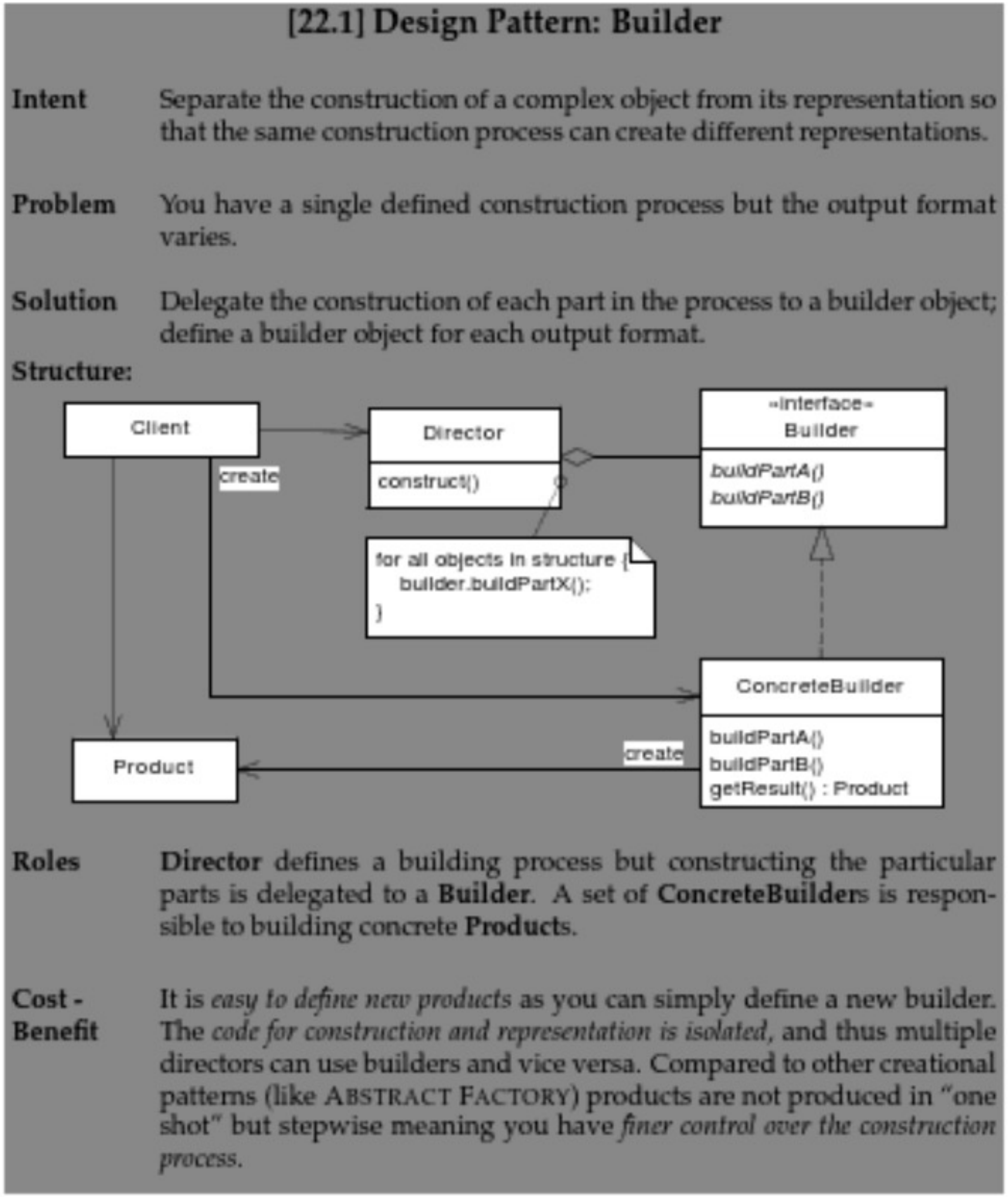
Roles **Director** defines a building process but constructing the particular parts is delegated to a **Builder**. A set of **ConcreteBuilder**s is responsible to building concrete **Product**s.

Cost - Benefit It is *easy to define new products* as you can simply define a new builder. The *code for construction and representation is isolated*, and thus multiple directors can use builders and vice versa. Compared to other creational patterns (like ABSTRACT FACTORY) products are not produced in "one shot" but stepwise meaning you have *finer control over the construction process*.

Builder Pattern

☺

- Separating construction process and concrete part building means we can use builders for other purposes
    - Favors object composition
- Easy to define new builders



[22.1] Design Pattern: Builder

Intent Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Problem You have a single defined construction process but the output format varies.

Solution Delegate the construction of each part in the process to a builder object; define a builder object for each output format.

Structure:

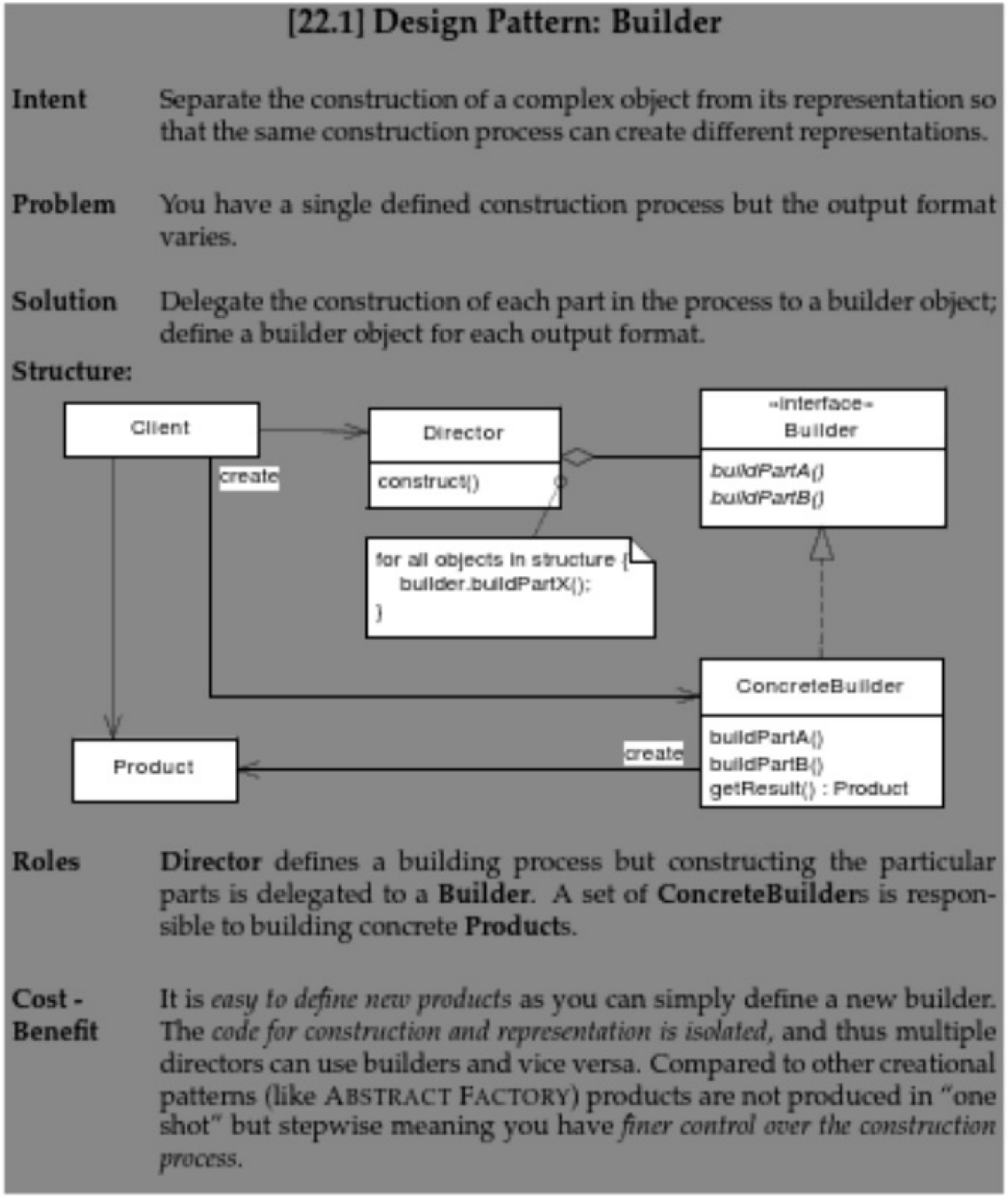
Roles **Director** defines a building process but constructing the particular parts is delegated to a **Builder**. A set of **ConcreteBuilders** is responsible to building concrete **Products**.

Cost - Benefit It is *easy to define new products* as you can simply define a new builder. The *code for construction and representation is isolated*, and thus multiple directors can use builders and vice versa. Compared to other creational patterns (like ABSTRACT FACTORY) products are not produced in "one shot" but stepwise meaning you have *finer control over the construction process.*

Builder Pattern

☹

- Complex setup of construction project
- Client must know product of the builder as well as concrete builder types



[22.1] Design Pattern: Builder

Intent Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Problem You have a single defined construction process but the output format varies.

Solution Delegate the construction of each part in the process to a builder object; define a builder object for each output format.

Structure:

Roles **Director** defines a building process but constructing the particular parts is delegated to a **Builder**. A set of **ConcreteBuilder**s is responsible to building concrete **Products**.

Cost - Benefit It is *easy to define new products* as you can simply define a new builder. The *code for construction and representation is isolated*, and thus multiple directors can use builders and vice versa. Compared to other creational patterns (like ABSTRACT FACTORY) products are not produced in "one shot" but stepwise meaning you have *finer control over the construction process.*

Command Pattern

Suppose we want to build an application with UI elements that lets the user customize shortcut keys/buttons, define macros, and perform undo operations

Command Pattern

Suppose we want to build an application with UI elements that lets the user customize shortcut keys/buttons, define macros, and perform undo operations

→Challenge: behavior is defined by object methods, which cannot be stored or passed as parameters

Command Pattern

Suppose we want to build an application with UI elements that lets the user customize shortcut keys/buttons, define macros, and perform undo operations

→Challenge: behavior is defined by object methods, which cannot be stored or passed as parameters

→Solution: Make objects that encapsulate the methods

Command Pattern

(3) Encapsulate what varies: We need to handle behavior as objects that can be assigned to keys, put in macro lists, be “executable” and “un-executable” to support undo

Command Pattern

(3) Encapsulate what varies: We need to handle behavior as objects that can be assigned to keys, put in macro lists, be “executable” and “un-executable” to support undo

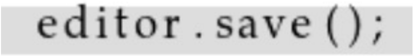
(1) Program to an interface: request objects must have a common interface to be exchanged across UI elements that enact them (**command** role, encapsulates “execute” and potentially “undo”)

Command Pattern

(3) Encapsulate what varies: We need to handle behavior as objects that can be assigned to keys, put in macro lists, be “executable” and “un-executable” to support undo

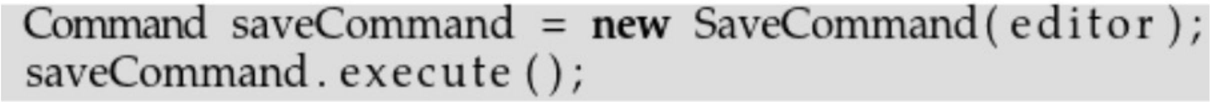
(1) Program to an interface: request objects must have a common interface to be exchanged across UI elements that enact them (**command** role, encapsulates “execute” and potentially “undo”)

(2) Favor object composition: Instead of UI elements with hardcoded behavior, they delegate to their assigned command objects



```
editor.save();
```

→



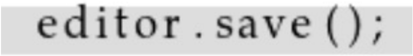
```
Command saveCommand = new SaveCommand(editor);
saveCommand.execute();
```

Command Pattern

(3) Encapsulate what varies: We need to handle behavior as objects that can be assigned to keys, put in macro lists, be “executable” and “un-executable” to support undo

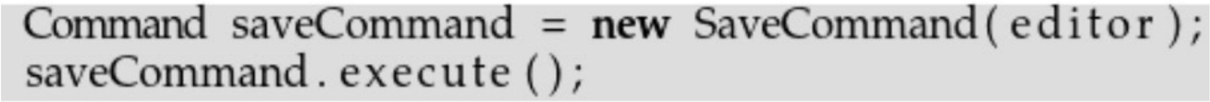
(1) Program to an interface: request objects must have a common interface to be exchanged across UI elements that enact them (**command** role, encapsulates “execute” and potentially “undo”)

(2) Favor object composition: Instead of UI elements with hardcoded behavior, they delegate to their assigned command objects



```
editor.save();
```

→



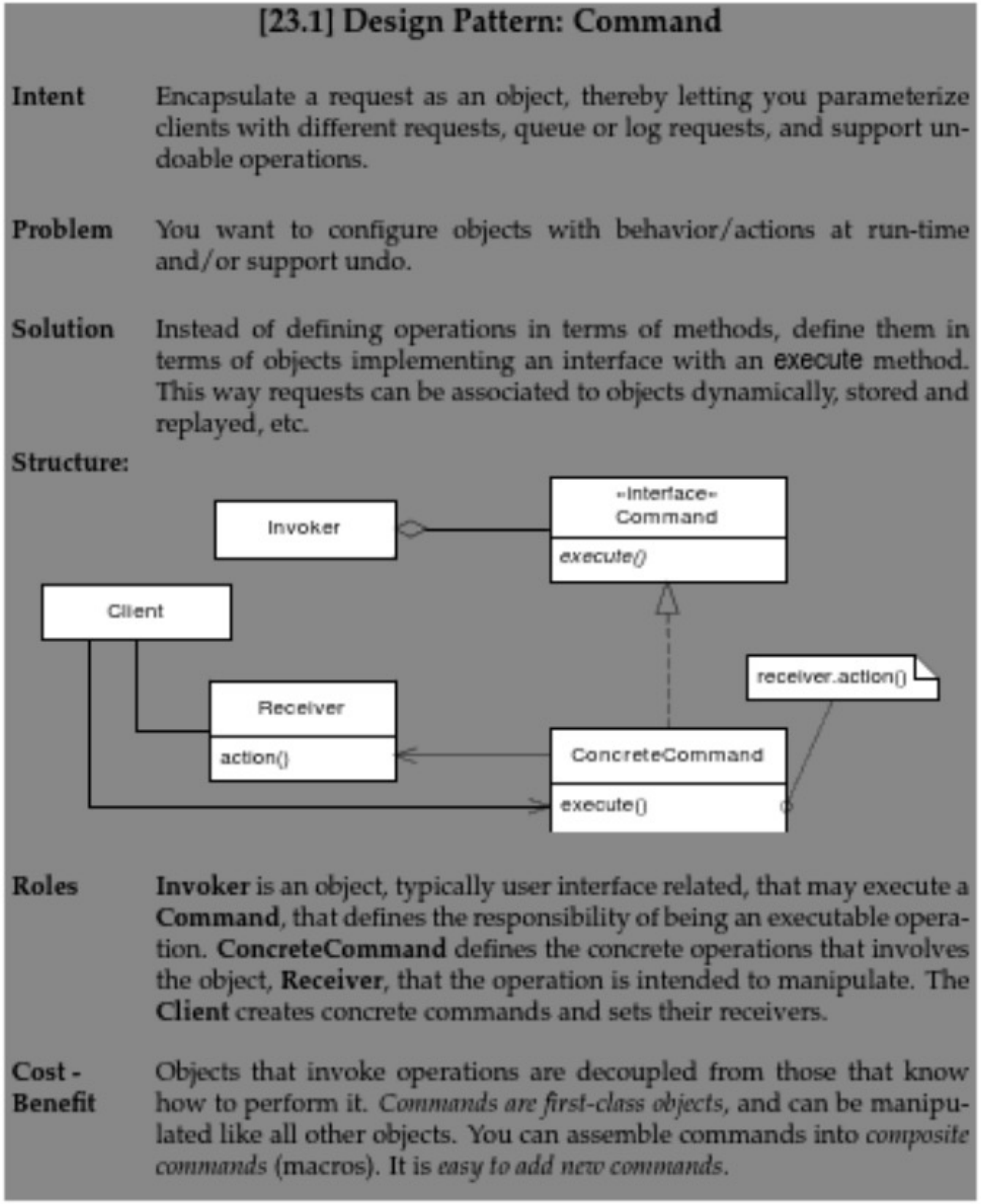
```
Command saveCommand = new SaveCommand(editor);
saveCommand.execute();
```

Command: Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations

Command Pattern

Problem: We want to configure objects with behavior/actions at runtime, and/or support undoing actions

Solution: Define operations in terms of objects implementing an interface with an **execute** method



[23.1] Design Pattern: Command

Intent Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Problem You want to configure objects with behavior/actions at run-time and/or support undo.

