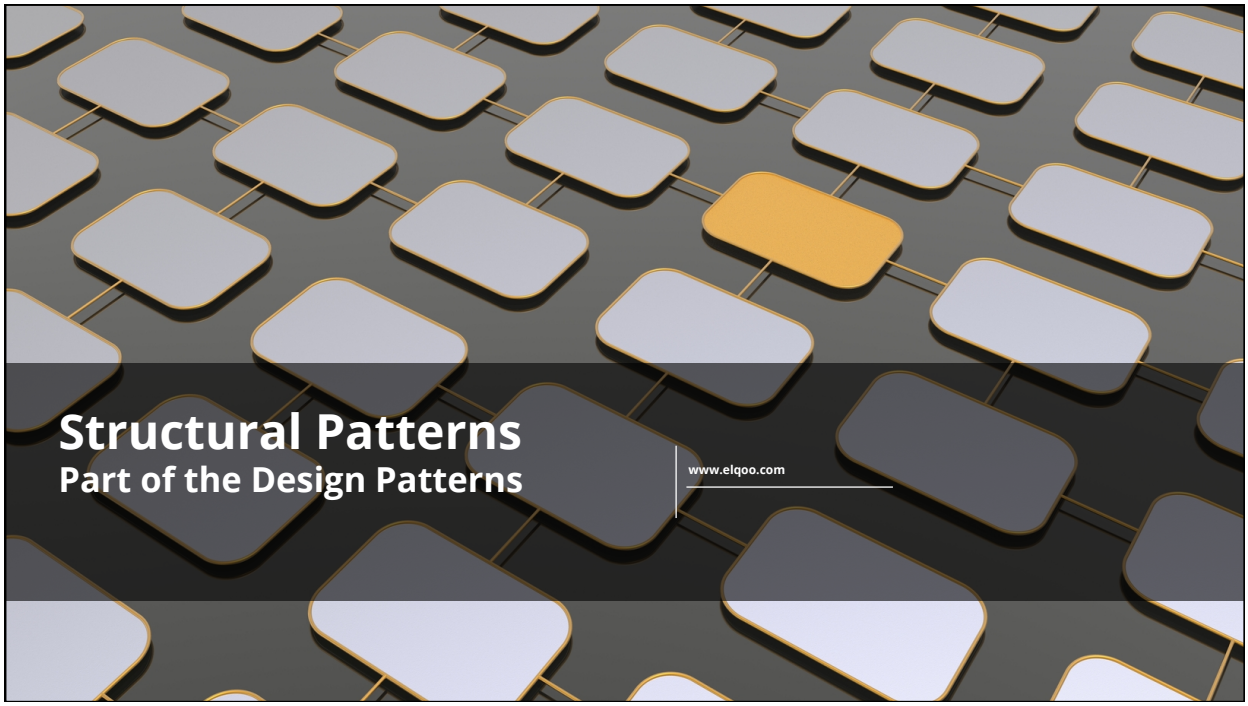
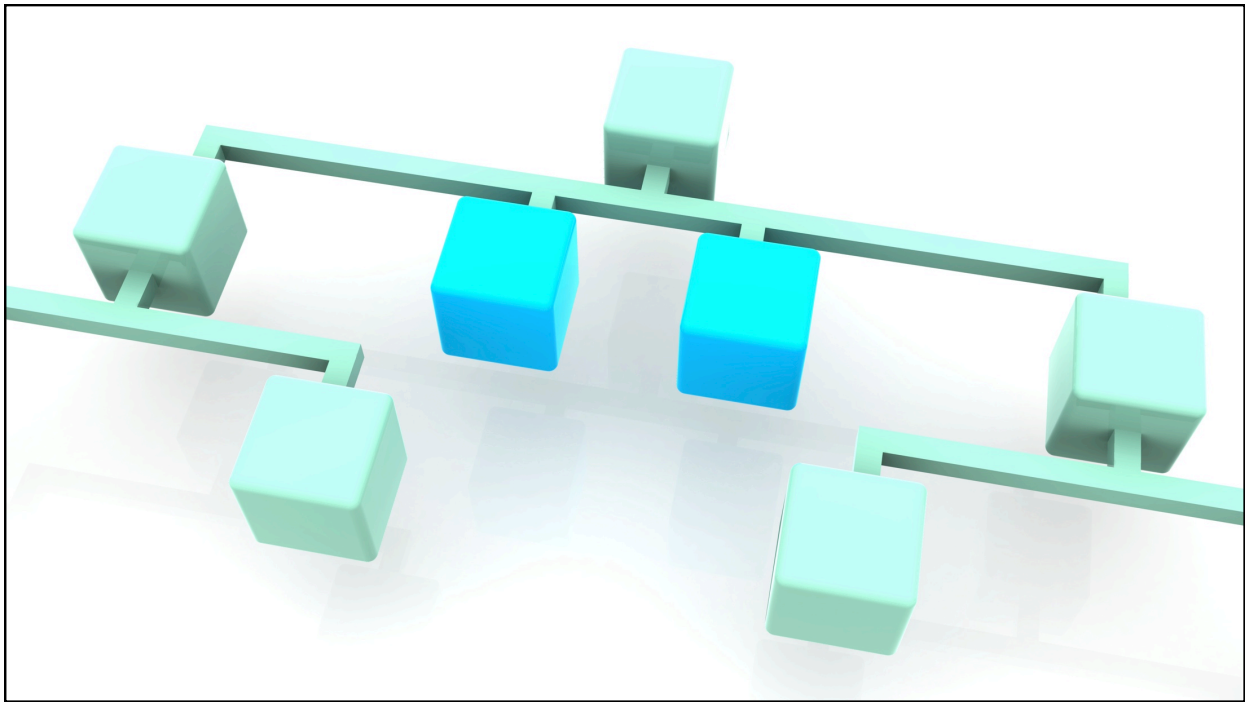
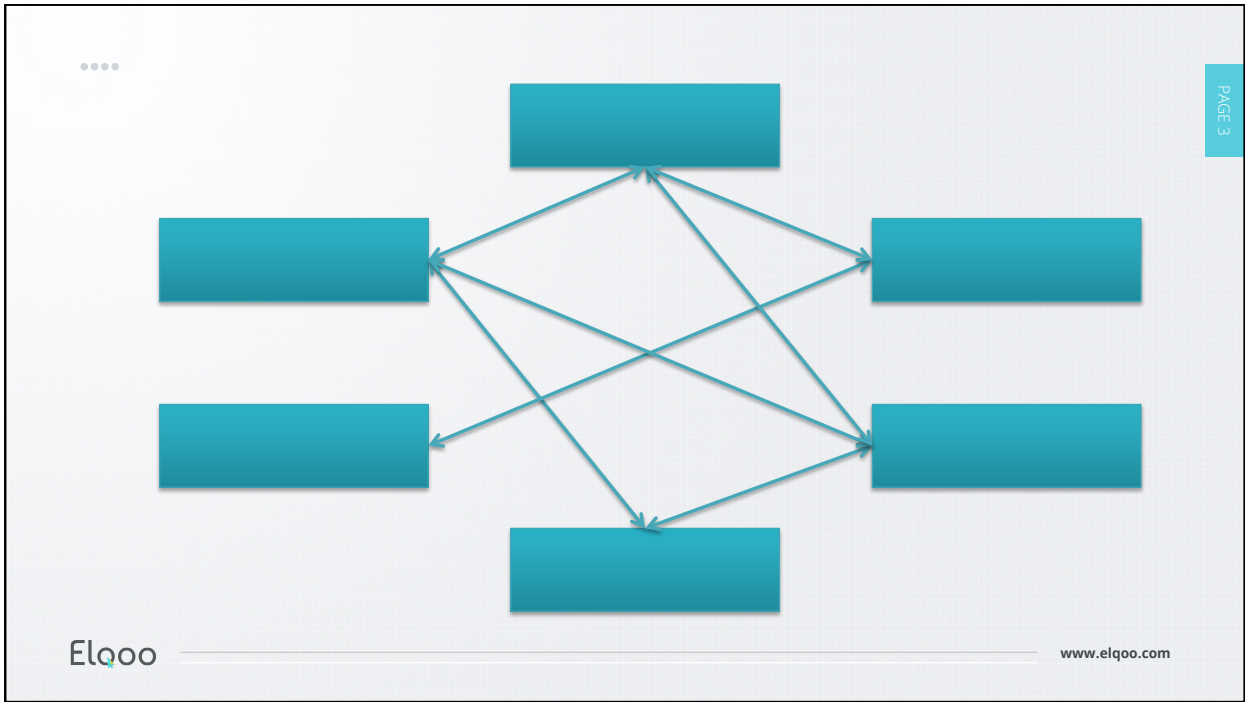
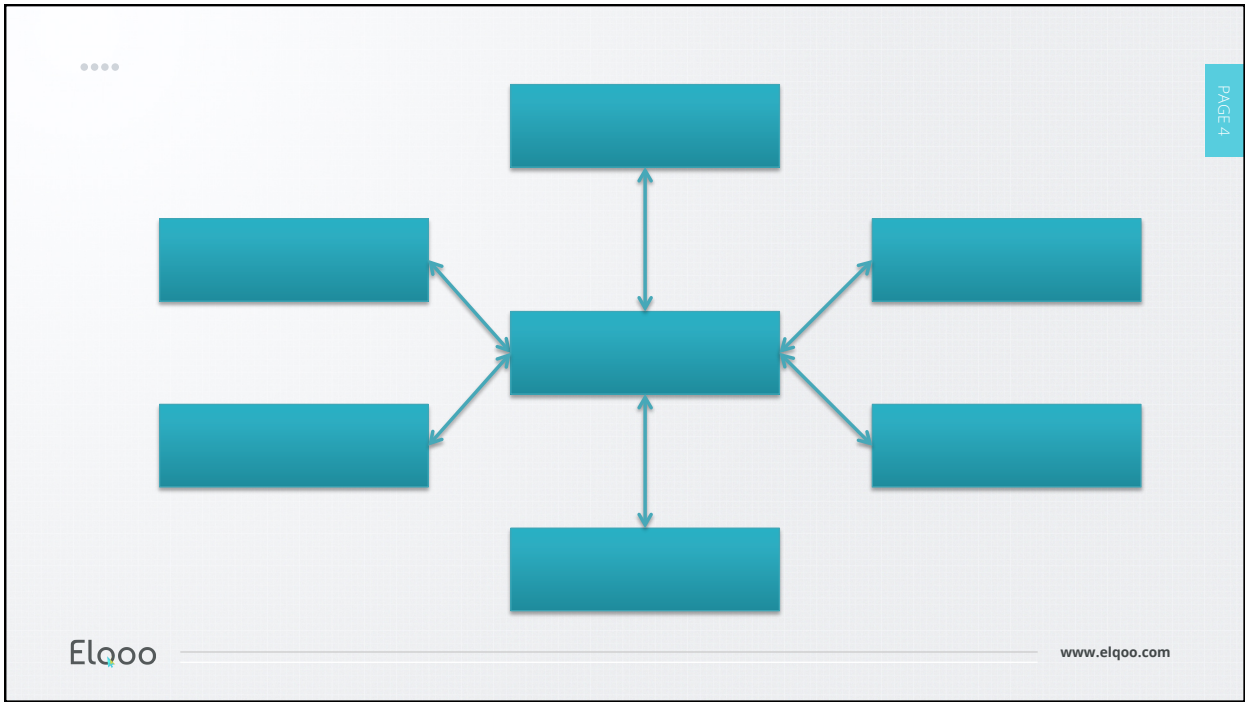
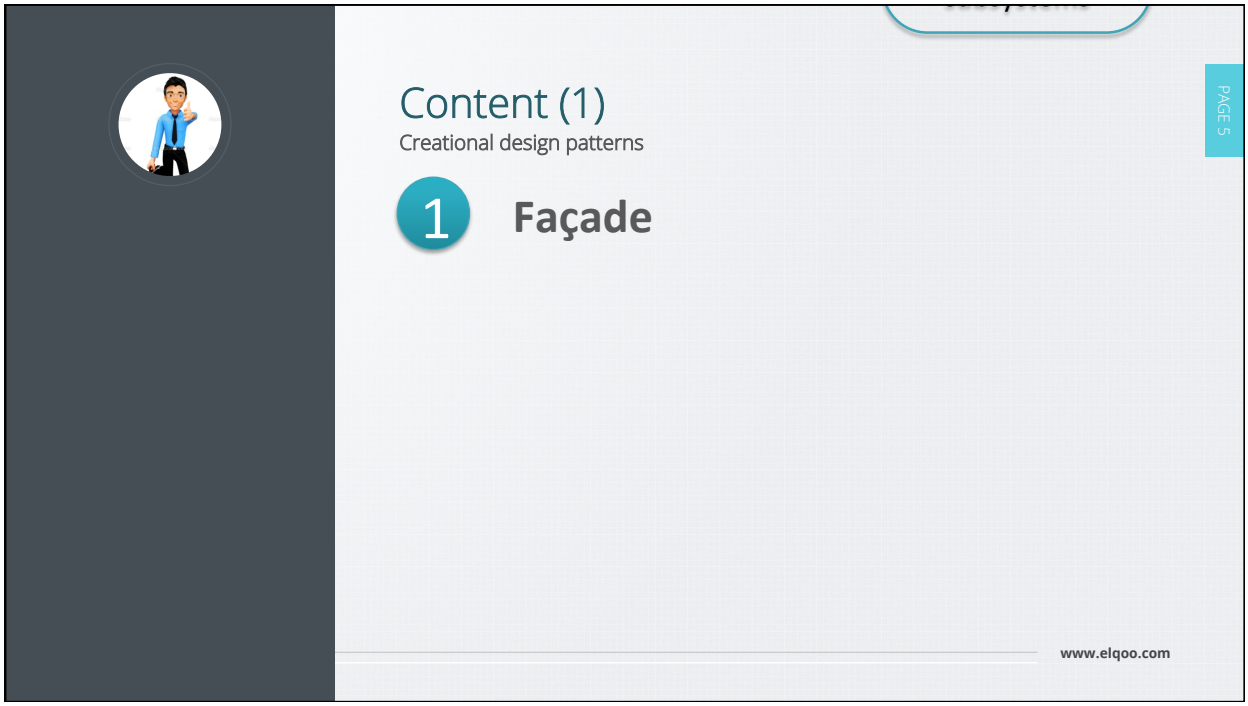

