

Loans

```
public class LoanHandler {
    public void handle ( List<Loan> loans ) {
        for ( Loan loan : loans ) {
            if ( ( loan.getBalance() < 2000 ) ||
                ( loan.getBalance() < 5000 &&
                  loan.getRate() < .03 ) ) {
                System.out.println("loan "+loan.getID()+" approved!");
            } else {
                System.out.println("loan "+loan.getID()+" denied!");
            }
        }
    }
}
```

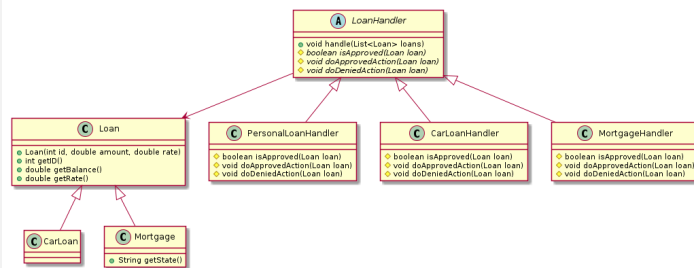
Does this satisfy OCP? (What might change?)

- properties of a loan ...not a problem as long as there's still balance and rate
- rules for approval
- actions taken on approval, denial

```
public abstract class LoanHandler {
    public void handle ( List<Loan> loans ) {
        for ( Loan loan : loans ) {
            if ( isApproved(loan) ) { doApprovedAction(loan); }
            else { doDeniedAction(loan); }
        }
    }
    protected abstract boolean isApproved ( Loan loan );
    protected abstract void doApprovedAction ( Loan loan );
    protected abstract void doDeniedAction ( Loan loan );
}
```

```
public class PersonalLoanHandler extends LoanHandler {
    protected boolean isApproved ( Loan loan ) {
        return ( loan.getBalance() < 2000 ) ||
            ( loan.getBalance() < 5000 &&
              loan.getRate() < .03 );
    }
    protected void doApprovedAction ( Loan loan ) {
        System.out.println("loan "+loan.getID()+" approved!");
    }
    protected void doDeniedAction ( Loan loan ) {
        System.out.println("loan "+loan.getID()+" denied!");
    }
}
```

Loans



Is this a good design?

- + can easily add new loan approval rules, handling actions, kinds of loans
 - more than one reason to change – repeated code if the same actions apply with different rules, or different actions with the same rules
 - approval rules are likely specific to a particular kind of loan – need to cast, can pass the wrong kind of loan to a handler (having the right kind of loan is a runtime rather than compile time check)

Inheritance

Inheritance is often talked about as a way to reuse existing classes or code – but while this often occurs, it is not why inheritance should be used.

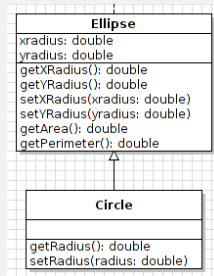
Create subclasses (only) when **both** –

- “is a”, “is a kind of” language makes logical sense, **and**
- everything inherited from the superclass makes sense for the subclass



- this is known as the Liskov Substitution Principle
- introduced by Barbara Liskov in 1987
- she won the 2008 Turing Award for work leading to the development of object-oriented programming (the Turing Award is kind of like the Nobel Prize for computer science – it's a big deal)

Inheritance and the Liskov Substitution Principle



Should Circle extend Ellipse?

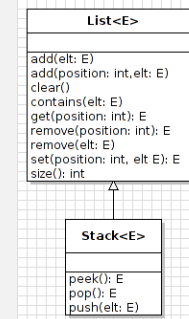
Circle “is a kind of” Ellipse...

But Circle inherits setXRadius() and setYRadius(), allowing the following –

```
Circle c = new Circle();
c.setXRadius(5);
c.setYRadius(10);
```

This doesn't make sense for Circle!
(so no, Circle should not extend Ellipse)

Example



a list is a linear data structure and a stack is a special kind of linear data structure, so Stack might be implemented to extend List

does this satisfy LSP?