Solution Instead of defining operations in terms of methods, define them in terms of objects implementing an interface with an execute method. This way requests can be associated to objects dynamically, stored and replayed, etc.

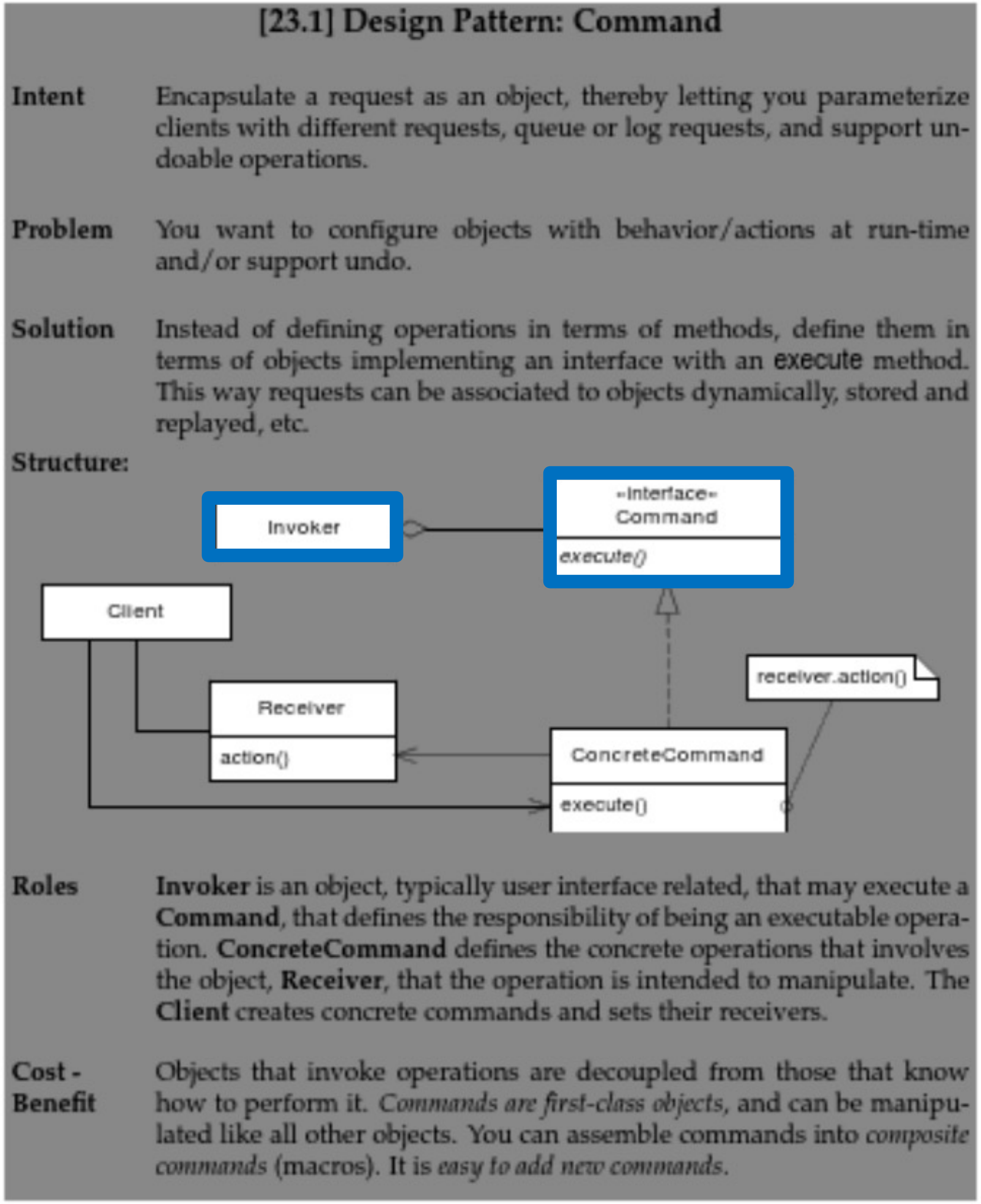
Structure:

Roles **Invoker** is an object, typically user interface related, that may execute a **Command**, that defines the responsibility of being an executable operation. **ConcreteCommand** defines the concrete operations that involves the object, **Receiver**, that the operation is intended to manipulate. The **Client** creates concrete commands and sets their receivers.

Cost - Benefit Objects that invoke operations are decoupled from those that know how to perform it. *Commands are first-class objects*, and can be manipulated like all other objects. You can assemble commands into *composite commands* (macros). It is *easy to add new commands*.

Command Pattern

Command defines the interface for execution, which is called by the **invoker**



[23.1] Design Pattern: Command

Intent Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Problem You want to configure objects with behavior/actions at run-time and/or support undo.

Solution Instead of defining operations in terms of methods, define them in terms of objects implementing an interface with an execute method. This way requests can be associated to objects dynamically, stored and replayed, etc.

Structure:

Roles **Invoker** is an object, typically user interface related, that may execute a **Command**, that defines the responsibility of being an executable operation. **ConcreteCommand** defines the concrete operations that involves the object, **Receiver**, that the operation is intended to manipulate. The **Client** creates concrete commands and sets their receivers.

Cost - Benefit Objects that invoke operations are decoupled from those that know how to perform it. *Commands are first-class objects*, and can be manipulated like all other objects. You can assemble commands into *composite commands* (macros). It is *easy to add new commands*.

Command Pattern

Command defines the interface for execution, which is called by the **invoker**

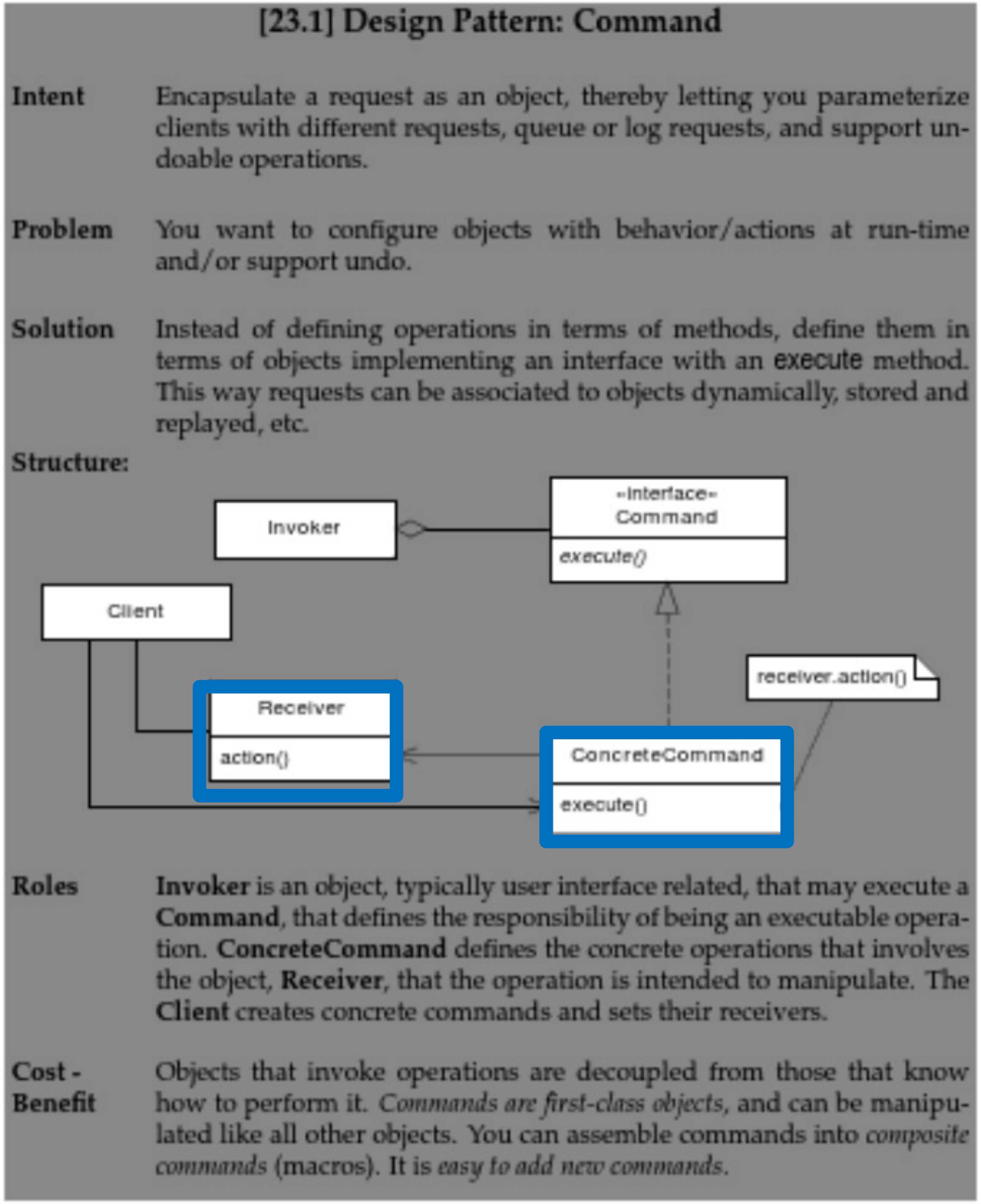
ConcreteCommand implements the command interface, and must know the **receiver** object to invoke the method on

→Command object must have the same set of parameters as the receiver

```
editor.save();
```

↓

```
Command saveCommand = new SaveCommand(editor);
saveCommand.execute();
```



[23.1] Design Pattern: Command

Intent Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Problem You want to configure objects with behavior/actions at run-time and/or support undo.

Solution Instead of defining operations in terms of methods, define them in terms of objects implementing an interface with an execute method. This way requests can be associated to objects dynamically, stored and replayed, etc.

Structure:

Roles **Invoker** is an object, typically user interface related, that may execute a **Command**, that defines the responsibility of being an executable operation. **ConcreteCommand** defines the concrete operations that involves the object, **Receiver**, that the operation is intended to manipulate. The **Client** creates concrete commands and sets their receivers.

Cost - Benefit Objects that invoke operations are decoupled from those that know how to perform it. *Commands are first-class objects*, and can be manipulated like all other objects. You can assemble commands into *composite commands* (macros). It is *easy to add new commands*.

Command Pattern

Command defines the interface for execution, which is called by the **invoker**

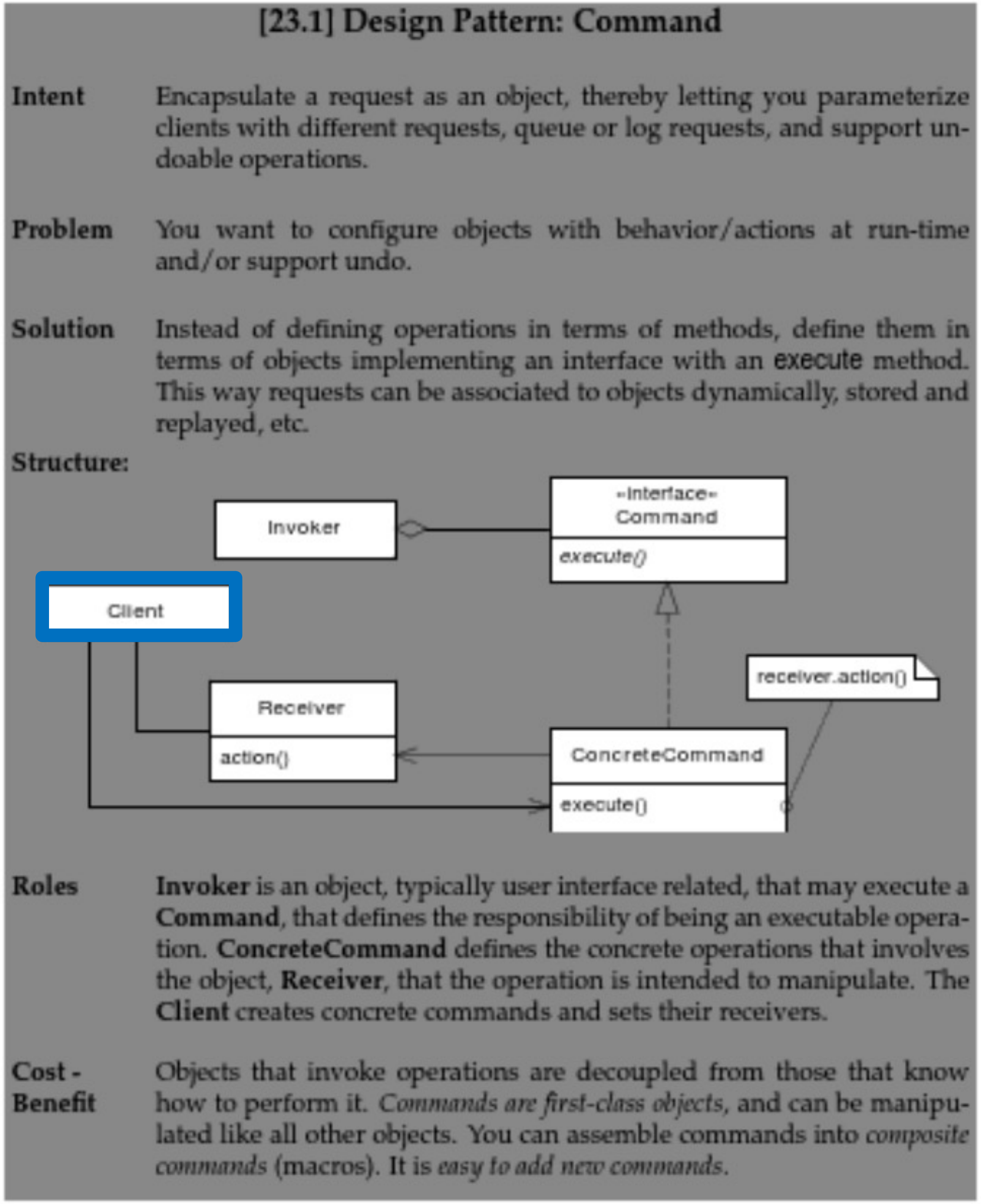
ConcreteCommand implements the command interface, and must know the **receiver** object to invoke the method on

→Command object must have the same set of parameters as the receiver

Client creates the command object and binds it to the receiver, and configures the invoker

```
Command saveCommand = new SaveCommand(editor);
```

```
saveCommand.execute();
```



[23.1] Design Pattern: Command

Intent Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Problem You want to configure objects with behavior/actions at run-time and/or support undo.