Structural Patterns
Part of the Design Patterns
www.elqoo.com



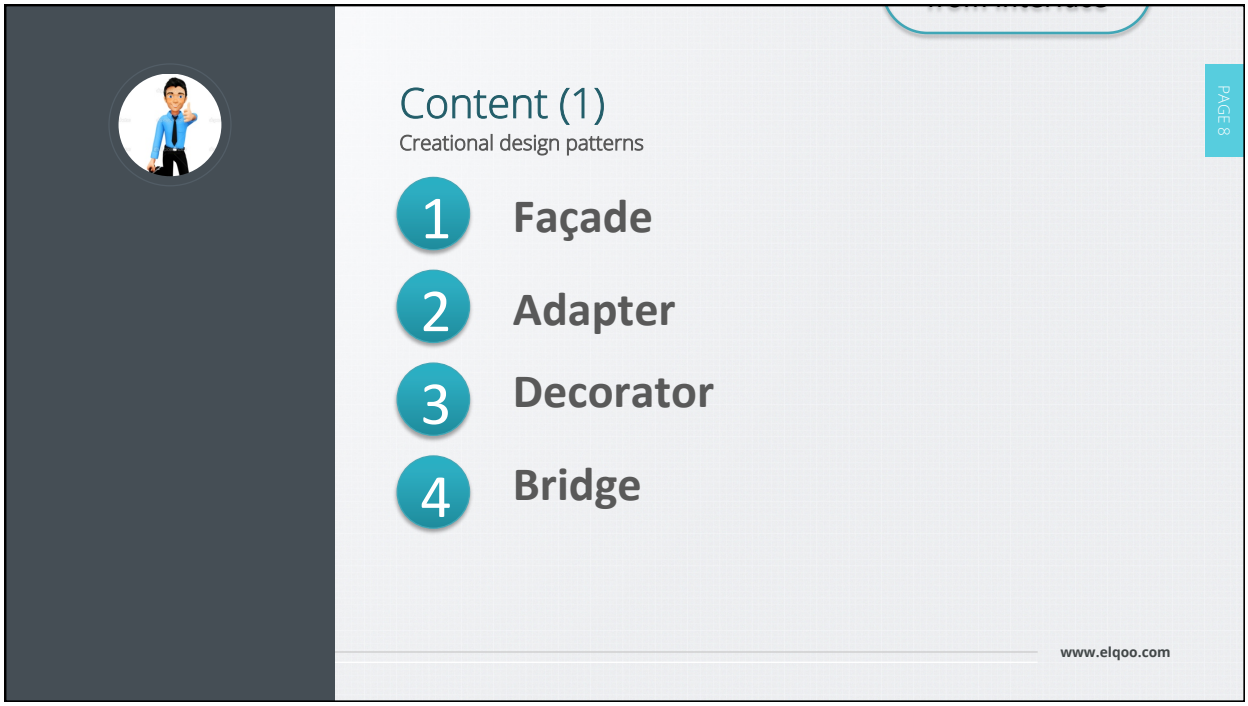

PAGE 3
Elqoo
www.elqoo.com


PAGE 4
Elqoo
www.elqoo.com


Content (1)
Creational design patterns
1 Façade
PAGE 5
www.elqoo.com


Content (1)
Creational design patterns
1 Façade
2 Adapter
PAGE 6
www.elqoo.com


Content (1)
Creational design patterns
1 Façade
2 Adapter
3 Decorator
PAGE 7
www.elqoo.com


Content (1)
Creational design patterns
1 Façade
2 Adapter
3 Decorator
4 Bridge
PAGE 8
www.elqoo.com

Content (1)

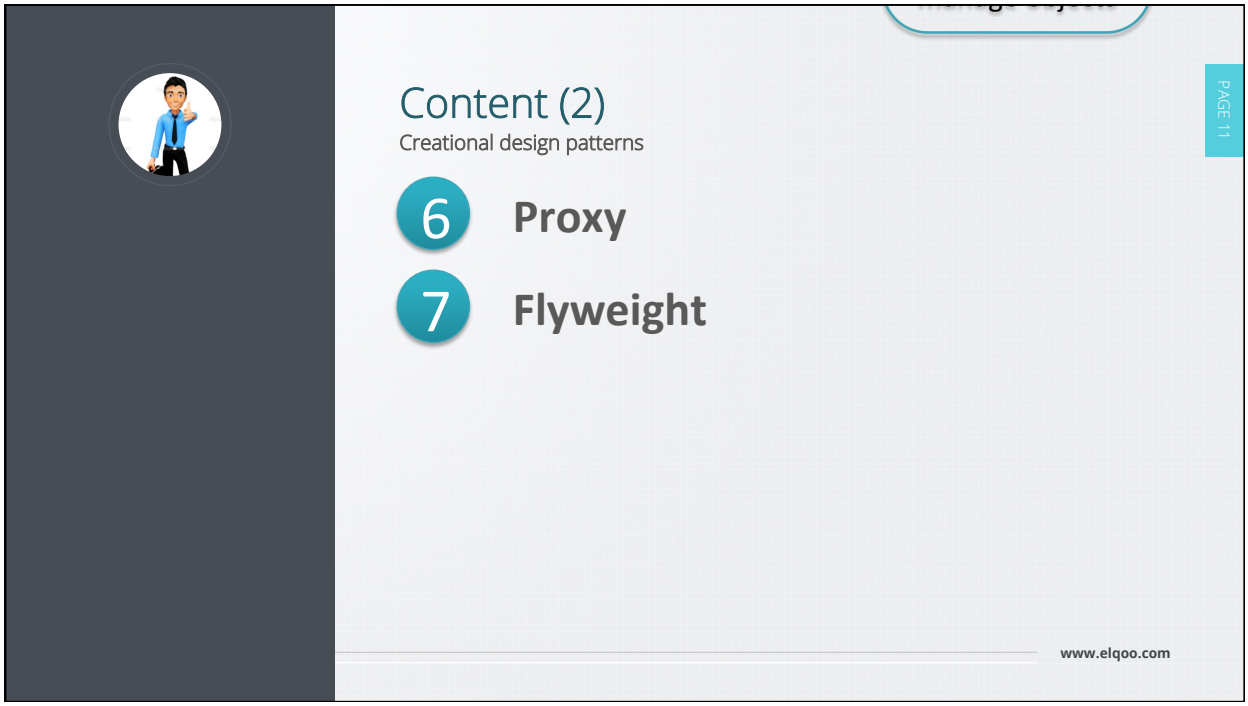
Creational design patterns

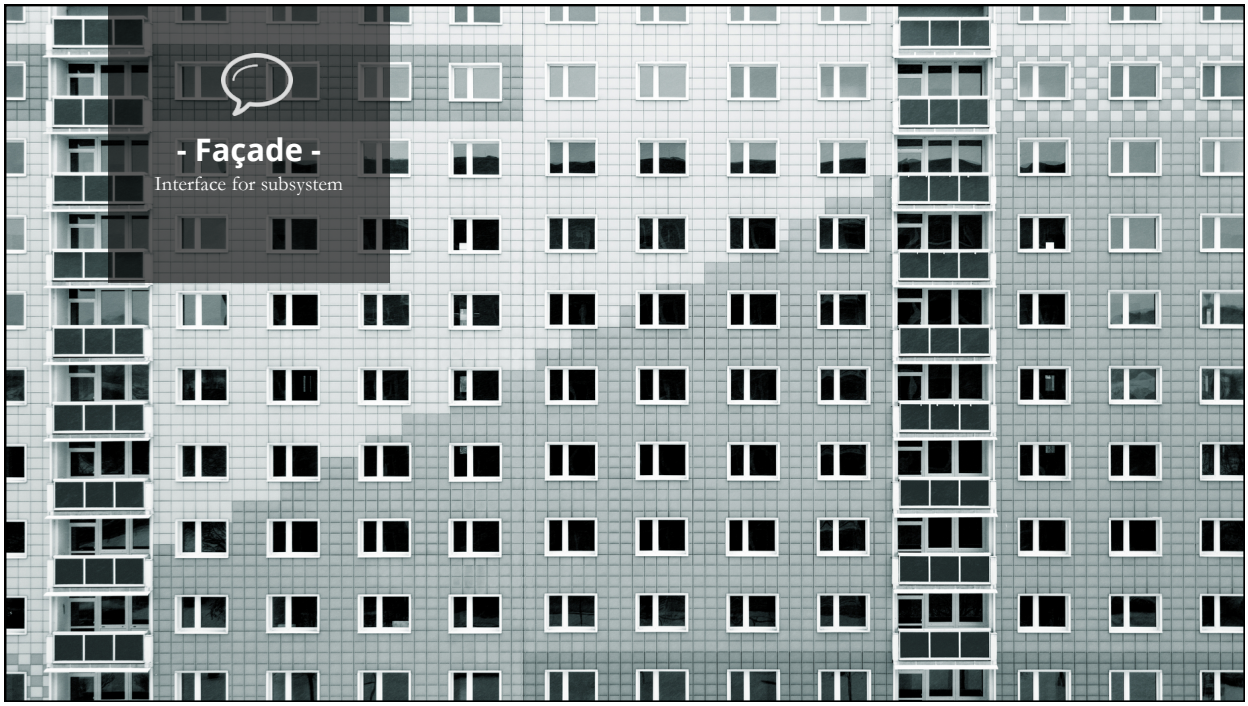
1. **Façade**
2. **Adapter**
3. **Decorator**
4. **Bridge**
5. **Composite**

Content (2)

Creational design patterns

6. **Proxy**


Content (2)
Creational design patterns
6 Proxy
7 Flyweight
PAGE 11
www.elqoo.com


- Façade -
Interface for subsystem

Problem Statement

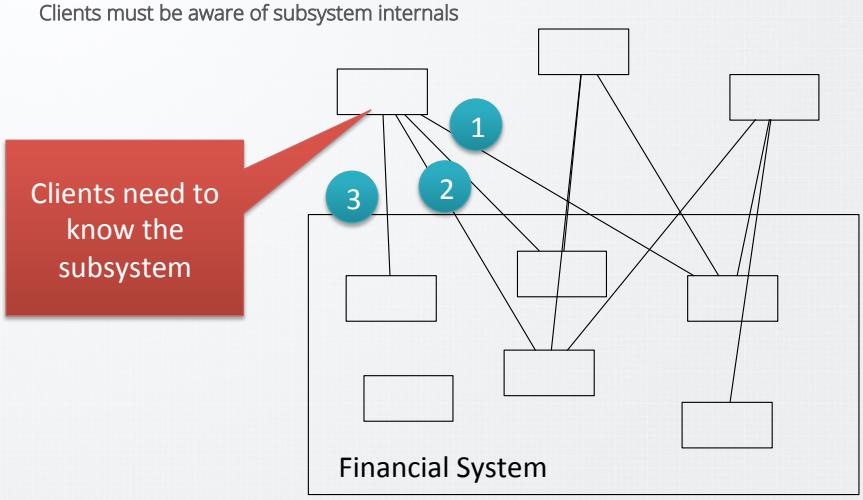
Brad

- Hello Brad
- Hi Suzy
- We need an extra mobile client for our financial system.
- Okido

