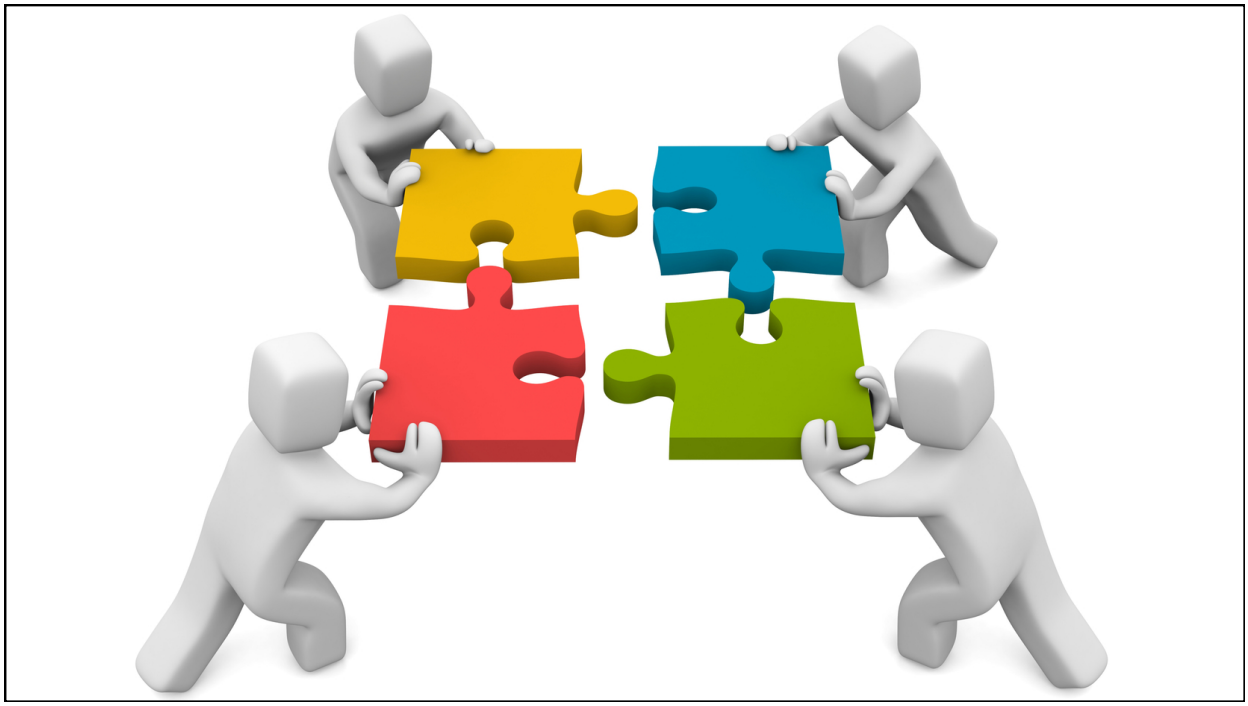
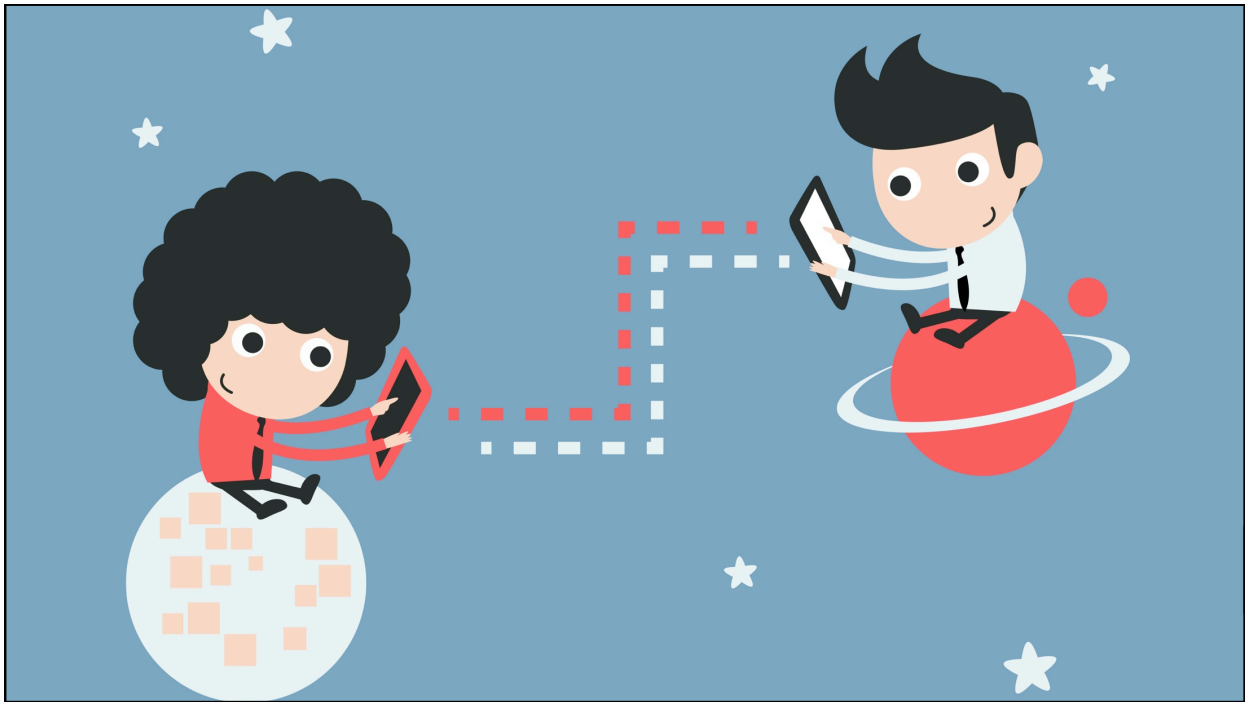

Behavioral Patterns
Part of the Design Patterns
www.elqoo.com

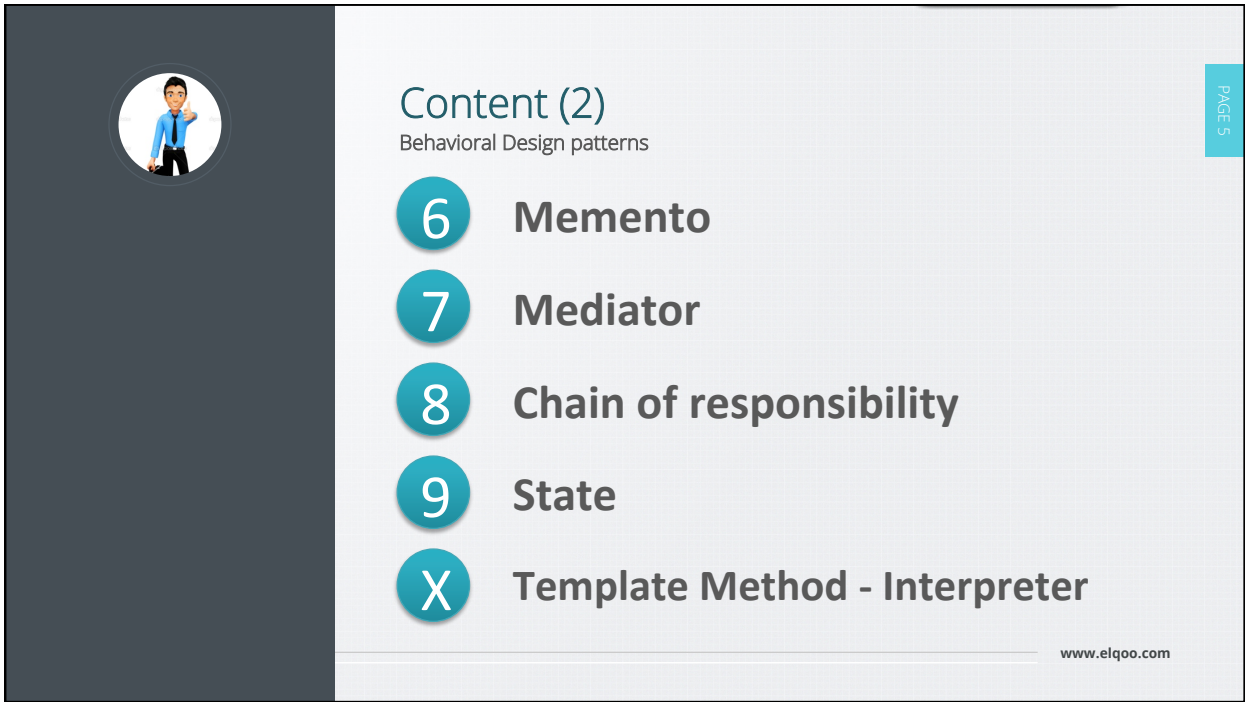




Content (1)

Behavioral Design patterns

1. Command
2. Strategy
3. Visitor
4. Observer
5. Iterator


Content (2)
Behavioral Design patterns
PAGE 5
6 Memento
7 Mediator
8 Chain of responsibility
9 State
X Template Method - Interpreter
www.elqoo.com


- Command Pattern–
Capture actions

Problem Statement

Brad

- Hello Brad
- Hi Suzy
- The end users would like to have an undo button in the application. Could you take care of it?
- Sure, no problem

Suzy

Problem Statement Overview (1)

Analysis of the undo button

Assumption: Text Editor

User can:

- Copy/Past
- Format Text
- Type Text
- Open Documents

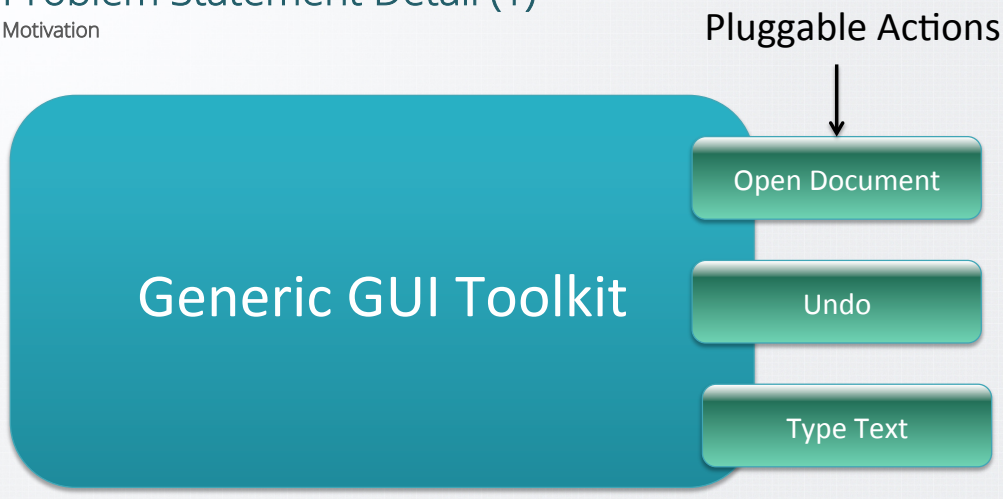
Problem Statement Overview (2)

Analysis of the undo button

- Where will we problem the undo button?
- We need to know
  - Which actions were last executed
  - How to undo those actions
- Problem
  - Each action is fundamentally different

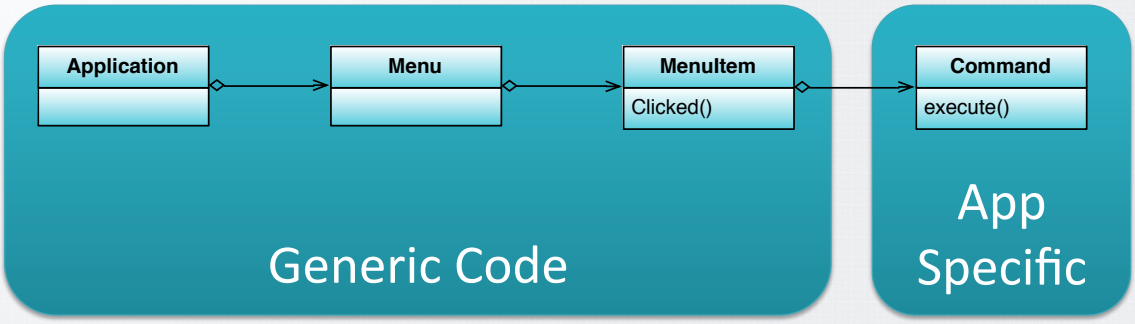
Problem Statement Detail (1)

Motivation



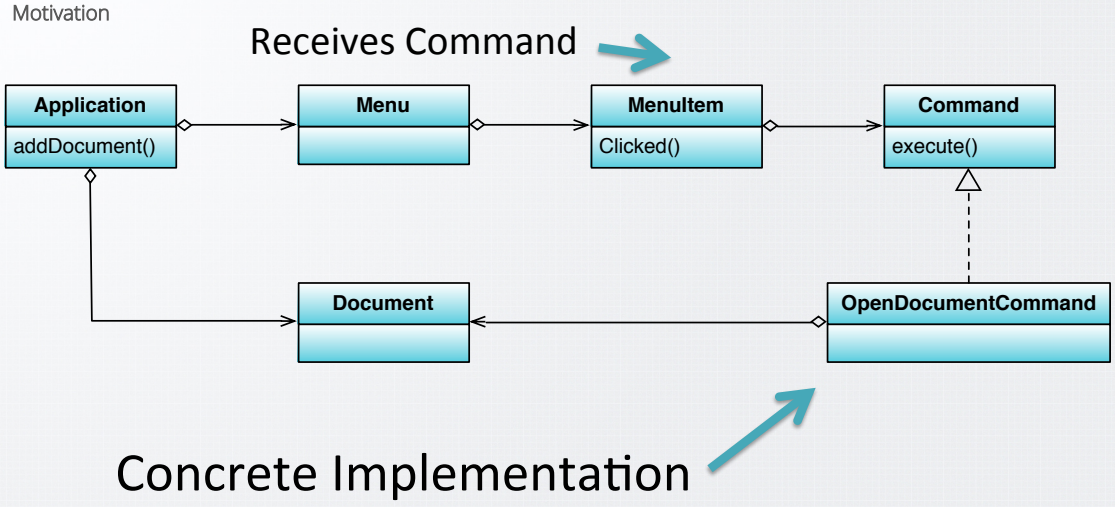
Problem Statement Detail (2)

Motivation



Problem Solution (1)

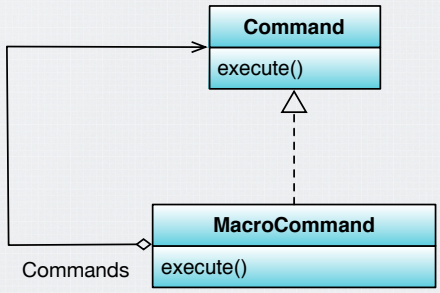
Motivation



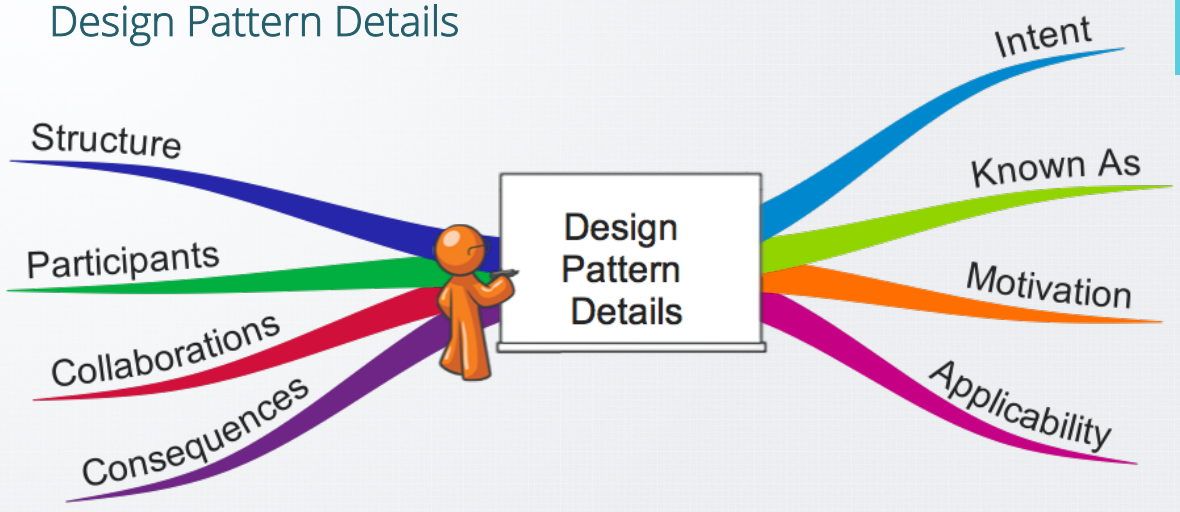
Problem Solution (2)

MacroCommand to execute multiple commands

MacroCommand
execute() → Call execute() for each command



Design Pattern Details



Command Pattern

Intent and Known As

Intent

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

Known As

Action, Transaction

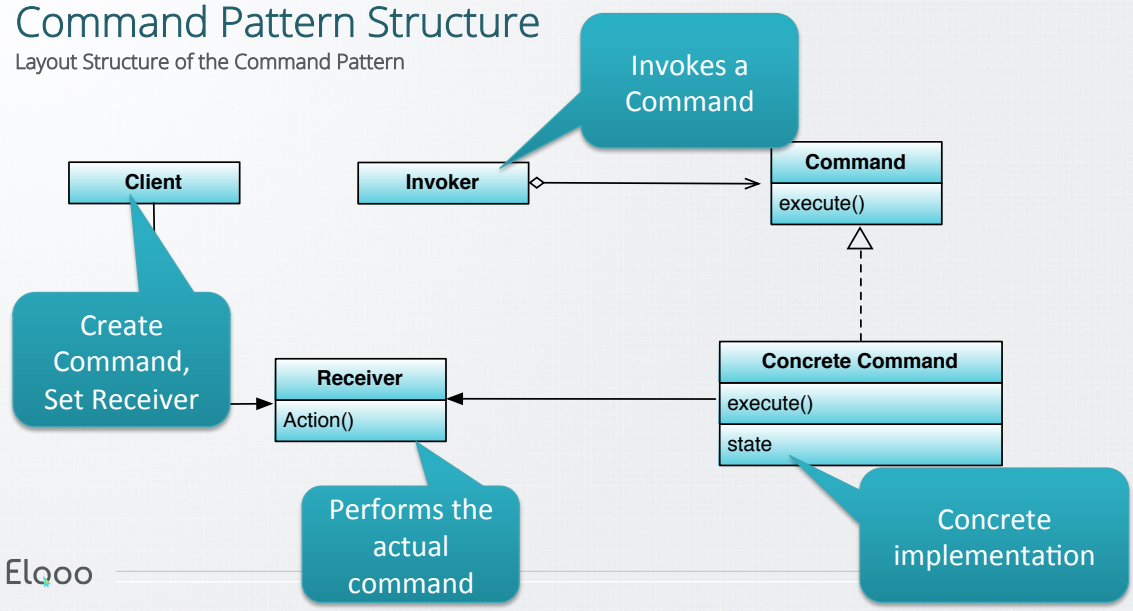
Apply Command Pattern

Applicability

- **Use**
  - Command as **parameter**
  - Pass command like general object
  - **Queue Request**
  - **Save** request state
  - Undo functionality
  - Provide an execute and undo method
  - Support Logging
  - **Re-execute** code in case of failure

Command Pattern Structure

Layout Structure of the Command Pattern



Elqoo

Command Pattern Consequences

- **Benefits**
  - **Decoupling** between
    - Object that invokes the operation
    - Object that knows how to perform it
  - Commands are **Objects**
    - All OO rules apply to them
  - Creating **new Commands** is easy
    - Just add a new class

Elqoo www.elqoo.com

Conclusion

- **Command pattern is great**
  - **Capture a request**
  - Centralize action functionality

Problem Statement

Brad

- Hello Brad
- Hi Suzy
- For our chess application: we need to be able to set the difficulty
- Sure not problem

Suzy

Problem Statement Overview

- **Multiple algorithms** to define to create an AI
- It needs to be **easily changed**
- No other code should be adapted in changing the algorithm

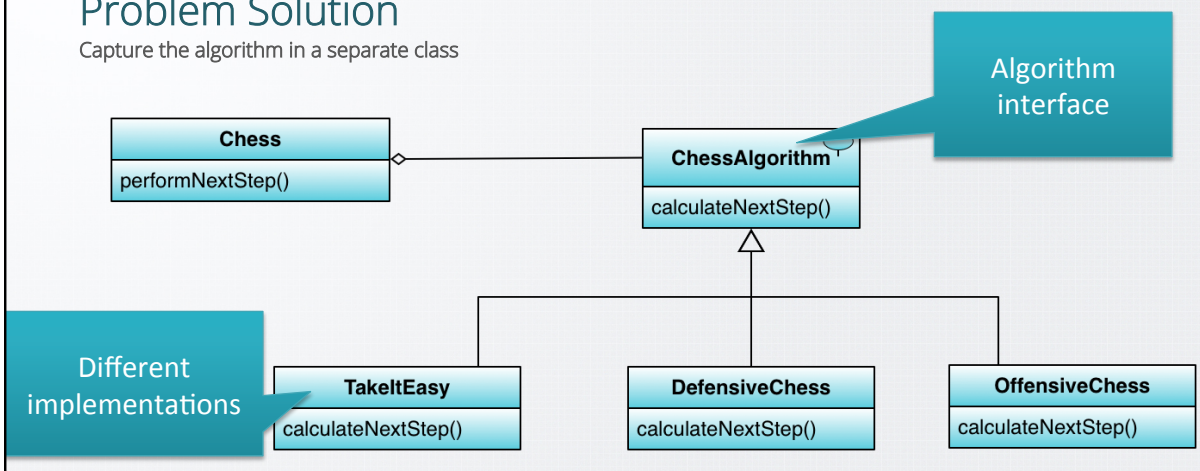
Problem Statement Detail

Motivation

- Programming **algorithms** into general **application classes**
  - More complex if algorithm grows
  - **Difficult** to **maintain**
  - Cannot switch between algorithms when needed

Problem Solution

Capture the algorithm in a separate class



Design Pattern Details

Strategy Pattern

Intent and Known As

Intent

Define a **family of algorithms**, encapsulate each one, and make them interchangeable. Strategy lets the **algorithm vary independently from clients** that use it.

