Supple Design

Marko Schütz-Schmuck

Department of Mathematical Sciences
University of Puerto Rico at Mayagüez
Mayagüez, PR

December 18, 2017

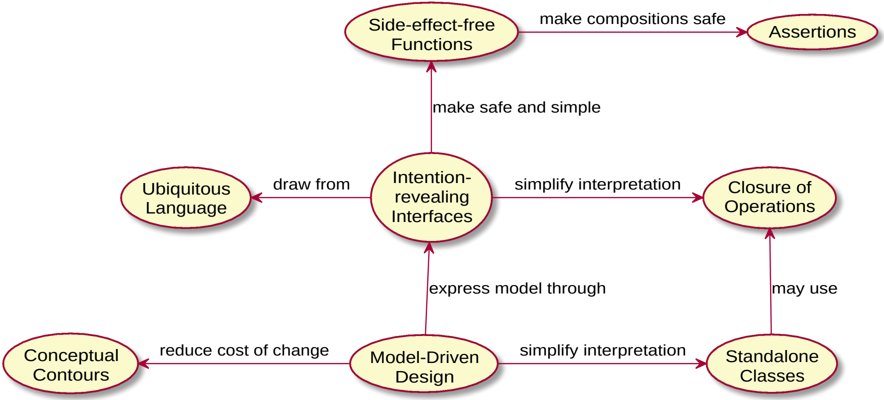
overview

- complex software without good design: hard to refactor or combine parts
- developer with doubts about full implications of any part of implementation will duplicate code
- monolithic design forces duplication: cannot be recombined
- to have a project accelerate requires a design that developers like to work with
- supple design complements deep modeling
- what kind of design should you try to arrive at?
- what experiments should you try on the way?

necessity of a supple design

- software that really empowers people often has a simple design
- "Simplicity is a great virtue but it requires hard work to achieve it and education to appreciate it. And to make matters worse: complexity sells better." Dijkstra
- developers have at least two roles when working with a design
  - client developer needs flexible and minimal set of loosely coupled concepts to express a range of scenarios from the domain
  - "foundation" developer needs to easily understand the design in order to be able to change it
- early versions of the design are usually stiff

patterns for supple design



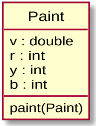
intention-revealing interfaces

- without a clear connection to the model it is difficult to understand the effect of code or anticipate the effect of a change
- if the interface does not tell the client developer what (s)he needs to know to use an object, the developer will have to dig into the internals to understand the details
- value of encapsulation is lost
- infer the purpose of an object of operation based on its implementation: may infer incidental purpose
  - client works for the moment . . .
- names must reflect purpose/concept
- all public elements of a design make up its interface
- each name can be used to reveal intention of design: type names, method names, parameter names, . . .

guide

- anything that can be named should reflect the effect and purpose in the name without revealing the means by which it is achieved
- relieves client developer of need to understand internals
- names should use ubiquitous language
- writing a test before creating a behavior focuses the developer to think like the client
- speak in terms of intentions, not means

example: refactoring paint mixing application 1



- need to read implementation to know what `paint` does

```
public void paint(Paint paint) {
  v = v + paint.getV(); // volume is summed
  // Omitted many lines of color mixing logic e
  // with the assignment of new r, b, and y va
}
```

- seems to combine two paints
- result has larger volume and mixed color

example: refactoring paint mixing application 2

- write test

```
public void testPaint() {
  // Create pure yellow paint of volume=100
  Paint yellow = new Paint(100.0, 0, 50, 0);
  // Create pure blue paint of volume=100
  Paint blue = new Paint(100.0, 0, 0, 50);

  // Mix the blue into the yellow
  yellow.paint(blue);

  // should have volume 200.0 of green paint
  assertEquals(200.0, yellow.getV(), 0.01);
  assertEquals(25, yellow.getB());
  assertEquals(25, yellow.getY());
  assertEquals(0, yellow.getR());
}
```

example: refactoring paint mixing application 3

- rewrite test to how we want to use paint

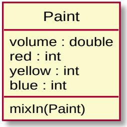
```
public void testPaint() {
  // Start with volume 100 of pure yellow paint
  Paint ourPaint = new Paint(100.0, 0, 50, 0);
  // Make volume 100 of pure blue paint
  Paint blue = new Paint(100.0, 0, 0, 50);

  // Mix the blue into the yellow
  ourPaint.mixIn(blue);

  // should have volume 200.0 of green paint
  assertEquals(200.0, ourPaint.getVolume(), 0.0
  assertEquals(25, ourPaint.getBlue());
  assertEquals(25, ourPaint.getYellow());
  assertEquals(0, ourPaint.getRed());
}
```