Suzy

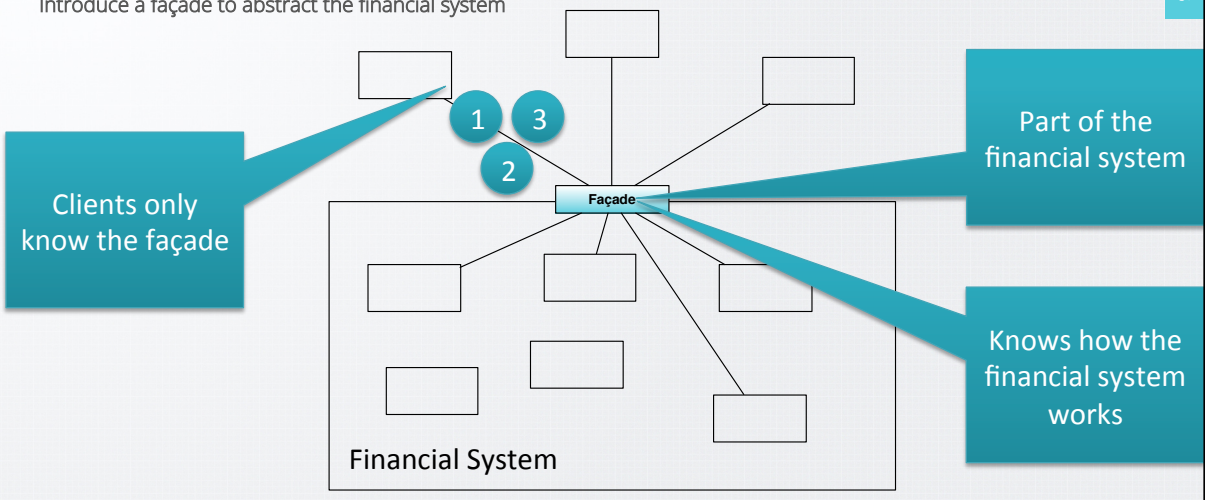
Problem Statement Overview

Clients must be aware of subsystem internals

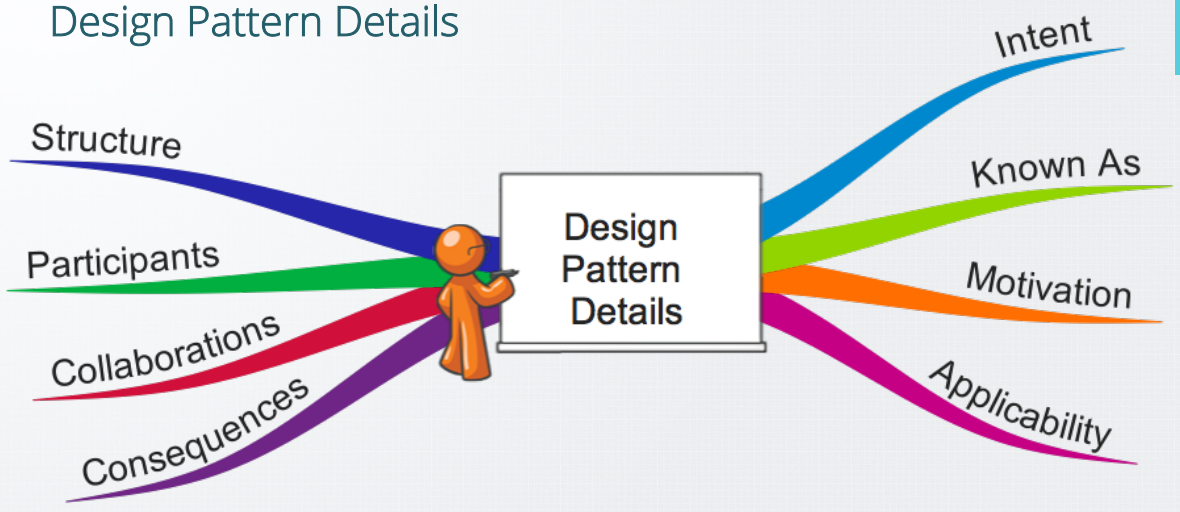


Problem Solution

Introduce a façade to abstract the financial system



Design Pattern Details



Façade Pattern

Intent and known as

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**.

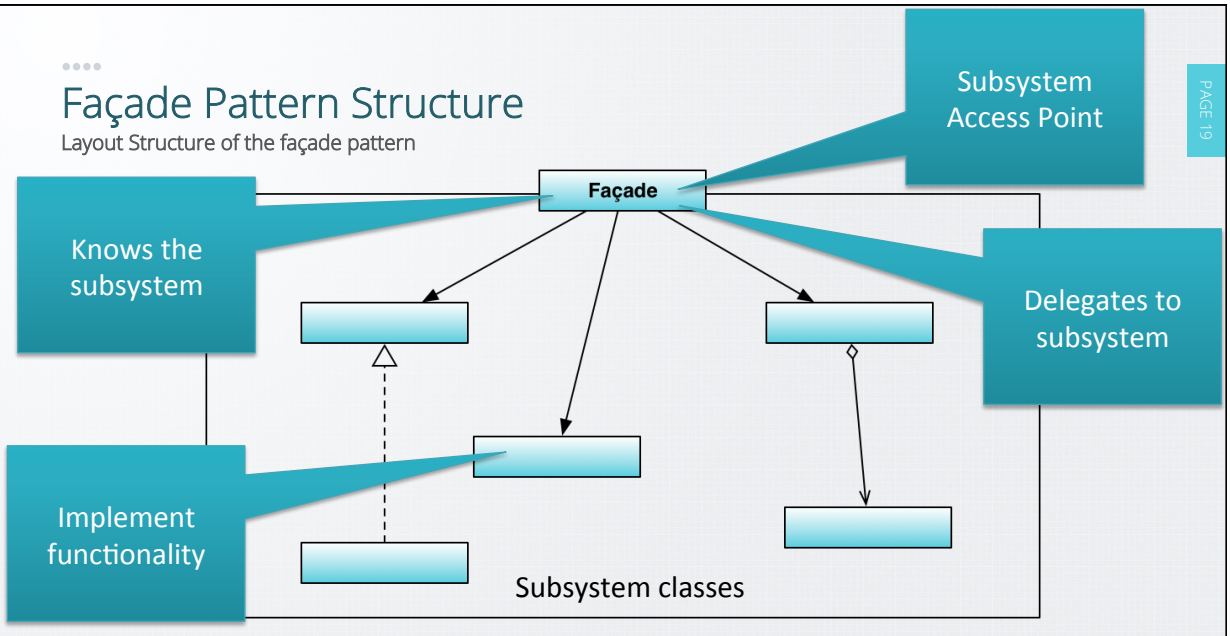
Apply Façade Pattern

Applicability

- **Use**
  - **Decouple** clients from subsystems
  - Provide **simple interface**
  - Subsystem layering (business, data and client services)

Façade Pattern Structure

Layout Structure of the façade pattern



Façade Pattern Collaboration

How objects work with a façade

- **Client → Façade → Subsystem**
  - Façade needs to translate request into subsystem specific objects
- Clients don't have access to the subsystem
  - Loose coupling

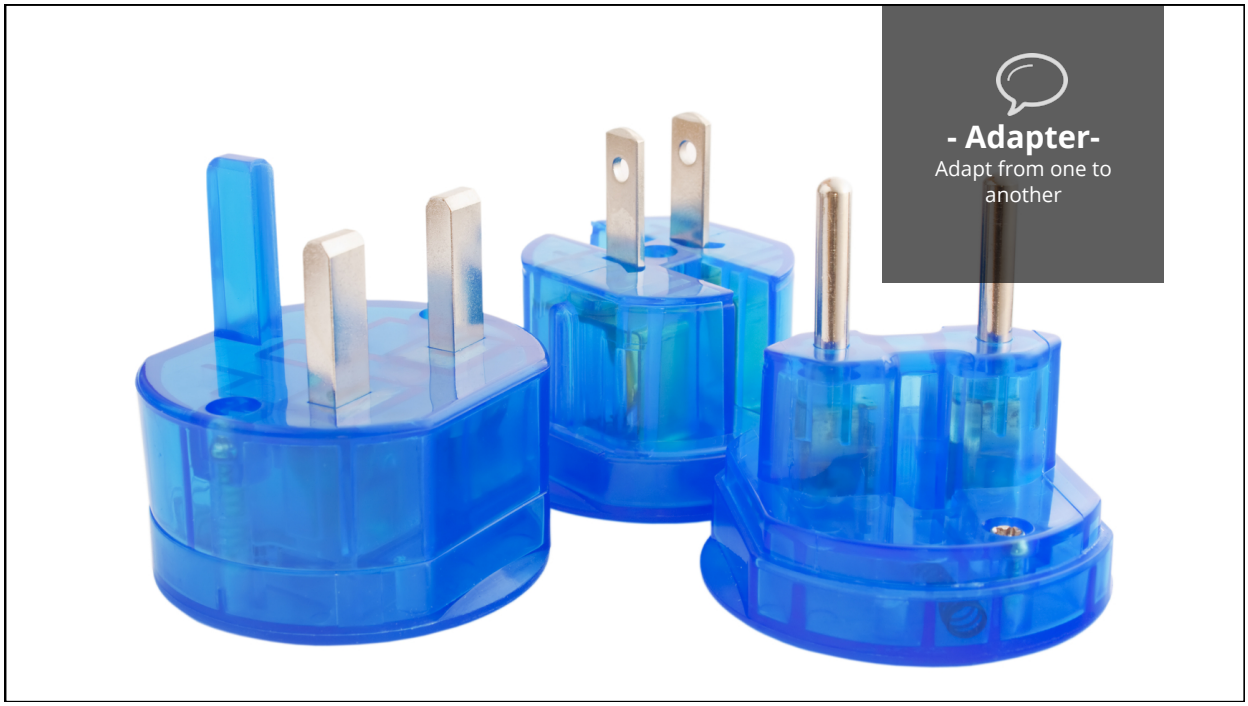
Façade Pattern Consequences

Benefits of the façade pattern

- **Benefits**
  - Subsystem **easier** to use
    - Client don't require specific knowledge
  - Loose coupling
  - Subsystem **can** still be **used directly** (if necessary)
- **Drawbacks**
  - Façade introduces an extra programming layer

Conclusion

- **Façade pattern is great**
  - **Abstract complex system** from client
  - Provide extra **loose coupling**