Solution Instead of defining operations in terms of methods, define them in terms of objects implementing an interface with an execute method. This way requests can be associated to objects dynamically, stored and replayed, etc.

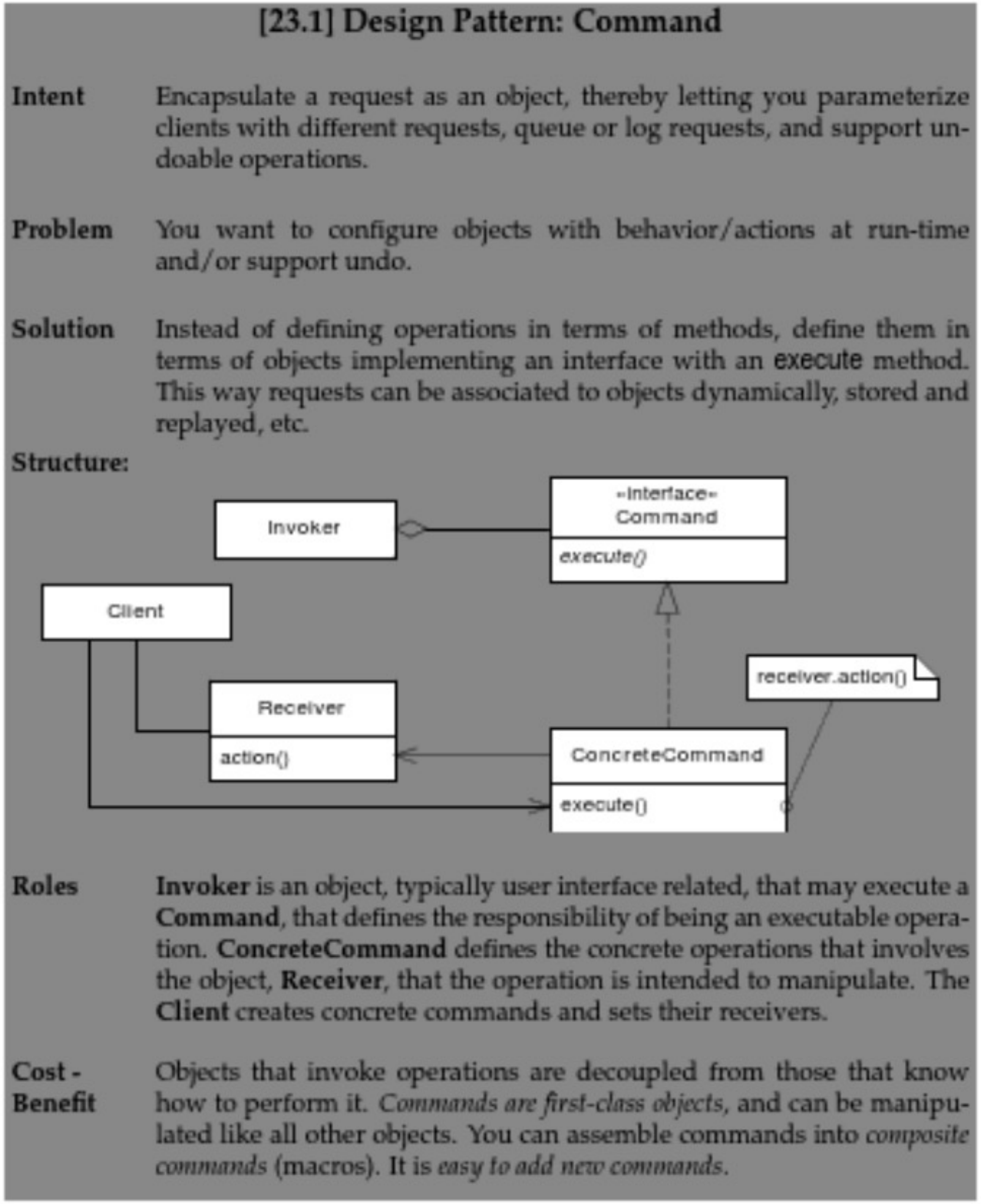
Structure:

Roles **Invoker** is an object, typically user interface related, that may execute a **Command**, that defines the responsibility of being an executable operation. **ConcreteCommand** defines the concrete operations that involves the object, **Receiver**, that the operation is intended to manipulate. The **Client** creates concrete commands and sets their receivers.

Cost - Benefit Objects that invoke operations are decoupled from those that know how to perform it. *Commands are first-class objects*, and can be manipulated like all other objects. You can assemble commands into *composite commands* (macros). It is *easy to add new commands*.

Command Pattern

- Clients are decoupled from set of commands
- Can extend command set at runtime
- Supports multiple ways to execute a command
- Can log and store commands



[23.1] Design Pattern: Command

Intent Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Problem You want to configure objects with behavior/actions at run-time and/or support undo.

Solution Instead of defining operations in terms of methods, define them in terms of objects implementing an interface with an execute method. This way requests can be associated to objects dynamically, stored and replayed, etc.

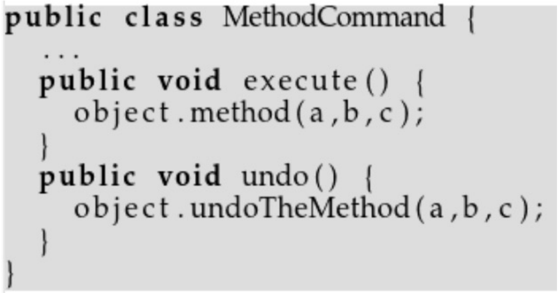
Structure:

Roles **Invoker** is an object, typically user interface related, that may execute a **Command**, that defines the responsibility of being an executable operation. **ConcreteCommand** defines the concrete operations that involves the object, **Receiver**, that the operation is intended to manipulate. The **Client** creates concrete commands and sets their receivers.

Cost - Benefit Objects that invoke operations are decoupled from those that know how to perform it. *Commands are first-class objects*, and can be manipulated like all other objects. You can assemble commands into *composite commands* (macros). It is *easy to add new commands*.

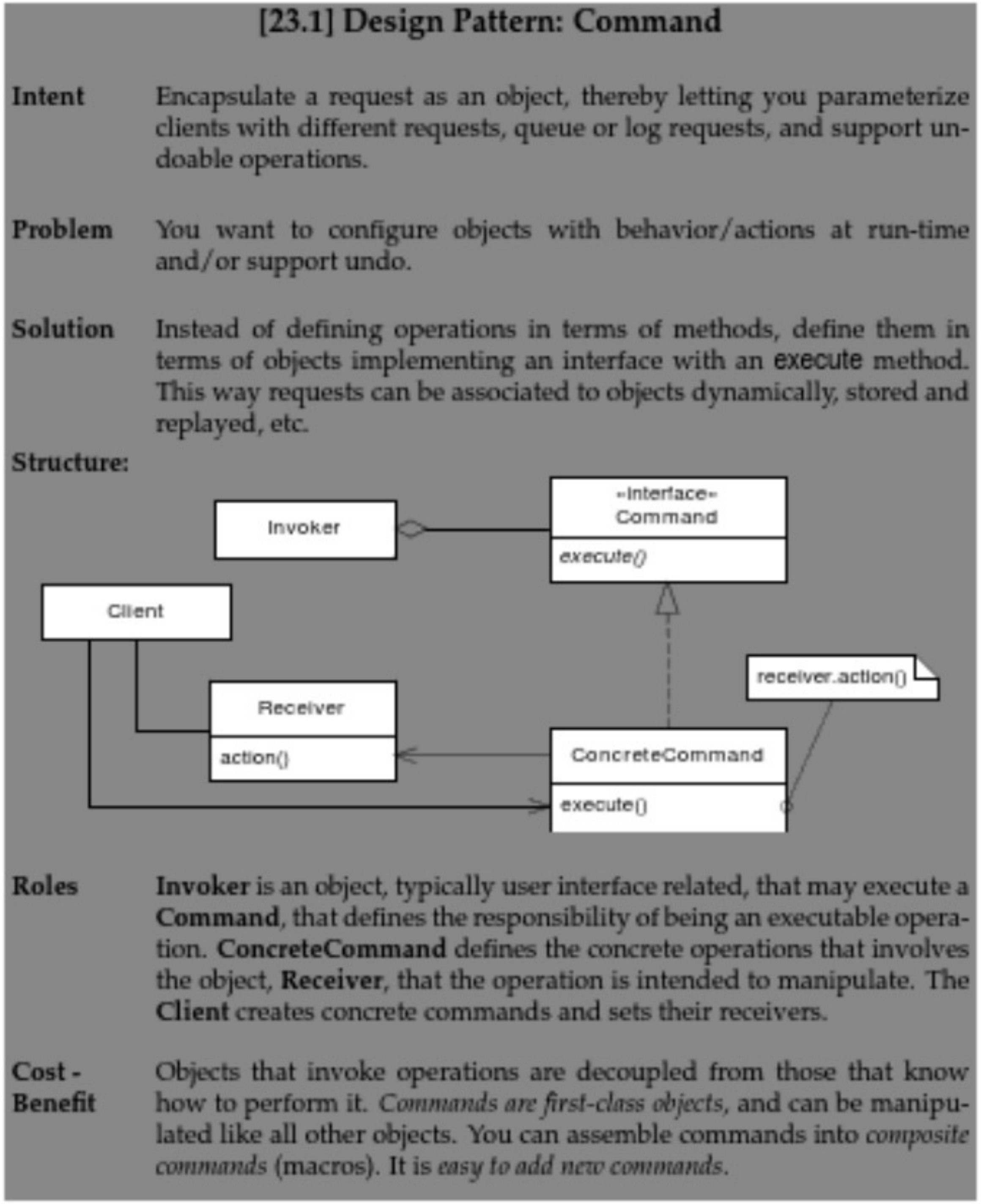
Command Pattern

- Clients are decoupled from set of commands
- Can extend command set at runtime
- Supports multiple ways to execute a command
- Can log and store commands
  - Support undo operations



```
public class MethodCommand {
  ...
  public void execute() {
    object.method(a,b,c);
  }
  public void undo() {
    object.undoTheMethod(a,b,c);
  }
}
```

  - Store executed commands on a stack
  - Define a macro as a composite of command objects



[23.1] Design Pattern: Command

Intent Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Problem You want to configure objects with behavior/actions at run-time and/or support undo.

Solution Instead of defining operations in terms of methods, define them in terms of objects implementing an interface with an execute method. This way requests can be associated to objects dynamically, stored and replayed, etc.

Structure:

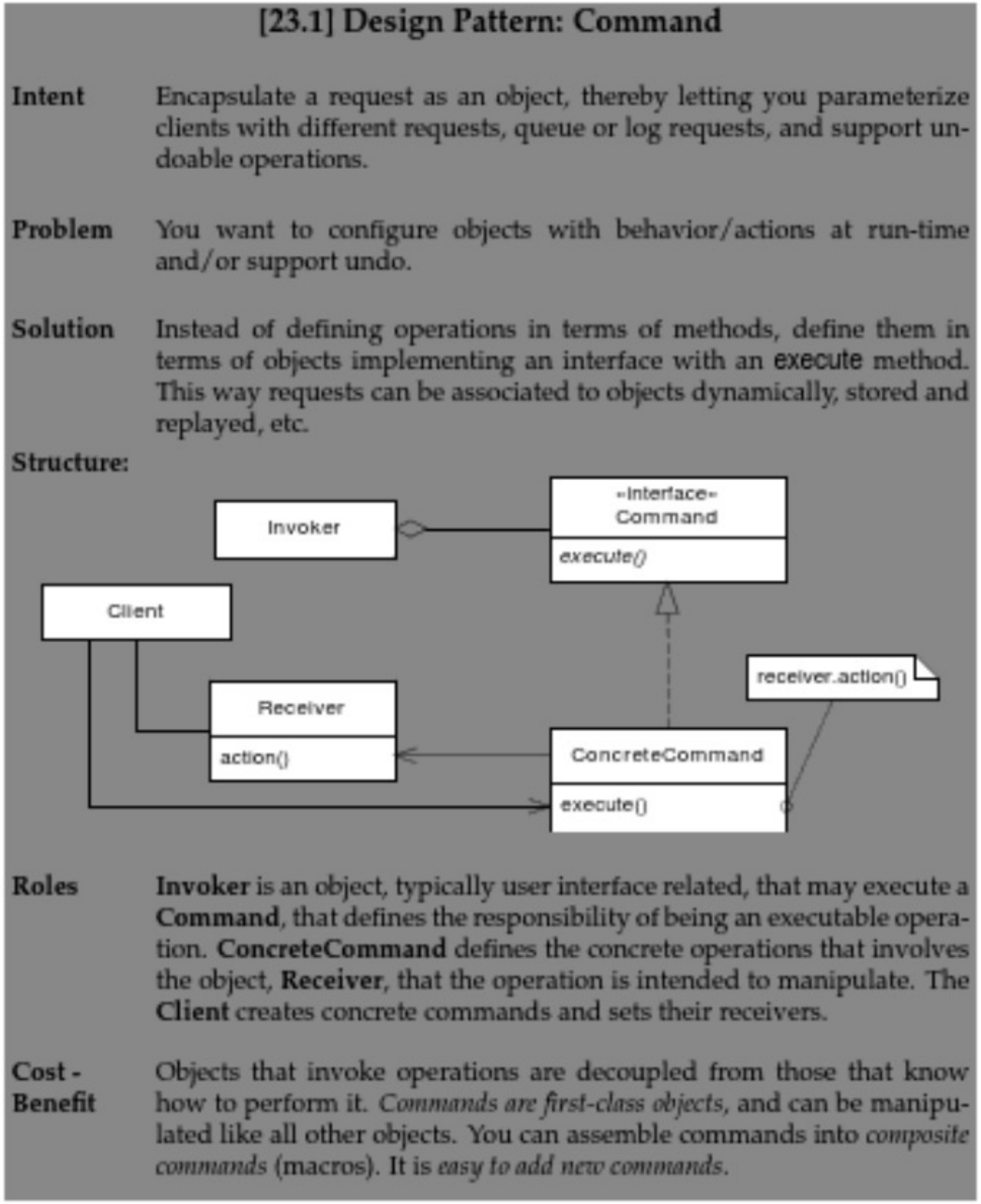
Roles **Invoker** is an object, typically user interface related, that may execute a **Command**, that defines the responsibility of being an executable operation. **ConcreteCommand** defines the concrete operations that involves the object, **Receiver**, that the operation is intended to manipulate. The **Client** creates concrete commands and sets their receivers.

