OOP – Design Principles

Hönnun og smíði hugbúnaðar

Haust 2022

Design principles

- KISS – Keep it simple stupid
- YAGNI – You aren't going to need it
- Rule of three
- DRY – Don't Repeat Yourself
- Knuth's Optimization Principle
- Loose Coupling
- Law of Demeter
- Program to an interface not an implementation
- Encapsulate what varies
- Favor composition over inheritance
- SOLID principles

Hvað eru design principles

- Reglur sem er gott að hafa í huga
- Myndar samhljóða álit á *clean code*
- Guidelines en ekki harðsettar reglur

Einkenni af slæmri hönnun

- Rigidity
  - Kerfið á erfitt með að breytast
  - Einn breyting leiðir til fleiri
- Fragility
  - Kerfið er brothætt
  - Breyting leiðir til villu
- Immobility
  - Erfitt að brjóta kerfið upp
- Viscosity
  - Erfitt að gera rétta hlutinn
  - Auðveldara að gera *hakkfix*
- Needless Complexity
  - Overdesign / Overengineering
- Needless Repetition
  - Gerir kerfið Rigid og Fragile
- Opacity
  - Óskipulag í því sem kóðinn er að reyna að *segja*
  - kóðinn lýsir ekki vel því sem hann gerir
  - Readability er king

Code smell / Design smell

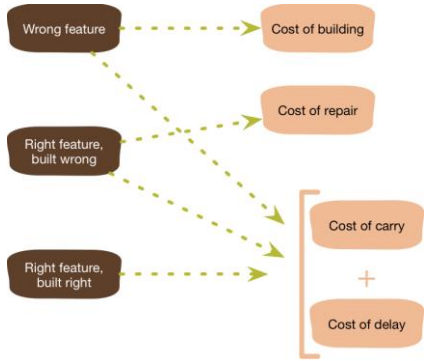
- Code smell er *sjúkdómseinkenni* í kóðanum þínum
- Fundið með *the code sense*
- *Ljótur* kóði/hönnun
- Orsakast of design principle brotum
- Orsakast af *needless complexity*
  - T.d. ofnot á design principles / patterns

KISS – Keep It Simple Stupid

- *“Everything should be made as simple as possible, but no simpler.”*
- Ekki simple að fylgja KISS
- Forðist *needless complexity*
- *“Clever is the enemy of clarity”*
- læsileiki er king
- Öll principle-inn í þessum glærum stuðla að simplicity

YAGNI – You aren‘t going to need it

- Ekki bæta við functionality fyrr en þú þarft það
- Ekki reyna að spá of mikið fyrir framtíðinni
- Requirements breytsat



https://martinfowler.com/bliki/Yagni.html

Rule of Three

- Hvað segir það?
  - Ekki minnka endurtekt með abstraction fyrr en þriðja dæmið er komið
- Af hverju?
  - Abstraction af tveimur dæmum-> overfitted **eða** overgeneralized hönnun
  - Endurtekt betra en slæmt abstraction
- Guideline en ekki harðsett regla
  - Eins og öll principle þá er þetta ekki harðsett regla
  - Notið readability og simplicity sem leiðarljós
  - Stundum þarftu 5 dæmi stundum þarftu bara 2, þrír er engin gullin tala

Rule of Three dæmi

- Einfaldur scraper fyrir bankasíður
- Hvað ef ég ætla síðan að bæta við nýjum banka?
  - Sem auðkennir öðruvísi?
  - Notar POST en ekki GET?
  - Þarf að fara á fleiri en eina síðu?
  - Etc.
- Dæmið er overfitt-að

```
class BaseScraper:
    def __init__(self, username, password):
        self._username = username
  self._password = password

    def scrape(self):
      session = requests.Session()
  sessions.get(self._LOGIN_URL,
               data={self._USERNAME_FORM_KEY: self._username,
               self._PASSWORD_FORM_KEY: self._password})
  sessions.get(self._STATEMENT_URL)


class ChaseScraper(BaseScraper):
    _LOGIN_URL = 'https://chase.com/rest/login.aspx'
    _STATEMENT_URL = 'https://chase.com/rest/download_current_statement.aspx'
    _USERNAME_FORM_KEY = 'username'
    _PASSWORD_FORM_KEY = 'password'


class CitibankScraper(BaseScraper):
    _LOGIN_URL = 'https://citibank.com/cgi-bin/login.pl'
    _STATEMENT_URL = 'https://citibank.com/cgi-bin/download-stmt.pl'
    _USERNAME_FORM_KEY = 'user'
    _PASSWORD_FORM_KEY = 'pass'
```

https://erikbern.com/2017/08/29/the-software-engineering-rule-of-3.html

DRY – Don‘t Repeat Yourself

- Ekki endurtaka þig
  - Reyndu að hafa kóða endurtekt eins lítið og mögulegt
  - Einangra það sem er sameiginlegt frá því sem er öðruvísi
  - Finna abstraction fyrir endurtekt
- Ástæða
  - Einfaldleiki, læsileiki, maintainability
- Mótsögn við Rule of Three
  - Principles ekki harðsettar reglur
  - Notið ykkar eigin huglæga mat
  - Forritarar eiga til með að abstract-a of snemma
    - Slæmt abstraction leiðir til illa maintainable, illa læsilegan og flókin kóða.
  - Áður en þið hugsið um DRY, hugsið um Rule of Three