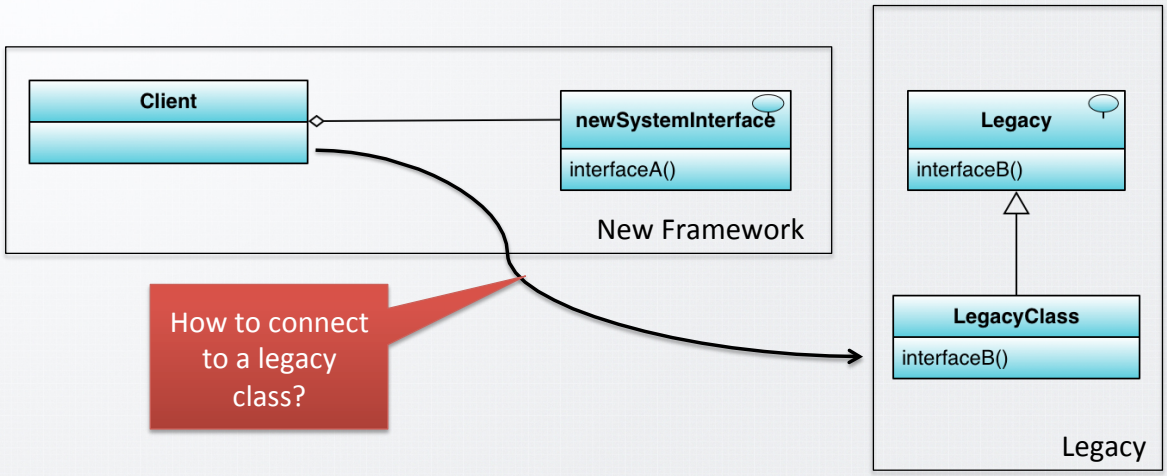
Problem Statement

Brad

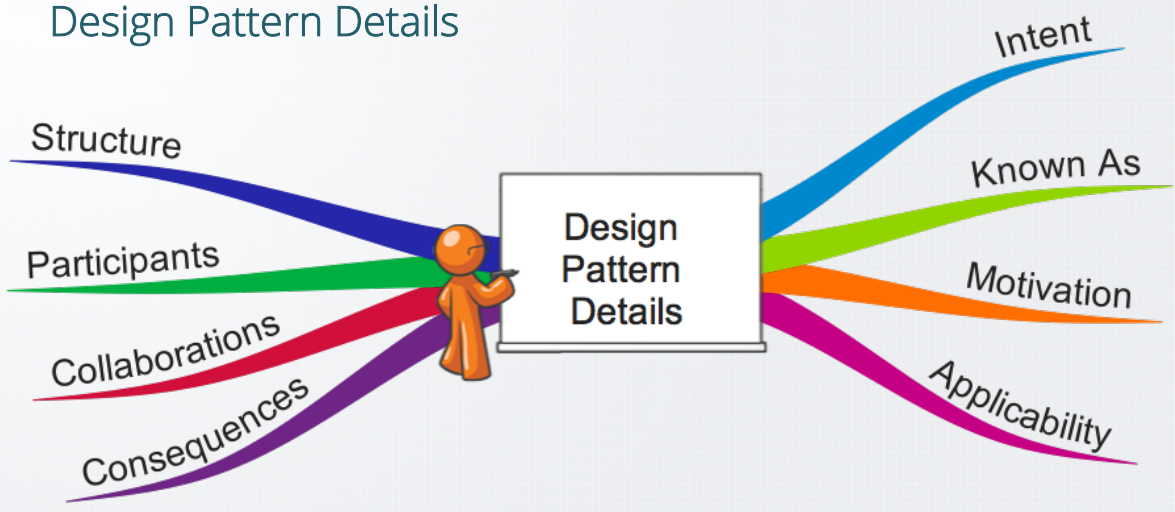
- Hello Brad
- Hi Suzy
- We want to connect to our old legacy systems, but there're not conform or standards. Can it be done?
- I'll have to investigate

Suzy

Problem Statement Overview



Design Pattern Details



Adapter Pattern

Intent and known as

Intent

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

Known As

Wrapper

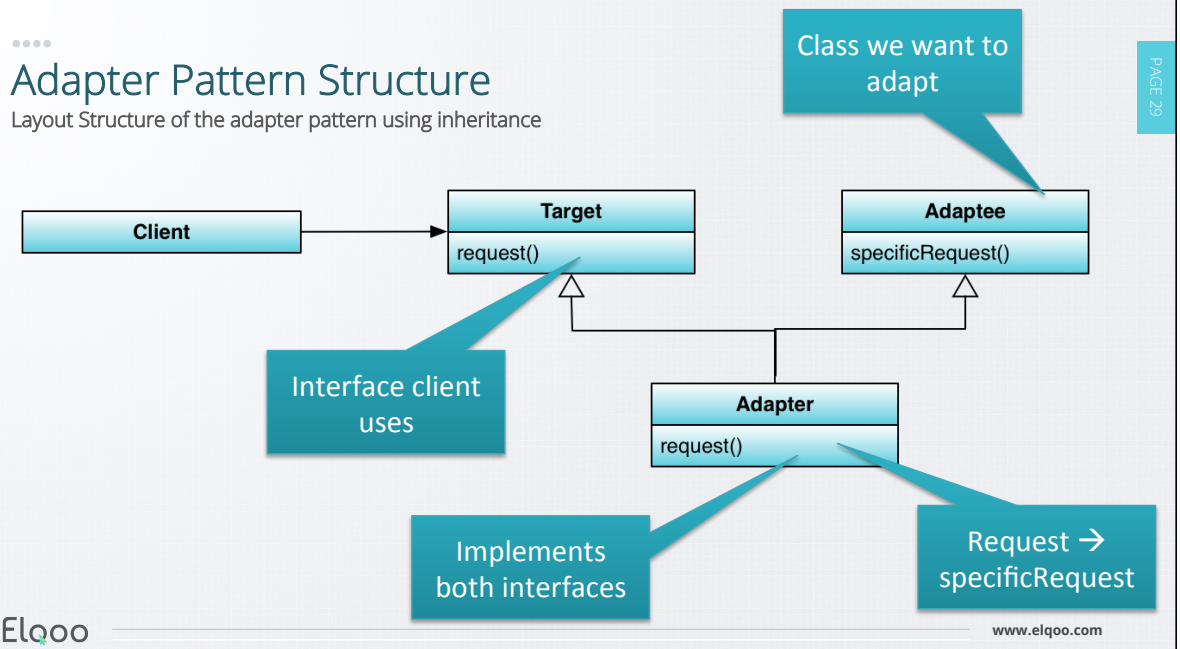
Apply Adapter Pattern

Applicability

- **Use**
  - **Re-use** an existing class
  - **Combine** unrelated **classes** with an incompatible interface

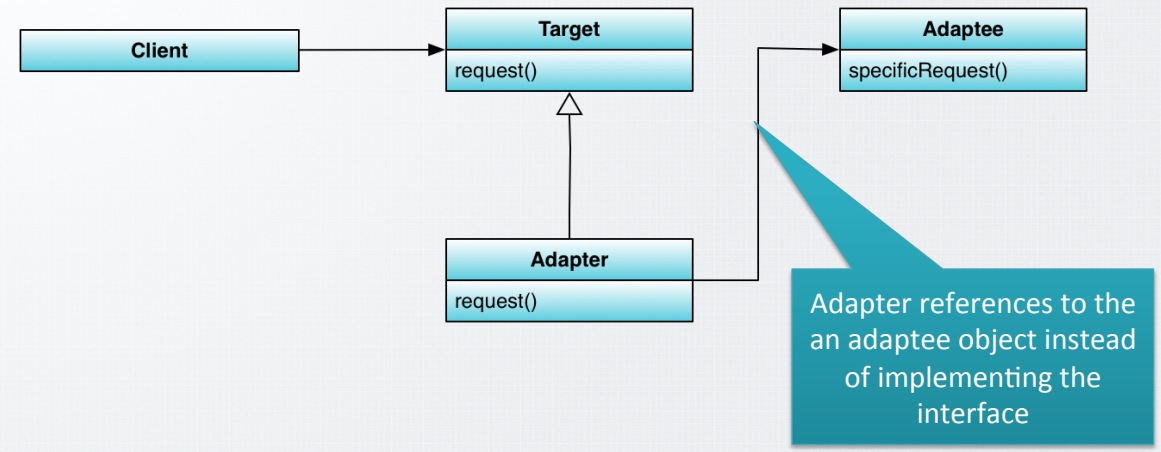
Adapter Pattern Structure

Layout Structure of the adapter pattern using inheritance



Adapter Pattern Structure Alternative

Layout structure of the adapter pattern using composition



Adapter Pattern Collaboration

Collaboration between objects

- Adapter **forwards request** to adaptee
  - This can be adapter itself (**inheritance**)
  - This can be a **reference** of the adaptee

Adapter Pattern Consequences (1)

Considerations for the adapter pattern

- **Benefits**
  - Adapter can **override** adaptee **behaviour** → it is a subclass
  - One adapter → Many adaptees
- **Drawbacks**
  - Adapter doesn't work for
    - Class with many subclasses → can't extend them all
  - Harder to override adaptee behaviour (do we want this?)

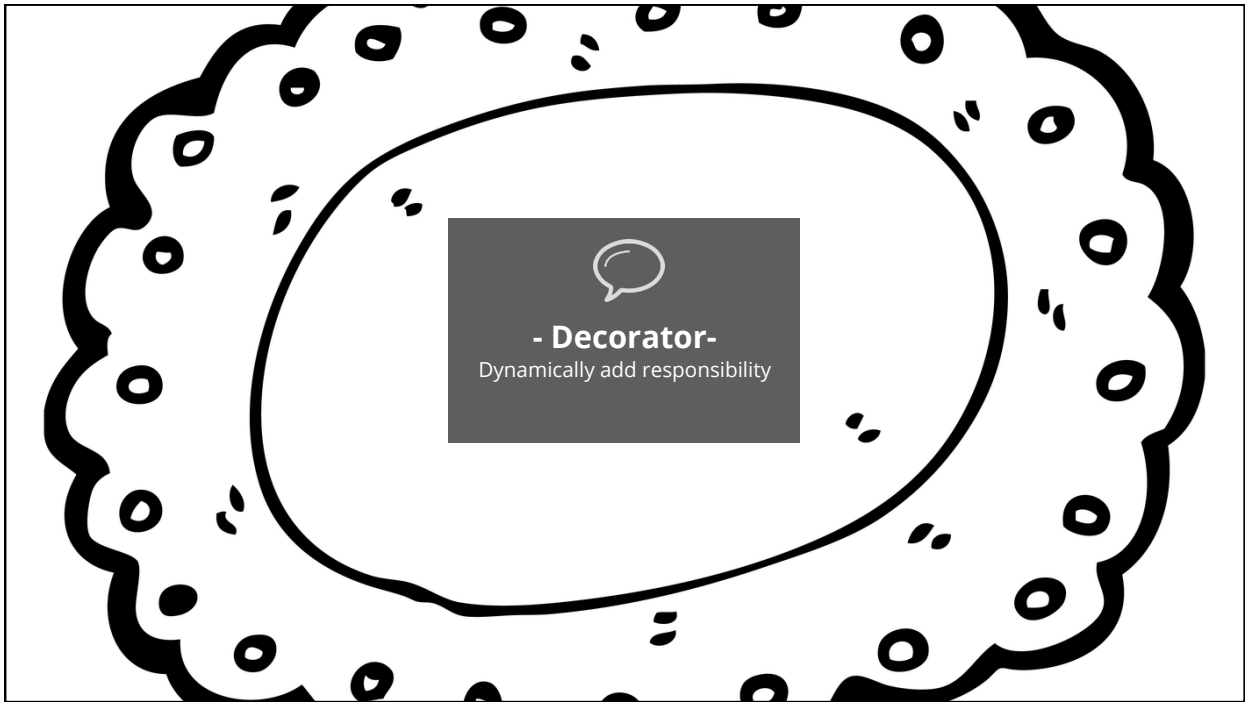
Adapter Pattern Consequences (2)

Considerations for the adapter pattern

- How much adapting does the adapter do?
  - **Complex adaptations** will be hard to write.