Cost - Benefit Objects that invoke operations are decoupled from those that know how to perform it. *Commands are first-class objects*, and can be manipulated like all other objects. You can assemble commands into *composite commands* (macros). It is *easy to add new commands*.

Command Pattern

☹

- Overhead in writing and executing commands



[23.1] Design Pattern: Command

Intent Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Problem You want to configure objects with behavior/actions at run-time and/or support undo.

Solution Instead of defining operations in terms of methods, define them in terms of objects implementing an interface with an execute method. This way requests can be associated to objects dynamically, stored and replayed, etc.

Structure:

Roles **Invoker** is an object, typically user interface related, that may execute a **Command**, that defines the responsibility of being an executable operation. **ConcreteCommand** defines the concrete operations that involves the object, **Receiver**, that the operation is intended to manipulate. The **Client** creates concrete commands and sets their receivers.

Cost - Benefit Objects that invoke operations are decoupled from those that know how to perform it. *Commands are first-class objects*, and can be manipulated like all other objects. You can assemble commands into *composite commands* (macros). It is *easy to add new commands*.

Proxy Pattern

Suppose we have a website or document with a lot of images

→ Loading all of the images at once will slow everything down

→ So, we want to only load the images that should currently be visible

Proxy Pattern

(3) Encapsulate what varies: images that become visible will fetch image data, while those that are not visible do not load

Proxy Pattern

(3) Encapsulate what varies: images that become visible will fetch image data, while those that are not visible do not load

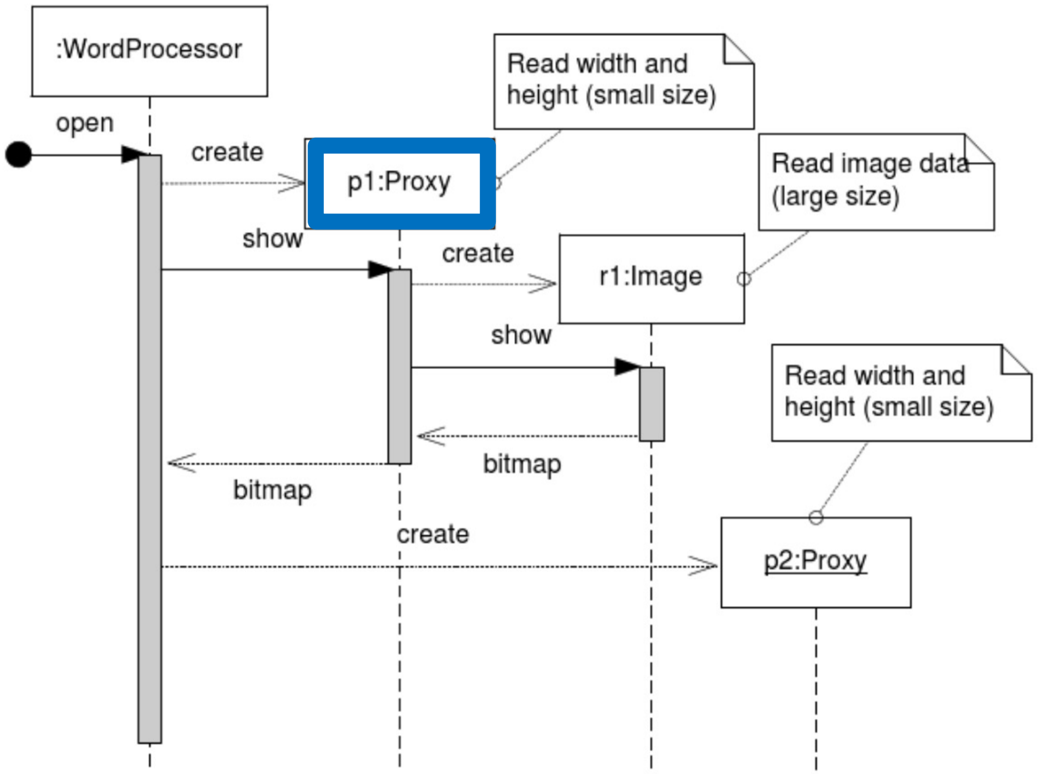
(1) Program to an interface: provide the client with an intermediate object that will defer loading until the show() method is called on an image object

Proxy Pattern

(3) Encapsulate what varies: images that become visible will fetch image data, while those that are not visible do not load

(1) Program to an interface: provide the client with an intermediate object that will defer loading until the show() method is called on an image object

(2) Object composition: put a **proxy** object in front of the image


:WordProcessor
open
create
p1:Proxy
Read width and height (small size)
show
create
r1:Image
Read image data (large size)
show
bitmap
bitmap
Read width and height (small size)
create
p2:Proxy

Proxy Pattern

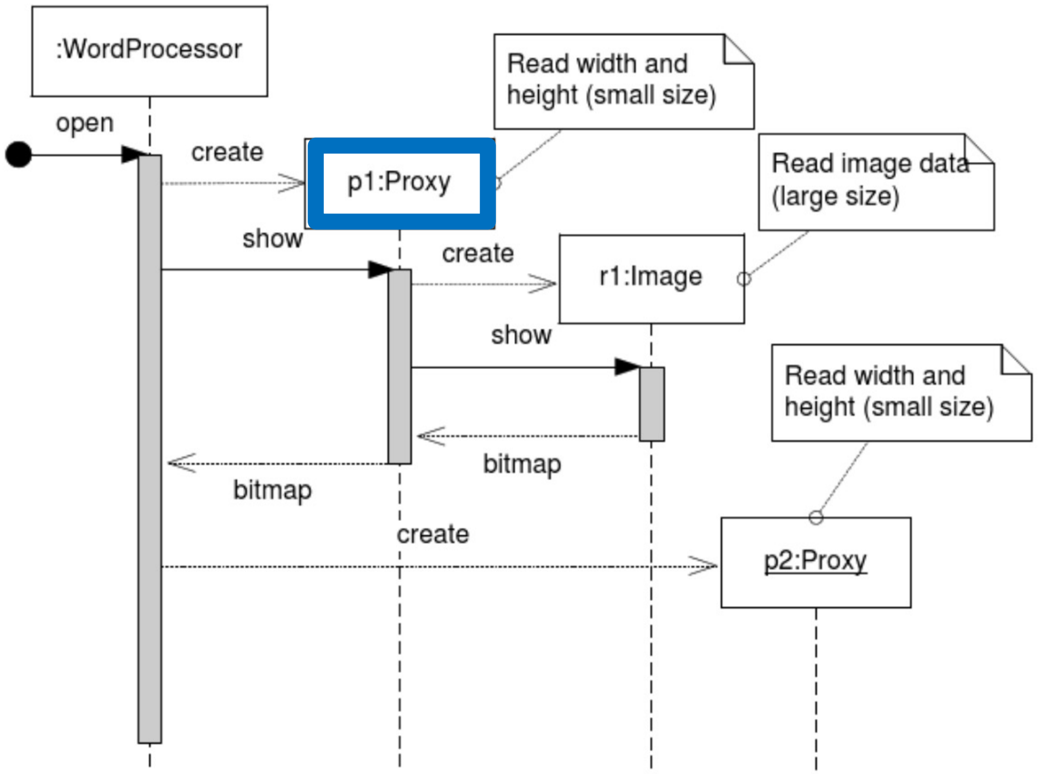
(3) Encapsulate what varies: images that become visible will fetch image data, while those that are not visible do not load

(1) Program to an interface: provide the client with an intermediate object that will defer loading until the show() method is called on an image object

(2) Object composition: put a **proxy** object in front of the image

→Have the proxy load essential but small data, e.g., image width and height

→The real image object is not created by the proxy until show() is called
(The first show() invocation will be slowest)


:WordProcessor
open
create
p1:Proxy
Read width and height (small size)
show
create
r1:Image
Read image data (large size)
show
bitmap
bitmap
Read width and height (small size)
create
p2:Proxy

Proxy Pattern

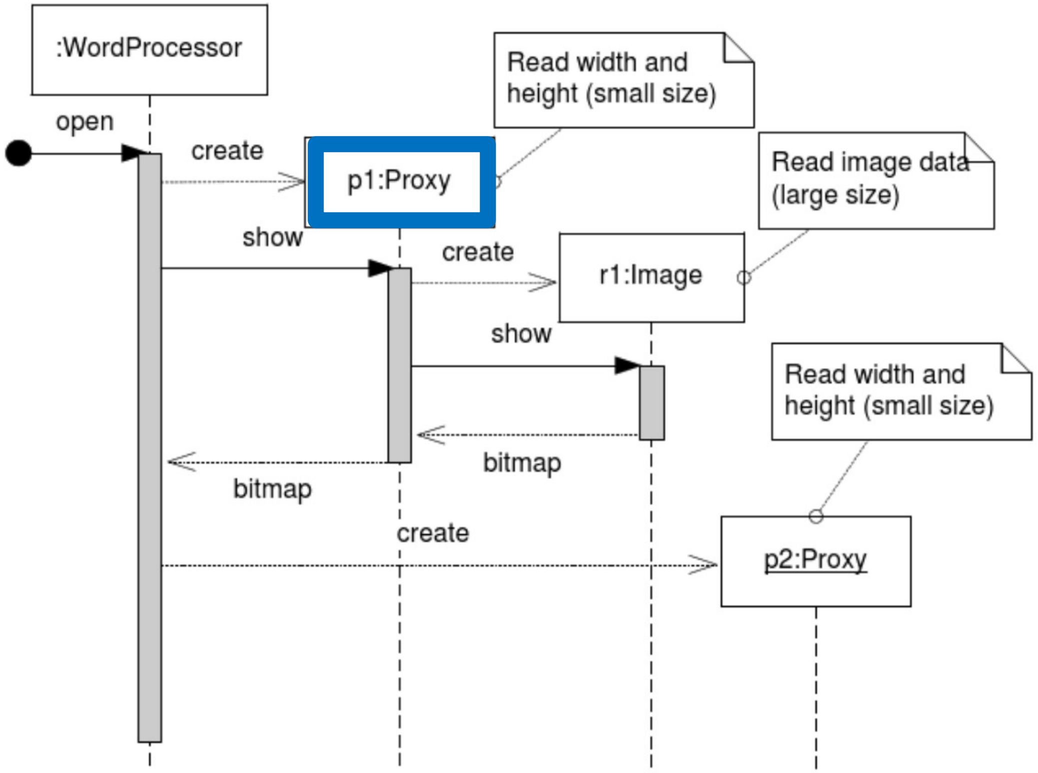
(3) Encapsulate what varies: images that become visible will fetch image data, while those that are not visible do not load

(1) Program to an interface: provide the client with an intermediate object that will defer loading until the show() method is called on an image object

(2) Object composition: put a **proxy** object in front of the image

→Have the proxy load essential but small data, e.g., image width and height

→The real image object is not created by the proxy until show() is called
(The first show() invocation will be slowest)

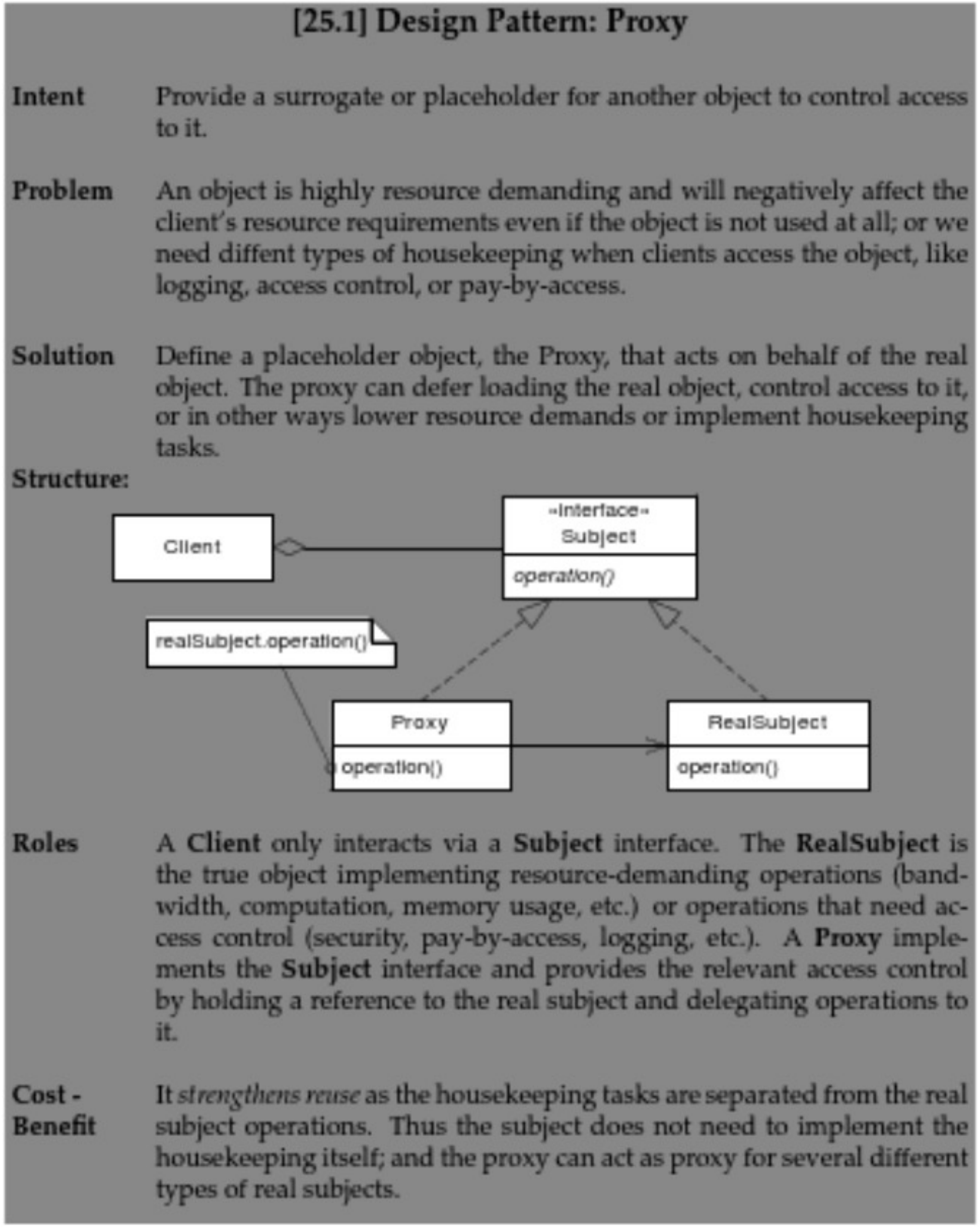

:WordProcessor
open
create
p1:Proxy
Read width and height (small size)
Read image data (large size)
show
create
r1:Image
show
Read width and height (small size)
bitmap
bitmap
create
p2:Proxy

Proxy: Provide a surrogate or placeholder for another object to control access to it

Proxy Pattern

Problem: An object is resource-heavy, even if not used; or we need different types of “housekeeping” when clients access an object

Solution: Define a **proxy** object that acts in place of the real object (to defer loading, control access, perform logging tasks, etc.)