example: refactoring paint mixing application 4

- refactor paint so tests pass



- new method name together with example from test allows reader to get started
- allows reader of client code to interpret intent

side-effect-free functions

- queries obtain information, commands affect state change
- side-effect: any program state change other than expression evaluation
- operations that return results without producing side-effects are expressions or functions
- referential transparency: a function call can be replace with the value it returns without changing behavior

mitigate problem of commands

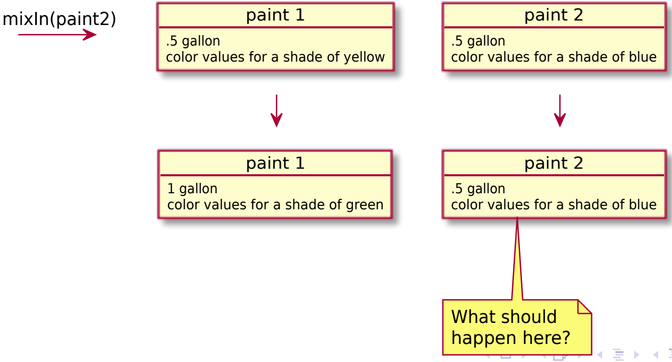
- segregate commands and queries
  - make commands as simple as possible
  - do queries and calculations only in functions that cause no observable side-effect
- look for model, design not modifying an existing object at all
  - instead value objects representing the result of computation can be returned
- since value objects are immutable, all their operations (except initialization) are functions
  - $\Longrightarrow$ easier to test and reason about
- operations involving query and command aspects can be separated
- *after* segregation: consider moving complex calculations into a value object
- side-effect can often be removed, by deriving a new value object instead of changing state

guidelines

- place as much as possible of the logic into functions
- segregate commands into very simple operations
- move complex logic into value objects
- side-effect-free functions allow safe combination of operations

refactoring the paint mixing application

```java
public void mixIn(Paint other) {
    volume = volume.plus(other.getVolume());
    // Many lines of complicated color-mixing logic
    // ending with the assignment of new red, blue,
    // and yellow values.
}
```

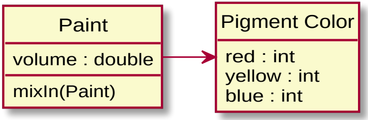


design critique

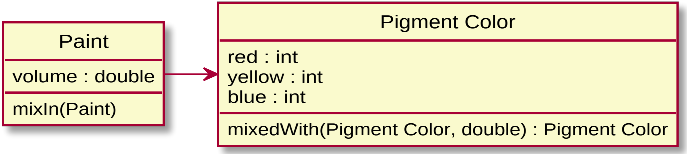
- segregates query from command
- volume of paint 2 is unchanged
- after `mixIn` we have more total volume than before
    - does not fit with domain understanding

more refactoring

- experiment: make color an explicit object
- using "pigment color" to distinguish from "light color"



- more insight communicated, but still same computation in `mixIn` method
- take behavior related to color to new object
- pigment color is value object



mixIn uses mixWith

```
public void mixIn(Paint other) {
  volume = volume + other.getVolume();
  double ratio = other.getVolume() / volume;
  pigmentColor = pigmentColor.mixedWith(
                        other.pigmentColor(), ratio);
}
```

- modification code is simple
- pigment color provides side-effect-free function
  - easy to test and understand
  - safe to combine with other operations

motivation for assertions

- some side effects will be left in the commands on entities
- must be understood
- *assertions* make side effects explicit
- simple command easy enough to understand
- a command may invoke other commands
- developer must understand effect of each underlying command
- object interfaces do not restrict side effects
  - two sub-classes that implement the same interface can have different side effects
  - abstraction and polymorphism is lost when a developer wants to know which one (s)he is using
- need to understand all resulting side effects without delving into internals

assertions

- post-conditions describe side effects of an operation
- pre-conditions describe what the client needs to ensure for the operation to proceed
- class invariants make assertions about the state of an object at the end of any operation
- invariants can be declared for entire aggregates
- describe state, not procedures

guidelines for assertions

- state post-conditions of operations and invariants of classes and aggregates
- if assertions cannot be written directly in the programming language, write automated tests
- write assertions in documentation and diagrams
- seek models with coherent sets of concepts: developers can infer intended assertions