Known As

Policy

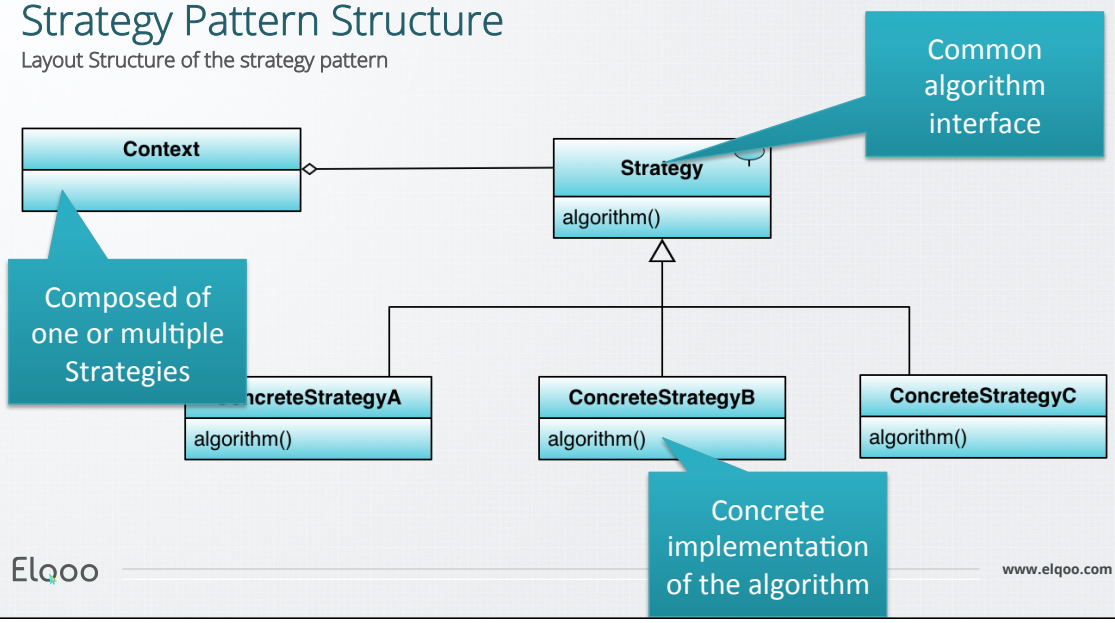
Apply Strategy Pattern

Applicability

- **Use**
  - Classes only change in behavior
  - Different variants of an **algorithm**
  - Algorithms that **use complex data** that clients shouldn't be aware of

Strategy Pattern Structure

Layout Structure of the strategy pattern



Strategy Pattern Collaboration

Collaboration between objects

- Context can **pass necessary data** to strategy
- Clients of the context can **pass a strategy**
- Context **forwards clients** requests to the strategy

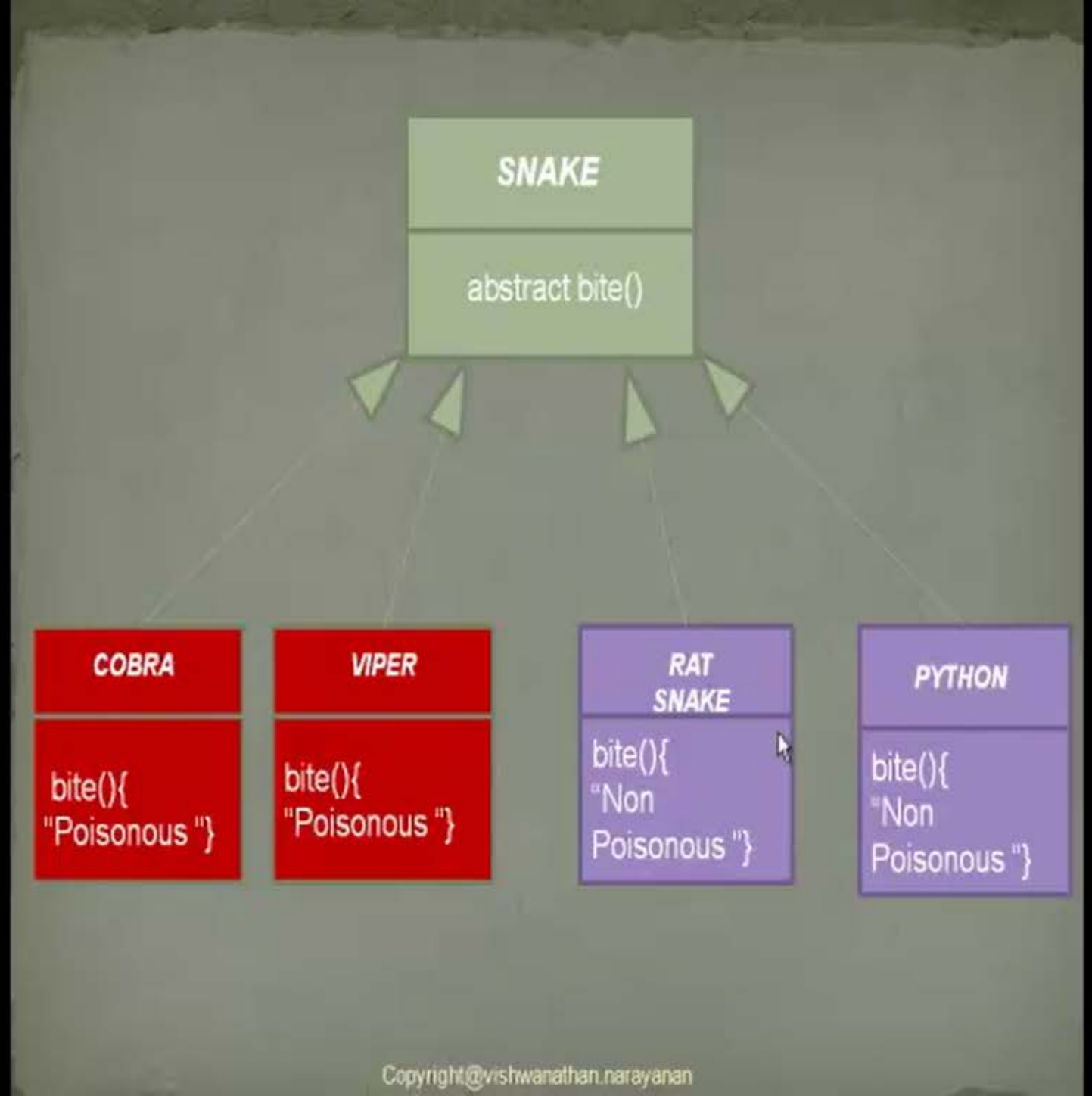
Builder Pattern Consequences

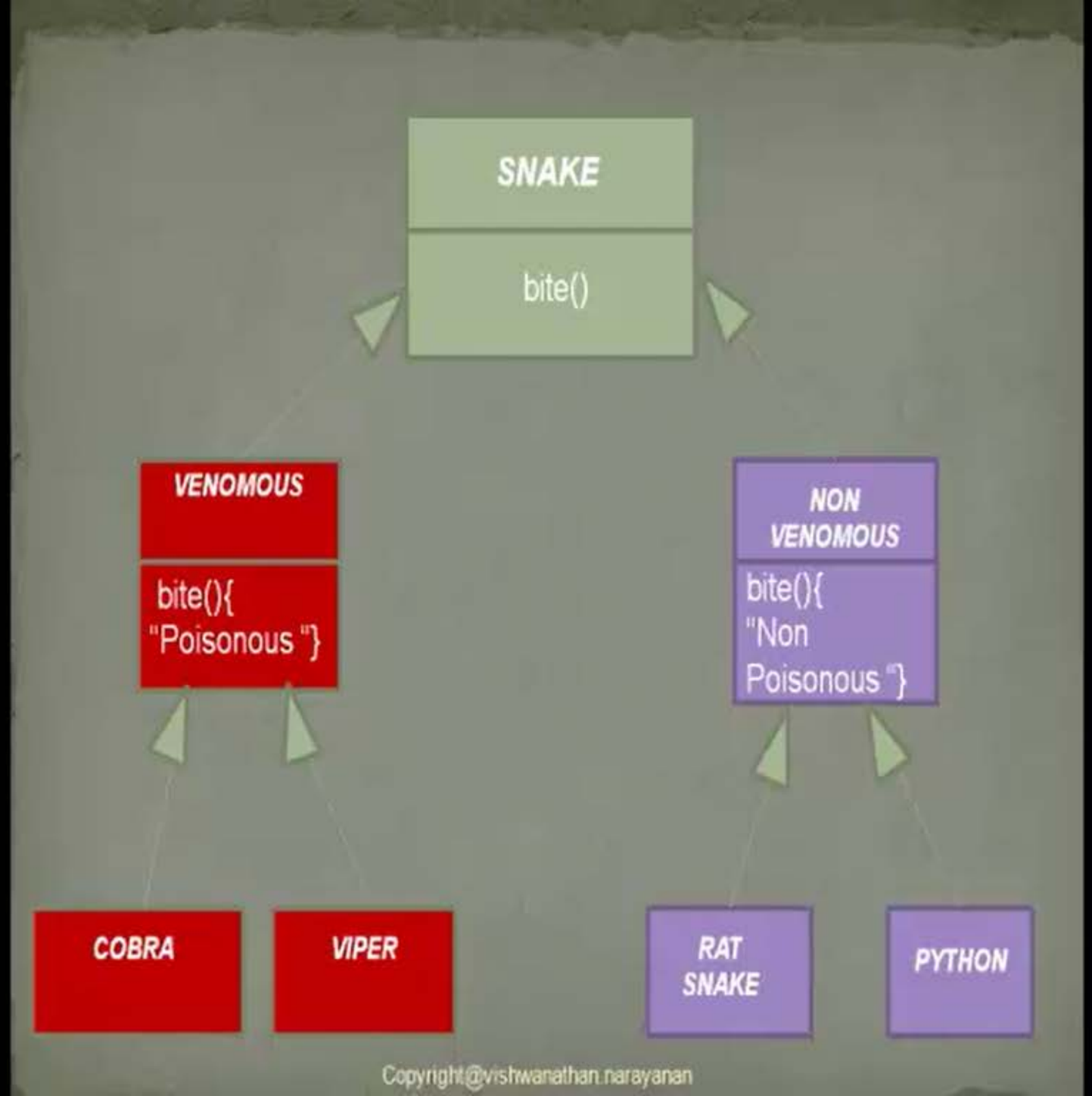
Benefits of the strategy pattern

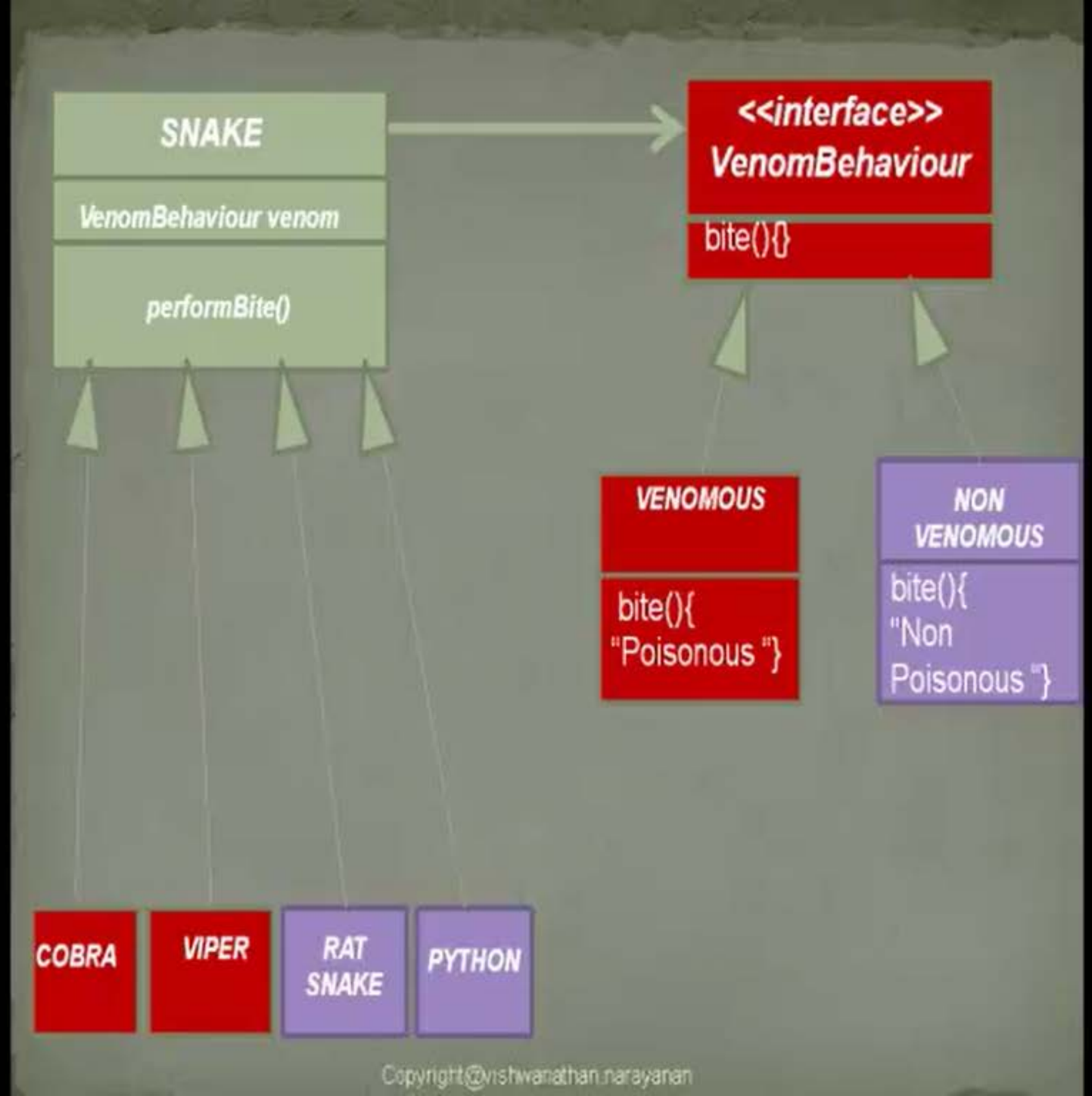
- **Benefits**
  - Algorithm families use inheritance for common parts
  - Avoid conditional statements using this pattern
  - **Clients** can **choose** the required **behavior**
- **Drawbacks**
  - Pattern increases nr of objects in application
    - Share strategies between objects.

Problem statement

- We are going to design a s/w solution for Indian Snake park
- Indian snake park is a sanctuary/research center for various types of snakes
- The authority wants to maintain various details especially pertaining to snake bites
- Also they want to track snakes turning from poisonous to non poisonous and vice versa
- For the sake of simplicity we will consider only 4 snakes COBRA, VIPER, RAT SNAKE & PYTHON


SNAKE
abstract bite()
COBRA
bite(){
"Poisonous "}
VIPER
bite(){
"Poisonous "}
RAT SNAKE
bite(){
"Non Poisonous "}
PYTHON
bite(){
"Non Poisonous "}
Copyright@vishwanathan.narayanan



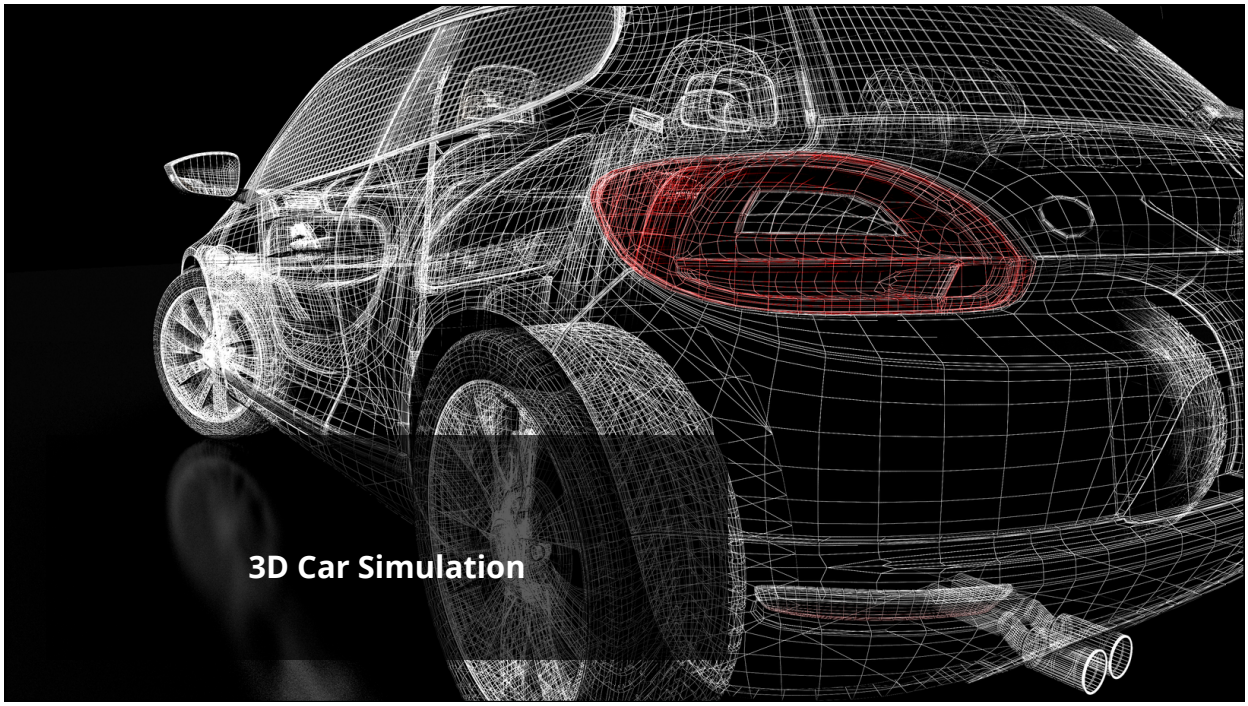

SNAKE
VenomBehaviour venom
performBite()
<<interface>>
VenomBehaviour
bite(){}
VENOMOUS
bite(){
"Poisonous "}
NON
VENOMOUS
bite(){
"Non
Poisonous "}
COBRA
VIPER
RAT
SNAKE
PYTHON
Copyright@vishwanathan.narayanan

Underlying principles of strategy design pattern

- Separate dynamic part from static part
- Prefer composition over inheritance
- Code to interface and not implementation

Conclusion

- **Strategy pattern is great**
  - **Encapsulate an algorithm**
  - Change algorithm at run-time



Problem Statement

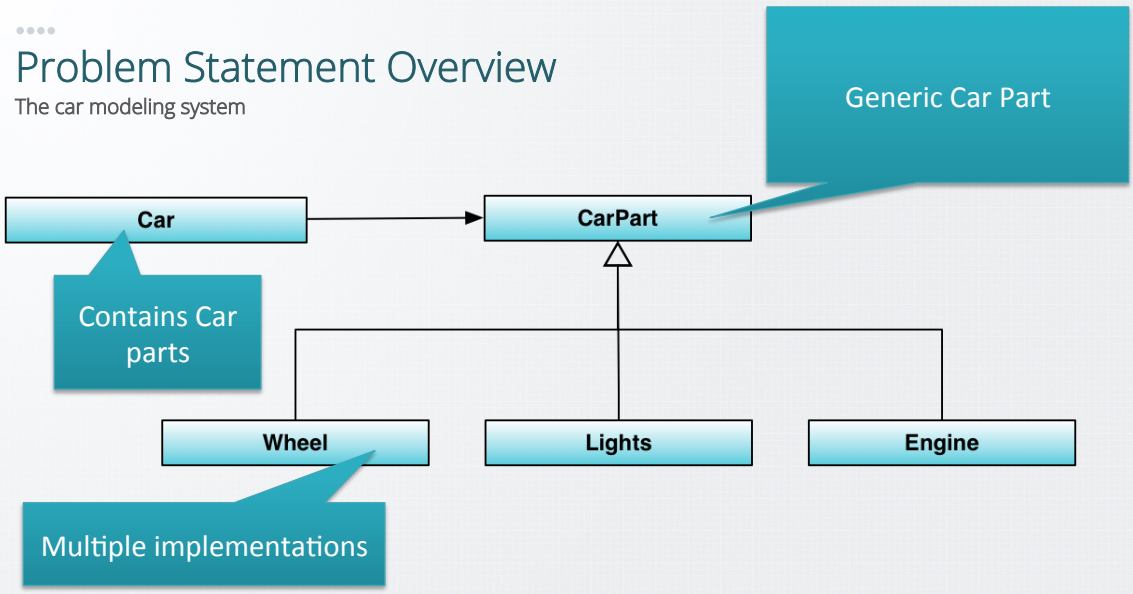
Brad

- Hello Brad
- Hi Suzy
- We want to print and render cars in our 3D simulation program
- Ok