[25.1] Design Pattern: Proxy

Intent Provide a surrogate or placeholder for another object to control access to it.

Problem An object is highly resource demanding and will negatively affect the client’s resource requirements even if the object is not used at all; or we need diffent types of housekeeping when clients access the object, like logging, access control, or pay-by-access.

Solution Define a placeholder object, the Proxy, that acts on behalf of the real object. The proxy can defer loading the real object, control access to it, or in other ways lower resource demands or implement housekeeping tasks.

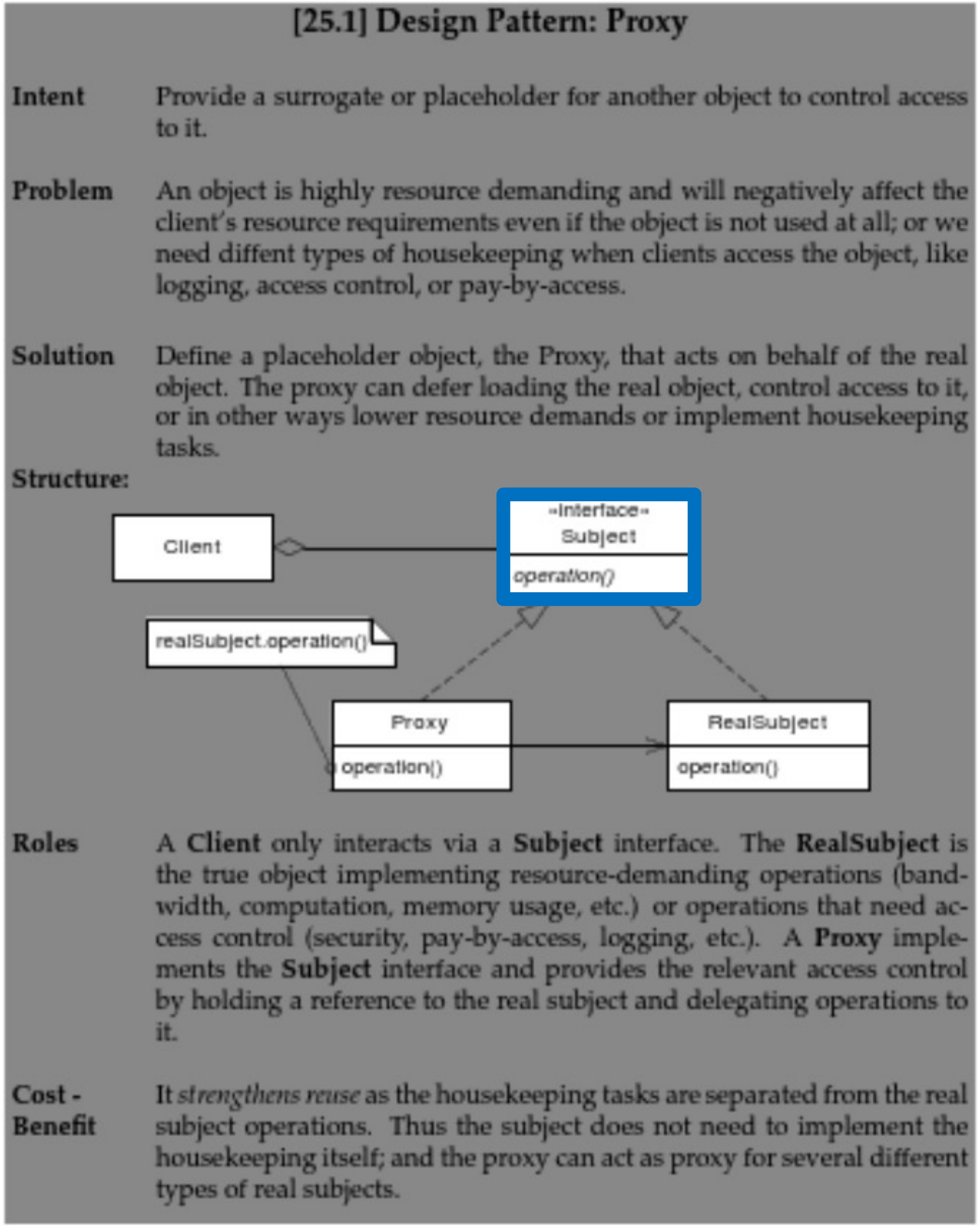
Structure:

Roles A **Client** only interacts via a **Subject** interface. The **RealSubject** is the true object implementing resource-demanding operations (bandwidth, computation, memory usage, etc.) or operations that need access control (security, pay-by-access, logging, etc.). A **Proxy** implements the **Subject** interface and provides the relevant access control by holding a reference to the real subject and delegating operations to it.

Cost - Benefit It *strengthens reuse* as the housekeeping tasks are separated from the real subject operations. Thus the subject does not need to implement the housekeeping itself; and the proxy can act as proxy for several different types of real subjects.

Proxy Pattern

The **subject** defines the interface for objects that can be proxied



[25.1] Design Pattern: Proxy

Intent Provide a surrogate or placeholder for another object to control access to it.

Problem An object is highly resource demanding and will negatively affect the client's resource requirements even if the object is not used at all; or we need diffent types of housekeeping when clients access the object, like logging, access control, or pay-by-access.

Solution Define a placeholder object, the Proxy, that acts on behalf of the real object. The proxy can defer loading the real object, control access to it, or in other ways lower resource demands or implement housekeeping tasks.

Structure:

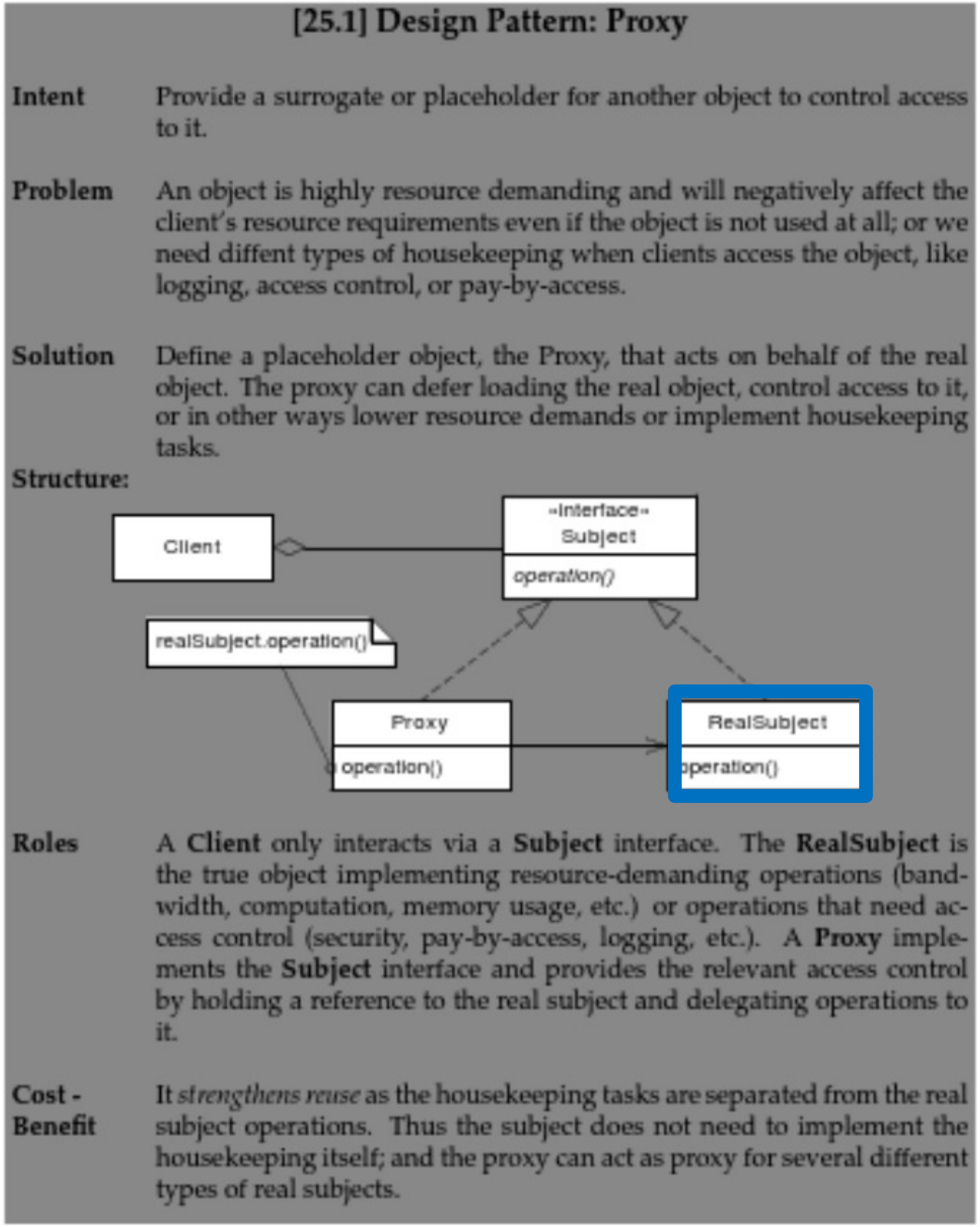
Roles A **Client** only interacts via a **Subject** interface. The **RealSubject** is the true object implementing resource-demanding operations (bandwidth, computation, memory usage, etc.) or operations that need access control (security, pay-by-access, logging, etc.). A **Proxy** implements the **Subject** interface and provides the relevant access control by holding a reference to the real subject and delegating operations to it.

Cost - Benefit It *strengthens reuse* as the housekeeping tasks are separated from the real subject operations. Thus the subject does not need to implement the housekeeping itself; and the proxy can act as proxy for several different types of real subjects.

Proxy Pattern

The **subject** defines the interface for objects that can be proxied

RealSubject is the real implementation of the object



[25.1] Design Pattern: Proxy

Intent Provide a surrogate or placeholder for another object to control access to it.

Problem An object is highly resource demanding and will negatively affect the client's resource requirements even if the object is not used at all; or we need diffent types of housekeeping when clients access the object, like logging, access control, or pay-by-access.

Solution Define a placeholder object, the Proxy, that acts on behalf of the real object. The proxy can defer loading the real object, control access to it, or in other ways lower resource demands or implement housekeeping tasks.

Structure:

Roles A **Client** only interacts via a **Subject** interface. The **RealSubject** is the true object implementing resource-demanding operations (bandwidth, computation, memory usage, etc.) or operations that need access control (security, pay-by-access, logging, etc.). A **Proxy** implements the **Subject** interface and provides the relevant access control by holding a reference to the real subject and delegating operations to it.

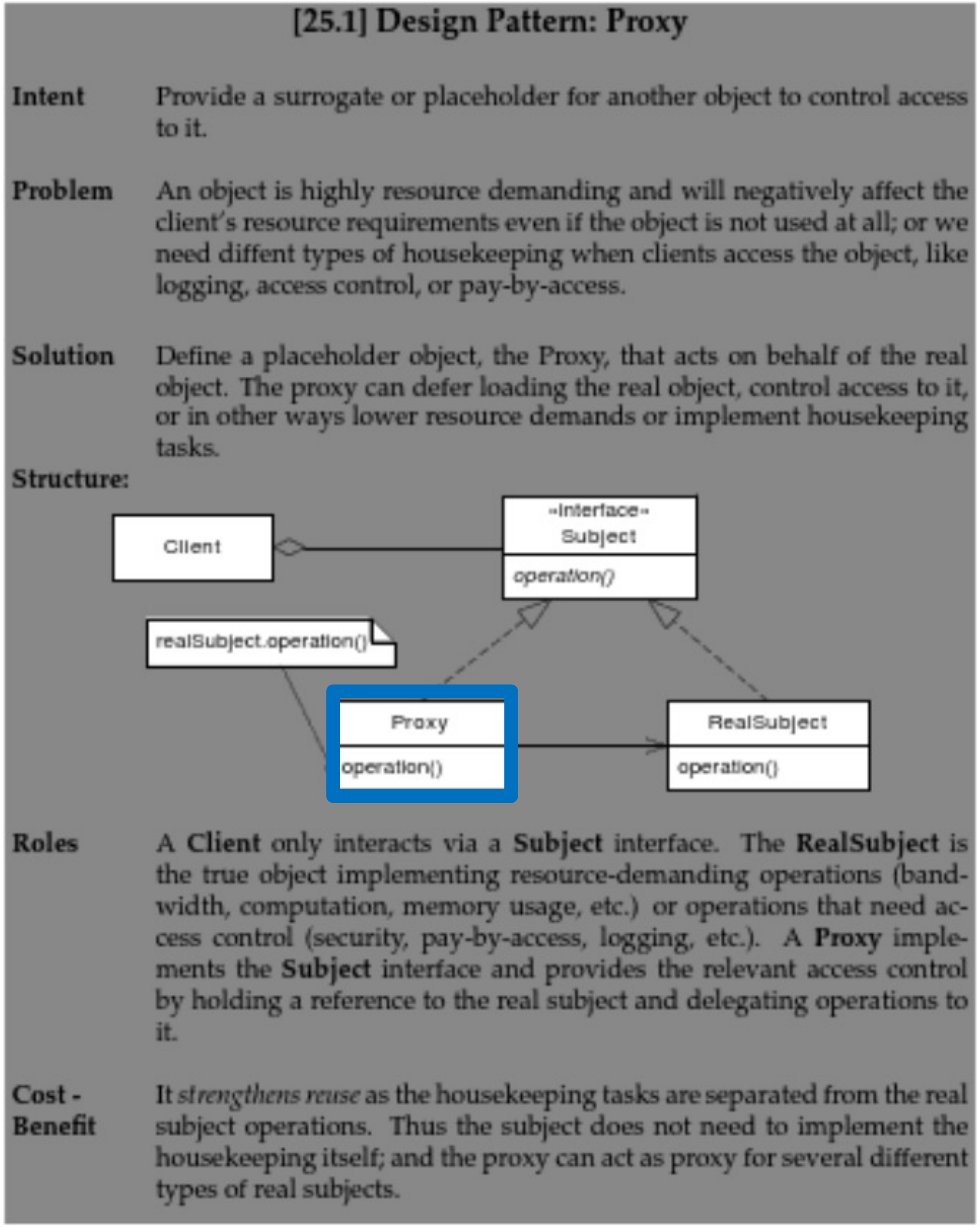
Cost - Benefit It *strengthens reuse* as the housekeeping tasks are separated from the real subject operations. Thus the subject does not need to implement the housekeeping itself; and the proxy can act as proxy for several different types of real subjects.