using assertions

- most languages do not support assertions directly
- assertions (even without language support) help thinking about design
- automated unit-tests can compensate lack of language support somewhat
- clearly stated invariants and pre- and post-conditions facilitate understanding consequences of use (operations, objects)

applying assertions to paint example 1

- current implementation does not consume mixed in paint
- `mixIn` could modify argument: risky
- what is the post-condition currently?
  After `p1.mixIn(p2)`: `p1.volume` is increased by `p2.volume`, `p1.volume` is unchanged
- could use invariant: total volume of paint is unchanged by mixing
- original design supports report of original paints that were mixed together
- making the volume model consistent with real world, would make it unsuitable for this requirement
- hidden concept "mixed paint" as opposed to "stock paint"

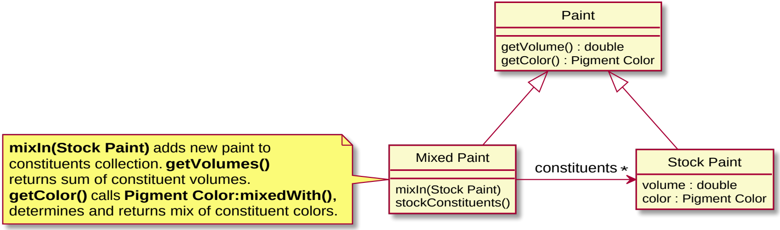
applying assertions to paint example 2

```
public void testMixingVolume {
  PigmentColor yellow = new PigmentColor(0, 50, 0);
  PigmentColor blue = new PigmentColor(0, 0, 50);

  StockPaint paint1 = new StockPaint(1.0, yellow);
  StockPaint paint2 = new StockPaint(1.5, blue);
  MixedPaint mix = new MixedPaint();

  mix.mixIn(paint1);
  mix.mixIn(paint2);
  assertEquals(2.5, mix.getVolume(), 0.01);
}
```

applying assertions to paint example 3



motivation for conceptual contours

- elements of model or design embedded in monolithic construct: functionality gets duplicated
  - external interface will not say everything the clients wants to know
  - meaning hard to understand, because several concepts are mixed
- breaking down classes and methods can complicate client
  - forces understanding how tiny pieces fit together
  - concepts might get lost completely
- when we find a model that resonates with some part of the domain, it likely fits with other parts discovered later

finding conceptual contours 1

- repeated refactoring leads to suppleness, because conceptual contours emerge
  - code is repeatedly adapted to newly understood concepts or requirements
- goal of high cohesion/low coupling applies at all levels: concepts as much as code
- frequently appeal to your intuition about the domain
- does this echo some contour of the domain or is it incident to the implementation?
- look for conceptually meaningful unit of functionality
  - design will be flexible and understandable
- example: "addition" of two domain objects has meaning in the domain
  - implement `add()` function, do not break in two steps

finding conceptual contours 2

- if the users do not add e.g. individual red pigments, but instead combine complete paints, our model should not separate these pigments
- clustering things together that do not need to be separated avoids clutter and focuses on the elements that are meant to be recombined

guidelines for conceptual contours

- decompose design units (operations, interfaces, classes, aggregates) into cohesive units
- use your intuition of the important divisions in the domain
- watch for change and stability across refactorings
- look for conceptual contours that explain these observed patterns

judging fitness of conceptual contours

- successive refactorings are localized, do not shake multiple broad concepts $\implies$ model fits
- a requirement forces excessive change in the breakdown of the objects and methods $\implies$ domain understanding needs refinement

summary conceptual contours

- intention-revealing interfaces allow clients to use objects as units of meaning rather than mechanism
- side-effect-free functions and assertions make support safe complex combinations of units
- conceptual contours stabilize parts of the model and make units more intuitive to use and combine

motivation for standalone classes

- dependencies/associations create cognitive load
- two dependencies $\implies$ need to consider three classes, nature of their relationships
- dependencies on dependencies: also need to be considered
- modules and aggregates help limit and control inter-dependencies
- even within a module interpreting design becomes difficult as dependencies as added
- dependencies limit the design complexity a developer can handle
- implicit concepts add even more to this than explicit references

sources of cognitive load

- the creation of pigment color did not increase the number of concepts nor dependencies
- made concepts mode explicit and easier to understand
- `size()` on a collection is a basic concept and a simple integer $\Longrightarrow$ adds very little cognitive load
- every dependency is suspect until proven basic to the concept modeled