Suzy

Problem Statement Overview

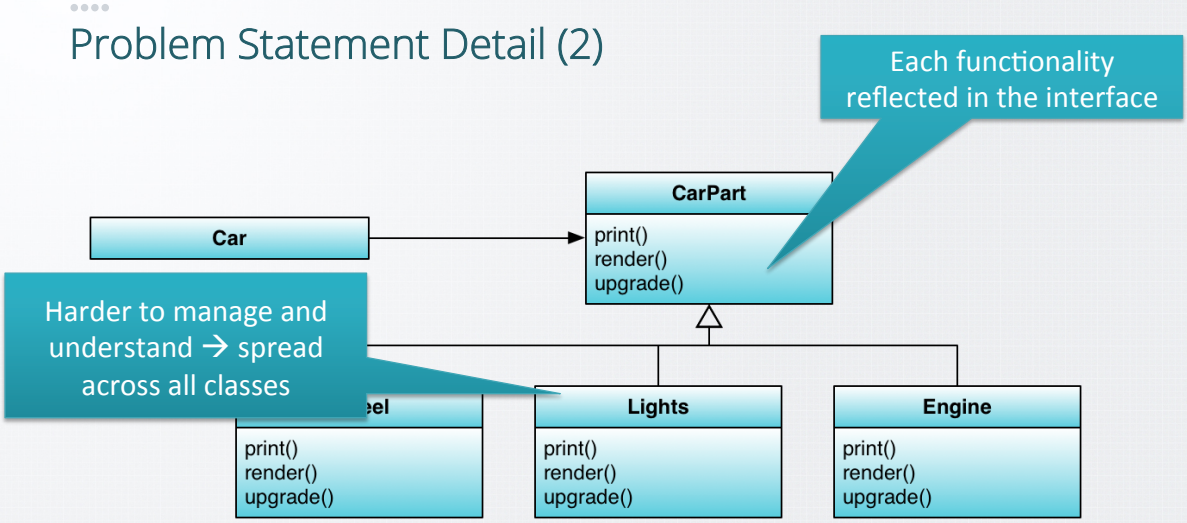
The car modeling system



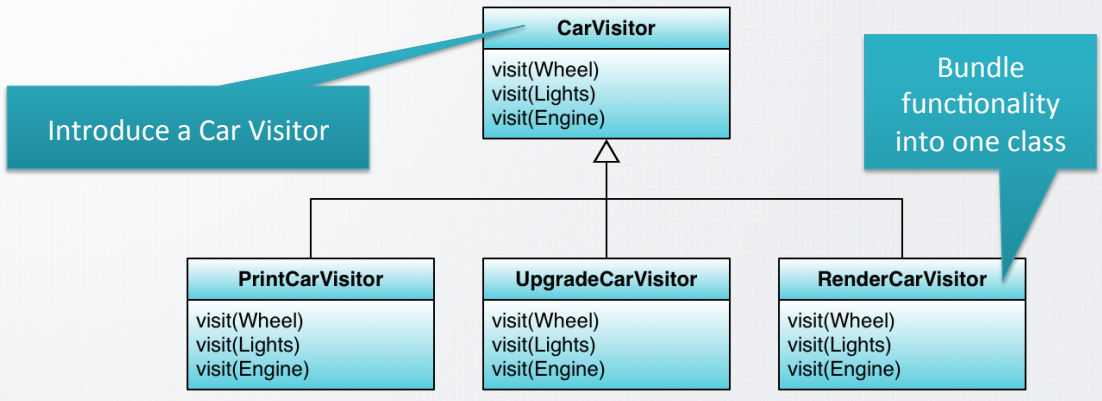
Problem Statement Detail (1)

- We want to
  - **Print** car details
  - **Render** car details
  - **Upgrade** car parts

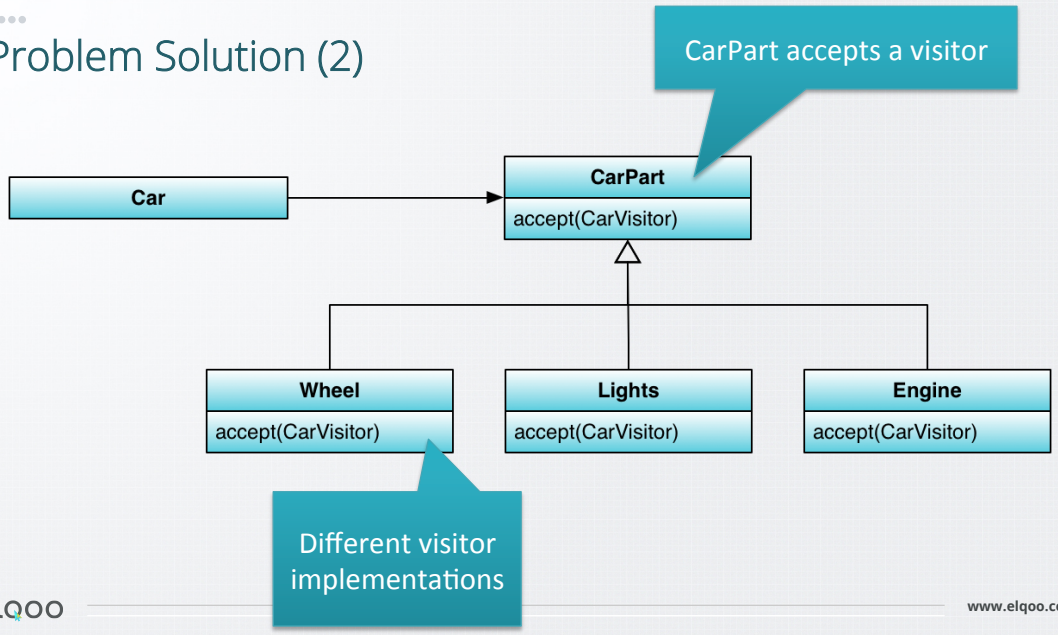
Problem Statement Detail (2)



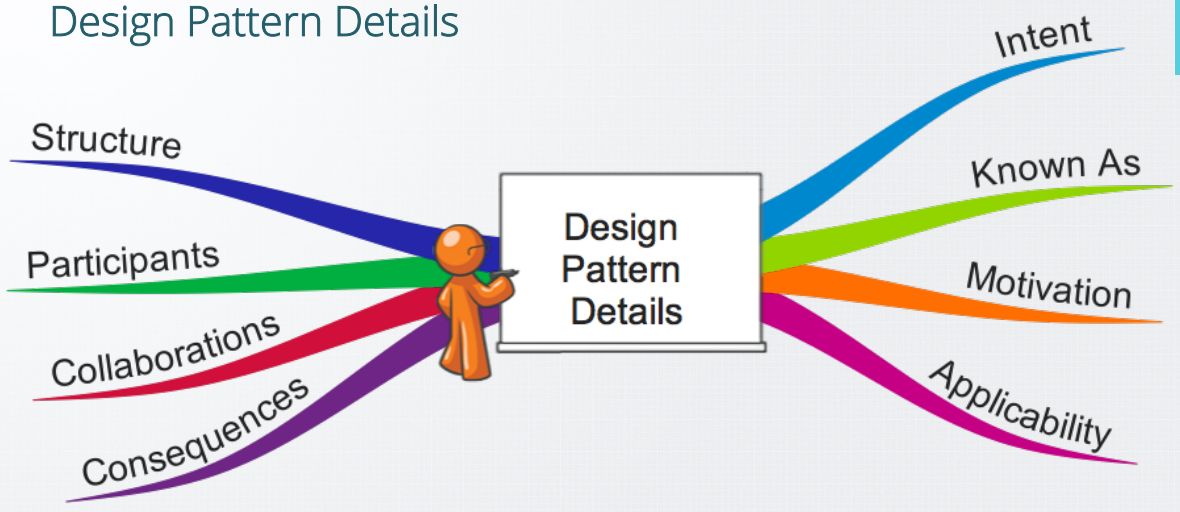
Problem Solution (1)



Problem Solution (2)



Design Pattern Details



Visitor Pattern

Intent and Known As

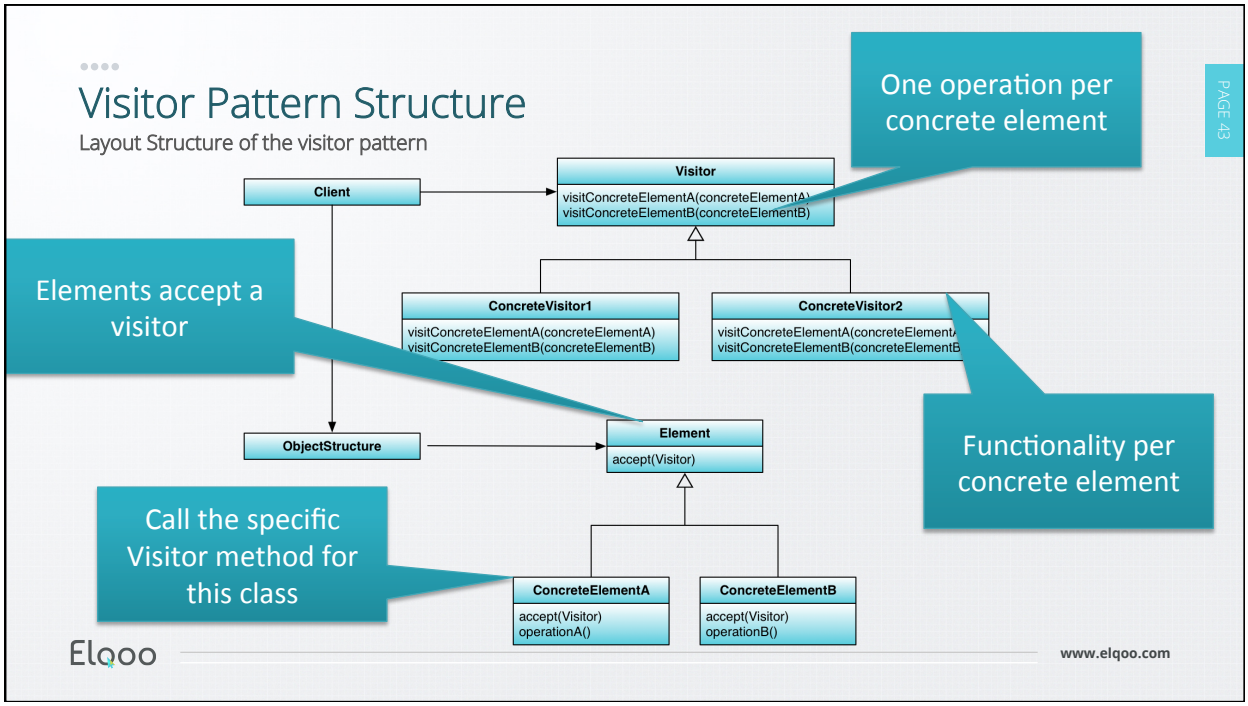
Intent

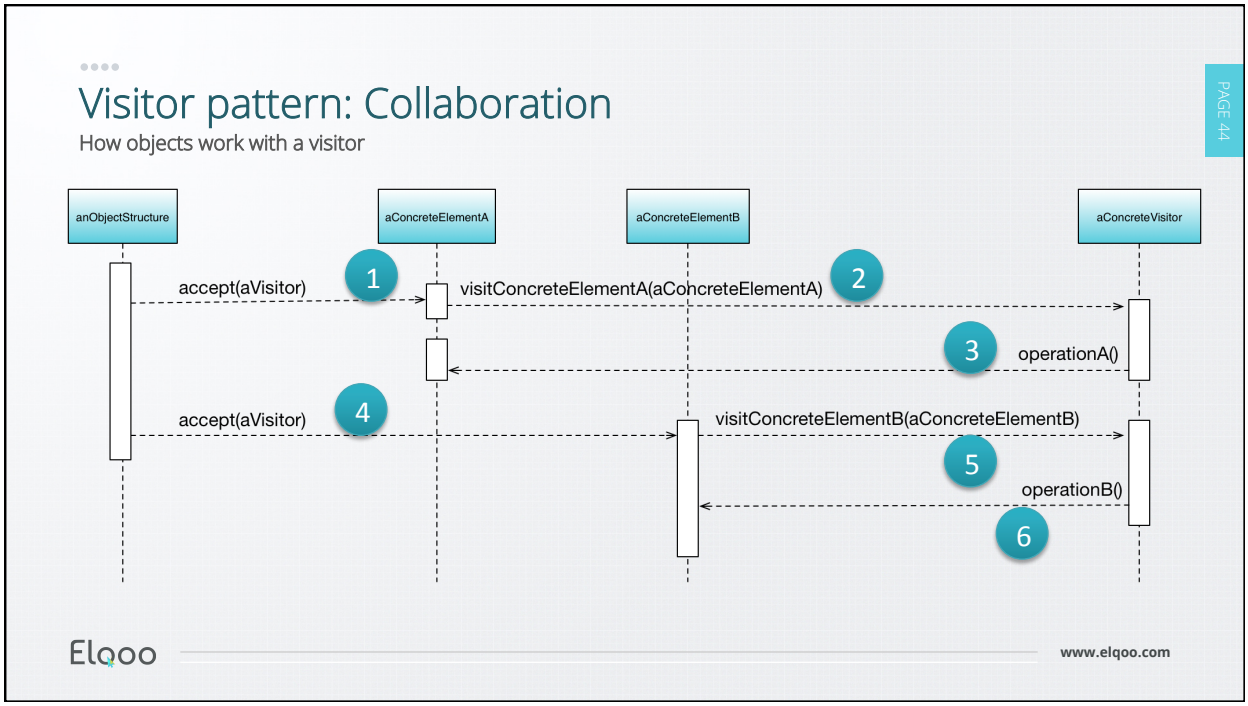
Represent an **operation** to be **performed** on the **elements** of **an object structure**. Visitor lets you **define a new operation** without changing the classes of the elements on which it operates.

Apply Visitor Pattern

Applicability

- **Use**
  - **Visit complex object structure** (inheritance)
    - Perform operations based upon concrete classes
  - **Avoid pollution** of concrete classes with many different operations
    - Visitor groups this functionality
  - Ability **to easily define new operations** without changing concrete classes.


Visitor Pattern Structure
Layout Structure of the visitor pattern
PAGE 43
One operation per concrete element
Client
Visitor
visitConcreteElementA(concreteElementA)
visitConcreteElementB(concreteElementB)
Elements accept a visitor
ConcreteVisitor1
visitConcreteElementA(concreteElementA)
visitConcreteElementB(concreteElementB)
ConcreteVisitor2
visitConcreteElementA(concreteElement
visitConcreteElementB(concreteElementB
Functionality per concrete element
ObjectStructure
Element
accept(Visitor)
Call the specific Visitor method for this class
ConcreteElementA
accept(Visitor)
operationA()
ConcreteElementB
accept(Visitor)
operationB()
Elqoo
www.elqoo.com


Visitor pattern: Collaboration
How objects work with a visitor
PAGE 44
anObjectStructure
aConcreteElementA
aConcreteElementB
aConcreteVisitor
accept(aVisitor)
1
visitConcreteElementA(aConcreteElementA)
2
3
operationA()
accept(aVisitor)
4
visitConcreteElementB(aConcreteElementB)
5
operationB()
6
Elqoo
www.elqoo.com

Visitor Pattern Consequences

Considerations for the Visitor pattern

- **Benefits**
  - Adding **new operations** is **easy**
  - Visitor separates operations that don't belong together
  - **Accumulate state**
    - Visitors can maintain state across the hierarchy
- **Drawbacks**
  - **Adding** new **concrete elements** is **hard**
    - Requires a new method for all concrete visitors
  - Visitors **break encapsulation**
    - They rely on the interface of the concrete element

Conclusion

- **Visitor pattern is great**
  - **Visit complex object structure**
  - Centralize logic

Problem Statement

Brad

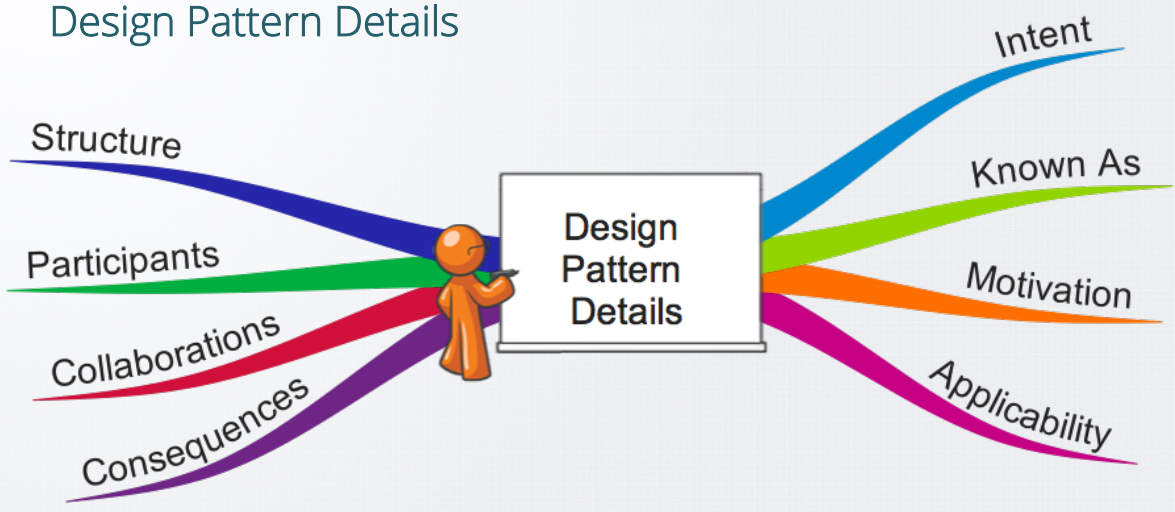
- Hello Brad
- Hi Suzy
- When our data changes, we want to automatically change the graphical representations in our business application
- No problem

Suzy

Problem Statement Overview

A = 50%
B= 25%
C = 25%

Design Pattern Details



Observer Pattern

Intent and Known As

Intent

Define a **one-to-many dependency** between objects so that when one object changes state, **all** its **dependents** are **notified** and updated automatically