Proxy Pattern

The **subject** defines the interface for objects that can be proxied

RealSubject is the real implementation of the object

Proxy is the intermediate object, needs a reference to the real subject to delegate requests



[25.1] Design Pattern: Proxy

Intent Provide a surrogate or placeholder for another object to control access to it.

Problem An object is highly resource demanding and will negatively affect the client's resource requirements even if the object is not used at all; or we need diffent types of housekeeping when clients access the object, like logging, access control, or pay-by-access.

Solution Define a placeholder object, the Proxy, that acts on behalf of the real object. The proxy can defer loading the real object, control access to it, or in other ways lower resource demands or implement housekeeping tasks.

Structure:

Roles A **Client** only interacts via a **Subject** interface. The **RealSubject** is the true object implementing resource-demanding operations (bandwidth, computation, memory usage, etc.) or operations that need access control (security, pay-by-access, logging, etc.). A **Proxy** implements the **Subject** interface and provides the relevant access control by holding a reference to the real subject and delegating operations to it.

Cost - Benefit It *strengthens reuse* as the housekeeping tasks are separated from the real subject operations. Thus the subject does not need to implement the housekeeping itself; and the proxy can act as proxy for several different types of real subjects.

Proxy Pattern

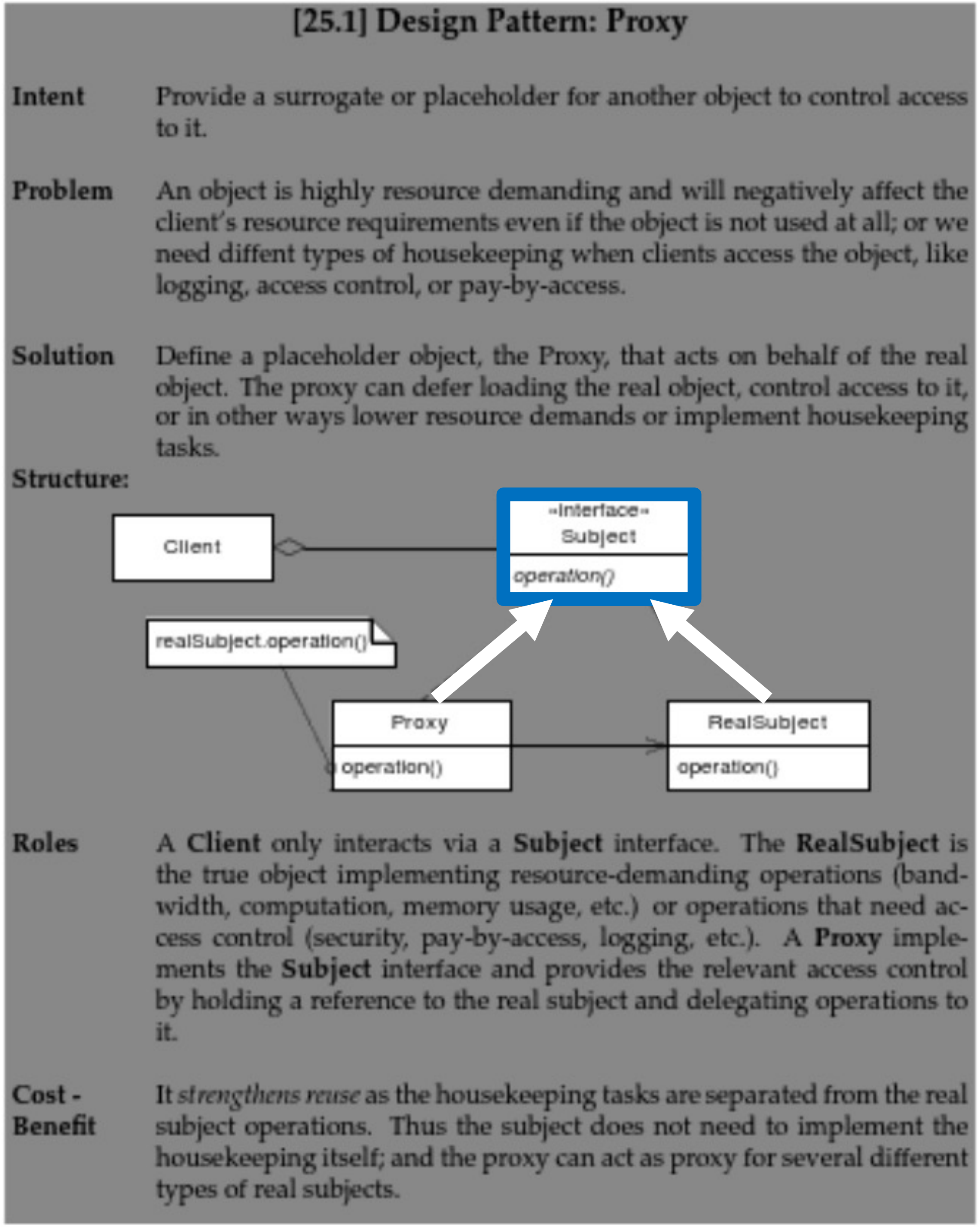
The **subject** defines the interface for objects that can be proxied

RealSubject is the real implementation of the object

Proxy is the intermediate object, needs a reference to the real subject to delegate requests

→Proxy and real subject roles have the same protocol and interface

→Client can use abstract factory to reduce coupling if it is creating instances of Subject



[25.1] Design Pattern: Proxy

Intent Provide a surrogate or placeholder for another object to control access to it.

Problem An object is highly resource demanding and will negatively affect the client's resource requirements even if the object is not used at all; or we need diffent types of housekeeping when clients access the object, like logging, access control, or pay-by-access.

Solution Define a placeholder object, the Proxy, that acts on behalf of the real object. The proxy can defer loading the real object, control access to it, or in other ways lower resource demands or implement housekeeping tasks.

Structure:

Roles A **Client** only interacts via a **Subject** interface. The **RealSubject** is the true object implementing resource-demanding operations (bandwidth, computation, memory usage, etc.) or operations that need access control (security, pay-by-access, logging, etc.). A **Proxy** implements the **Subject** interface and provides the relevant access control by holding a reference to the real subject and delegating operations to it.

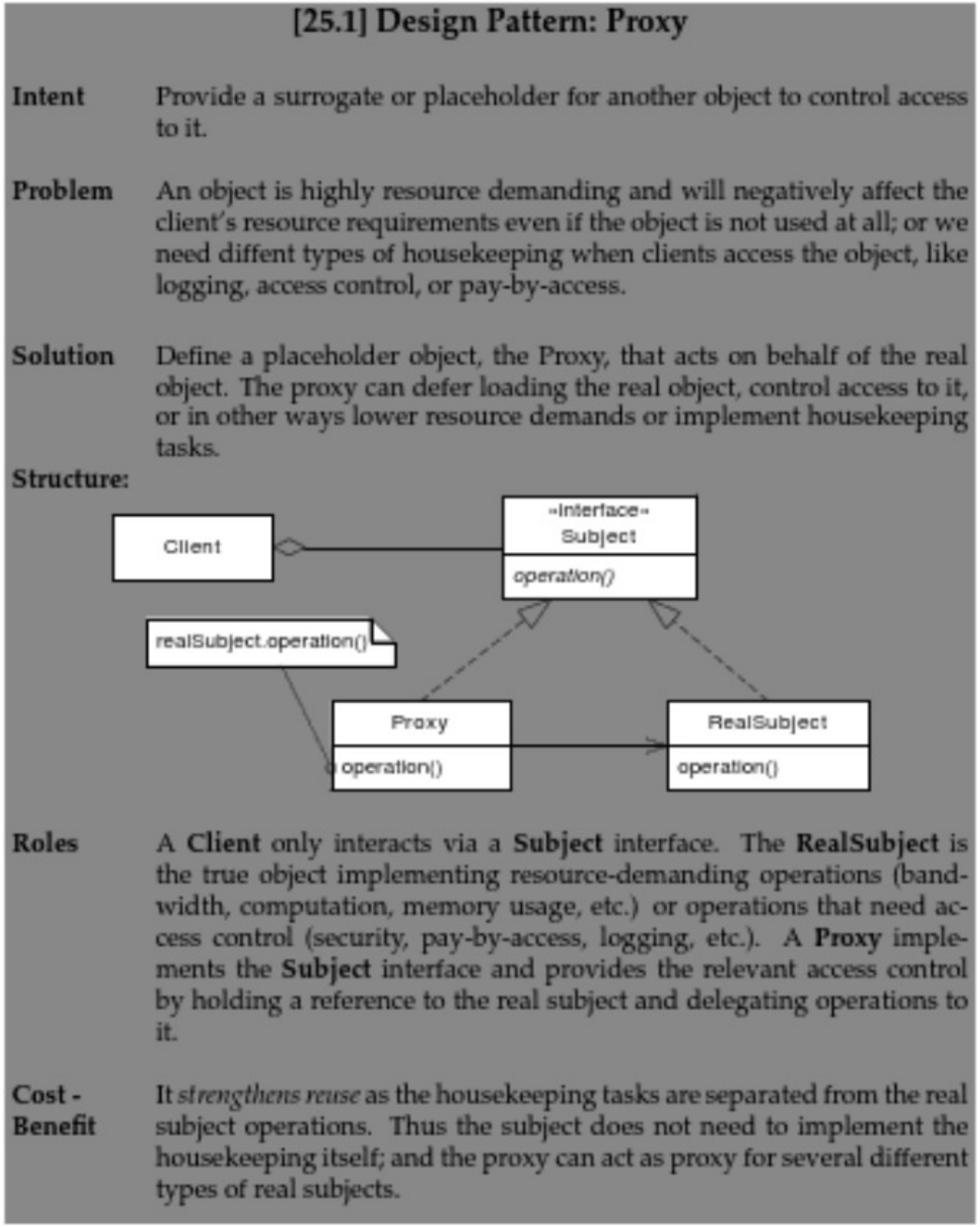
Cost - Benefit It *strengthens reuse* as the housekeeping tasks are separated from the real subject operations. Thus the subject does not need to implement the housekeeping itself; and the proxy can act as proxy for several different types of real subjects.

Proxy Pattern

Similar benefits and liabilities as all compositional designs

Uses:

- Protection proxies / access control
- Virtual proxies / performance control
- Remote proxies / remote access



[25.1] Design Pattern: Proxy

Intent Provide a surrogate or placeholder for another object to control access to it.

Problem An object is highly resource demanding and will negatively affect the client's resource requirements even if the object is not used at all; or we need diffent types of housekeeping when clients access the object, like logging, access control, or pay-by-access.

Solution Define a placeholder object, the Proxy, that acts on behalf of the real object. The proxy can defer loading the real object, control access to it, or in other ways lower resource demands or implement housekeeping tasks.

Structure:

Roles A **Client** only interacts via a **Subject** interface. The **RealSubject** is the true object implementing resource-demanding operations (bandwidth, computation, memory usage, etc.) or operations that need access control (security, pay-by-access, logging, etc.). A **Proxy** implements the **Subject** interface and provides the relevant access control by holding a reference to the real subject and delegating operations to it.

Cost - Benefit It *strengthens reuse* as the housekeeping tasks are separated from the real subject operations. Thus the subject does not need to implement the housekeeping itself; and the proxy can act as proxy for several different types of real subjects.

Proxy Pattern

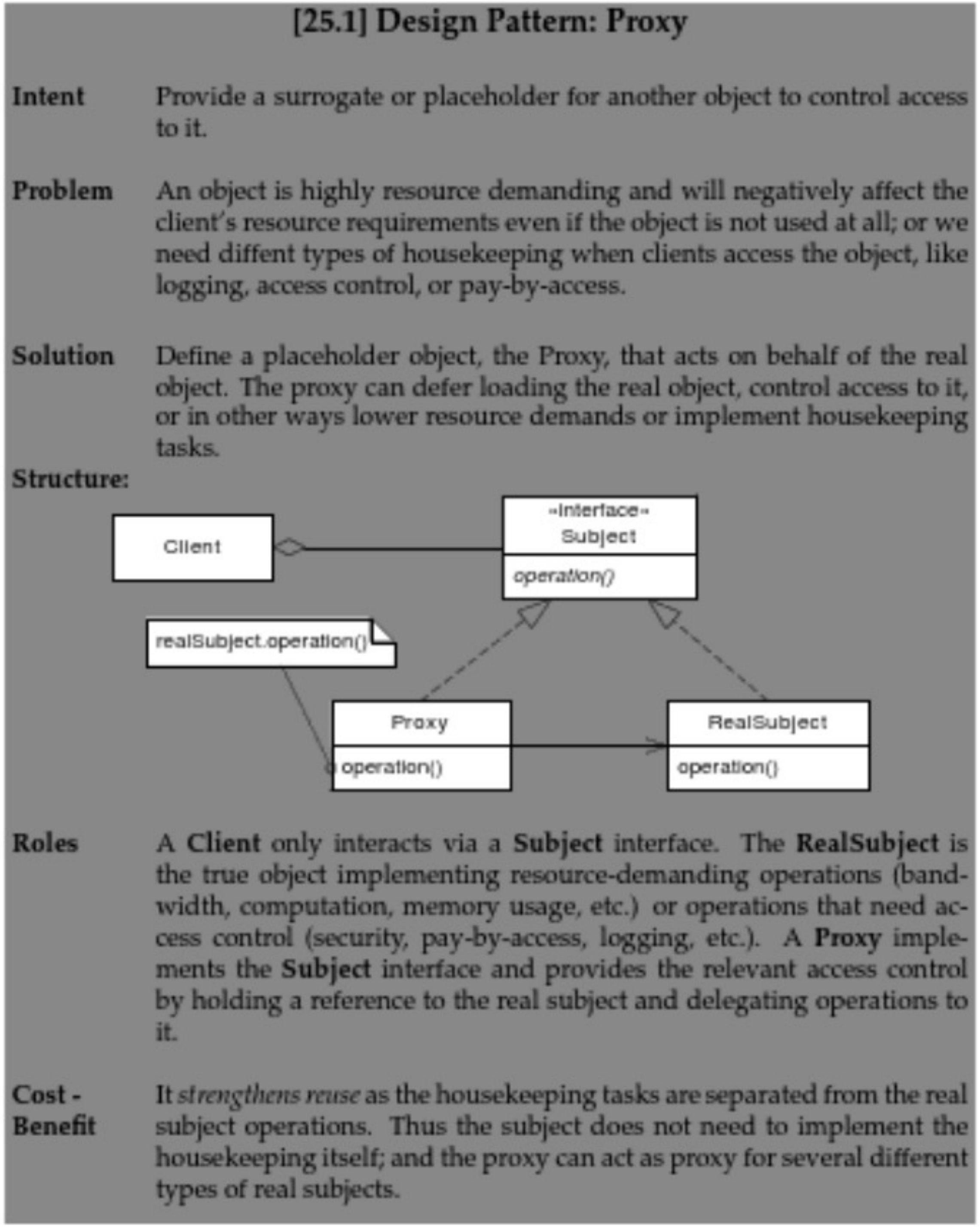
Compared to decorator:

- Decorator looks at the object itself (looking inside)

→ Adding behavior to the object itself

- Proxy looks at the user of the object (looking outside)

→ Monitoring/controlling access



[25.1] Design Pattern: Proxy

Intent Provide a surrogate or placeholder for another object to control access to it.