Conclusion

- **Adapter pattern is great**
  - **Connect to legacy system**
  - Adapt one interface to another



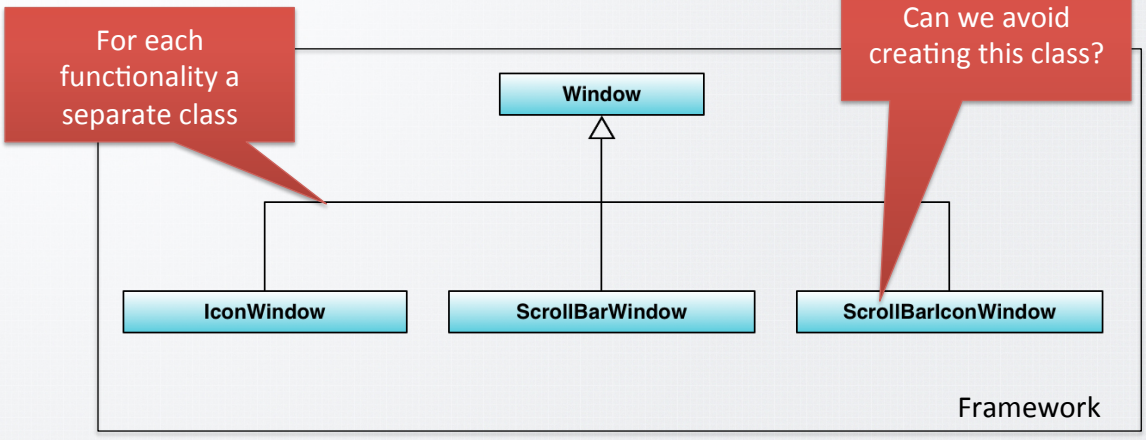
Problem Statement

Brad

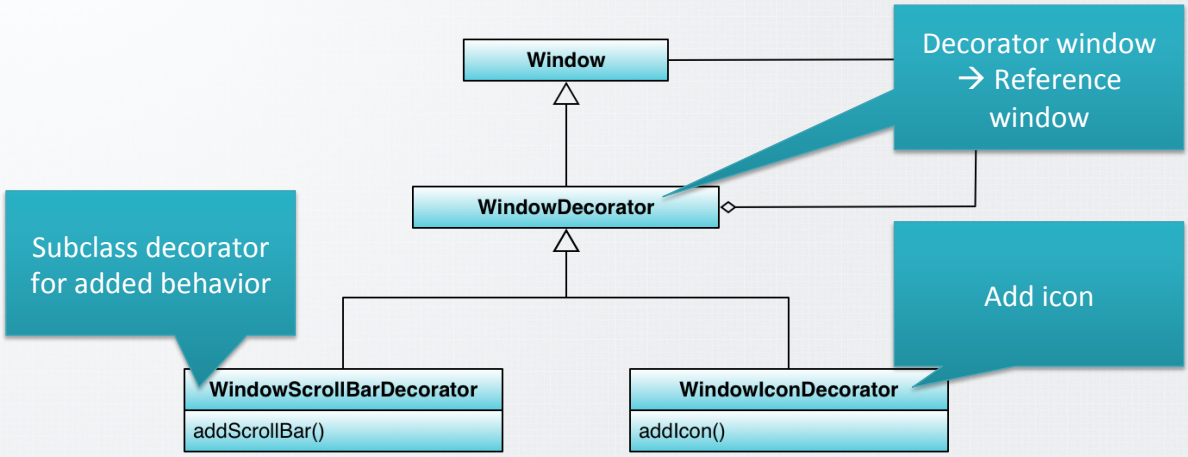
- Hello Brad
- Hi Suzy
- We would like to have a framework that supports a visual window with a scrollbar and an icon.
- This is easy

Suzy

Problem Statement Overview

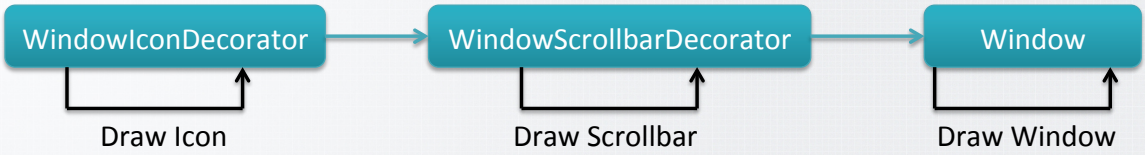


Problem Solution (1)

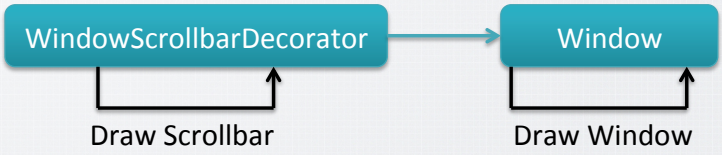


Problem Solution (2)

1. We need a window with an **icon** and a **scrollbar**

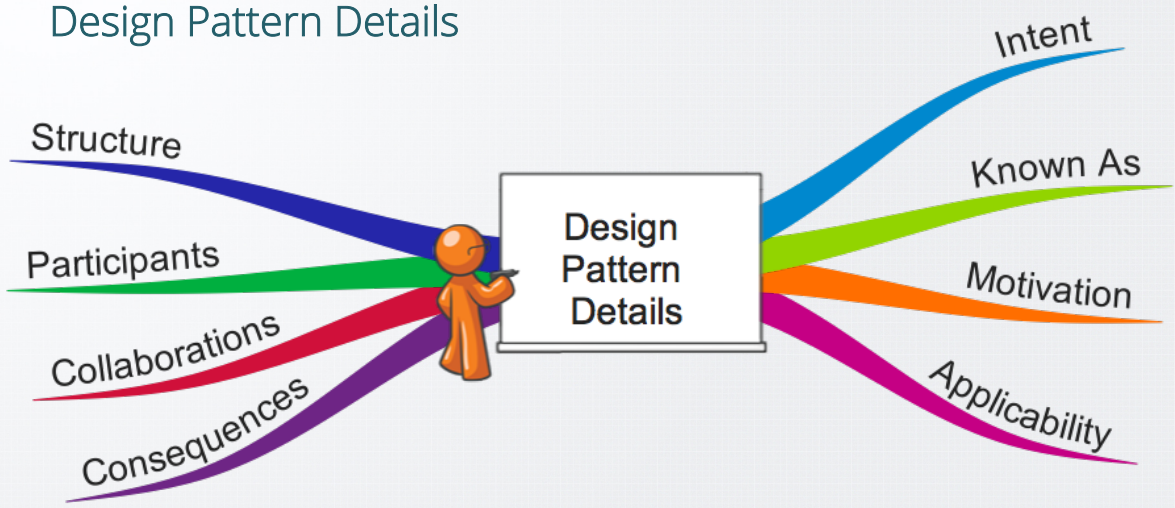


2. We need a window with **just** a **scrollbar**



Elqoo www.elqoo.com

Design Pattern Details



Elqoo www.elqoo.com

Decorator Pattern

Intent and known as

Intent

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

Known As

Wrapper

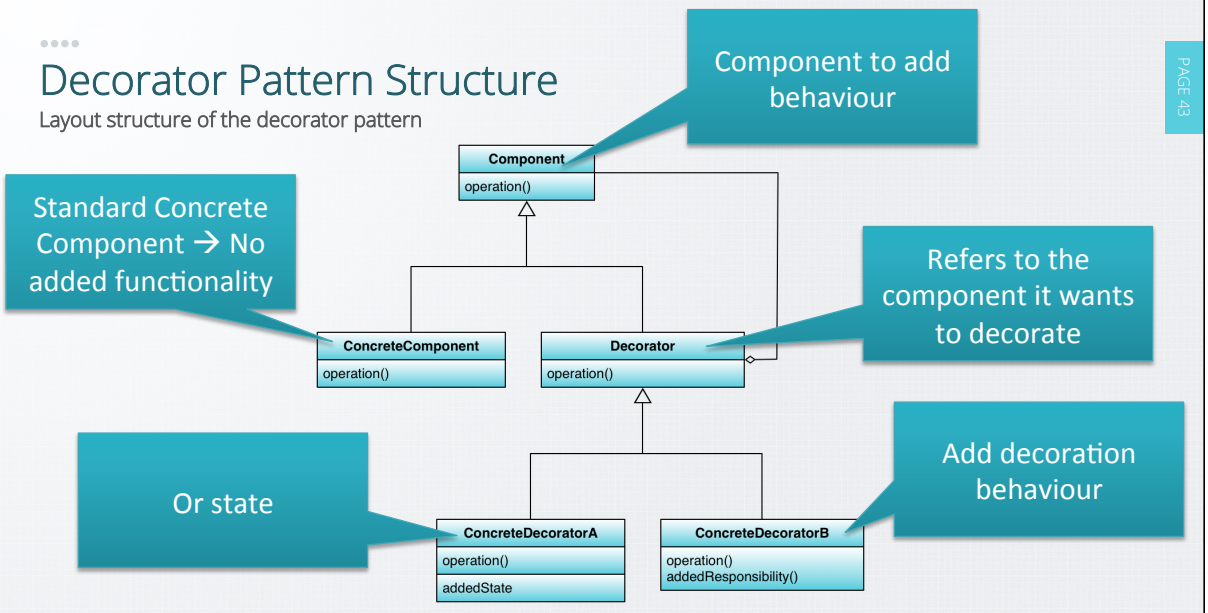
Apply Decorator Pattern

Applicability

- **Use**
  - **Add functionality** to objects, without affecting other objects
  - Functionalities can be taken away in the future
  - **Extension** by subclassing **is difficult**

Decorator Pattern Structure

Layout structure of the decorator pattern



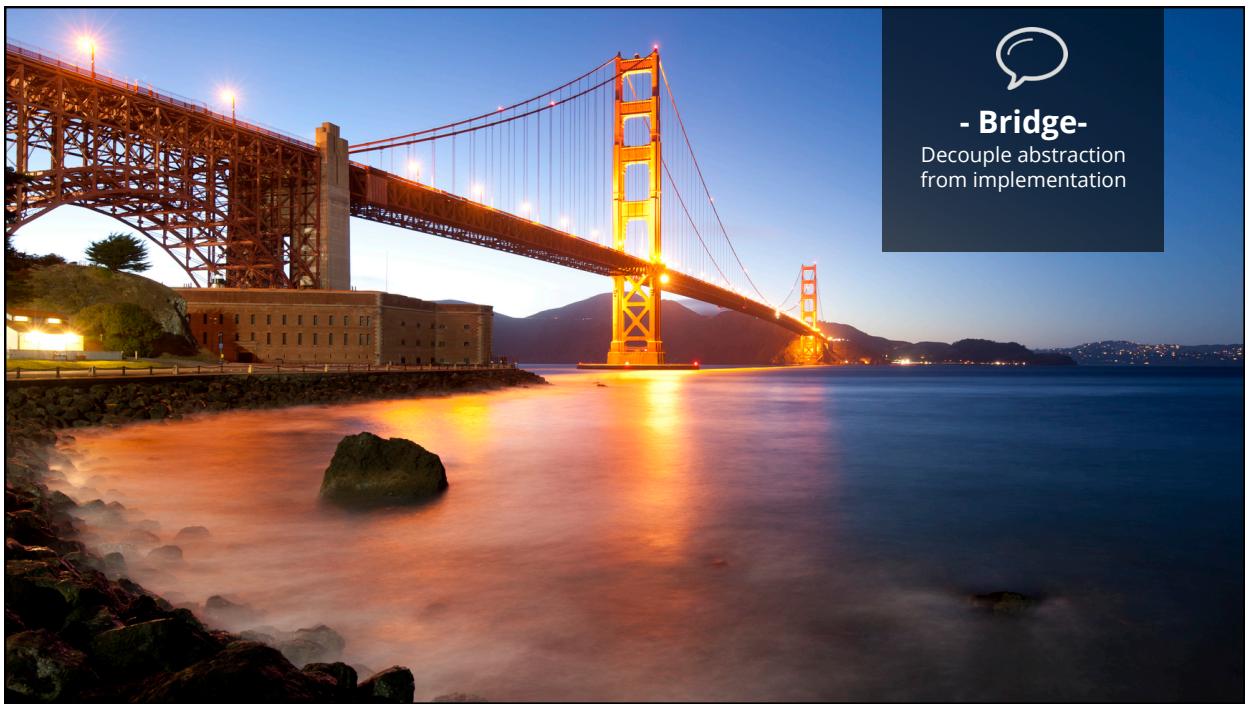
Decorator Pattern Consequences

Considerations for the decorator pattern

- **Benefits**
  - More **flexible** than inheritance
  - **Implement** functionality **when** you **need it**
    - Where you need it in the class hierarchy
- **Drawbacks**
  - **Object identity** cannot be used with decorators
  - **Too many** small decorator **objects**

Conclusion

- **Decorator pattern is great**
  - **Add behavior** to objects
  - **Higher flexibility** than **extension**