goals for standalone classes

- low coupling is essential to object design
- aim to eliminate all non-essential dependencies
- class becomes self-contained and can be studied alone
- self-contained classes ease understanding of module
- dependencies within the module are less harmful than external ones
- do not dumb down the model in order to eliminate dependencies
- a standalone class is an extreme of low coupling

closure under operations

- stripping interfaces down to the primitives impoverishes them
- unnecessary dependencies and even concepts get introduced at interfaces
- most interesting objects end up doing something that is not easily characterized by primitives

guidelines for closure under operations

- provided it fits: define an operation whose return type is the same as the type of its parameters
- if the implementer has state, it is itself an argument to the operation $\implies$ type of arguments and return value should be the implementer
- implementer type is closed under the operation
- high-level interface without dependencies on other concepts

using closure under operations

- rarely used with entities
  - entities have identity and life cycle in the domain, so they are not normally the result of computation
- can abstract a type to close it under operations
  - specific arguments can be of different concrete types
- helps even if reached only in part
  - argument matches implementer, but return does not, or return type matches implementer, but argument does not

example closure under operations 1

- to select a subset in Java

```
Set employees = (some Set of Employee objects);
Set lowPaidEmployees = new HashSet();
Iterator it = employees.iterator();
while (it.hasNext()) {
  Employee anEmployee = it.next();
  if (anEmployee.salary() < 40000)
    lowPaidEmployees.add(anEmployee);
}
```

- determine subset of set
- but two concepts iterator and set, not closed

example closure under operations 2

- in Smalltalk

```
employees := (some Set of Employee objects).
lowPaidEmployees := employees select:
    [:anEmployee | anEmployee salary < 40000].
```

- while `select:` returns a collection, the argument is a "block", not closed
- blocks belong to foundation of Smalltalk $\Longrightarrow$ do not increase cognitive load
- return type is a collection so selections can be strung together

motivation for declarative design

- assertions help improve designs
  - cannot rule out e.g. additional side-effects that the assertion did not explicitly exclude
- a lot of boilerplate code has to be written for every application
- ideal: write a program (part thereof) as executable specification

problems with code generation from declarative model description

- declaration language not expressive enough to express everything needed with a framework that makes extension difficult
- code generation techniques that merge generated and handwritten code, so regeneration becomes destructive
- unintended consequence: model and application are dumbed dow as developers are trapped by limitations of framework

rule-based programming

- uses rules and inference engine
- provides declarative design
- some side-effects are needed: "control predicates"
- no longer purely declarative: programmer has to be careful to keep effect of code obvious just as with OOP

domain-specific languages

- client code written in language tailor-made to a specific model of a particular domain
- shipping DSL would include terms such as *cargo* or *route*
- program in DSL then compiles to e.g. regular OO language
- programs expressed in DSL can be very expressive

drawbacks of DSLs

- to refine the model requires changing the language
  - modify underlying class libraries
  - change grammar rules & other language translation features
- too many skills to ask for?
- refactoring to the modified DSL might be difficult
- tendency of two different teams: framework builders / application builders
- support for DSLs: OCaml, Scheme, Clojure, Idris

declarative style of design 1

- many benefits of declarative design follow from combine-able elements that communicate their meaning and characterized/obvious effects or no effects
- specification is an adaptation of predicates
- logical combination of specifications is specification: closed under these operations

```java
public interface Specification {
   boolean isSatisfiedBy(Object candidate);
}
```

```
interface Specification (t : Type) where
  isSatisfiedBy : t -> Bool
```

declarative style of design 2

```
public class ContainerSpecification
    implements Specification {
  private ContainerFeature requiredFeature;

  public ContainerSpecification(ContainerFeature
      required) {
    requiredFeature = required;
  }

  boolean isSatisfiedBy(Object candidate){
    if (!candidate instanceof Container) return false;

    return (Container)aContainer.getFeatures()
          .contains(requiredFeature);
  }
}
```

- extend interface to

```
public interface Specification {
  boolean isSatisfiedBy(Object candidate);

  Specification and(Specification other);
  Specification or(Specification other);
  Specification not();
}
```