Problem An object is highly resource demanding and will negatively affect the client's resource requirements even if the object is not used at all; or we need diffent types of housekeeping when clients access the object, like logging, access control, or pay-by-access.

Solution Define a placeholder object, the Proxy, that acts on behalf of the real object. The proxy can defer loading the real object, control access to it, or in other ways lower resource demands or implement housekeeping tasks.

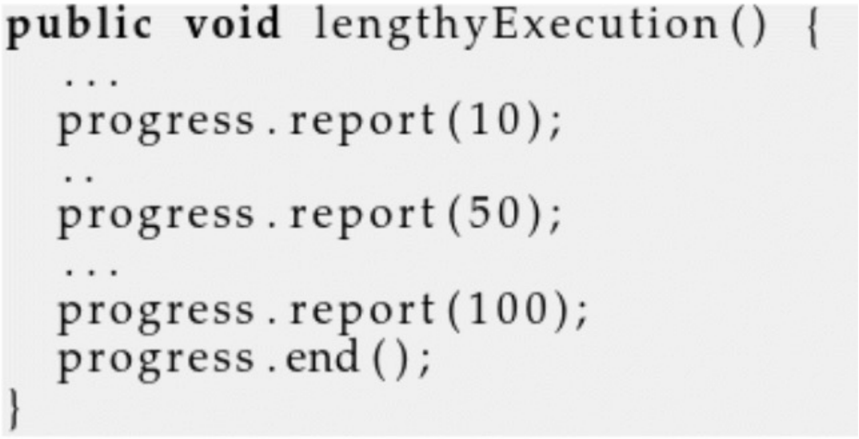
Structure:

Roles A **Client** only interacts via a **Subject** interface. The **RealSubject** is the true object implementing resource-demanding operations (bandwidth, computation, memory usage, etc.) or operations that need access control (security, pay-by-access, logging, etc.). A **Proxy** implements the **Subject** interface and provides the relevant access control by holding a reference to the real subject and delegating operations to it.

Cost - Benefit It *strengthens reuse* as the housekeeping tasks are separated from the real subject operations. Thus the subject does not need to implement the housekeeping itself; and the proxy can act as proxy for several different types of real subjects.

Null Object Pattern

Suppose we have a class that contains some methods with long execution time, so we want to test a progress indicator

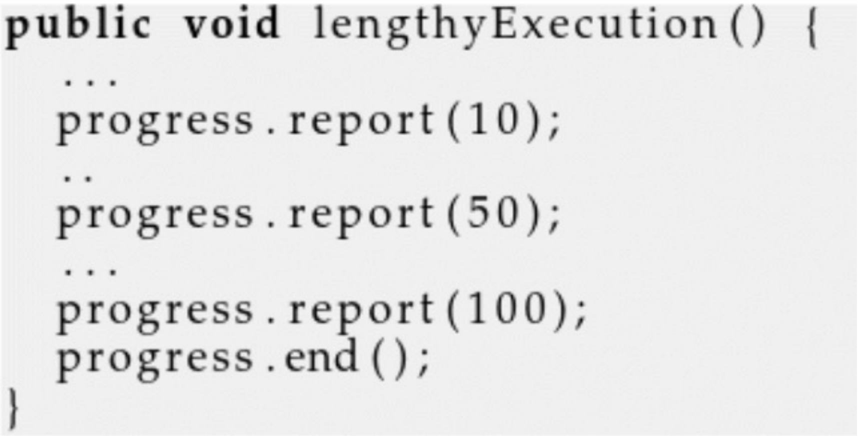


```
public void lengthyExecution() {
  ...
  progress.report(10);
  ..
  progress.report(50);
  ...
  progress.report(100);
  progress.end();
}
```

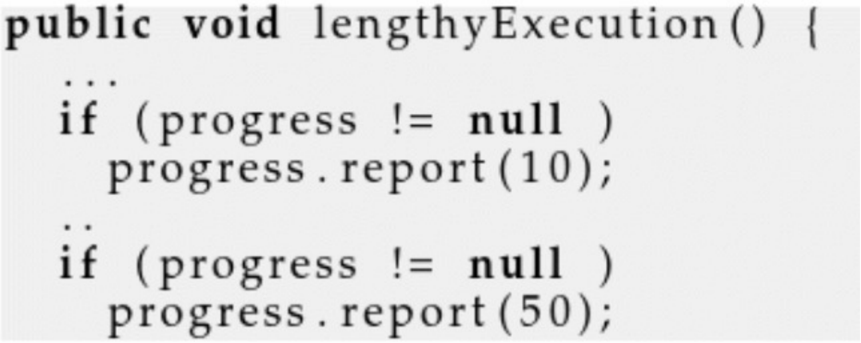
Null Object Pattern

Suppose we have a class that contains some methods with long execution time, so we want to test a progress indicator

We don’t want to bring up dialogs while testing this, so what if we set the object reference to null when testing?



```
public void lengthyExecution() {
  ...
  progress.report(10);
  ..
  progress.report(50);
  ...
  progress.report(100);
  progress.end();
}
```

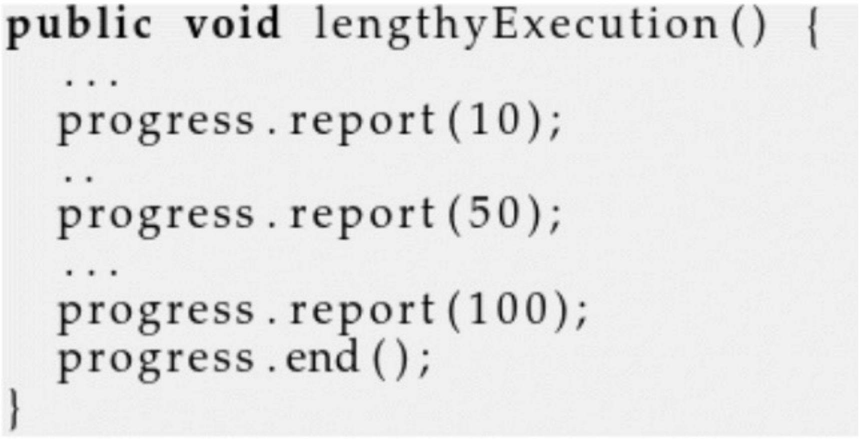


```
public void lengthyExecution() {
  ...
  if (progress != null )
    progress.report(10);
  ..
  if (progress != null )
    progress.report(50);
```

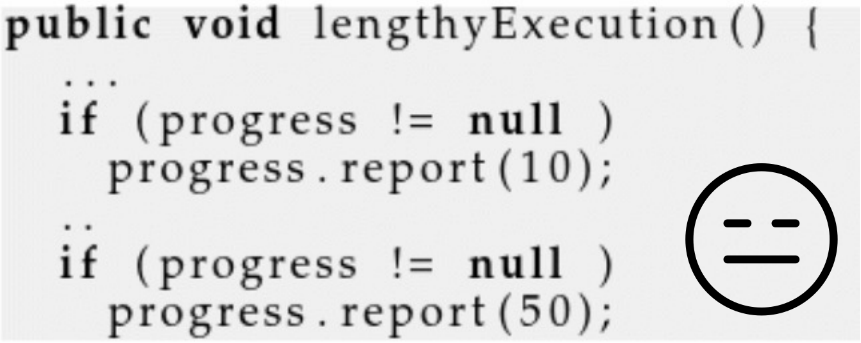
Null Object Pattern

Suppose we have a class that contains some methods with long execution time, so we want to test a progress indicator

We don’t want to bring up dialogs while testing this, so what if we set the object reference to null when testing?



```
public void lengthyExecution() {
   ...
   progress.report(10);
   ..
   progress.report(50);
   ...
   progress.report(100);
   progress.end();
}
```

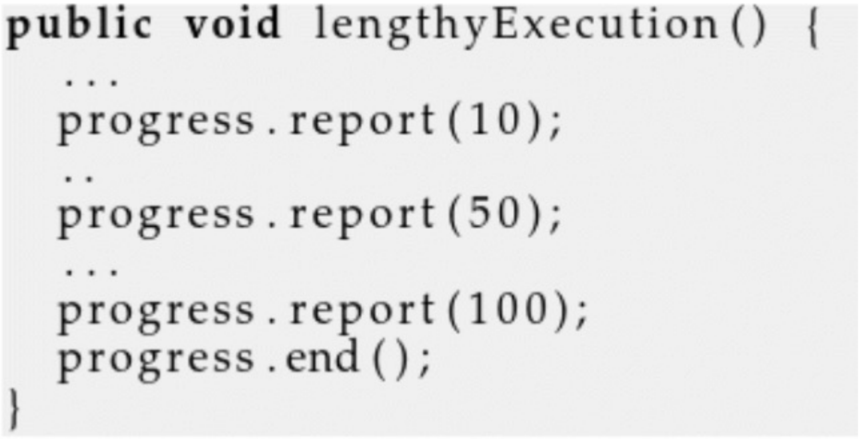


```
public void lengthyExecution() {
   ...
   if (progress != null )
     progress.report(10);
   ..
   if (progress != null )
     progress.report(50);
```

😑

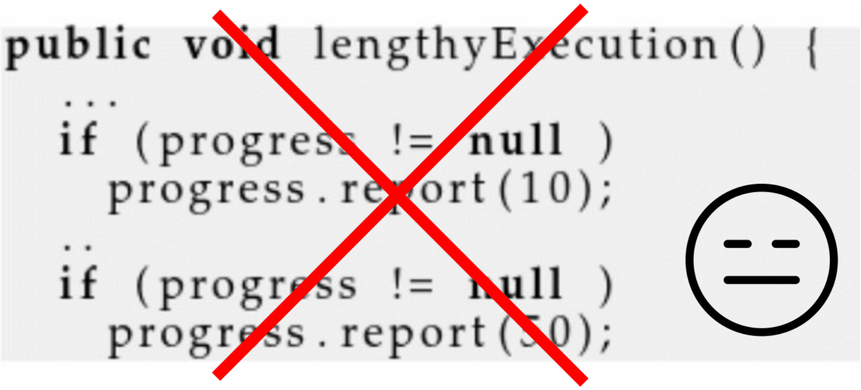
Null Object Pattern

Suppose we have a class that contains some methods with long execution time, so we want to test a progress indicator



```
public void lengthyExecution() {
   ...
   progress.report(10);
   ..
   progress.report(50);
   ...
   progress.report(100);
   progress.end();
}
```

We don’t want to bring up dialogs while testing this, so what if we set the object reference to null when testing?



```
public void lengthyExecution() {
   ...
   if (progress != null )
      progress.report(10);
   ..
   if (progress != null )
      progress.report(50);
```

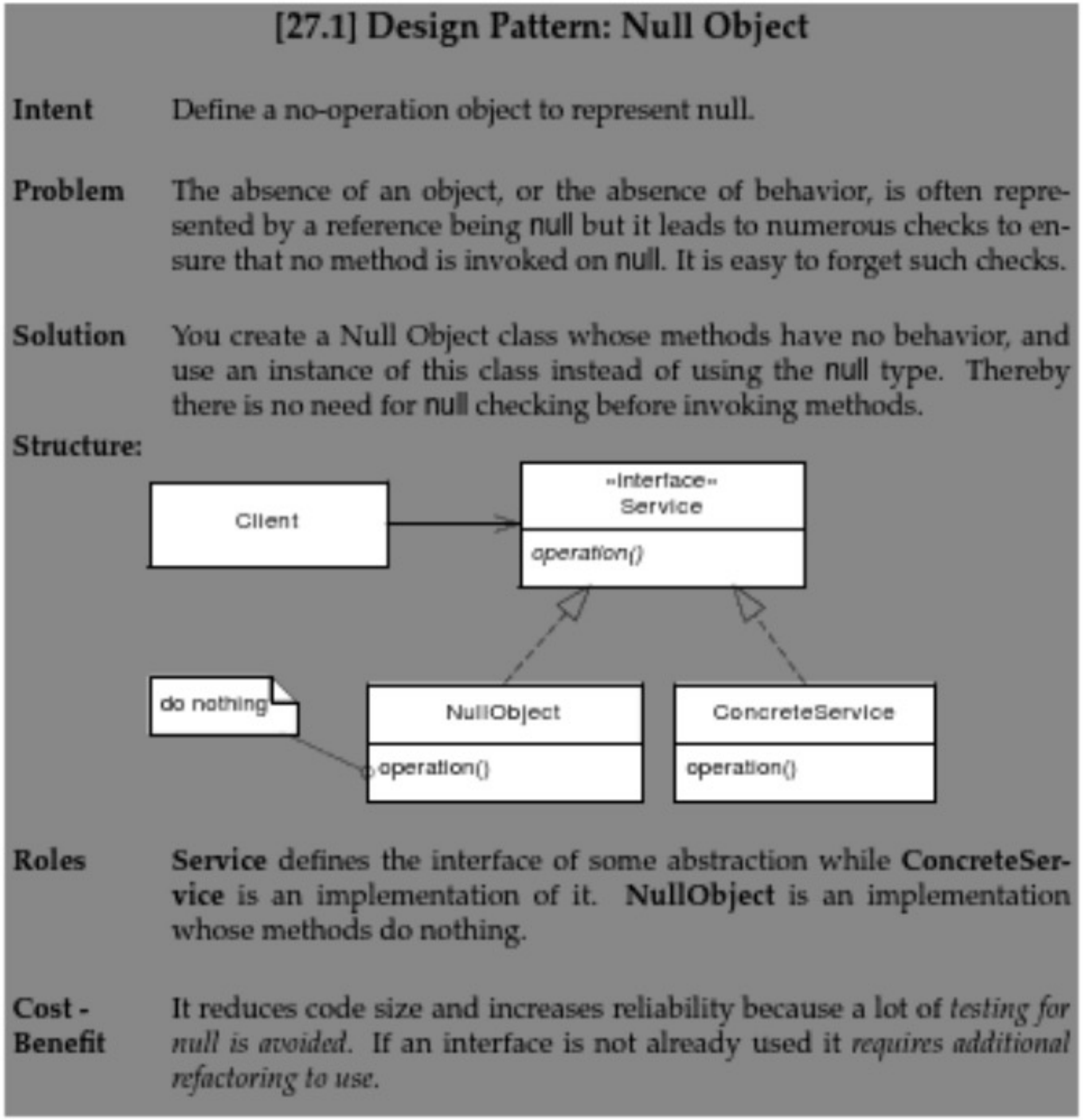
Rather than absence of object, what we actually need is absence of **behavior**

→ Use a **null object**, whose methods do nothing

Null Object Pattern

Problem: The absence of an object represented by a null reference leads to numerous checks before invoking methods

Solution: Use a **null object** whose methods have no behavior, use this instead of null



[27.1] Design Pattern: Null Object

Intent Define a no-operation object to represent null.

Problem The absence of an object, or the absence of behavior, is often represented by a reference being null but it leads to numerous checks to ensure that no method is invoked on null. It is easy to forget such checks.