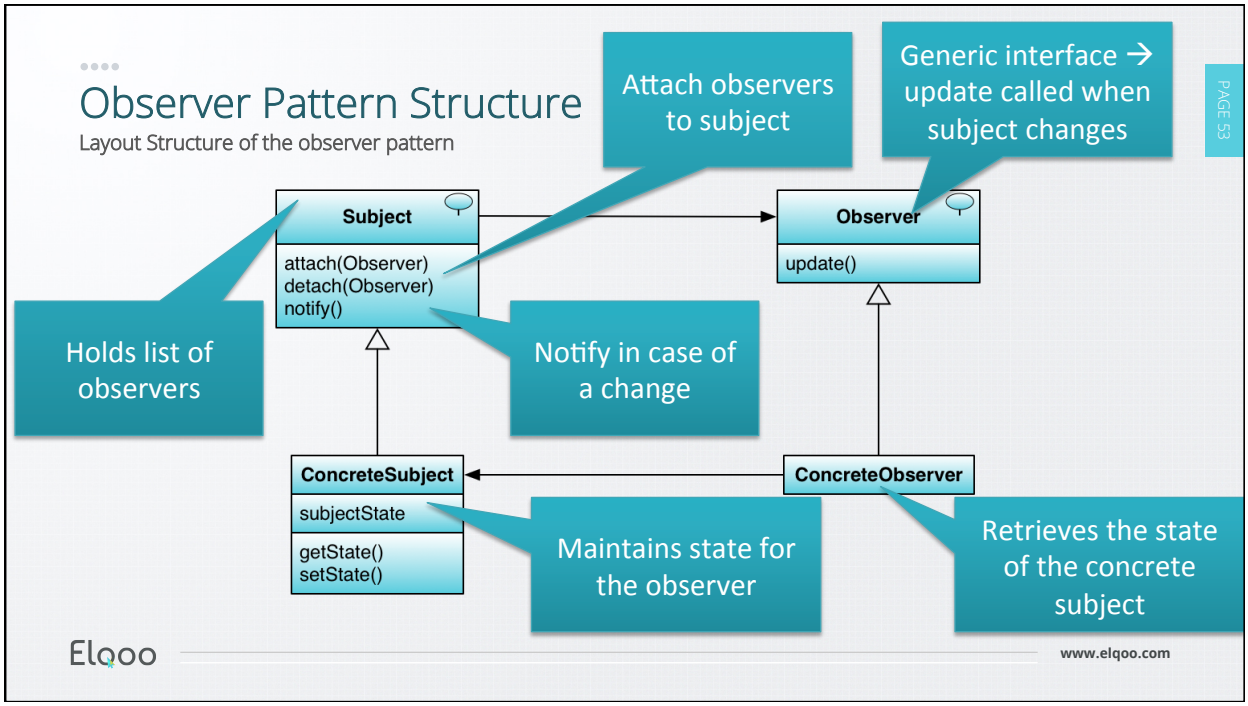
Known As

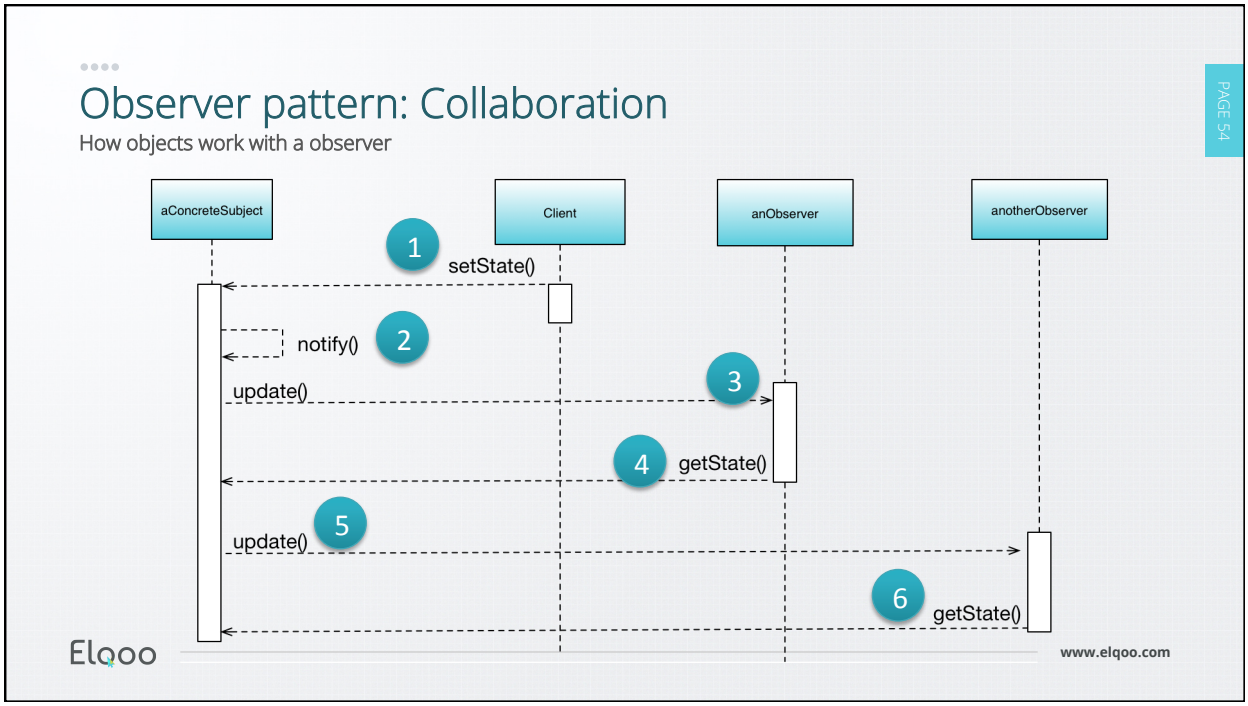
Dependents, Publish-Subscribe

Apply Observer Pattern

Applicability

- **Use**
  - **Change one** object → **changes others**
  - No idea how many objects need to be changed
  - Object change notification
    - With preserving loose coupling
  - One object may notify another without knowing them directly


Observer Pattern Structure
Layout Structure of the observer pattern
PAGE 53
Attach observers to subject
Generic interface → update called when subject changes
Subject
attach(Observer)
detach(Observer)
notify()
Observer
update()
Holds list of observers
Notify in case of a change
ConcreteSubject
subjectState
getState()
setState()
ConcreteObserver
Maintains state for the observer
Retrieves the state of the concrete subject
Elqoo
www.elqoo.com


Observer pattern: Collaboration
How objects work with a observer
PAGE 54
aConcreteSubject
Client
anObserver
anotherObserver
1
setState()
notify()
2
update()
3
4
getState()
5
update()
6
getState()
Elqoo
www.elqoo.com

Observer Pattern Consequences

Considerations for the observer pattern

- **Benefits**
  - **Loose coupling** between observers and subjects
  - Supporting a **broadcast model**
- **Drawbacks**
  - One change can result in multiple **unnecessary notifications**
  - Clients don't know the **ripple effects**

Conclusion

- **Observer pattern is great**
  - **Observe state in other objects**
  - Preserve loose coupling

Problem Statement

Brad

- Hello Brad
- Hi Suzy
- Some of the new developers need help with traversing through some data.
- Ok

Suzy

Problem Statement Overview

How to iterate through a collection without knowing how it's implemented

Interface Collection<E>

Type Parameters:

E - the type of elements in this collection

All Superinterfaces:

Iterable<E>

All Known Subinterfaces:

BeanContext, BeanContextServices, BlockingDeque<E>, BlockingQueue<E>, Deque<E>, List<E>, NavigableSet<E>, Queue<E>, Set<E>, SortedSet<E>, TransferQueue<E>

All Known Implementing Classes:

AbstractCollection, AbstractList, AbstractQueue, AbstractSequentialList, AbstractSet, ArrayBlockingQueue, ArrayDeque, ArrayList, AttributeList, BeanContextServicesSupport, BeanContextSupport, ConcurrentLinkedDeque, ConcurrentLinkedQueue, ConcurrentSkipListSet, CopyOnWriteArrayList, CopyOnWriteArraySet, DelayQueue, EnumSet, HashSet, JobStateReasons, LinkedBlockingDeque, LinkedBlockingQueue, LinkedHashSet, LinkedList, LinkedTransferQueue, PriorityBlockingQueue, PriorityQueue, RoleList, RoleUnresolvedList, Stack, SynchronousQueue, TreeSet, Vector

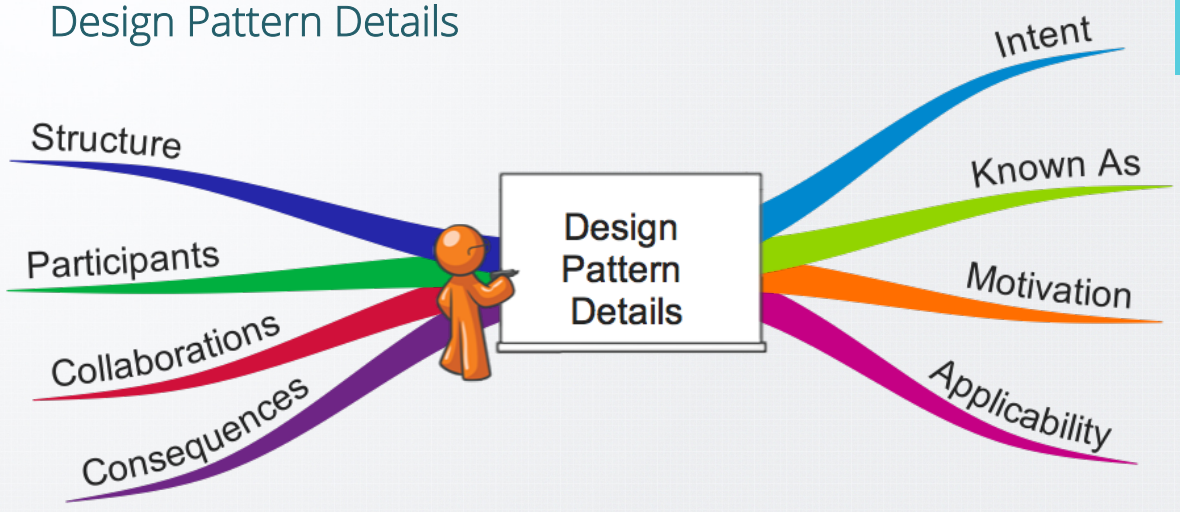
Expand the Collection interface with traversal methods?

What if we need a new iteration method e.g. back to forth

Different implementations of a Collection

Elqoo www.elqoo.com

Design Pattern Details



Elqoo www.elqoo.com

Iterator Pattern

Intent and Known As

Intent

Provide a way to **access the elements of an aggregate object** sequentially **without exposing** its underlying representation.

Known As

Cursor

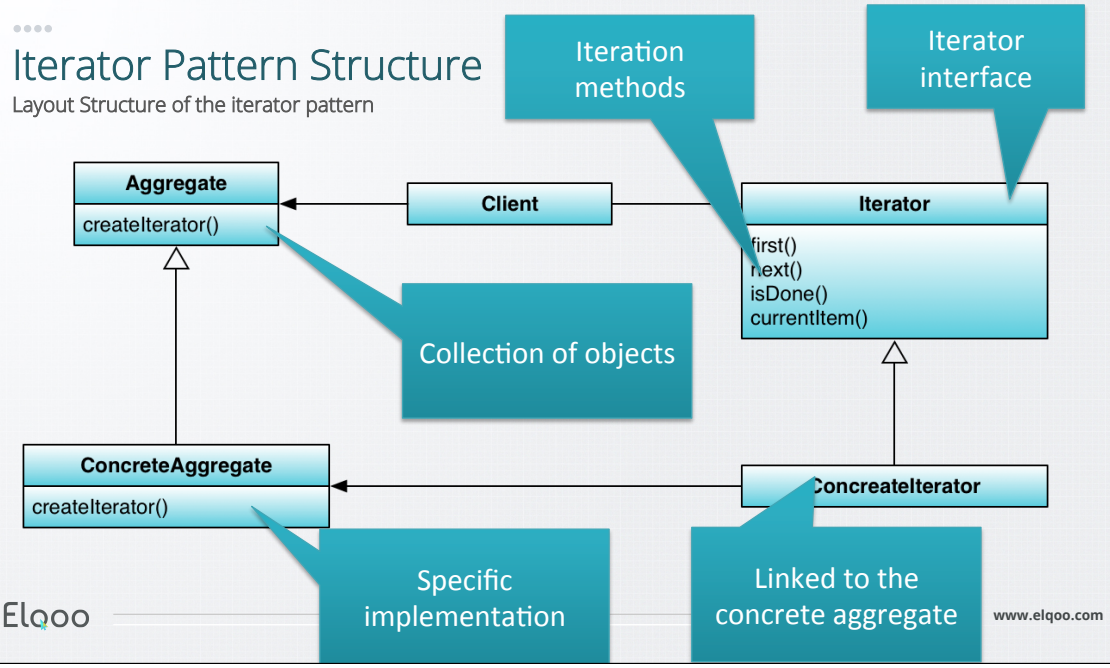
Apply Iterator Pattern

Applicability

- **Use**
  - **Access aggregated object's** contents **without exposing** its representation
  - Support multiple traversal of aggregated objects
  - Provide **uniform interface** to
    - Traverse aggregated objects
    - Might be of different classes

Iterator Pattern Structure

Layout Structure of the iterator pattern



Iterator Pattern Collaboration

- Iterator **knows representation** of aggregate
  - Keeps track of current object
  - **Knows** how to **move forward/backwards**

Iterator Pattern Consequences

Considerations for the Iterator pattern

- **Benefits**
  - **Multiple iteration** variations are possible
  - Aggregate or Collection **interface is simplified**
  - Each iterator keeps track where he is
    - Use **different iterators** to traverse an object **at same time**

Conclusion

- **Iterator pattern is great**
  - **Traverse** aggregated **objects**
  - **Provide unified interface**

Problem Statement

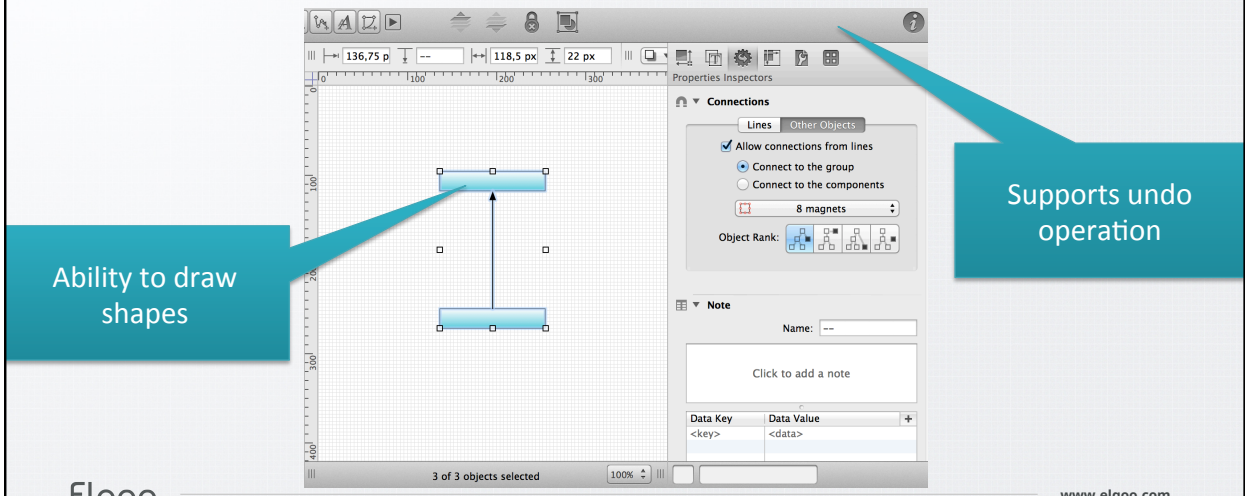
Brad

- Hello Brad
- Hi Suzy
- We want to support undo operations in our graphical drawing tool
- Sounds easy

Suzy

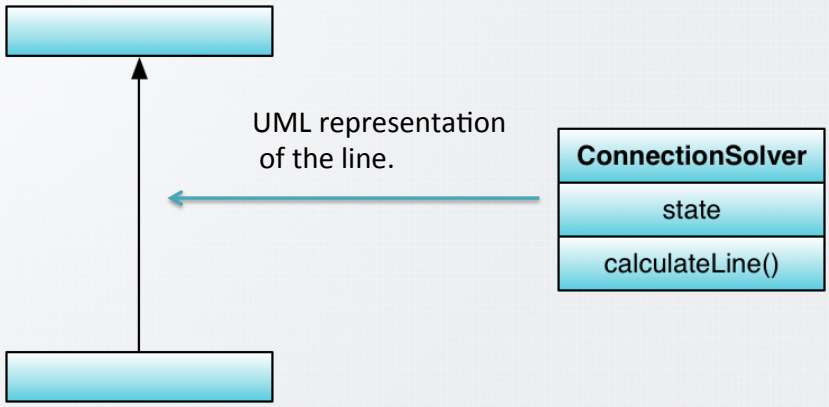
Problem Statement Overview

Example of a graphical drawing tool



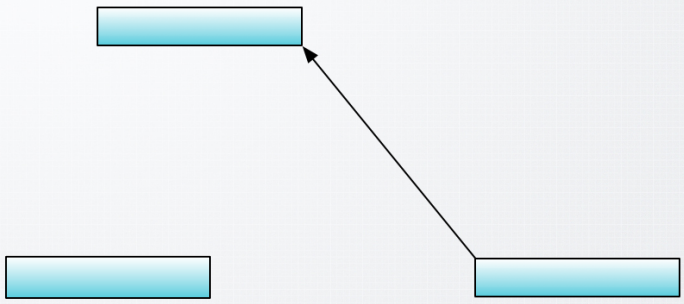
Problem Statement Detail (1)

Implementing line connectivity



Problem Statement Detail (2)

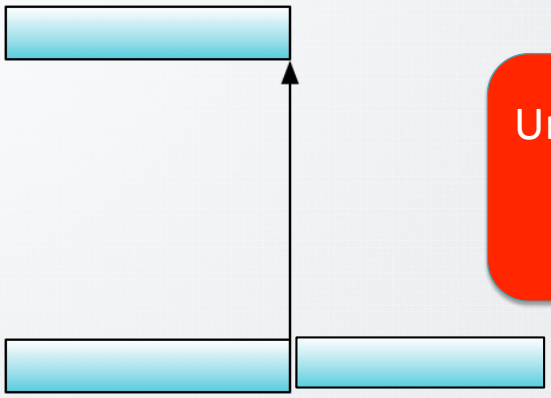
How would the undo functionality work?



Move Rectangle

Undo Move?

Problem Statement Detail (3)



Undo Move FAIL → We lost original information

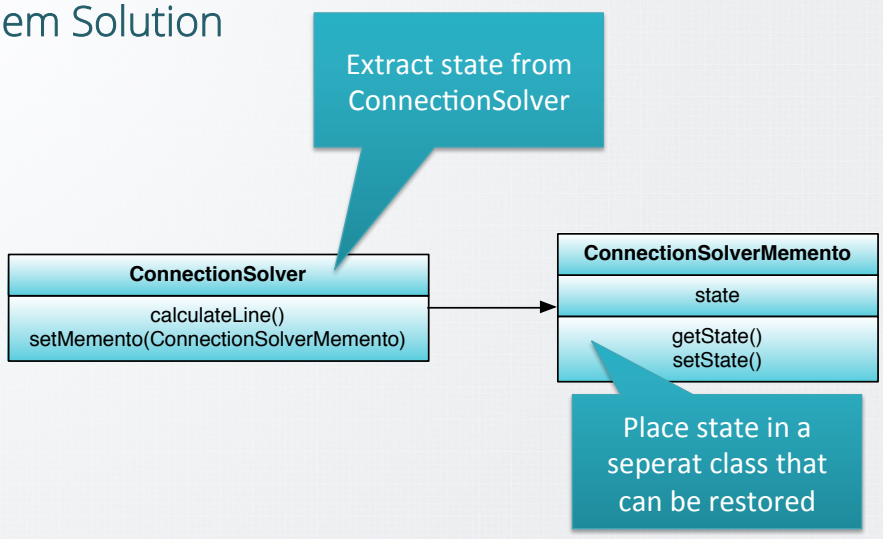
Problem Solution

Solution to the undo graphical drawing tool problem

- We need to **restore the original state** of the ConnectionSolver
  - Without breaking encapsulation

1. Memento pattern: **Save state** ConnectionSolver
2. Store memento state for later use
3. **Restore ConnectionSolver state** on undo

Problem Solution



Design Pattern Details

Memento Pattern

Intent and Known As

Intent

Without violating encapsulation, **capture and externalize an object's internal state** so that the object can be **restored** to this state later.