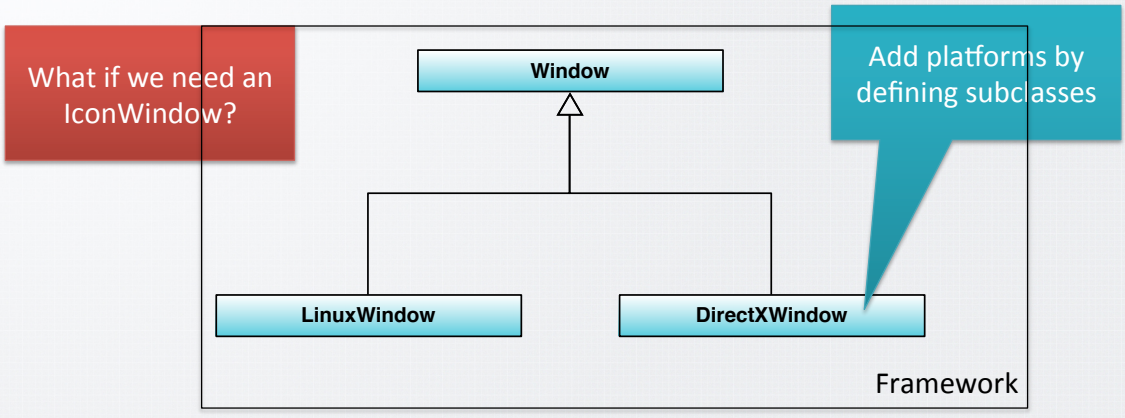
Problem Statement

Brad

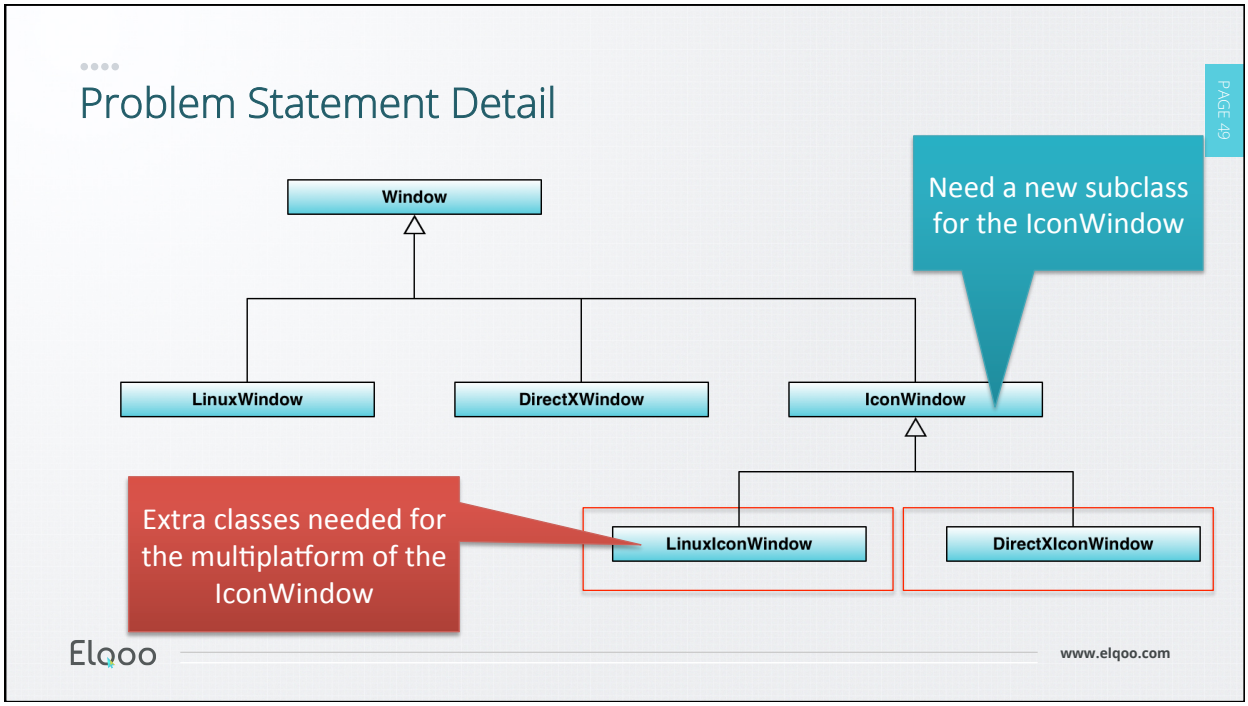
- Hello Brad
- Hi Suzy
- We want to create a multiplatform window toolkit
- Sounds a challenge

Suzy

Problem Statement Overview

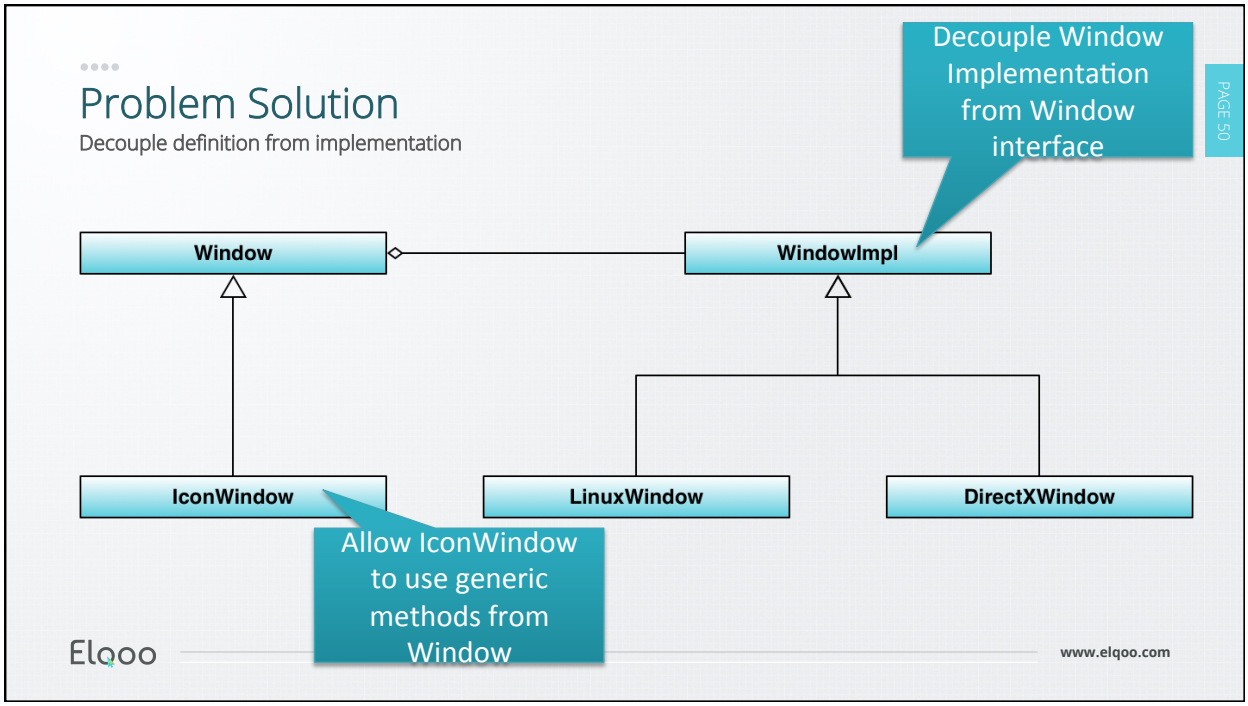


Problem Statement Detail



Problem Solution

Decouple definition from implementation



Design Pattern Details

Bridge Pattern

Intent and known as

Intent

Decouple an **abstraction** from its **implementation** so that the two can vary independently.

Known As

Handle, Body

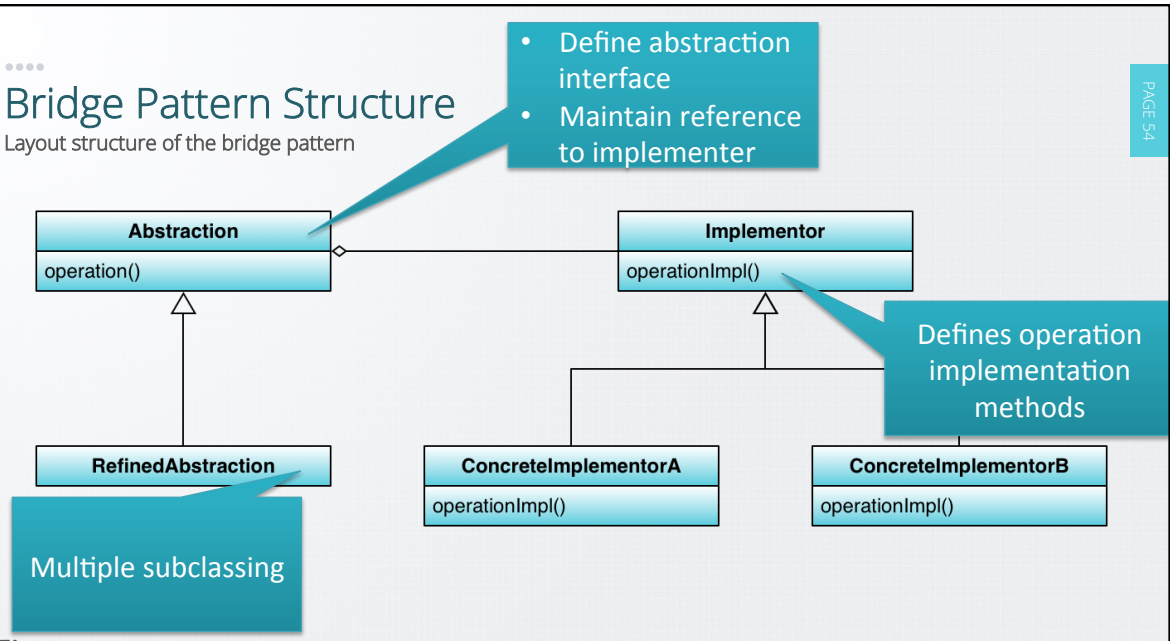
Apply Bridge Pattern

Applicability

- **Use**
  - **Avoid binding** between interface and implementation
  - Possible subclasses for abstraction and implementation
  - Must be possible to **change implementation at runtime** without affecting clients

Bridge Pattern Structure

Layout structure of the bridge pattern



Bridge Pattern Consequences

Considerations for the bridge pattern

- **Benefits**
  - **Decouple** interface and implementation
  - Improve **extensibility**
  - Hide implementation details

Conclusion

- **Bridge pattern is great**
  - **Decouple interface** from implementation
  - **Reduce** nr of **subclasses**

Bridge Design Pattern

Intent

- Decouple an abstraction from its implementation so that the two can vary independently.
- Publish interface in an inheritance hierarchy, and bury implementation in its own inheritance hierarchy.
- Beyond encapsulation, to insulation

Problem

"Hardening of the software arteries" has occurred by using subclassing of an abstract base class to provide alternative implementations. This locks in compile-time binding between interface and implementation. The abstraction and implementation cannot be independently extended or composed.

Motivation

Consider the domain of "thread scheduling".

There are two types of thread schedulers, and two types of operating systems or "platforms". Given this approach to specialization, we have to define a class for each permutation of these two dimensions. If we add a new platform (say ... Java's Virtual Machine), what would our hierarchy look like?

What if we had three kinds of thread schedulers, and four kinds of platforms? What if we had five kinds of thread schedulers, and ten kinds of platforms? The number of classes we would have to define is the product of the number of scheduling schemes and the number of platforms.

The Bridge design pattern proposes refactoring this exponentially explosive inheritance hierarchy into two orthogonal hierarchies – one for platform-independent abstractions, and the other for platform-dependent implementations.

Discussion

Decompose the component's interface and implementation into orthogonal class hierarchies. The interface class contains a pointer to the abstract implementation class. This pointer is initialized with an instance of a concrete implementation class, but all subsequent interaction from the interface class to the implementation class is limited to the abstraction maintained in the implementation base class. The client interacts with the interface class, and it in turn "delegates" all requests to the implementation class.

The interface object is the "handle" known and used by the client; while the implementation object, or "body", is safely encapsulated to ensure that it may continue to evolve, or be entirely replaced (or shared at run-time.

Use the Bridge pattern when:

- you want run-time binding of the implementation,
- you have a proliferation of classes resulting from a coupled interface and numerous implementations,
- you want to share an implementation among multiple objects,
- you need to map orthogonal class hierarchies.

Consequences include:

- decoupling the object's interface,
- improved extensibility (you can extend (i.e. subclass) the abstraction and implementation hierarchies independently),
- hiding details from clients.

Bridge is a synonym for the "handle/body" idiom. This is a design mechanism that encapsulates an implementation class inside of an interface class. The former is the body, and the latter is the handle. The handle is viewed by the user as the actual class, but the work is done in the body. "The handle/body class idiom may be used to decompose a complex abstraction into smaller, more manageable classes. The idiom may reflect the sharing of a single resource by multiple classes that control access to it (e.g. reference counting)."

Structure

The Client doesn't want to deal with platform-dependent details. The Bridge pattern encapsulates this complexity behind an abstraction "wrapper".

Bridge emphasizes identifying and decoupling "interface" abstraction from "implementation" abstraction.

Example

The Bridge pattern decouples an abstraction from its implementation, so that the two can vary independently. A household switch controlling lights, ceiling fans, etc. is an example of the Bridge. The purpose of the switch is to turn a device on or off. The actual switch can be implemented as a pull chain, simple two position switch, or a variety of dimmer switches.