- can use

```
Specification ventilated = new ContainerSpecification(
                                 VENTILATED);
Specification armored = new ContainerSpecification(
                              ARMORED);
Specification both = ventilated.and(armored);
```

angles of attack

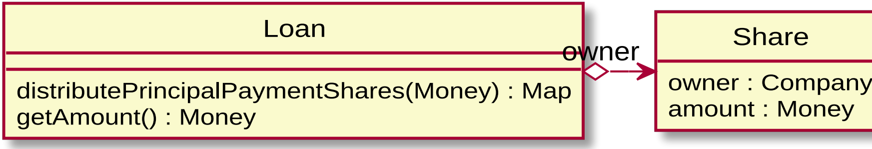
- how do we approach making a large system supple?
- carve off sub-domains
  - pick some aspects, factor out, work over, separate
- example: complex rules restricting state changes
  - pull out into separate model or simple framework
    - extracted model becomes clearer
    - rest is smaller
- go after one part at a time

draw on established formalism

- cannot create a tight conceptual framework every day
- use and adapt conceptual systems that are long established and are refined and distilled over centuries
- accounting, math, ...

example: shares math 1

- syndicated loan system
- requirement: when the borrower makes a principal payment, by default, money is prorated according to lenders' shares in the loan
- initial design



example: shares math 2

```
public class Loan {
  private Map shares;

  public Map distributePrincipalPayment(double
      paymentAmount) {
    Map paymentShares = new HashMap();
    Map loanShares = getShares();
    double total = getAmount();
    Iterator it = loanShares.keySet().iterator();
    while(it.hasNext()) {
      Object owner = it.next();
      double initialLoanShareAmount = getShareAmount(
                                                    owner);
      double paymentShareAmount =
        initialLoanShareAmount / total * paymentAmount
      Share paymentShare =
        new Share(owner, paymentShareAmount);
```

example: shares math 3

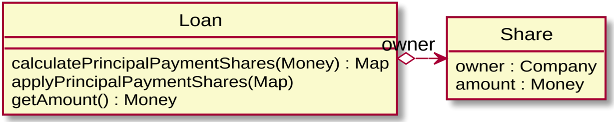
```
    paymentShares.put(owner, paymentShare);
    double newLoanShareAmount =
      initialLoanShareAmount - paymentShareAmount;
    Share newLoanShare =
      new Share(owner, newLoanShareAmount);
    loanShares.put(owner, newLoanShare);
  }
  return paymentShares;
}
```

example: shares math 4

```
  public double getAmount() {
    Map loanShares = getShares();
    double total = 0.0;
    Iterator it = loanShares.keySet().iterator();
    while(it.hasNext()) {
      Share loanShare = (Share)loanShares.get(it.next(
      total = total + loanShare.getAmount();
    }
    return total;
  }
}
```

example: shares math 5

- `distributePrincipalPayment()` calculates and modifies object
- separate commands and side-effect-free functions



example: shares math 6

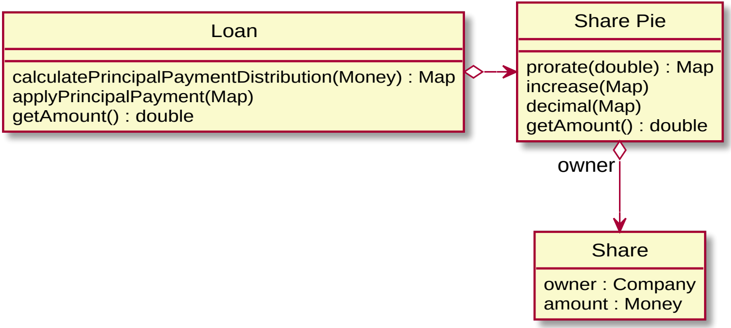
```
public void applyPrincipalPaymentShares(
    Map paymentShares) {
  Map loanShares = getShares();
  Iterator it = paymentShares.keySet().iterator();
  while(it.hasNext()) {
    Object lender = it.next();
    Share paymentShare = (Share)paymentShares
                                    .get(lender);
    Share loanShare = (Share)loanShares.get(lender);
    double newLoanShareAmount = loanShare.getAmount()
      - paymentShare.getAmount();
    Share newLoanShare =
      new Share(lender, newLoanShareAmount);
    loanShares.put(lender, newLoanShare);
  }
}
```

example: shares math 7

```
public Map calculatePrincipalPaymentShares(
    double paymentAmount) {
  Map paymentShares = new HashMap();
  Map loanShares = getShares();
  double total = getAmount();
  Iterator it = loanShares.keySet().iterator();
  while(it.hasNext()) {
    Object lender = it.next();
    Share loanShare = (Share) loanShares.get(lender);
    double paymentShareAmount =
      loanShare.getAmount() / total * paymentAmount;
    Share paymentShare
      = new Share(lender, paymentShareAmount);
    paymentShares.put(lender, paymentShare);
  }
  return paymentShares;
}
```

example: shares math 8

- shares are passive
  - not handled individually, but in groups
- is there a missing implicit concept?
- share pie represents distribution of the loan
  - entity in loan aggregate