Known As

Token

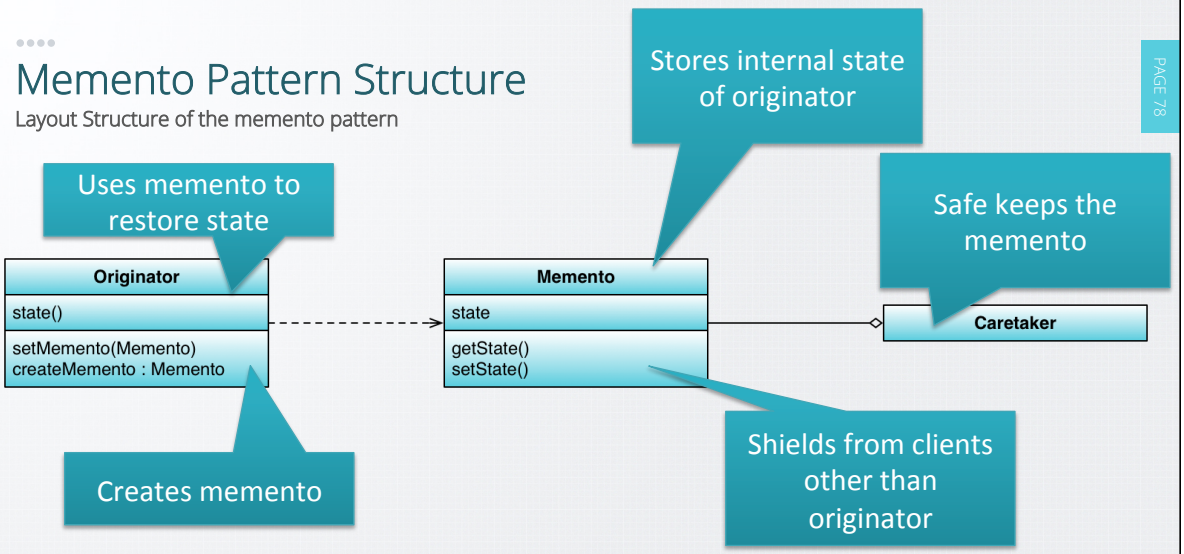
Apply Memento Pattern

Applicability

- **Use**
  - **Save** a **snapshot** of an objects state
  - Direct interface to object state would violate encapsulation

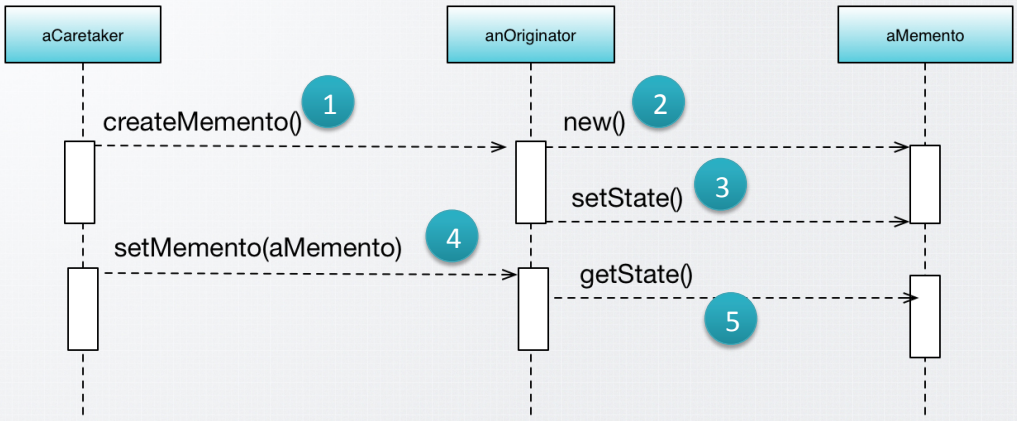
Memento Pattern Structure

Layout Structure of the memento pattern



Memento pattern Collaboration

How objects work with a memento



Memento Pattern Consequences

Considerations for the memento pattern

- **Benefits**
  - **Preserve encapsulation**
  - Simplifies originator
- **Drawbacks**
  - Might be **expensive**
    - **Object creation**
  - Language must facilitate that only the originator can access the memento's state
  - **Extra management** for caretaking the mementos

Conclusion

- **Memento pattern is great**
  - **Access state** but preserve **encapsulation**
  - Save **snapshot** of object data

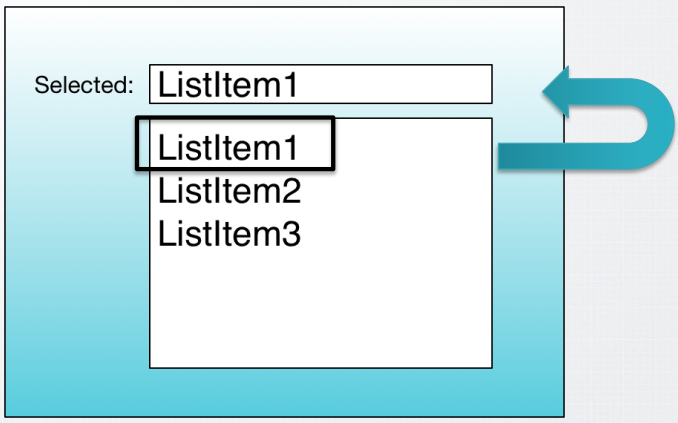
Problem Statement

Brad

- Hello Brad
- Hi Suzy
- One of the developers is stuck with the implementation of a screen, could you have a look?
- Sure

Suzy

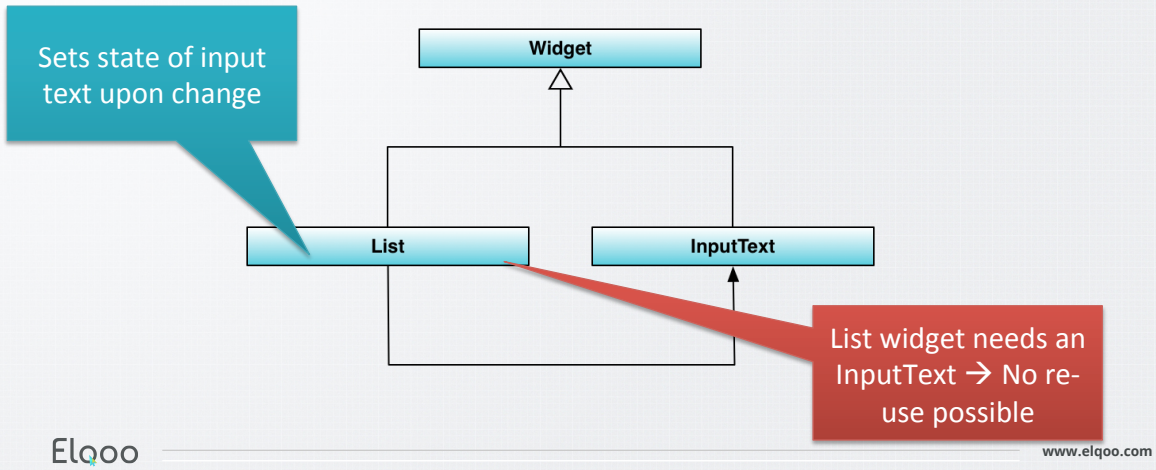
Problem Statement Overview



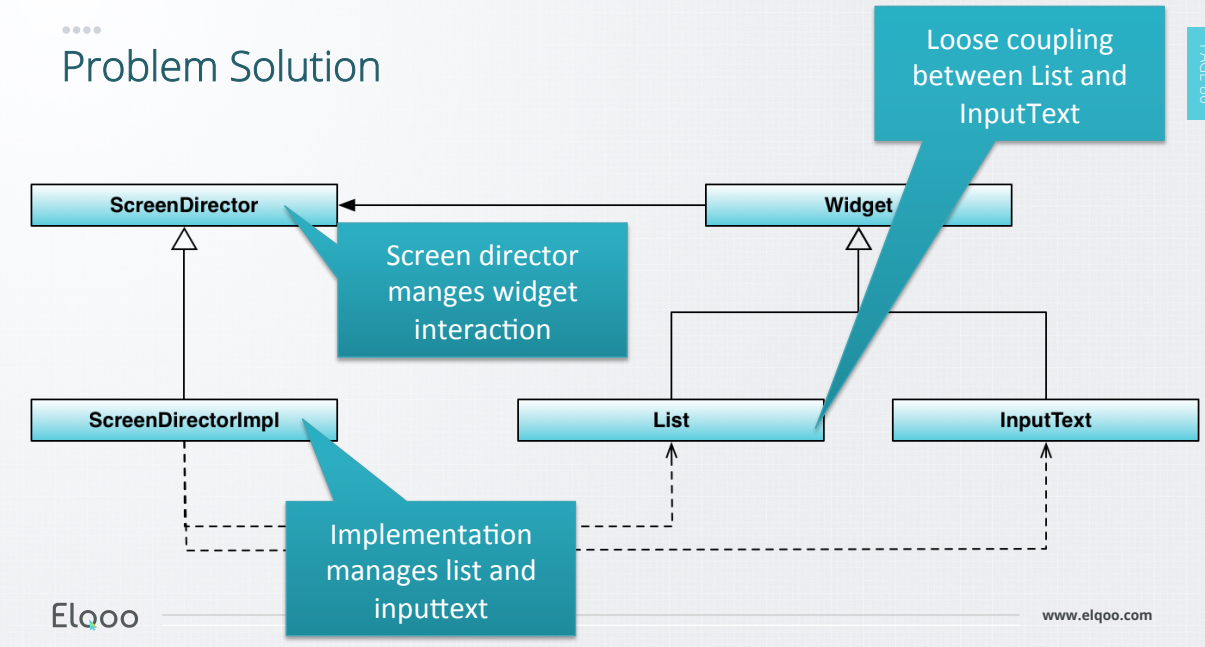
Changes upon selection of one item in the list

Problem Statement Overview

The screen implementation



Problem Solution



Design Pattern Details

Mediator Pattern

Intent and Known As

Intent

Define an object that **encapsulates** how a **set of objects interact**. Mediator **promotes loose coupling** by keeping objects from referring to each other explicitly, and lets you vary their interaction independently.

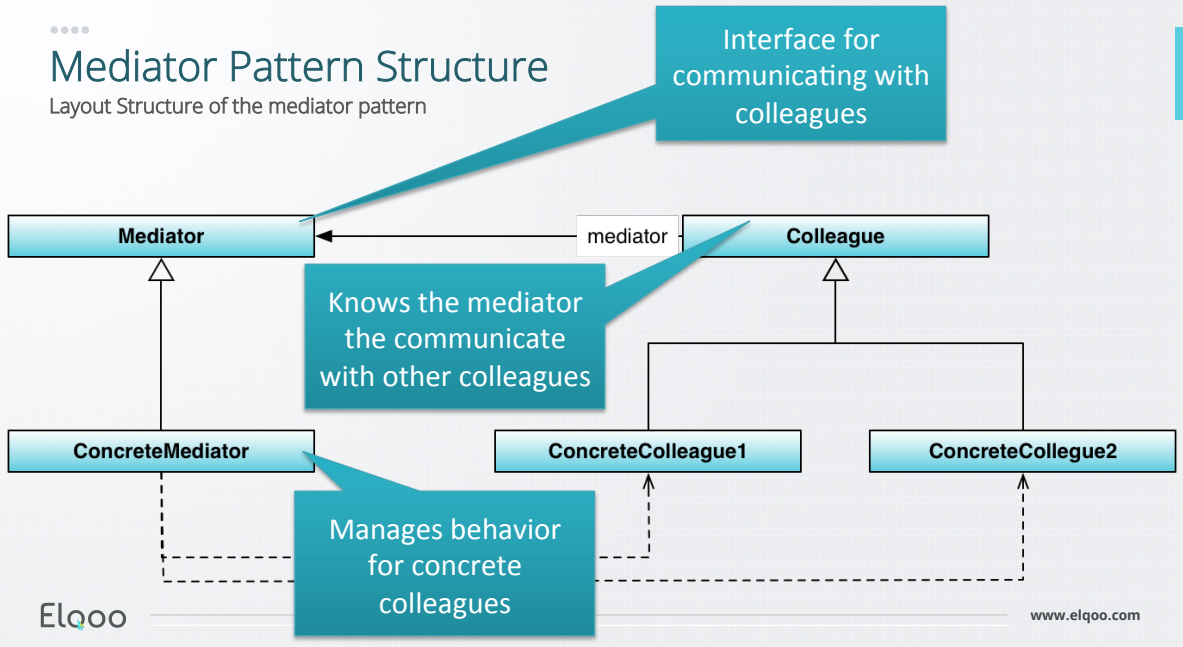
Apply Mediator Pattern

Applicability

- **Use**
  - Objects have **complex communication**, but it's well defined
  - Hard to identify how the communication actually works
  - Object **re-use** is **difficult**
    - These objects require a set of different objects (waterfall of dependencies)
  - **Centralize behavior** between classes.

Mediator Pattern Structure

Layout Structure of the mediator pattern



Mediator Pattern Consequences

Considerations for the Mediator pattern

- **Benefits**
  - **Centralize behavior**
    - Would otherwise be distributed among other objects
  - **Decoupling** of colleagues
  - Changes **many-to-many** interaction **to one-to-many** interaction
  - Object collaboration is abstracted
  - **Centralized control**

Conclusion

- **Mediator pattern is great**
  - **Centralize behavior** and **control**
  - **Loose coupling** between colleagues

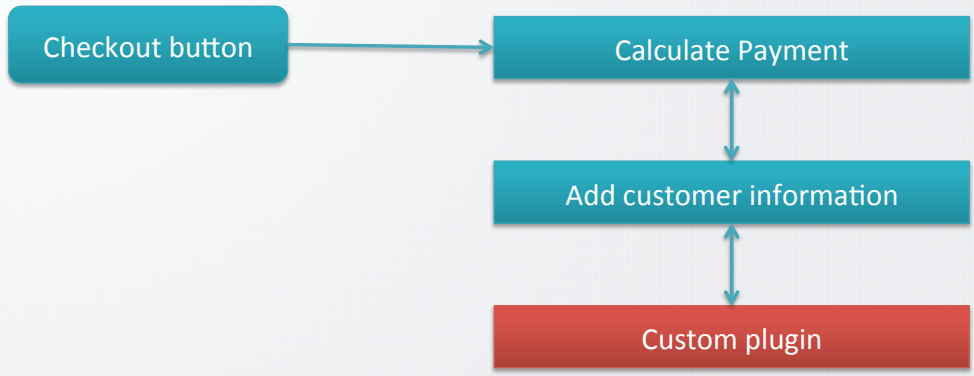
Problem Statement

Brad

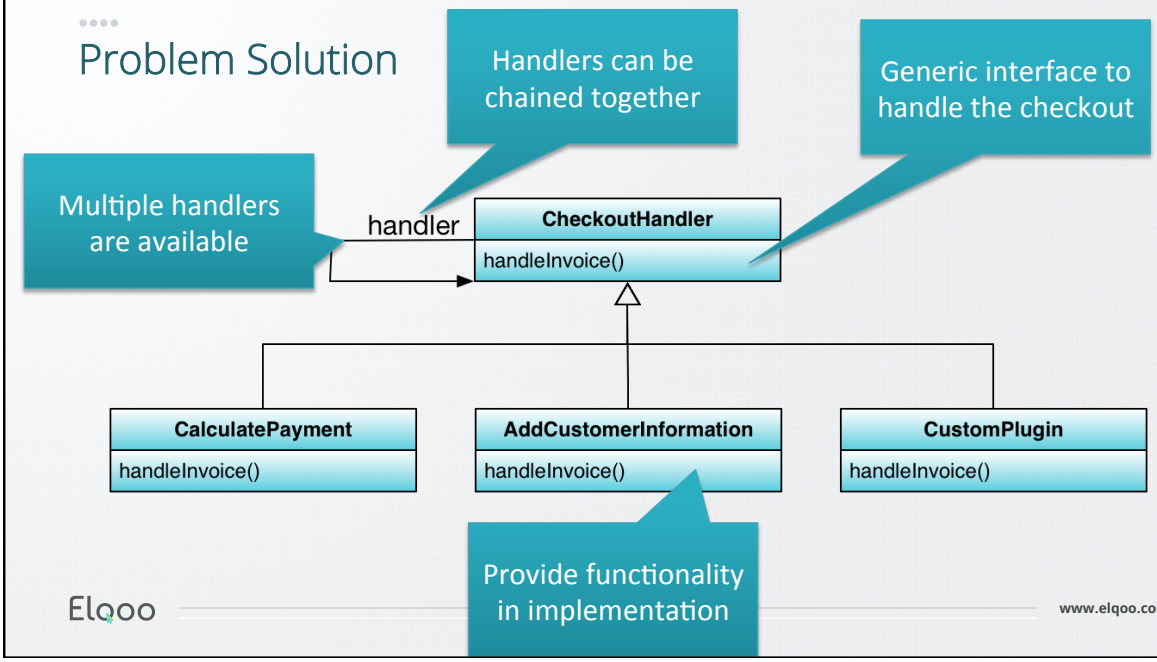
- Hello Brad
- Hi Suzy
- The users of our e-commerce system want to add plugins to the checkout page.
- No problem

Suzy

Problem Statement Overview



Problem Solution



Design Pattern Details

Chain Of Responsibility Pattern

Intent and Known As

Intent

Avoid coupling the **sender** of a **request** to its **receiver** by giving **more than one object** a change to **handle the request**. Chain the receiving objects and pass the request along the chain until an object handles it.

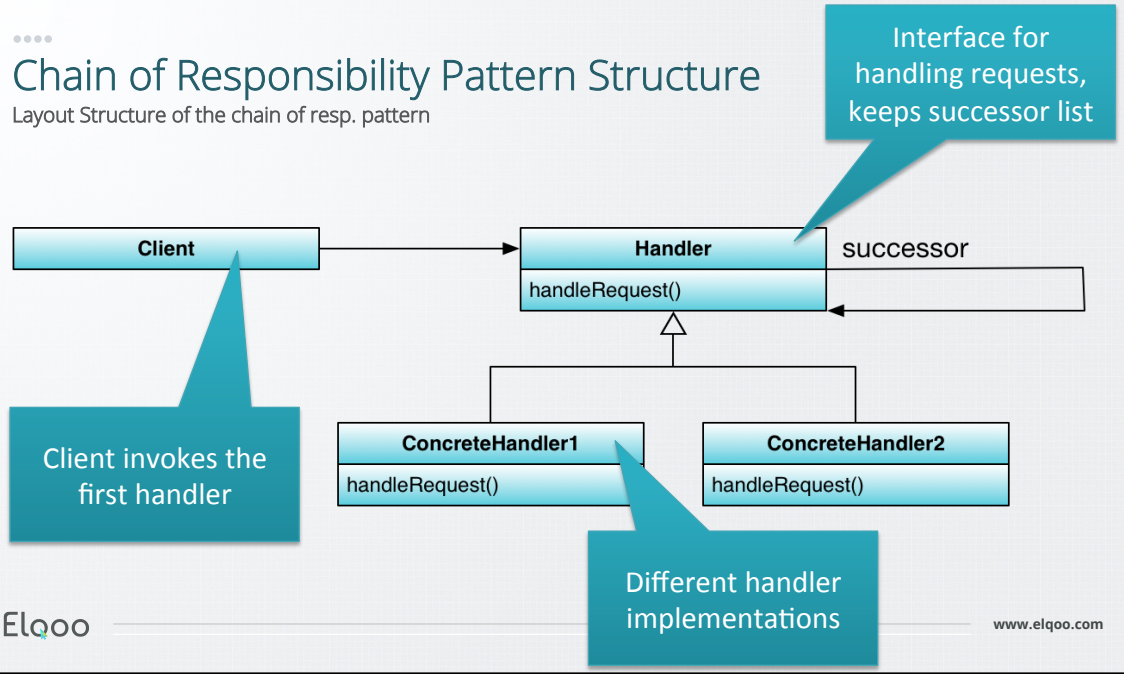
Apply Chain Of Responsibility Pattern

Applicability

- **Use**
  - **Multiple objects** need to **handle a request**
  - Isn't clear upfront **who will handle it**
  - Who can handle the request should be dynamic

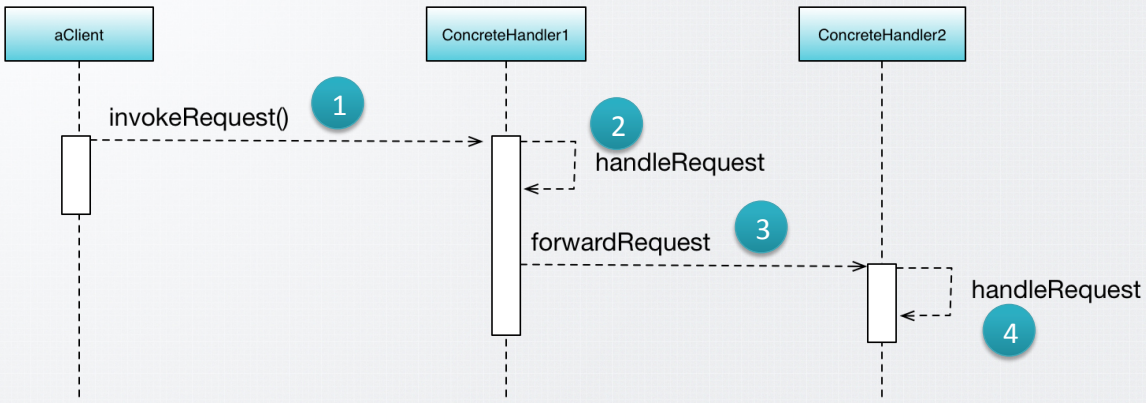
Chain of Responsibility Pattern Structure

Layout Structure of the chain of resp. pattern



Chain Of Responsibility Pattern Collaboration

Collaboration of objects using the chain of responsibility pattern



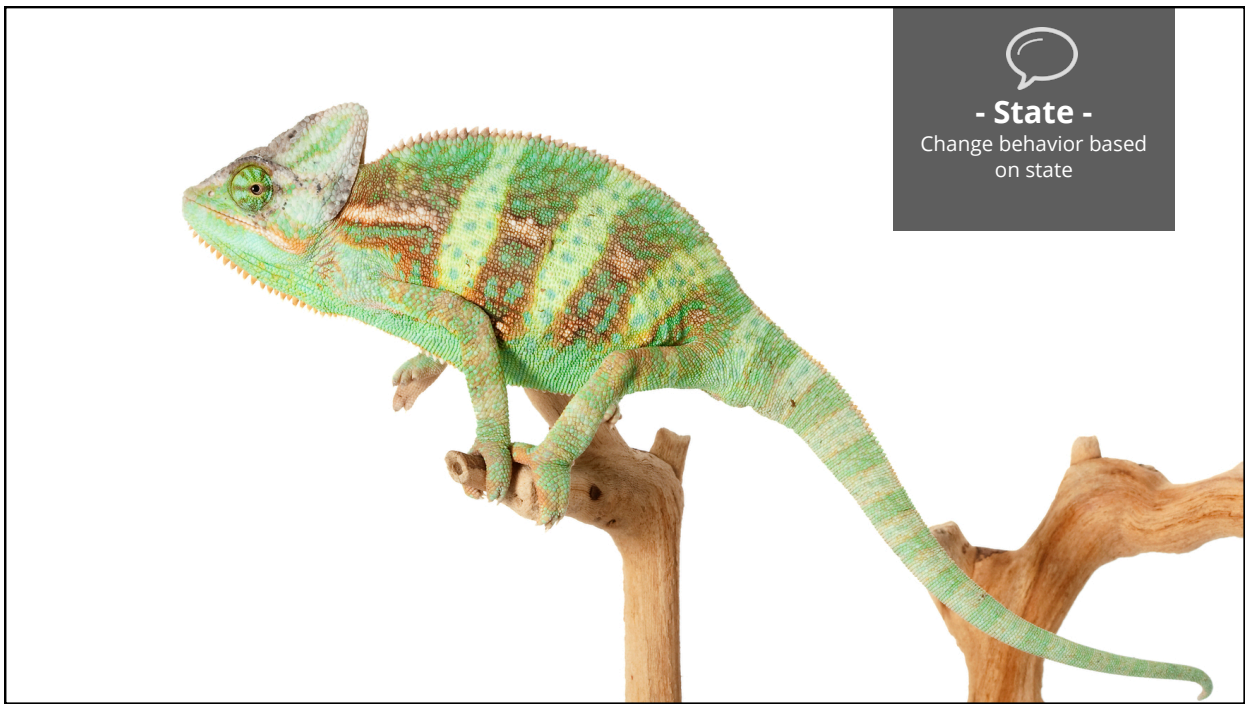
Chain Of Responsibility Pattern Consequences