Check list

1. Decide if two orthogonal dimensions exist in the domain. These independent concepts could be: abstraction/platform, or domain/infrastructure, or front-end/back-end, or interface/implementation.
2. Design the separation of concerns: what does the client want, and what do the platforms provide.
3. Design a platform-oriented interface that is minimal, necessary, and sufficient. Its goal is to decouple the abstraction from the platform.
4. Define a derived class of that interface for each platform.
5. Create the abstraction base class that "has a" platform object and delegates the platform-oriented

functionality to it.

6. Define specializations of the abstraction class if desired.

Rules of thumb

- Adapter makes things work after they're designed; Bridge makes them work before they are.
- Bridge is designed up-front to let the abstraction and the implementation vary independently. Adapter is retrofitted to make unrelated classes work together.
- State, Strategy, Bridge (and to some degree Adapter) have similar solution structures. They all share elements of the "handle/body" idiom. They differ in intent - that is, they solve different problems.
- The structure of State and Bridge are identical (except that Bridge admits hierarchies of envelope classes, whereas State allows only one). The two patterns use the same structure to solve different problems: State allows an object's behavior to change along with its state, while Bridge's intent is to decouple an abstraction from its implementation so that the two can vary independently.
- If interface classes delegate the creation of their implementation classes (instead of creating/coupling themselves directly), then the design usually uses the Abstract Factory pattern to create the implementation objects.

Read next

Code examples

Java	**Bridge in Java**	**Bridge in Java**	**Bridge in Java**
C++	**Bridge in C++**		
PHP	**Bridge in PHP**		
Python	**Bridge in Python**		

RETURN

Adapter Design Pattern

READ NEXT

Composite Design Pattern

Problem Statement

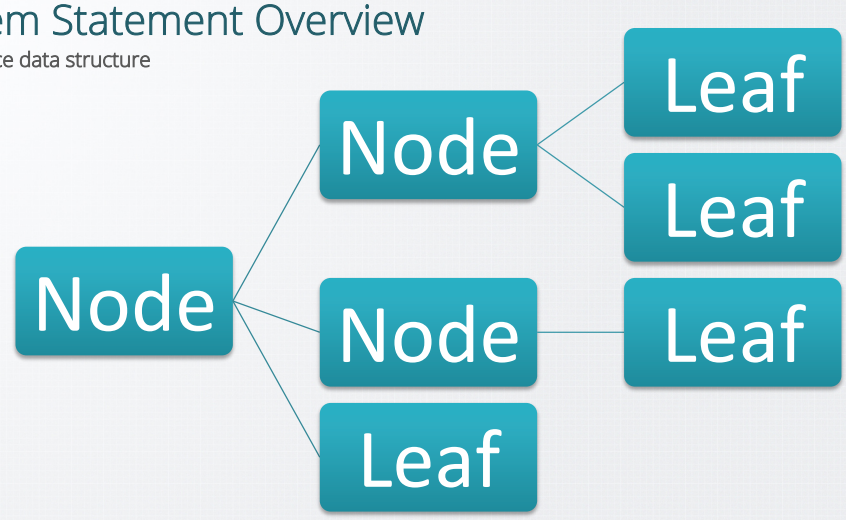
Brad

- Hello Brad
- Hi Suzy
- We want to be able to group objects in our graphical user interface
- No problem

Suzy

Problem Statement Overview

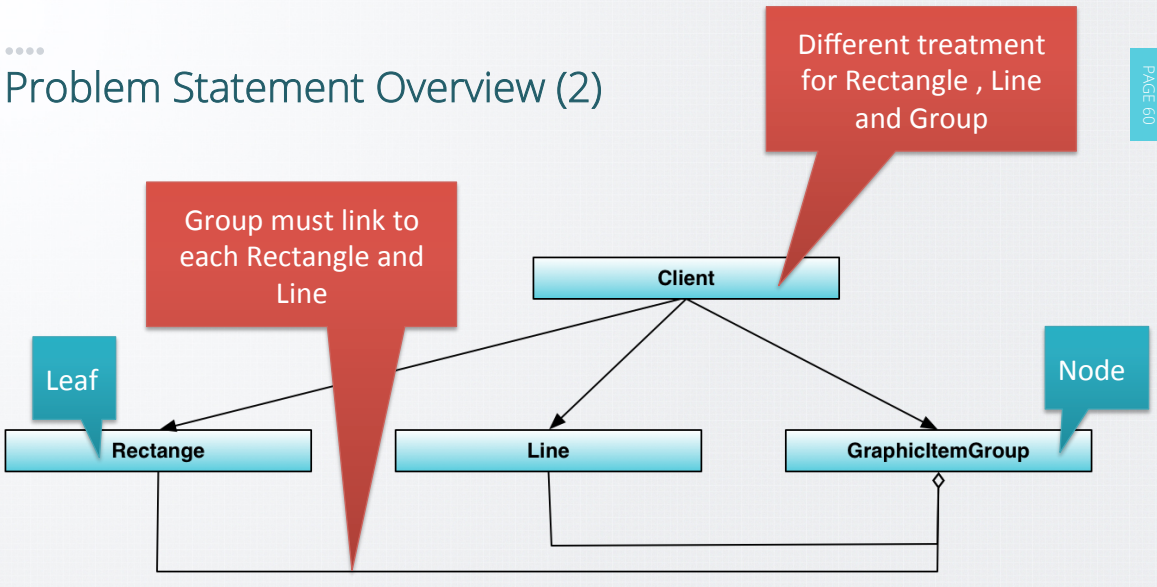
User interface data structure



PAGE 59

Elqoo www.elqoo.com

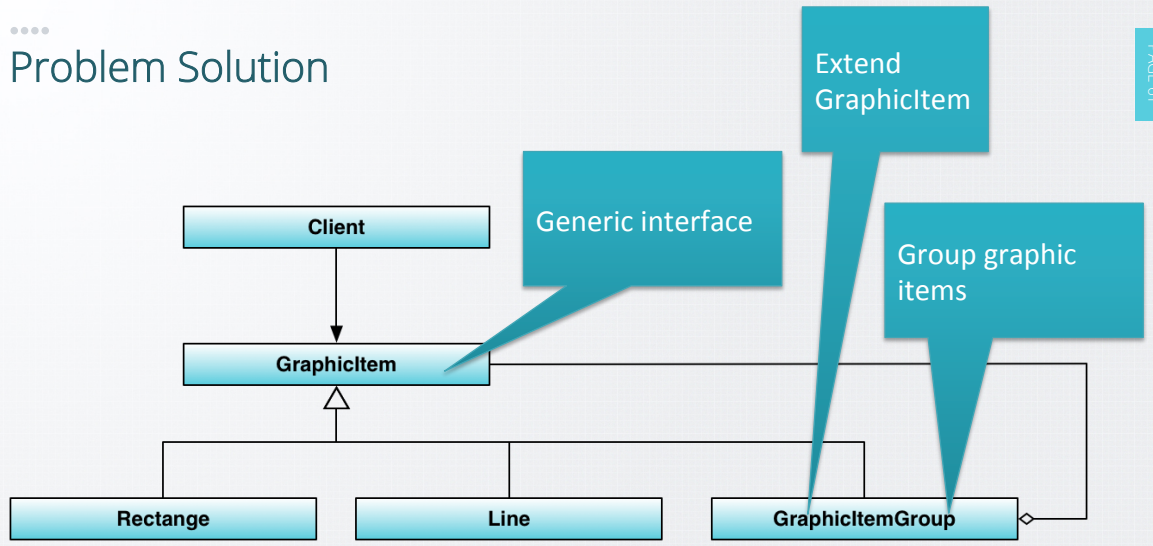
Problem Statement Overview (2)



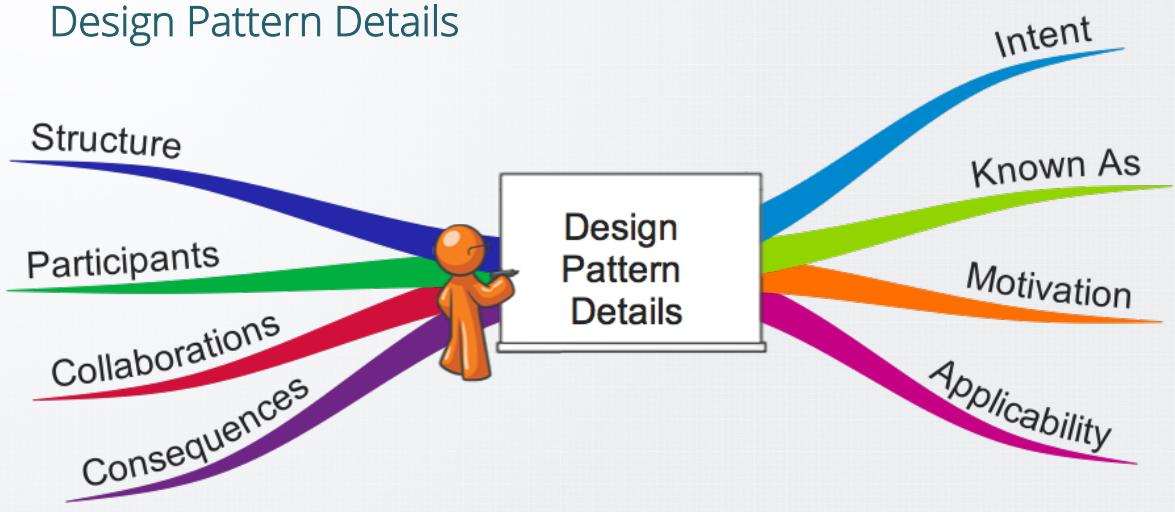
PAGE 60

Elqoo www.elqoo.com

Problem Solution



Design Pattern Details



Composite Pattern

Intent and known as

Intent

Compose **objects into tree structures** to represent part-whole hierarchies. Composite lets clients **treat individual objects** and compositions of objects **uniformly**

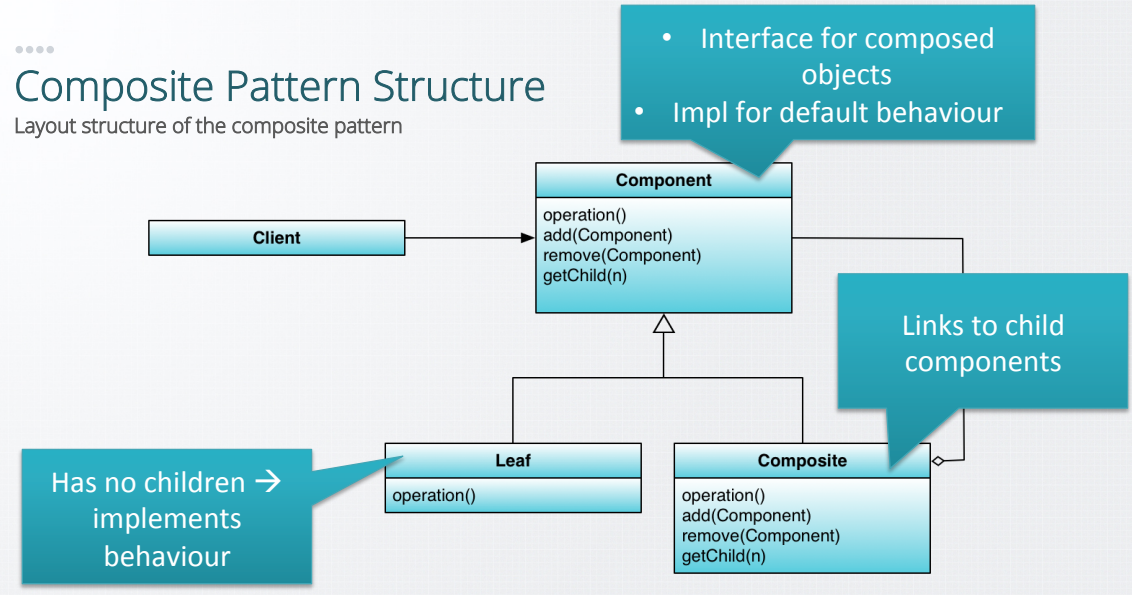
Apply Composite Pattern

Applicability

- **Use**
  - **Ignore differences** between **compositions** and **individual items**
  - **Represent** part-whole **hierarchies of objects**

Composite Pattern Structure

Layout structure of the composite pattern



Composite Pattern Consequences

Considerations for the composite pattern

- **Benefits**
  - **Define class hierarchies** of consistent objects
  - Client is simplified
    - **No distinction** between child and composite objects
  - Easy to add new components
- **Drawbacks**
  - Can be hard for new components with different expectations

Conclusion

- **Composite pattern is great**
  - **Represent** part-whole **hierarchies of objects**
  - **Abstraction** from client

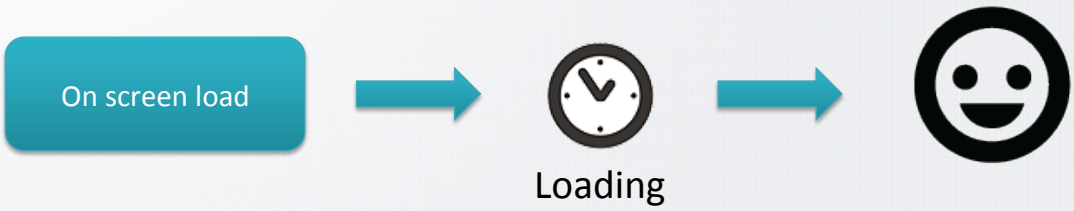
Problem Statement

Brad

- Hello Brad
- Hi Suzy
- We want to add on-demand loading to our graphics editor
- It might be a lot of work to do, but I'll look into it.