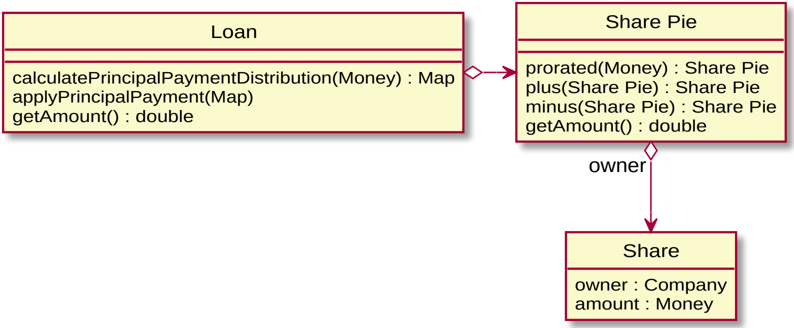
example: shares math 9

```
public class Loan {
  private SharePie shares;
  //...
  public Map calculatePrincipalPaymentDistribution(
      double paymentAmount) {
    return getShares().prorated(paymentAmount);
  }
  public void applyPrincipalPayment(
      Map paymentShares) {
    shares.decrease(paymentShares);
  }
}
```

- loan simplified, share calculation centralized in value object with precisely this responsibility

example: shares math 10

- experiment: what would happen if we made share pie a value object?
- `increase(Map)` and `decrease(Map)` need to change because of immutability
- even further: try for closure under operations
  - add two share pies together to get a new share pie



example: shares math 11

```
public class SharePie {
  private Map shares = new HashMap();
  //...
  public double getAmount() {
    double total = 0.0;
    Iterator it = shares.keySet().iterator();
    while(it.hasNext()) {
      //The whole is equal to the sum of its parts.
      Share loanShare = getShare(it.next());
      total = total + loanShare.getAmount();
    }
    return total;
  }
```

example: shares math 12

```
public SharePie minus(SharePie otherShares) {
  //difference in two share pies is the
  //difference in each owner's shares
  SharePie result = new SharePie();
  Set owners = new HashSet();
  owners.addAll(getOwners());
  owners.addAll(otherShares.getOwners());
  Iterator it = owners.iterator();
  while(it.hasNext()) {
    Object owner = it.next();
    double resultShareAmount = getAmount(owner)
      - otherShares.getAmount(owner);
    result.add(owner, resultShareAmount);
  }
  return result;
}
```

example: shares math 13

```
public SharePie plus(SharePie otherShares) {
 //sum of share pies, is sum of each owner's
 //shares. ...
}
public SharePie prorated(double amountToProrate) {
  SharePie proration = new SharePie();
  double basis = getAmount();
  Iterator it = shares.keySet().iterator();
  while(it.hasNext()) {
    Object owner = it.next();
    Share share = getShare(owner);
      double proratedShareAmount =
        share.getAmount() / basis * amountToProrate;
      proration.add(owner, proratedShareAmount);
  }
  return proration;
}
```

example: shares math 14

```
public class Loan {
  private SharePie shares;
  //...
  public SharePie calculatePrincipalPaymentDistributio
      double paymentAmount) {
    return shares.prorated(paymentAmount);
  }
  public void applyPrincipalPayment(
      SharePie paymentShares) {
    setShares(shares.minus(paymentShares));
  }
}
```

– code now reads as a conceptual definition of the business transaction rather than a calculation

example: shares math 15

- can now incorporate transaction types based on our domain language

```
public class Facility {
  private SharePie shares;
  //...
  public SharePie calculateDrawdownDefaultDistribution
      double drawdownAmount) {
    return shares.prorated(drawdownAmount);
  }
}

public class Loan {
  //...
  public void applyDrawdown(SharePie drawdownShares) {
    setShares(shares.plus(drawdownShares));
  }
}
```

example: shares math 16

- characteristics of share pie design that make it easy to recombine the code
  - complex logic encapsulated in specialized value objects with side-effect-free functions
    - we can build analytical features, because use immutable objects (before we changed the object)
  - state-modifying operations are simple and characterized by assertions
  - model concepts are decoupled; operations entangle a minimum of other types
    - some operations are closed
    - others use basic types: add little to cognitive load
    - only associated with one other class: share
  - familiar formalism makes protocol easy to grasp