Considerations for the chain of responsibility pattern

- **Benefits**
  - **Loose coupling** between **requester** and **receiver**
  - Objects can **spread responsibility** in handling requests.
- **Drawbacks**
  - No guarantee the **request** will be **handled**

Conclusion

- **Chain of responsibility pattern is great**
  - **Handle** requests **dynamically**
  - **Enhanced loose coupling** between sender and receiver



Problem Statement

Brad

- Hello Brad
- Hi Suzy
- We would like you to build us a networking framework.
- No problem

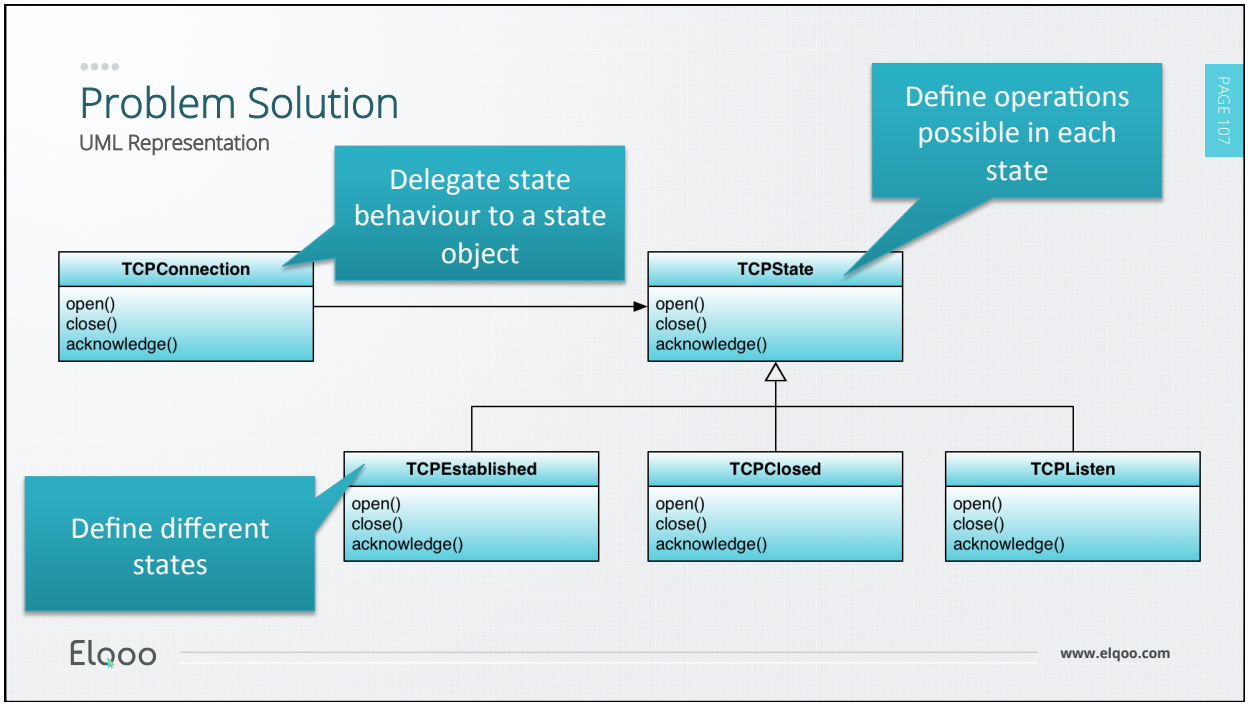
Suzy

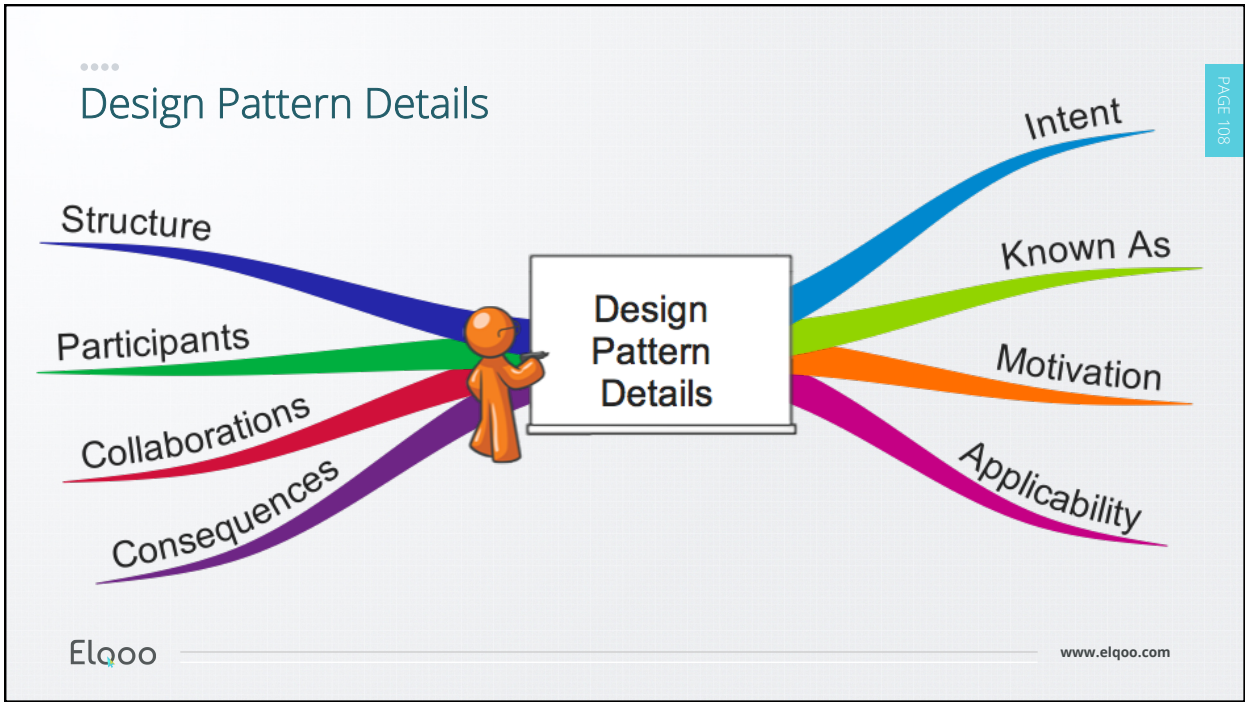
Problem Statement Overview

TCP Connection

- Has different **states**
  - Open, closed, acknowledge
- The **behavior** for the connection is **different in each state**
  - e.g. only an opened connection can be closed
- Should we keep the state and it's implementation in the TCP Connection class itself?

TCP Connection


Problem Solution
UML Representation
PAGE 107
Define operations possible in each state
Delegate state behaviour to a state object
TCPConnection
open()
close()
acknowledge()
TCPState
open()
close()
acknowledge()
TCPEstablished
open()
close()
acknowledge()
TCPClosed
open()
close()
acknowledge()
TCPListen
open()
close()
acknowledge()
Define different states
Elqoo
www.elqoo.com


Design Pattern Details
PAGE 108
Intent
Known As
Motivation
Applicability
Structure
Participants
Collaborations
Consequences
Design Pattern Details
Elqoo
www.elqoo.com

State Pattern

Intent and Known As

Intent

Allow an object **to alter its behavior** when its internal **state changes**. The object will appear to change its class.

Known As

Object for states

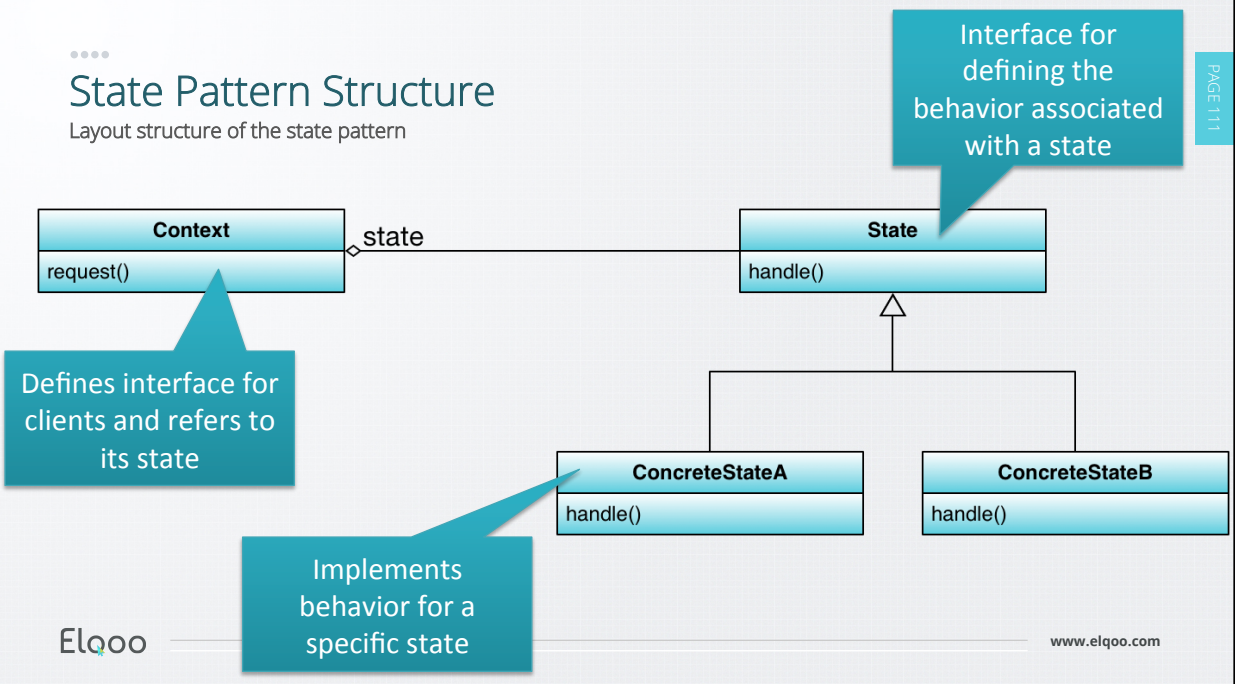
Apply State Pattern

Applicability

- **Use**
  - Object **behavior depends** on **object state**.
  - **Avoid complex if-else-structures**
    - When state changes, simply change the state object
    - Implementation is done in the state object

State Pattern Structure

Layout structure of the state pattern



State Pattern Collaboration

How objects work with a state

- All **state behavior** is **delegated** to the **state implementation**
- Arguments can be past from context to state
- Only **context** is **available** for **clients to see**
- Which state is next?
  - Can be decided by the Context or ConcreteState impl.

State Pattern Consequences

Considerations for the state pattern

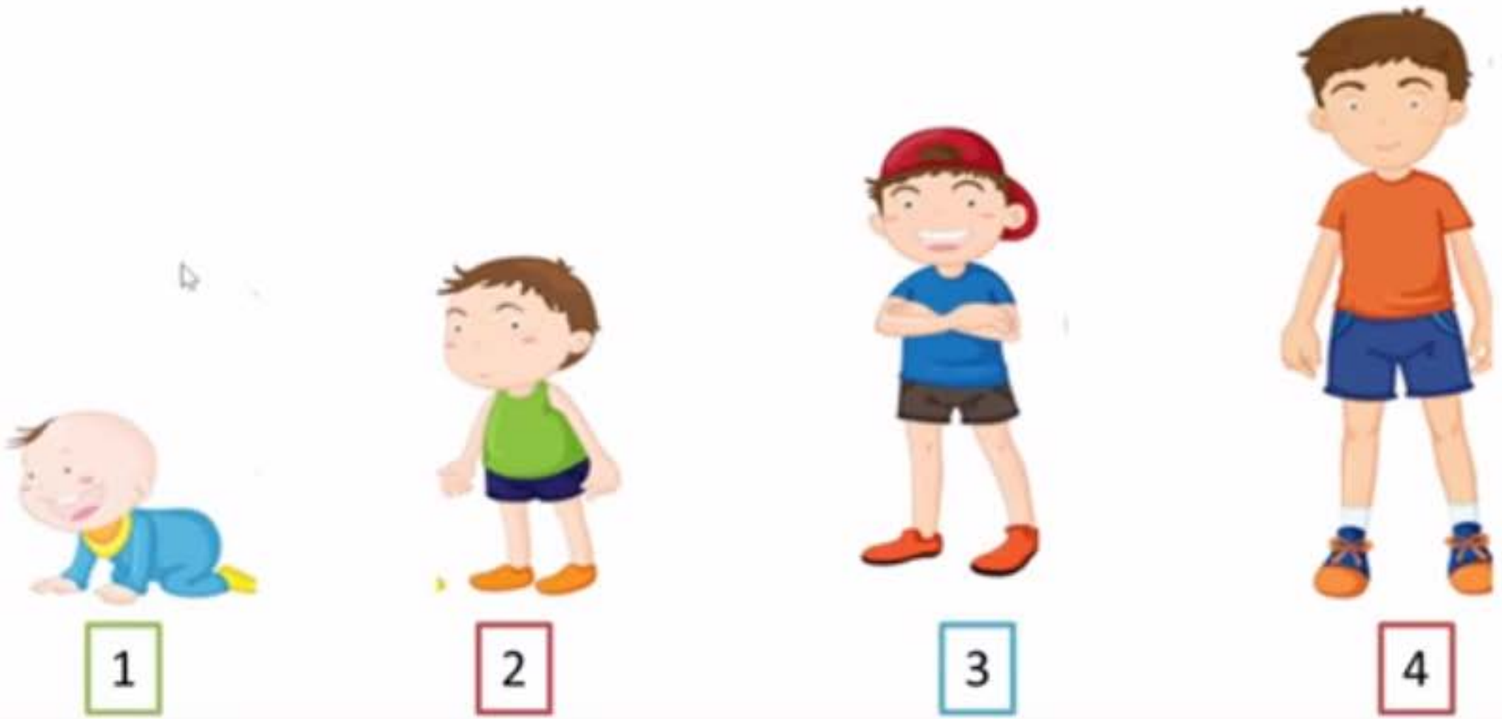
- **Benefits**
  - State related **behavior** is **centralized**
  - State **transitions** are **explicit** → state object must be changed
  - State **changes** can be **atomic** → only one variable in the Context
- **Drawbacks**
  - State objects must be **shareable**

Conclusion

- **State pattern is great**
  - **State behavior** is **centralized**
  - **Reduce complexity** due to state

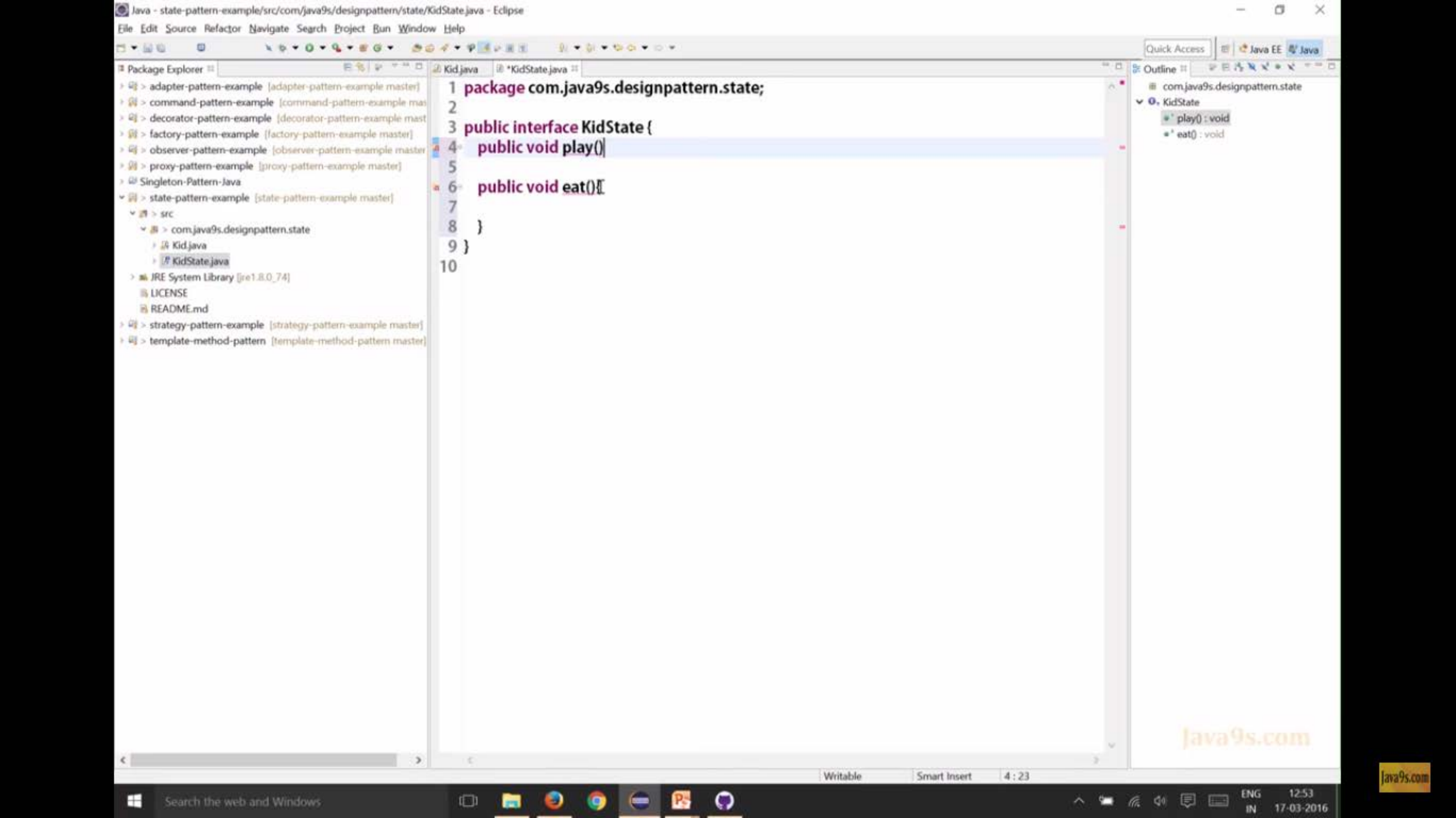
State Pattern in Java | State Design pattern example | Design Pattern | Java9s.com