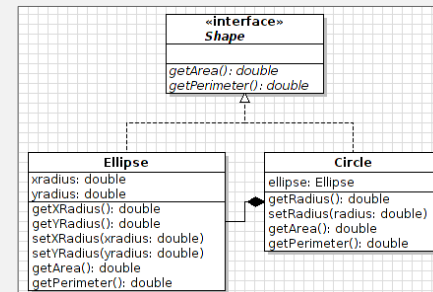
no – remember that all methods are inherited, and List operations that access/manipulate positions other than the top are not appropriate for Stack

Favor Composition Over Inheritance

- inheritance – captures “is-a”
 - tightly bound to superclass's implementation
 - relationship is fixed at compile time
- composition – captures “has-a”
 - a class holds a reference to another object and delegates to it
 - swappable at runtime
- *favor composition over inheritance*
 - only use inheritance when LSP holds
 - use composition to reuse or delegate behavior

Avoiding LSP Violations

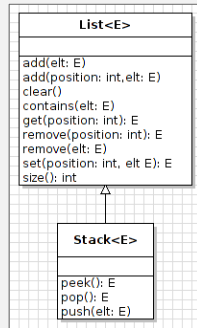
Reserve inheritance for cases which satisfy LSP.
Otherwise favor composition over inheritance.



Circle contains an instance of Ellipse rather than inheriting from Ellipse

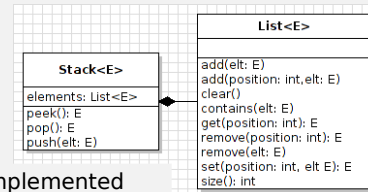
→ reuse the Ellipse code to implement Circle, but the use of Ellipse is hidden so bad usage is prevented

Example



how to fix?

Example



a Stack is implemented using a List

```

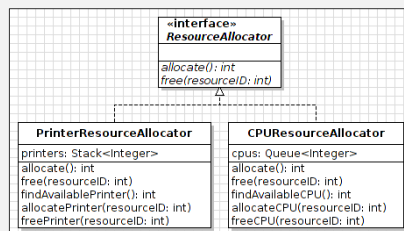
public void push ( E elt ) {
    elements_.add(elt);
}

public E pop () {
    return elements_.remove(elements_.size()-1);
}
    
```

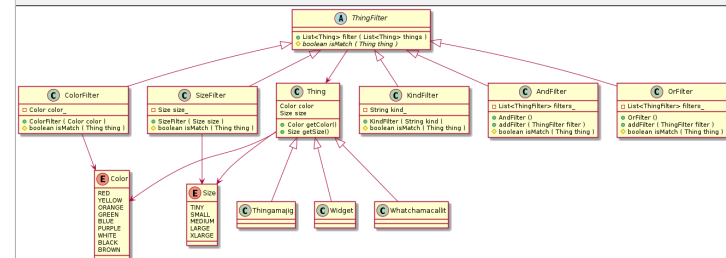
OCP, Encapsulate What Varies, and LSP

The original form of the Open-Closed Principle specified subclasses as the way to extend behavior.

That has been superseded by the *Polymorphic OCP* which says that only interfaces should be inherited.

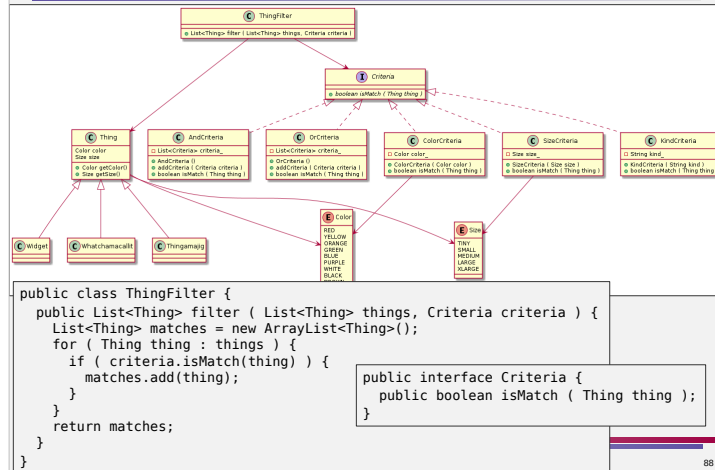


OCP, Encapsulate What Varies, and LSP



- extending ThingFilter isn't a violation of LSP, but it doesn't follow the polymorphic OCP
- solution: favor composition over inheritance
 - pull what varies (isMatch) into a separate class

Favoring Composition Over Inheritance



Favoring Composition Over Inheritance

```

public class ColorCriteria implements Criteria {
    private Color color_;
    public ColorCriteria ( Color color ) { color_ = color; }
    public boolean isMatch ( Thing thing ) {
        return thing.getColor() == color_;
    }
}