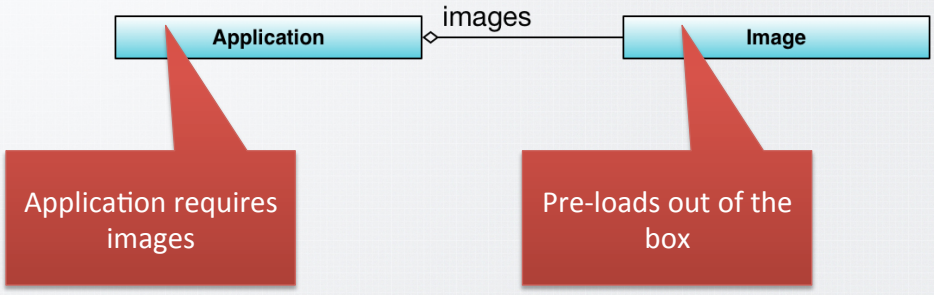
Suzy

Problem Statement Overview



Problem Statement Detail

Takes too much time to load all images out of the box



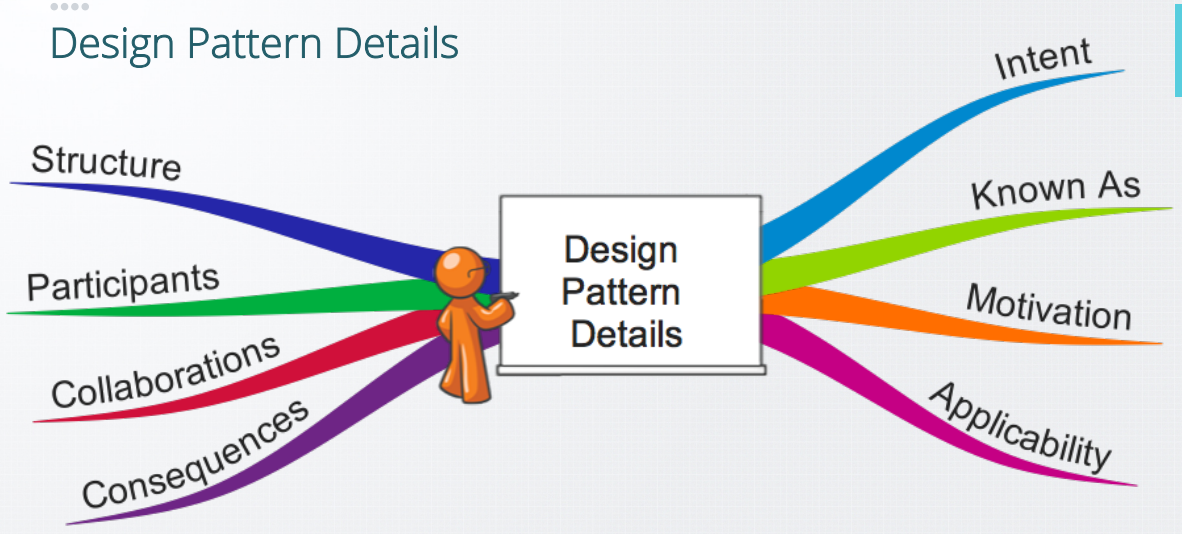
Application requires images

Pre-loads out of the box

Problem Solution

Loads image when drawing is requested

Design Pattern Details



Proxy Pattern

Intent and known as

Intent

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

Known As

Surrogate

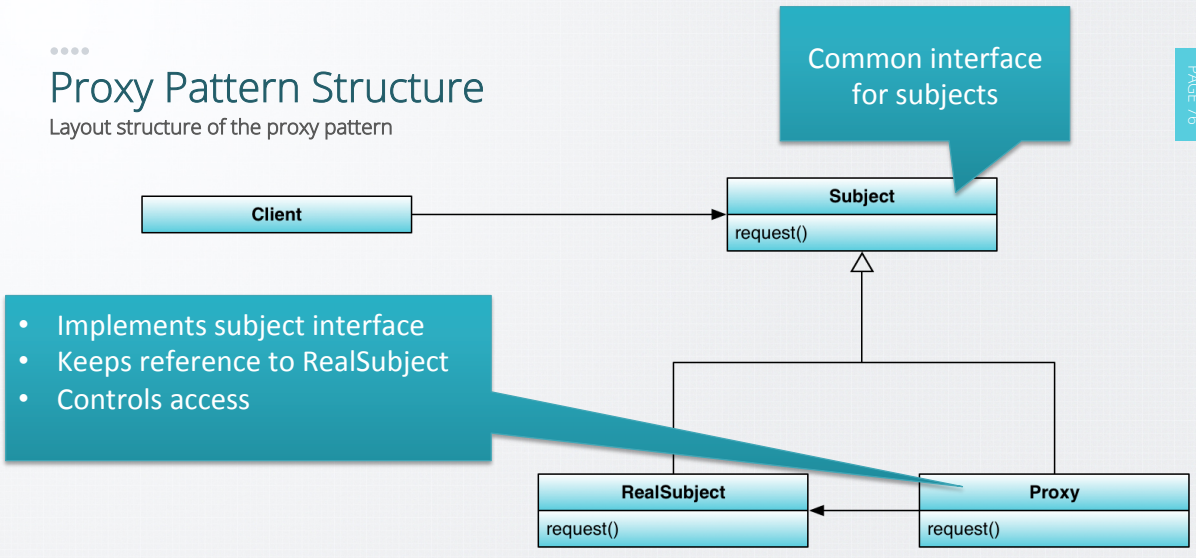
Apply Proxy Pattern

Applicability

- **Use**
  - Extra functionality is required
    - **Transparancy**
    - More than just a reference
- **Example**
  - Remote proxy (access network separated classes)
  - Virtual Proxy (create expensive objects on demand)
  - Protection proxy (access management)
  - Added functionality e.g. count nbr of references

Proxy Pattern Structure

Layout structure of the proxy pattern



Proxy Pattern Consequences

Considerations for the proxy pattern

- Adds **indirection**
- Can be beneficial in many cases
  - Remote proxy
  - Virtual proxy
  - Smart references

Conclusion

- **Proxy pattern is great**
  - **Transparently add functionality**

Problem Statement

Brad

- Hello Brad
- Hi Suzy
- Our text editor is running slow. Could you fix it.
- Sure

Suzy

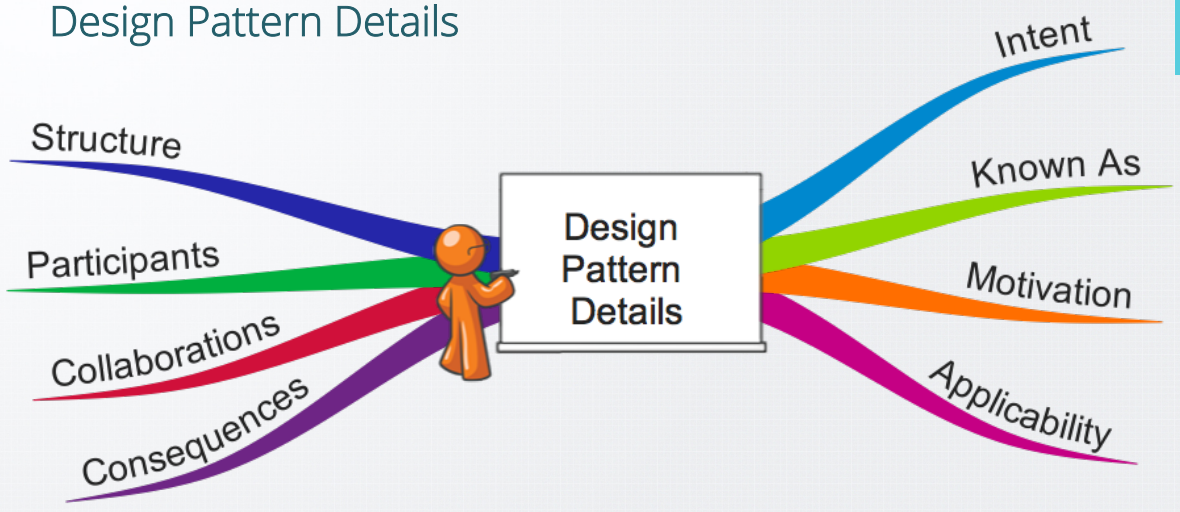
Problem Statement Overview

A B C D E F G

Each Letter = One object in memory

Serious Memory and performance overhead

Design Pattern Details



Flyweight Pattern

Intent and known as

Intent

Use Sharing to support large numbers of fine-grained objects efficiently.

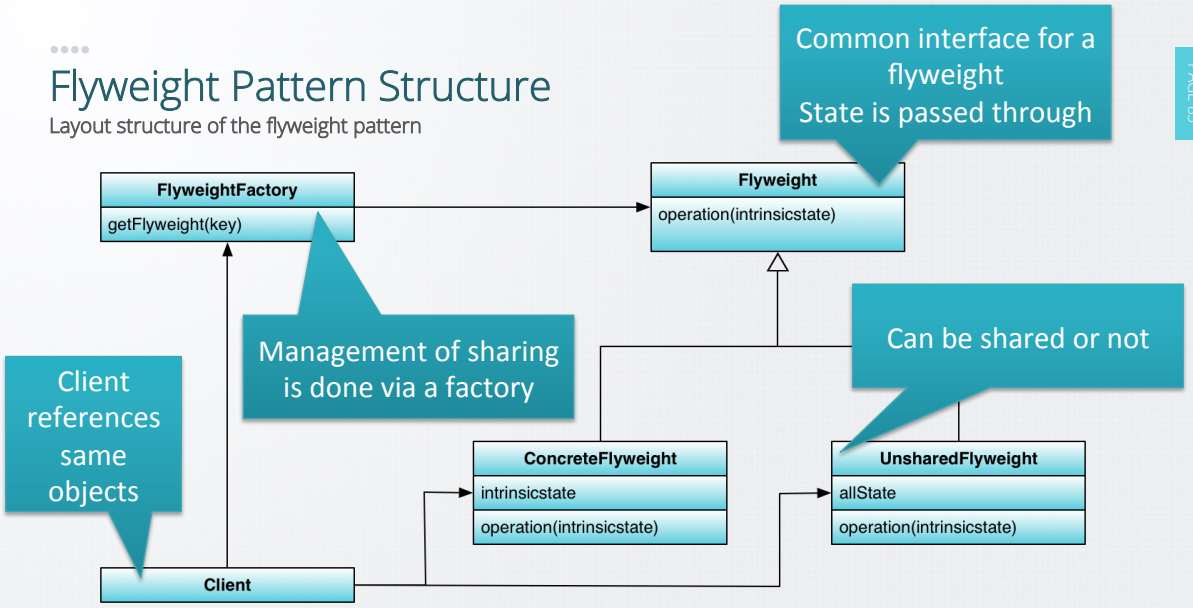
Apply Flyweight Pattern

Applicability

- **Use**
  - **Large number** of objects
  - High **storage costs**
  - Extrinsic state (**shareable**)
  - Many objects → replaced by few objects
  - Object identity isn't necessary

Flyweight Pattern Structure

Layout structure of the flyweight pattern



Flyweight Pattern Consequences

Considerations for the flyweight pattern

- **Benefits**
  - **Reduce** the total number of **instances**
  - Share intrinsic state per object
- **Drawbacks**
  - Factory needs to manage shared instances → run-time overhead

Use with Composite → Leaf nodes as flyweight

Conclusion

- **Flyweight pattern is great**
  - **Improve performance**
  - **Manage objects more efficiently**