Solution You create a Null Object class whose methods have no behavior, and use an instance of this class instead of using the null type. Thereby there is no need for null checking before invoking methods.

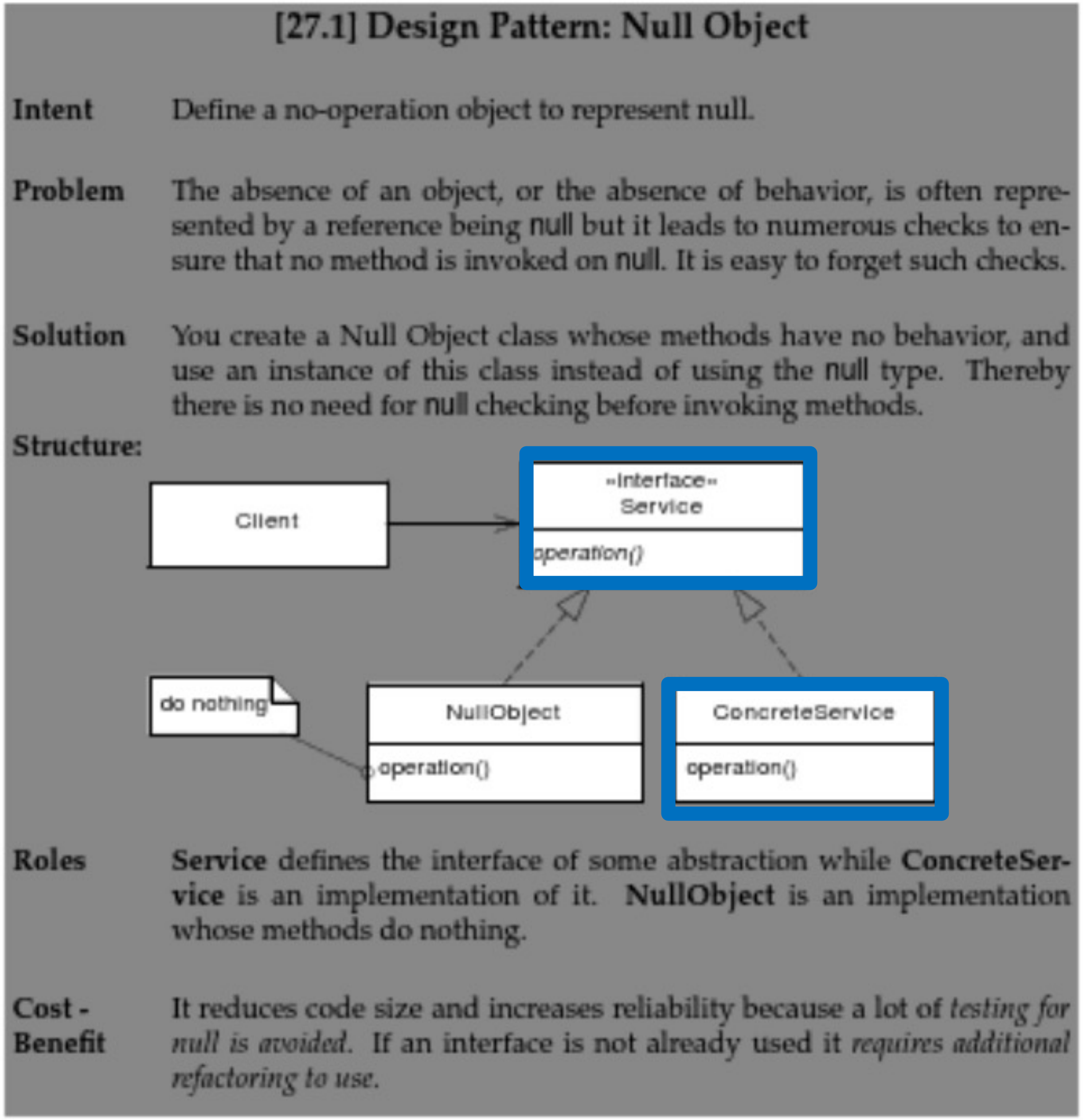
Structure:

Roles **Service** defines the interface of some abstraction while **ConcreteService** is an implementation of it. **NullObject** is an implementation whose methods do nothing.

Cost - Benefit It reduces code size and increases reliability because a lot of *testing for null is avoided*. If an interface is not already used it *requires additional refactoring to use.*

Null Object Pattern

Service defines the interface, **ConcreteService** implements the service



[27.1] Design Pattern: Null Object

Intent Define a no-operation object to represent null.

Problem The absence of an object, or the absence of behavior, is often represented by a reference being null but it leads to numerous checks to ensure that no method is invoked on null. It is easy to forget such checks.

Solution You create a Null Object class whose methods have no behavior, and use an instance of this class instead of using the null type. Thereby there is no need for null checking before invoking methods.

Structure:

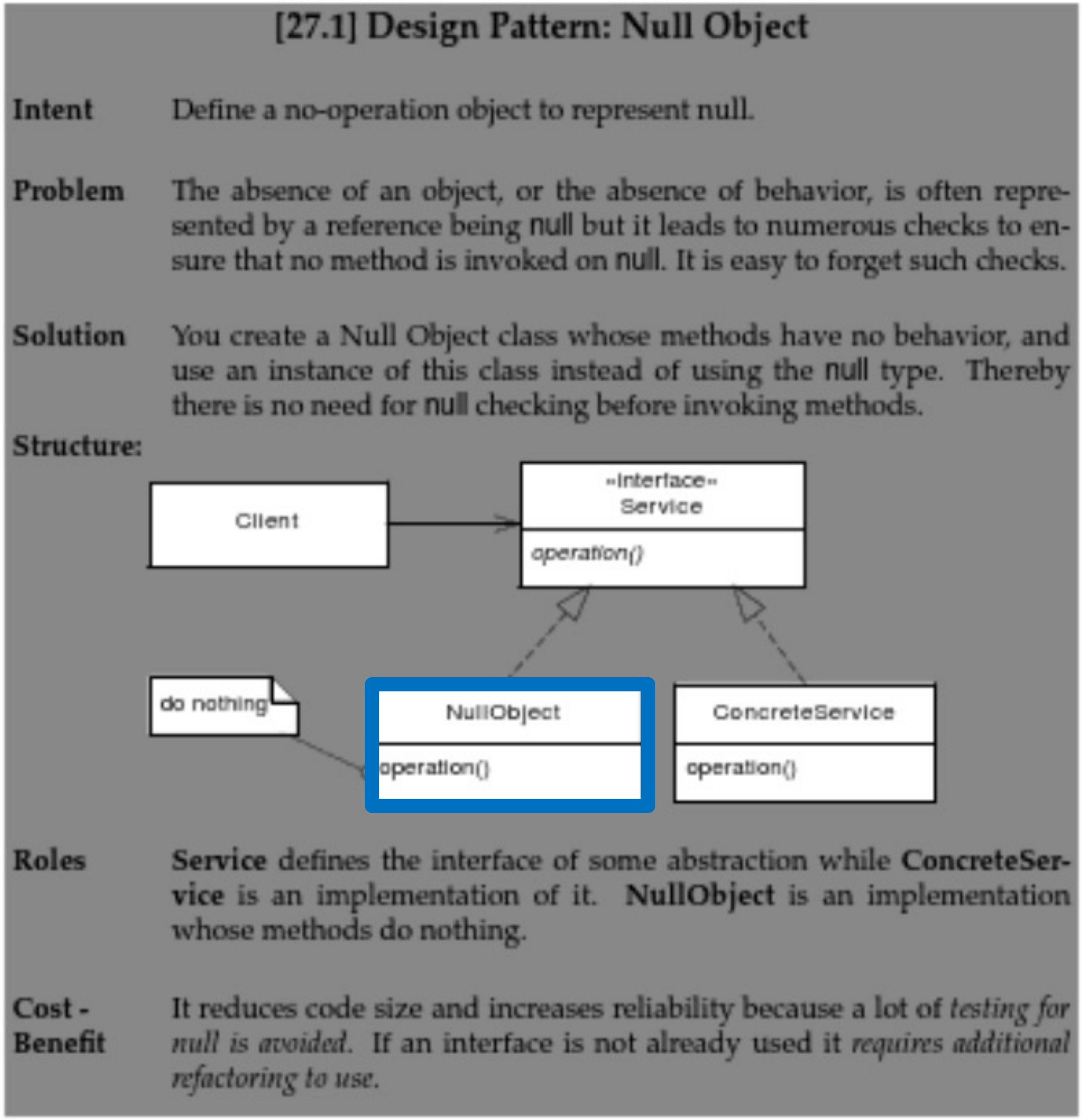
Roles **Service** defines the interface of some abstraction while **ConcreteService** is an implementation of it. **NullObject** is an implementation whose methods do nothing.

Cost - Benefit It reduces code size and increases reliability because a lot of *testing for null is avoided*. If an interface is not already used it *requires additional refactoring to use*.

Null Object Pattern

Service defines the interface, **ConcreteService** implements the service

Null Object contains empty implementations of all methods in the service interface



[27.1] Design Pattern: Null Object

Intent Define a no-operation object to represent null.

Problem The absence of an object, or the absence of behavior, is often represented by a reference being null but it leads to numerous checks to ensure that no method is invoked on null. It is easy to forget such checks.

Solution You create a Null Object class whose methods have no behavior, and use an instance of this class instead of using the null type. Thereby there is no need for null checking before invoking methods.

Structure:

Roles **Service** defines the interface of some abstraction while **ConcreteService** is an implementation of it. **NullObject** is an implementation whose methods do nothing.

Cost - Benefit It reduces code size and increases reliability because a lot of *testing for null is avoided*. If an interface is not already used it *requires additional refactoring to use*.

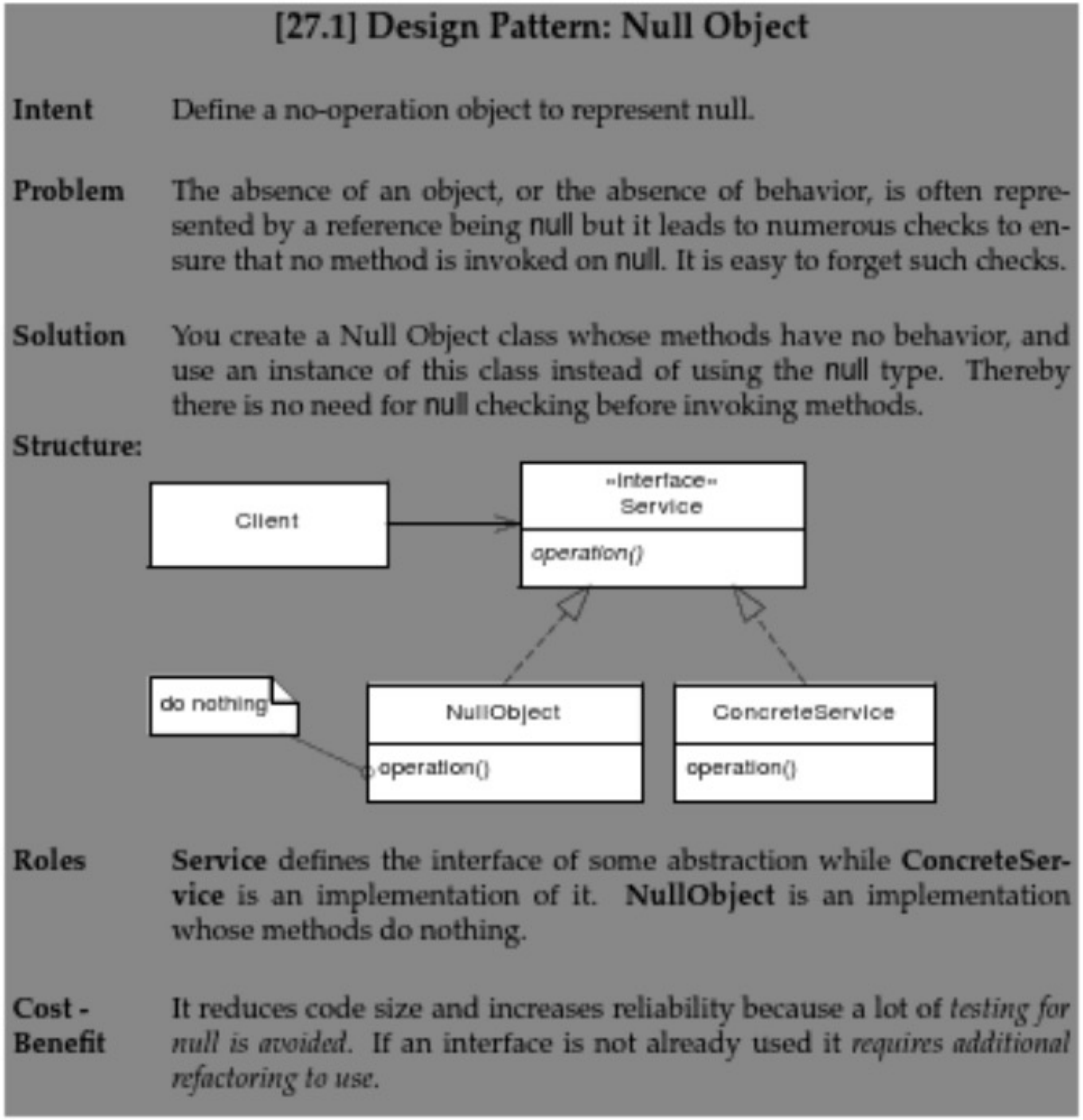
Null Object Pattern

☺

- Ensures object reference is always valid
    - Lower risk of null pointer exceptions
    - Eliminates checks, for analyzability

☹

- Beware of coupling client to ConcreteService



[27.1] Design Pattern: Null Object

Intent Define a no-operation object to represent null.

Problem The absence of an object, or the absence of behavior, is often represented by a reference being null but it leads to numerous checks to ensure that no method is invoked on null. It is easy to forget such checks.

Solution You create a Null Object class whose methods have no behavior, and use an instance of this class instead of using the null type. Thereby there is no need for null checking before invoking methods.

Structure:

Roles **Service** defines the interface of some abstraction while **ConcreteService** is an implementation of it. **NullObject** is an implementation whose methods do nothing.

Cost - Benefit It reduces code size and increases reliability because a lot of *testing for null is avoided*. If an interface is not already used it *requires additional refactoring to use.*

Summary: Pattern Catalog

- **Strategy**: encapsulate each of a family of business rules / algorithms and make them interchangeable
- **State**: allow an object to alter its behavior when internal state changes
- **Abstract Factory:** interface for creating families of related objects
- **Façade**: provide a unified interface to a set of interfaces in a subsystem
- **Decorator**: attach responsibilities to an object dynamically
- **Adapter**: convert interface for a class into another interface that clients expect
- **Builder**: separate construction of an object from its representation, same construction process can create different representations
- **Command**: encapsulate a request as an object, parameterize clients with different requests, queue and log requests, support undo
- **Proxy**: provide a surrogate or placeholder for another object to control access
- **Null Object**: define a no-operation object to represent null
- Later: Model View Controller, Observer

Next time: Systematic Testing