```

```

public class SizeCriteria implements Criteria {
    private Size size_;
    public SizeCriteria ( Size size ) { size_ = size; }
    public boolean isMatch ( Thing thing ) {
        return thing.getSize() == size_;
    }
}

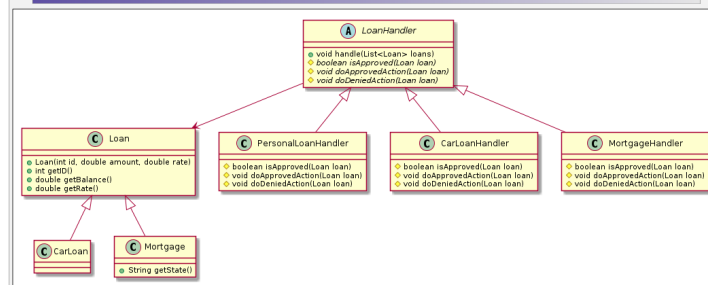
```

```

public class KindCriteria implements Criteria {
    private String kind_;
    public KindCriteria ( String kind ) { kind_ = kind; }
    public boolean isMatch ( Thing thing ) {
        return thing.getKind().equals(kind_);
    }
}

```

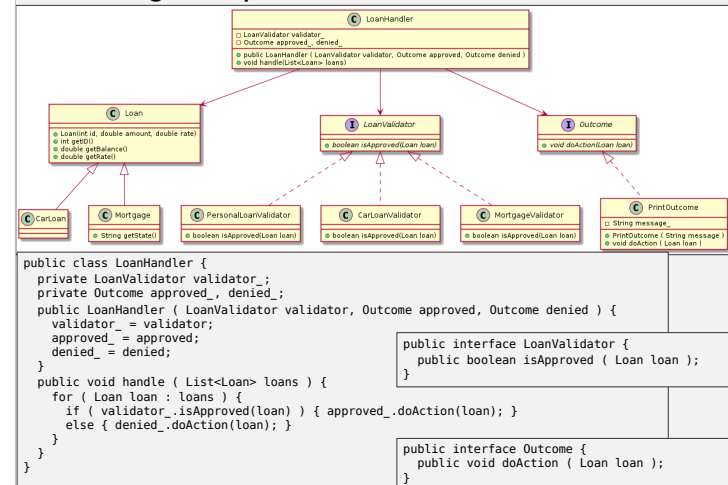
Loans



Is this a good design?

- + can easily add new loan approval rules, handling actions, kind of loans
- more than one reason to change: repeated code if the same actions apply with different rules, or different actions with the same rules
- approval rules are likely specific to a particular kind of loan - need to cast, can pass the wrong kind of loan to a handler (having the right kind of loan is a runtime rather than compile time check)

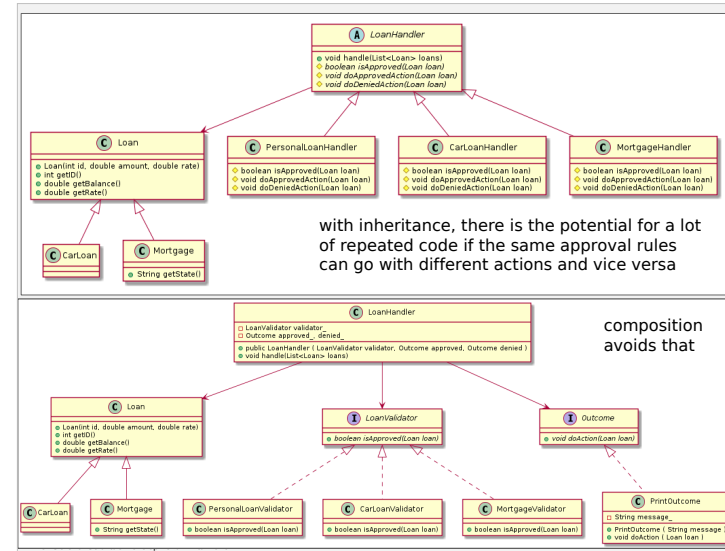
Favoring Composition Over Inheritance



Favoring Composition Over Inheritance

```
public class PersonalLoanValidator implements LoanValidator {
    public boolean isApproved ( Loan loan ) {
        return ( loan.getBalance() < 2000 ) ||
            ( loan.getBalance() < 5000 &&
              loan.getRate() < .03 );
    }
}
```

```
public class PrintOutcome implements Outcome {
    private String message_;
    public PrintOutcome ( String message ) { message_ = message; }
    public void doAction ( Loan loan ) {
        System.out.println("loan "+loan.getID()+" "+message_);
    }
}
```