Object - States

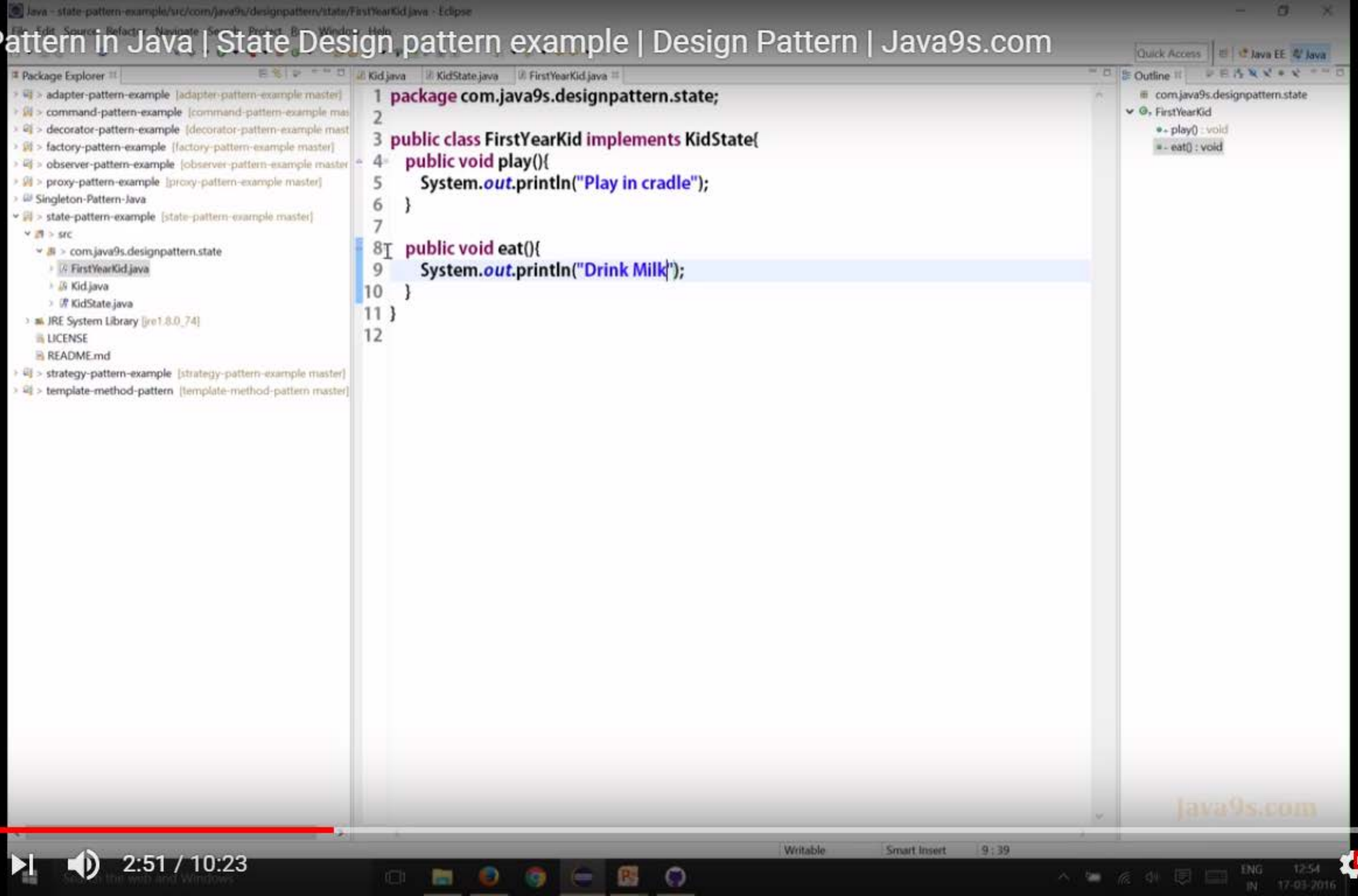


Settings

0:20 / 10:23

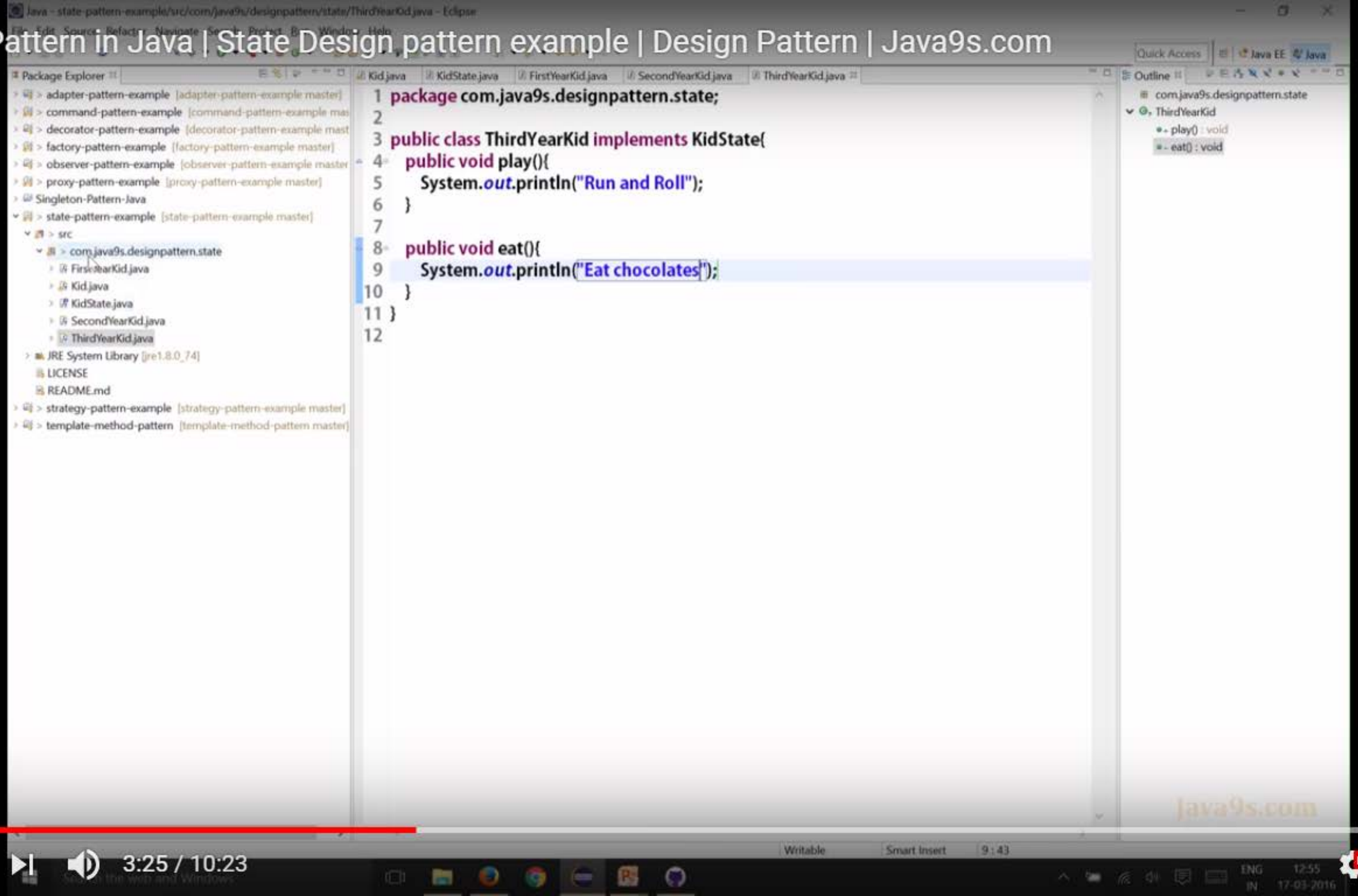

Java - state-pattern-example/src/com/java9s/designpattern/state/KidState.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Java EE
Java
Package Explorer
adapter-pattern-example [adapter-pattern-example master]
command-pattern-example [command-pattern-example mas
decorator-pattern-example [decorator-pattern-example mast
factory-pattern-example [factory-pattern-example master]
observer-pattern-example [observer-pattern-example master
proxy-pattern-example [proxy-pattern-example master]
Singleton-Pattern-Java
state-pattern-example [state-pattern-example master]
src
com.java9s.designpattern.state
Kid.java
KidState.java
JRE System Library [jre1.8.0_74]
LICENSE
README.md
strategy-pattern-example [strategy-pattern-example master]
template-method-pattern [template-method-pattern master]
Kid.java
*KidState.java
1 package com.java9s.designpattern.state;
2
3 public interface KidState {
4 public void play()
5
6 public void eat(){
7
8 }
9 }
10
Outline
com.java9s.designpattern.state
KidState
play() : void
eat() : void
Java9s.com
Writable
Smart Insert
4 : 23
Search the web and Windows
ENG
IN
12:53
17-03-2016
Java9s.com

State Pattern in Java | State Design pattern example | Design Pattern | Java9s.com



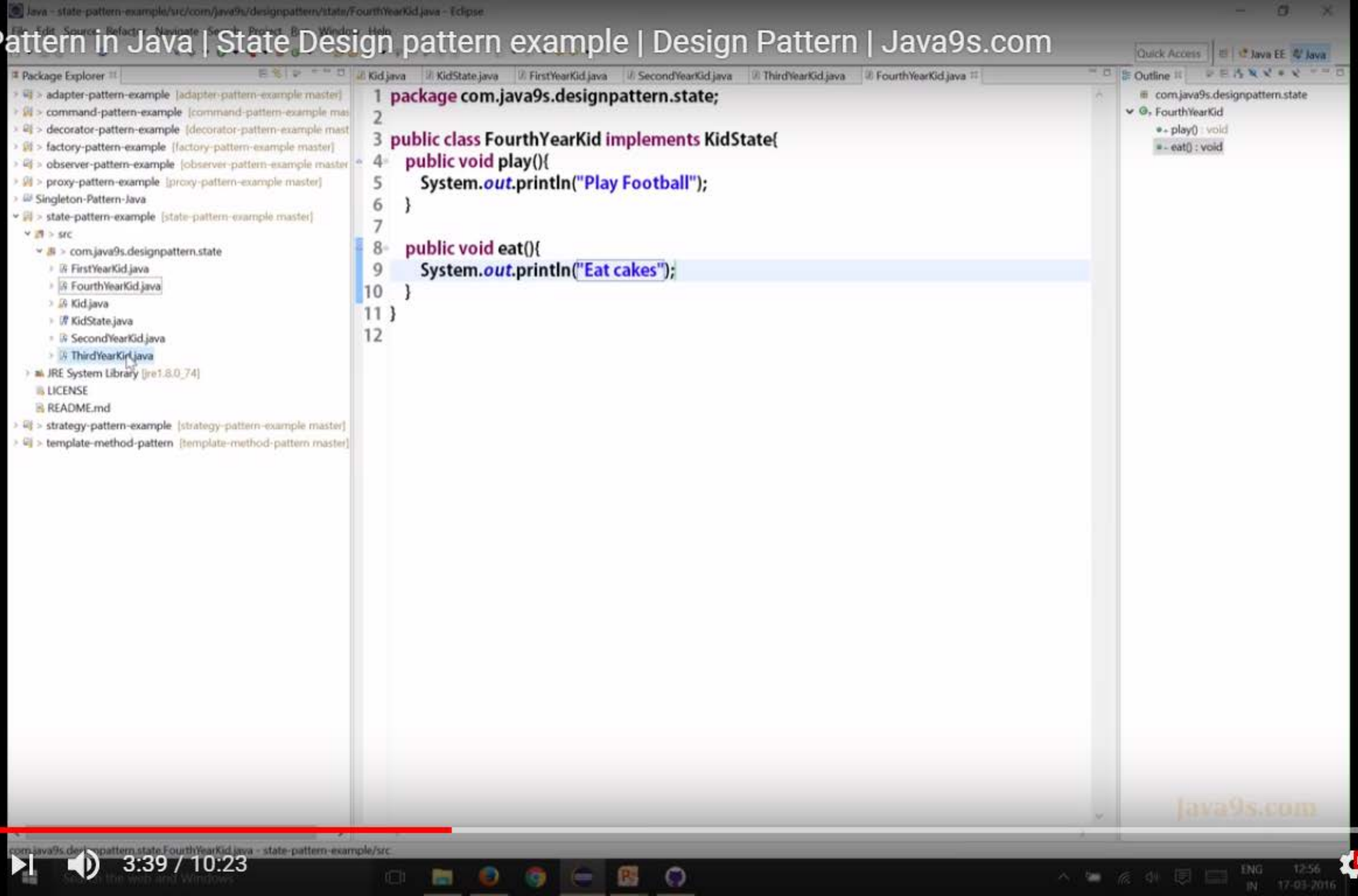
2:51 / 10:23

State Pattern in Java | State Design pattern example | Design Pattern | Java9s.com



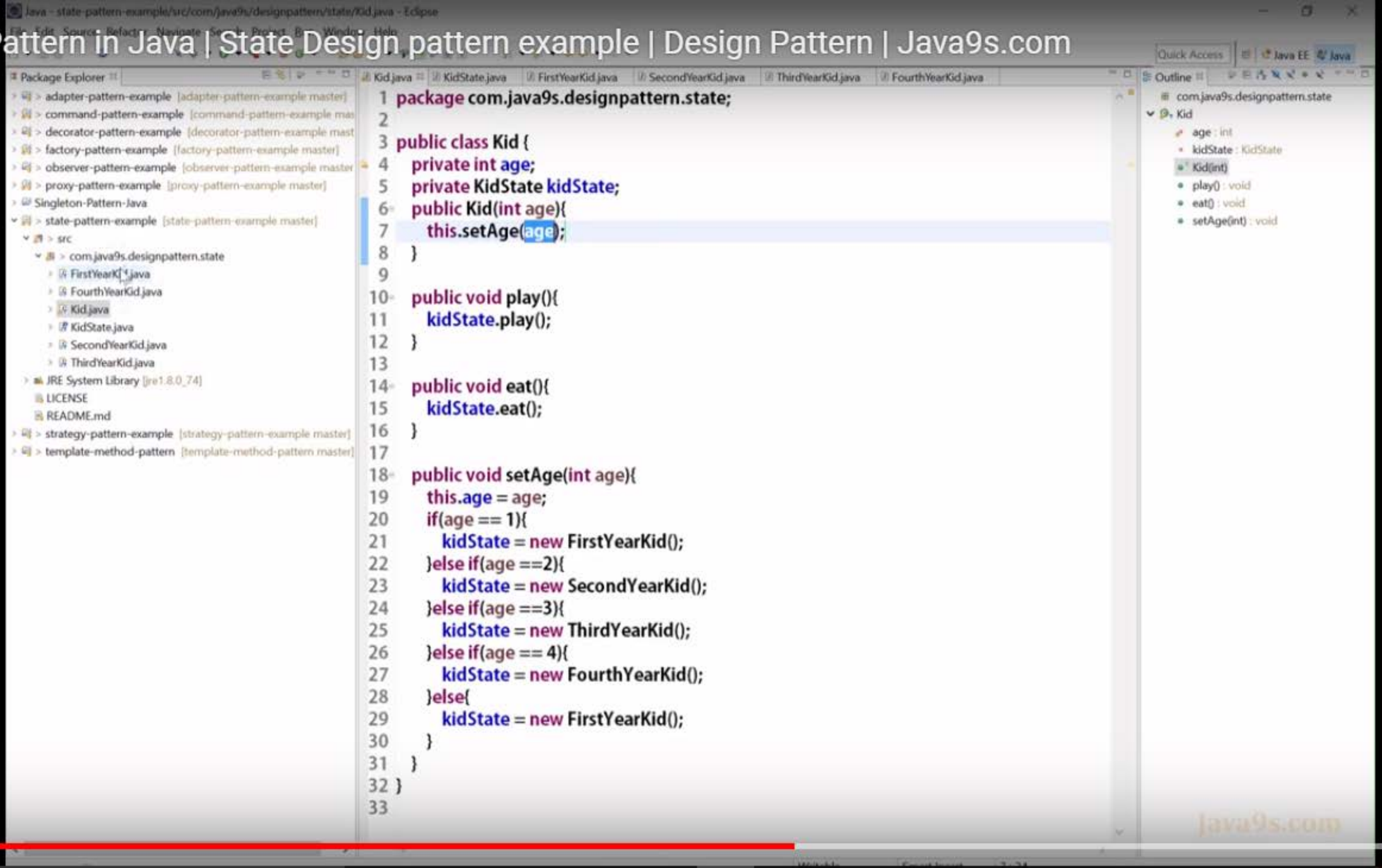
3:25 / 10:23

State Pattern in Java | State Design pattern example | Design Pattern | Java9s.com

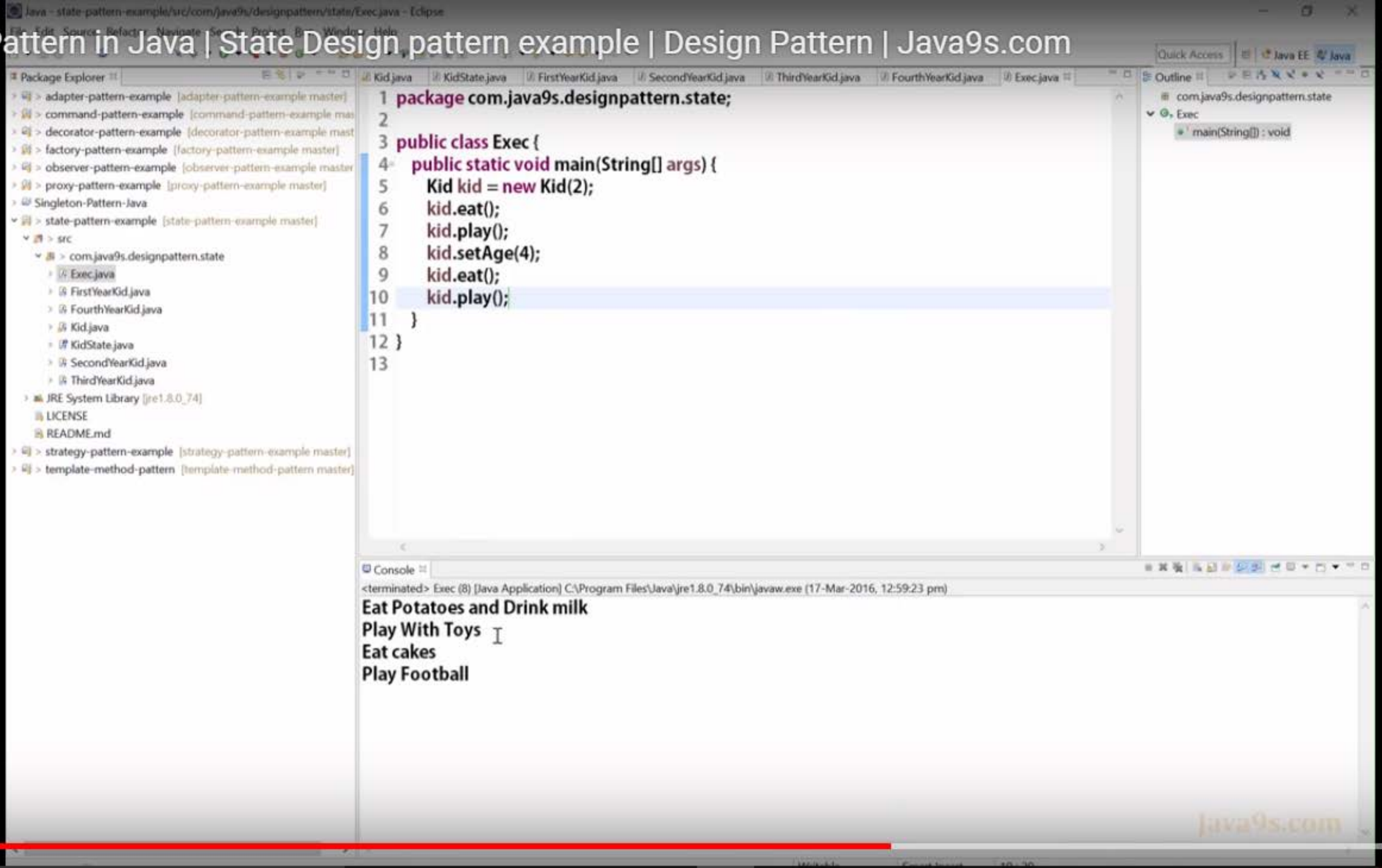


3:39 / 10:23

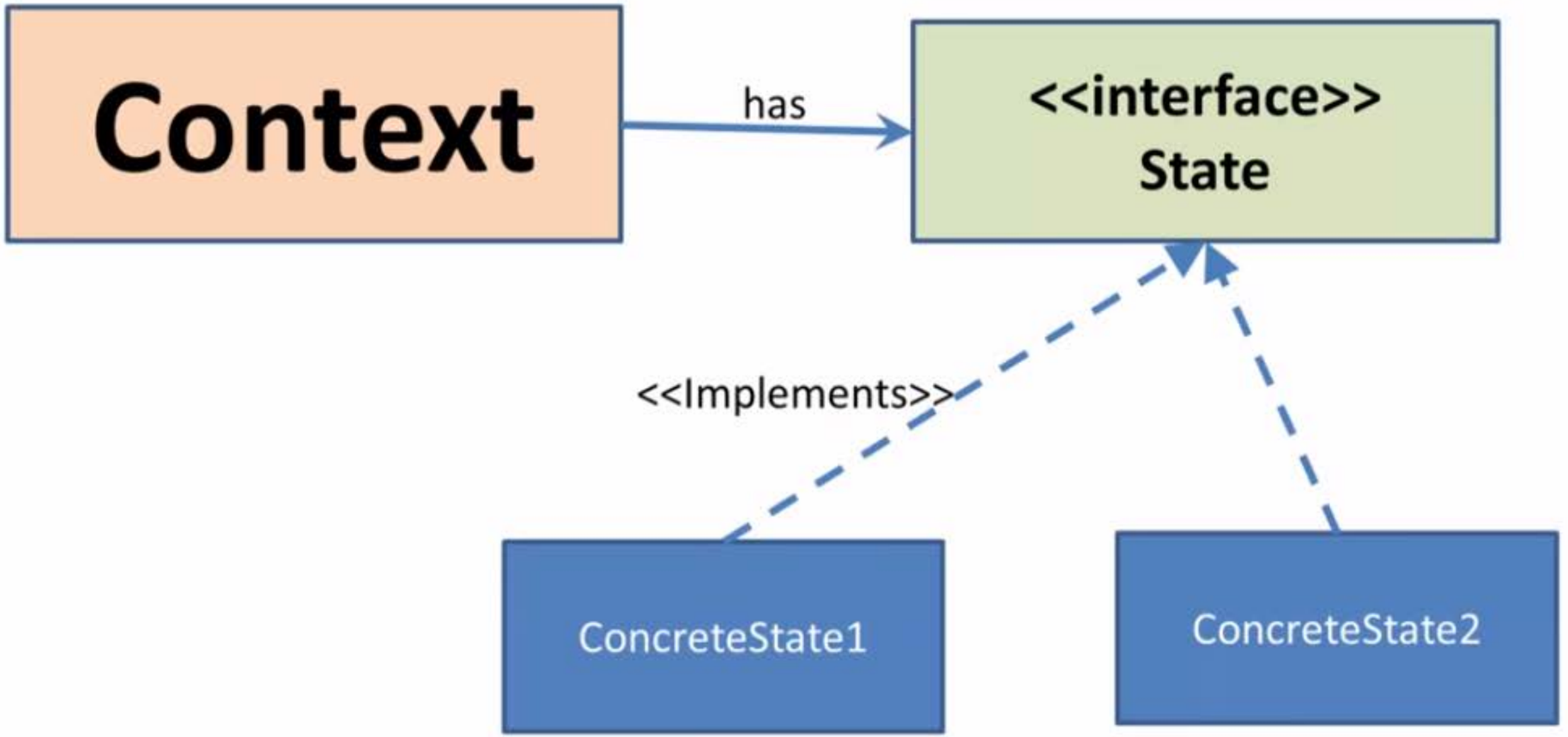
State Pattern in Java | State Design pattern example | Design Pattern | Java9s.com