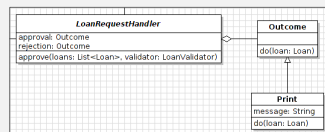


Favoring Composition over Inheritance

When to use composition instead of subclassing?

- when inheritance is not appropriate
 - such as an LSP violation
- when there are many combinations

with just two possible Outcomes for approval or rejection, would need four different LoanRequestHandler subclasses to handle all possible combinations



- when SRP and reusability dictate
 - e.g. LoanHandler has more than one reason to change if it is handling going through a collection of loans and implementing approval rules and taking approved/denied actions – using composition for the rules and actions variations removes them from LoanHandler
 - e.g. if Criteria or Outcome are useful as concepts outside of ThingFilter or LoanHandler – encapsulating them separately means they can be used elsewhere

Interface Segregation Principle (ISP)

- no client should be forced to depend on interface methods it doesn't use
- importance
 - avoids unnecessary dependencies
 - loose coupling – reduces dependencies between classes
 - flexibility – allows more targeted implementations of interfaces

ISP Example

a basic ATM that only supports withdrawals still has to implement the whole ATM interface

```
interface ATMOperations {
    void withdrawCash ( double amount );
    void depositCash ( double amount );
    double checkBalance ();
    void transferFunds ( String toAccount, double amount );
    void printMiniStatement ();
}
```

```
class BasicATM implements ATMOperations {
    public void withdrawCash ( double amount ) { ... }
    public double checkBalance () { ... }

    public void depositCash ( double amount ) {
        throw new UnsupportedOperationException("no deposit slot");
    }
    public void transferFunds ( String toAccount, double amount ) {
        throw new UnsupportedOperationException("not supported at this machine");
    }
    public void printMiniStatement () {
        throw new UnsupportedOperationException("no printer");
    }
}
```

98

ISP Example

fix: split the interface by capability

```
interface CashWithdrawal { void withdrawCash ( double amount ); }
interface CashDeposit { void depositCash ( double amount ); }
interface BalanceInquiry { double checkBalance (); }
interface FundsTransfer { void transferFunds ( String toAccount,
                                                double amount ); }
interface MiniStatement { void printMiniStatement (); }
```

```
class BasicATM implements CashWithdrawal, BalanceInquiry {
    public void withdrawCash ( double amount ) { ... }
    public double checkBalance () { ... }
}

class FullServiceATM implements CashWithdrawal, CashDeposit,
    BalanceInquiry, FundsTransfer,
    MiniStatement {
    ...
}
```

CPSC 329: Software Development • Fall 2026

99

ISP Example

```
interface Document {
    void open ();
    String getText ();
    void setText ( String text );
    void save ();
    void print ();
    List<Comment> getComments ();
    void addComment ( String text, int position );
    void resolveComment ( int commentId );
}
```

- split by what different clients (don't) need
 - read only viewer – doesn't need setText, save, addComment, resolveComment
 - reviewer – doesn't need setText, save
 - full editor – needs everything

CPSC 329: Software Development • Fall 2026

100

ISP Example

```
interface DocumentViewing {
    void open ();
    String getText ();
    List<Comment> getComments ();
}

interface CommentAuthoring {
    void addComment ( String text, int position );
    void resolveComment ( int commentId );
}

interface DocumentEditing {
    void setText( String text );
    void save ();
}

interface Printing {
    void print ();
}
```

```
class ReadOnlyViewer implements DocumentViewing, Printing { ... }
class ReviewerClient implements DocumentViewing, CommentAuthoring { ... }
class FullEditor implements DocumentViewing, CommentAuthoring,
    DocumentEditing, Printing { ... }
```

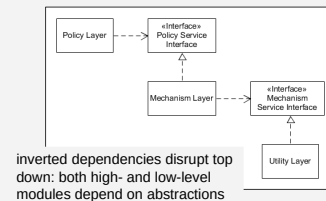
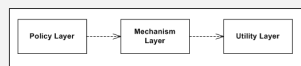
CPSC 329: Software Development • Fall 2026

101

Dependency Inversion Principle (DIP)

- only depend on abstractions, not concretes
 - high-level modules should not import anything from low-level modules – both should depend only on abstractions
 - abstractions should not depend on details – details should depend on abstractions

traditional dependencies are top down:
high-level modules depend directly on low-level modules

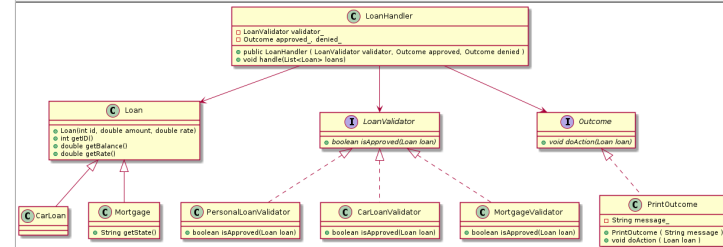


- importance
 - loose coupling – reduces dependencies between classes
 - flexibility – enables changes to implementations without affecting clients
 - testability – enables substitution of test doubles
 - maintainability – makes code easier to understand and modify

https://en.wikipedia.org/wiki/Dependency_inversion_principle

12

DIP Example



- the LoanValidator and Outcome interfaces are DIP
 - the instance variable types in LoanHandler are the abstractions, not the concrete classes
 - the concrete instances of the validator and the outcomes are passed in – not hardcoded into LoanHandler

CPSC 329: Software Development • Fall 2026

103

DIP Example

DIP isn't only about extensibility and fixing OCP

```
class Loan {
    private LocalDate dueDate_;
    public boolean isOverdue () {
        return LocalDate.now().isAfter(dueDate_);
    }
}
```

- Loan depends on the concrete details of the system clock
 - the system clock isn't likely to change, but we have a testing problem...

CPSC 329: Software Development • Fall 2026

104

DIP Example

having Loan instead depend on an abstraction
allows substituting the concrete details for testing

- production version – still using the system clock...

```
interface Clock {
    LocalDate today ();
}

class SystemClock implements Clock {
    public LocalDate today () { return LocalDate.now(); }
}

class Loan {
    private LocalDate dueDate_;
    private Clock clock_;

    public Loan ( LocalDate dueDate, Clock clock ) {
        dueDate_ = dueDate;
        clock_ = clock;
    }

    public boolean isOverdue() {
        return clock_.today().isAfter(dueDate_);
    }
}
```

CPSC 329: Software Development • Fall 2026

105

DIP Example

having Loan instead depend on an abstraction
allows substituting the concrete details for testing

- ...but can now substitute a controllable clock for testing

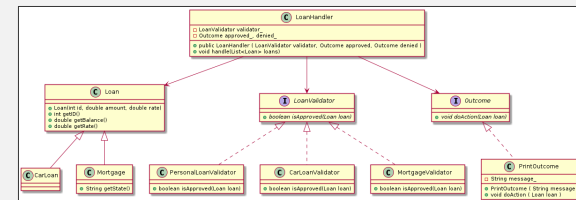
```
class FixedClock implements Clock {
    private LocalDate fixedDate_;
    public FixedClock ( LocalDate date ) { fixedDate_ = date; }
    public LocalDate today () { return fixedDate_; }
}

@Test
void loanIsNotOverdueOnItsDueDate () {
    Loan loan = new Loan(LocalDate.of(2026, 10, 1),
        new FixedClock(LocalDate.of(2026, 10, 1)));
    assertFalse(loan.isOverdue());
}

@Test
void loanIsOverdueTheDayAfter () {
    Loan loan = new Loan(LocalDate.of(2026, 10, 1),
        new FixedClock(LocalDate.of(2026, 10, 2)));
    assertTrue(loan.isOverdue());
}
```

OCP vs DIP

- OCP is about whether new behavior can be added without modifying existing code
- DIP is about which way the dependencies point



OCP –

- we can write new kinds of validators and outcomes and plug them into **LoanHandler** without touching **LoanHandler**'s code

DIP –

- the instance variable types in **LoanHandler** are the abstractions, not the concrete classes
- the concrete instances of the validator and the outcomes are passed in – not hardcoded into **LoanHandler**