State Pattern in Java | State Design pattern example | Design Pattern | Java9s.com

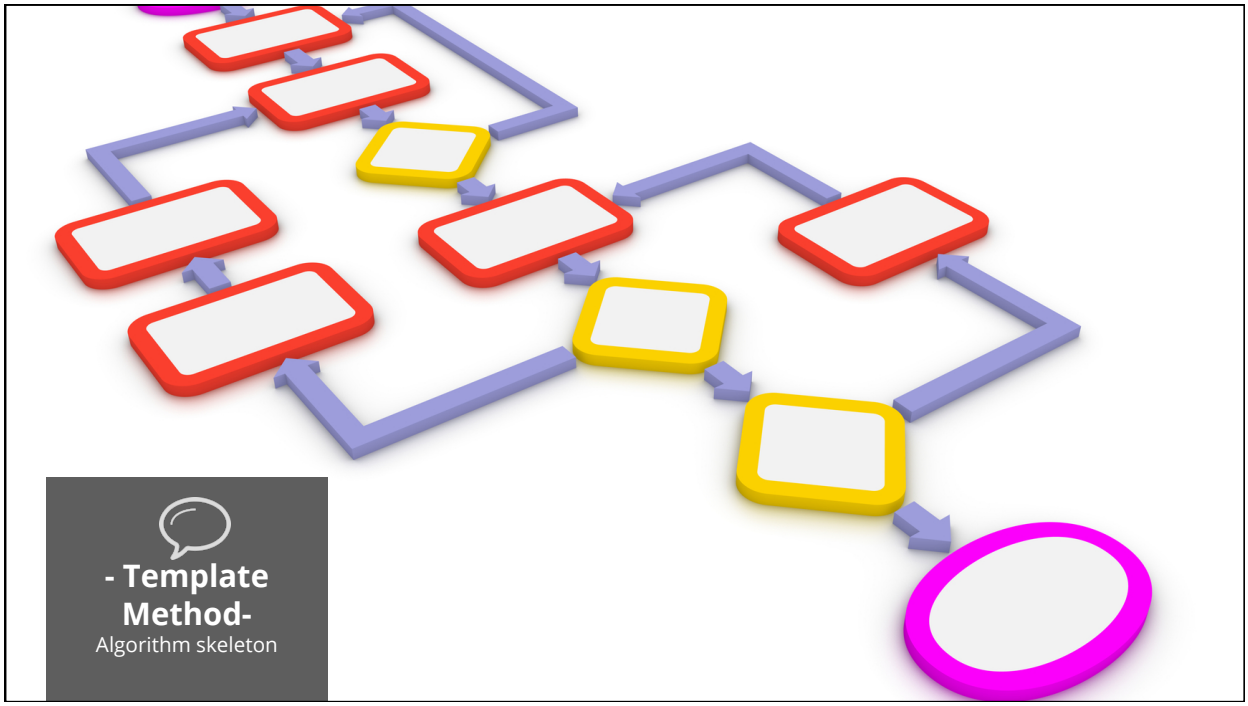


State Pattern - Design



State Pattern - Advantages

- Encapsulates the changing behaviour
- Any changes to the behaviours based on state will only change state objects.
- Easy for extensions as we can create any number of State objects

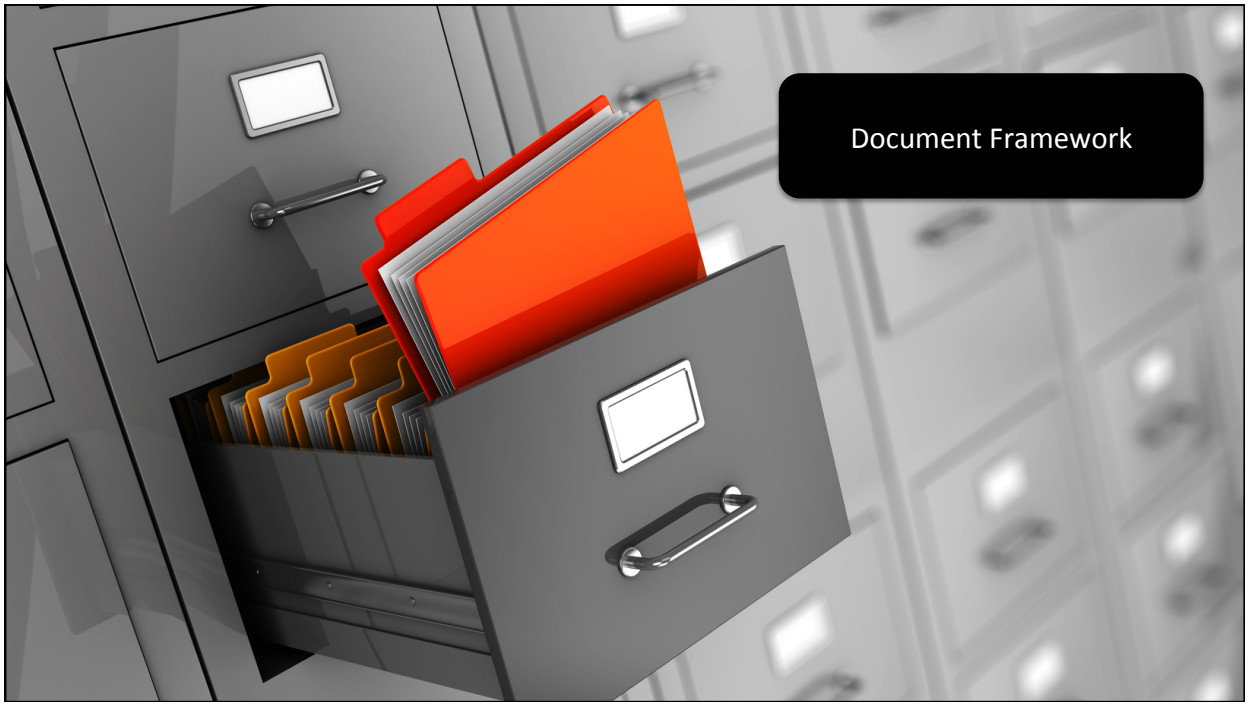


Problem Statement

Brad

- Hello Brad
- Hi Suzy
- Can you help our developers to only focus on the new functionality to implement for our document framework?
- No problem

Suzy

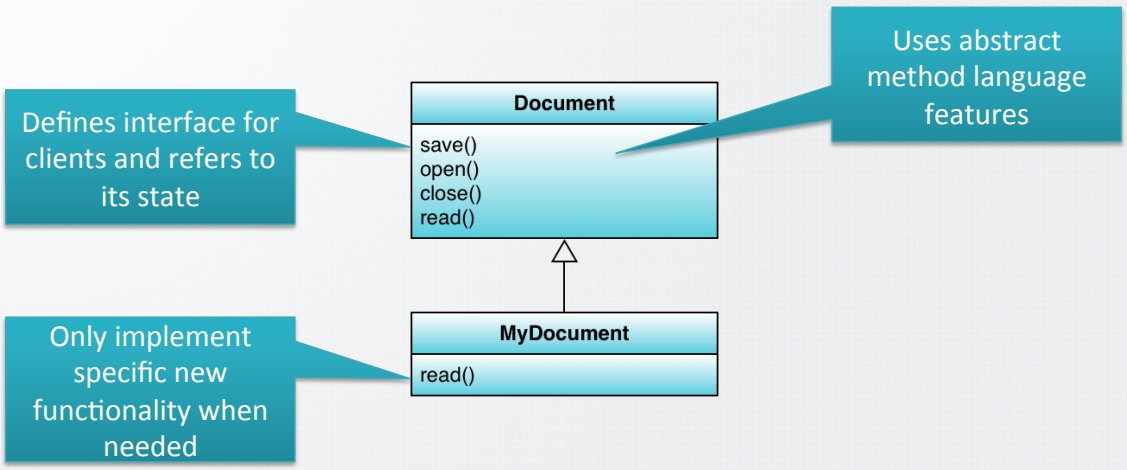


Problem Statement Overview

- Each **document** has the **same lifecycle**
  - Save
  - Open
  - Close
  - Read
- This lifecycle returns for each document

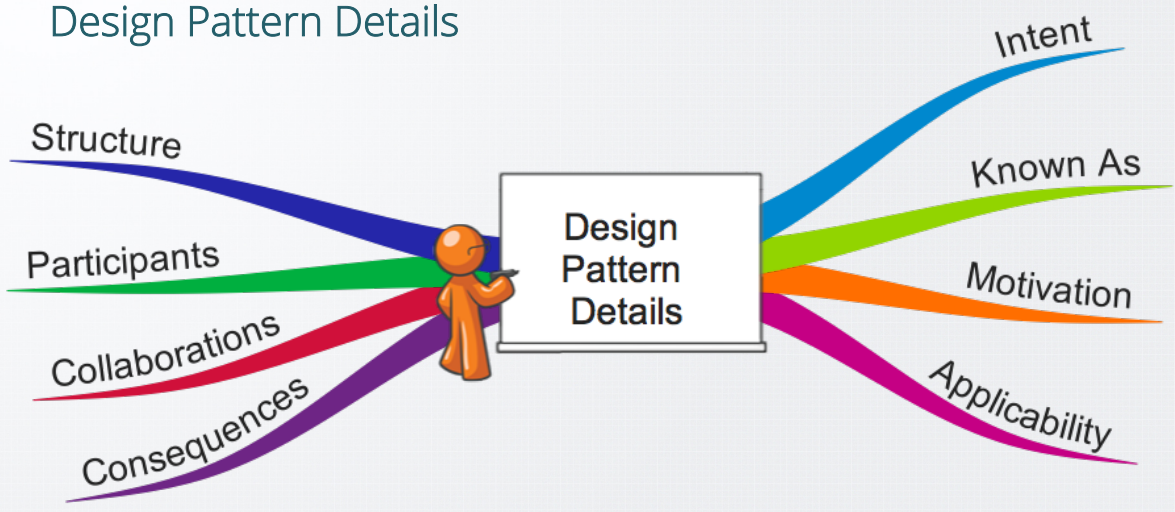
  → Do we need to implement it again?

Problem Solution



Elqoo www.elqoo.com

Design Pattern Details



Elqoo www.elqoo.com

Template Method Pattern

Intent and Known As

Intent

Define the **skeleton of an algorithm** in an operation, deferring some steps to subclasses. Template Method **lets subclasses redefine certain steps** of an algorithm without changing the algorithm's structure.

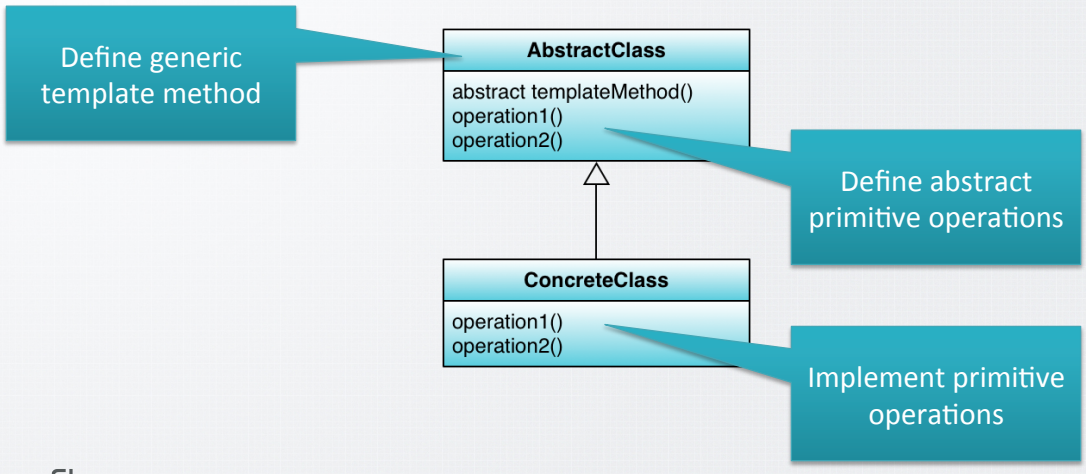
Apply Template Method Pattern

Applicability

- **Use**
  - **Implement** an **algorithm once**
    - Subclasses can provide different implementation
  - **Avoid code duplication**
  - Define how a class should be extended
    - Implementing hooks

Template Method Pattern Structure

Layout Structure of the observer pattern using inheritance



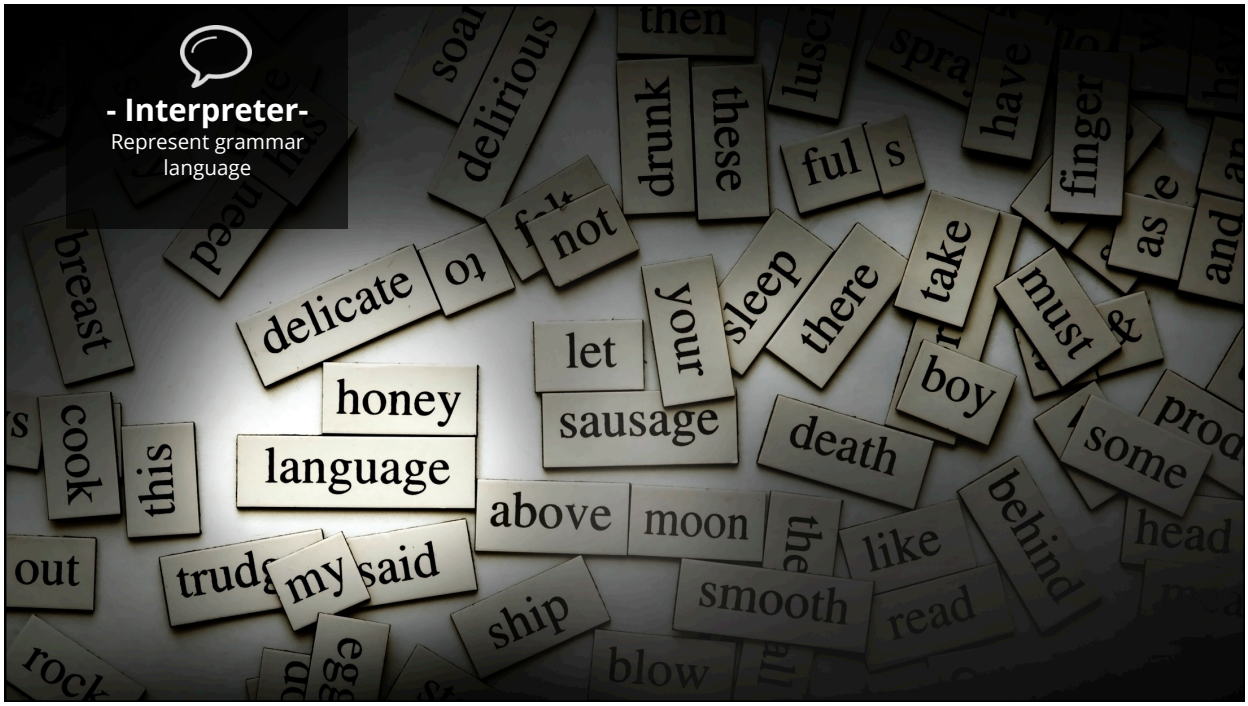
Template Method Pattern Consequences

Considerations for the template method pattern

- **Benefits**
  - Ultimate **code re-use**
    - Important for class libraries
  - **Provide hooks**
  - The abstract base class generally doesn't implement a hook → forces clients to have an implementation

Conclusion

- **Template method pattern is great**
  - Heavily **re-use code**
  - Provide **hooks**



Problem Statement

Brad

- Hello Brad
- Hi Suzy
- We have created a new calculation language and need a problem to help us.
- No problem

Suzy

Problem Statement Overview (1)

The language definition


```
expression ::= plus | minus | variable | number
plus ::= expression expression '+'
minus ::= expression expression '-'
variable  ::= 'a' | 'b' | 'c' | ... | 'z'
digit = '0' | '1' | ... | '9'
number ::= digit | digit number
```

Problem Statement Overview (2)

Example:

a = 1
b = 2
c = 3

a b + c a - -

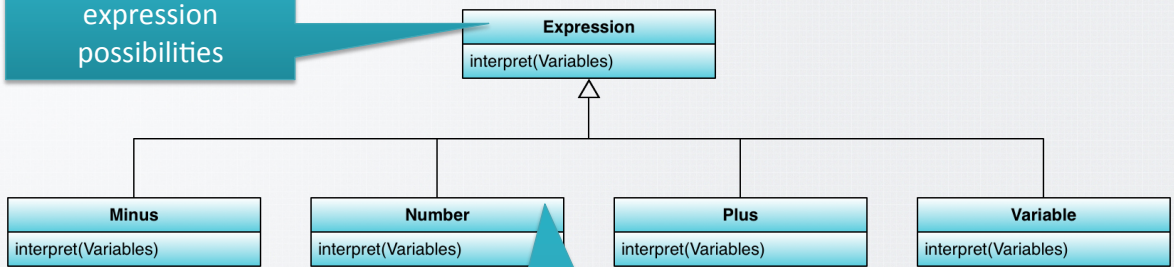
3 2

1

Problem Solution

UML language representation

Base class for all expression possibilities



Each Base class is a separate expression

Design Pattern Details

Interpreter Pattern

Intent and Known As

Intent

Given a **language**, **define a representation** for its grammar along with an interpreter that uses the representation to **interpret sentences in the language**

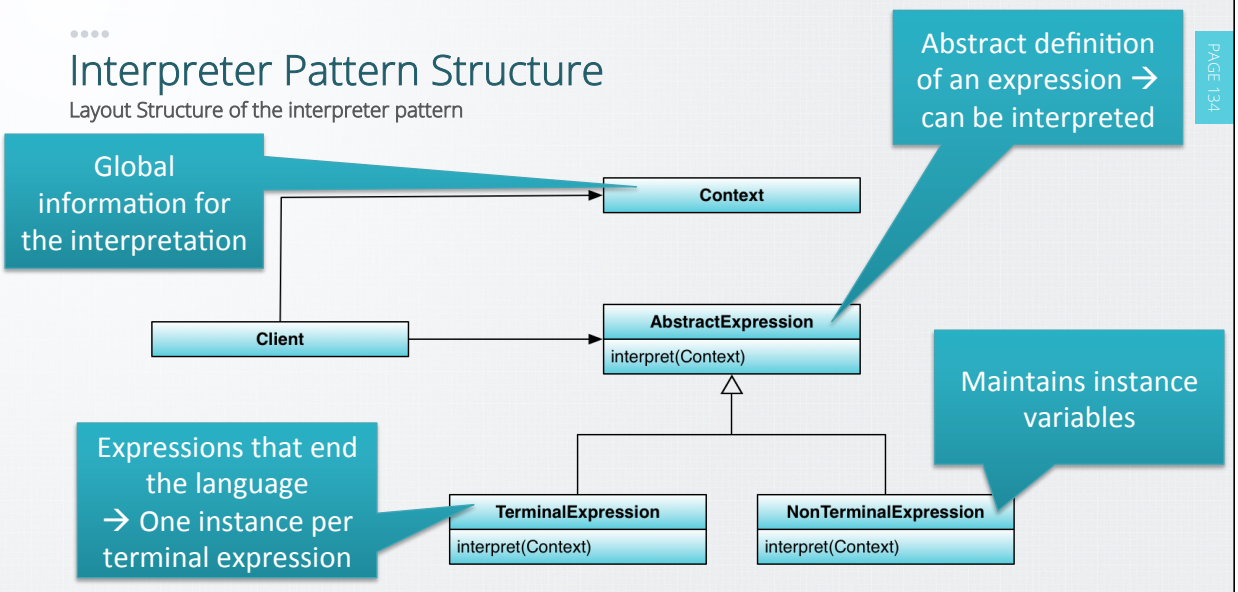
Apply Interpreter Pattern

Applicability

- **Use**
  - **A language needs to be interpreted**
    - The language is simple
  - **Efficiency** is **not critical**
    - Note: languages are normally translated into a state machine

Interpreter Pattern Structure

Layout Structure of the interpreter pattern



Interpreter Pattern: Collaborations

- **Client** is responsible for **building** the **syntax tree**
  - Representations of terminal & nonterminal expressions

Interpreter Pattern Consequences

Considerations for the interpreter pattern

- **Benefits**
  - **Easy to change** or **extend** the grammar
  - New interpretation expressions can be added easily
    - E.g. a pretty print
- **Drawbacks**
  - Hard for complex grammar

Conclusion

- **Interpreter pattern is great**
  - **Simple syntax trees**
  - Transform **language into code objects**