Knuth‘s Optimization Principle

- Quote
  - *“premature optimization is the root of all evil.” – Donald Knuth*
  - *“Never sacrifice clarity for some perceived efficiency.”*
  - *“clever is the enemy of clear”*

- Readability er king
  - Ekki fórna læsileika fyrir optimization (nema ef alvöru þörf er á)

- Flest optimization eru óþarfi og gefa engan marktækan hraðamun

Knuth's Optimization Principle dæmi

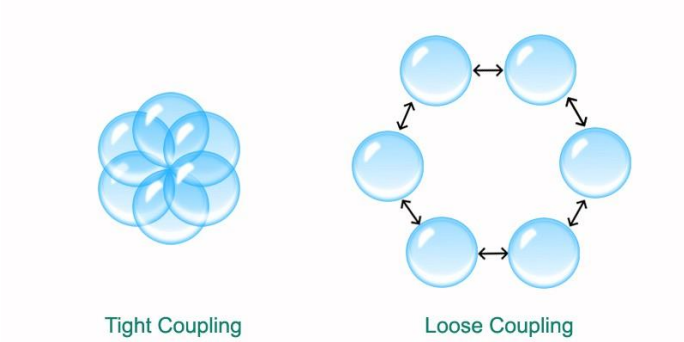
```
[HttpPut]
[Route(template:"api/orders/{orderId}")]
0 references
public IActionResult UpdatePrice(int orderId, decimal newPrice)
{
    var order = dbContext.Orders.Find(orderId);
    order.Price = newPrice;
    dbContext.SaveChanges();

    return Ok();
}
```

vs

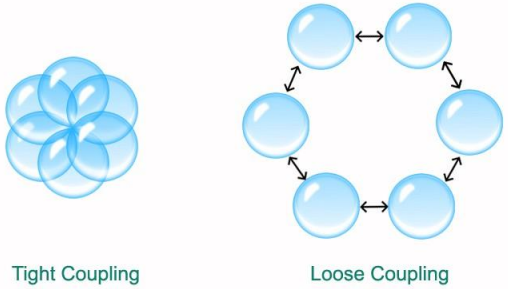
```
[HttpPut]
[Route(template:"api/orders/{orderId}")]
0 references
public IActionResult UpdatePrice(int orderId, decimal newPrice)
{
    dbContext.Database.ExecuteSqlRaw(
        $@"update orders
           set price={newPrice}
           where Id={orderId}"
        );

    return Ok();
}
```

Strive for loosely coupled design

- Componentar vita eins lítið um hvort annað og mögulega
  - Hafa samskipti en vita lítið um innri virkni
  - Gera ekki ályktanir
  - Kerfið er byggt upp af mörgum litlum sjálfstæðum einingum
  - Klasi ætti t.d. bara að tala við interface-ið á öðrum klasa
- Skýr mörk
  - Componentar ættu að hafa skýr mörk
  - Klasar verða eins og púsluspil
  - Eins og bíll, hægt að skipta um parta án þess að rífa allt í sundur og endursmíða
- Vörn gegn breytingum
  - Breytingar í einum klasa leiða ekki til breytingar í öðrum



Loose coupling kostir



- Flexibility
  - Kerfið verður plug and play
- Endurnýtanleiki
  - Auðvelt að endurnýta klasa í mörgum samhengjum
- Breytingar verða auðveldari og hættu minni
  - Breyting í einum klasa ætti ekki að hafa afleiðingar í öðrum
- Auðveldara að testa
  - Auðvelt að test-a klasa sér í lagi

Loose Coupling, Hvernig?

- Látið klasa vita eins lítið um aðra klasa og mögulega
- Engar ályktanir um innri virkni
- Látið klasa vera einangraðir með sitt eigið *scope*
- Fylgið Design Principles
- Fylgið Hönnunarmynstrum

Decoupled vs Coupled design dæmi


```python
class Car:
    def __init__(self):
        self.engine = DiselEngine()

    def accelerate(self):
        self.engine.do_engine_stuff()
```

Car og DieselEngine er coupled saman ✗

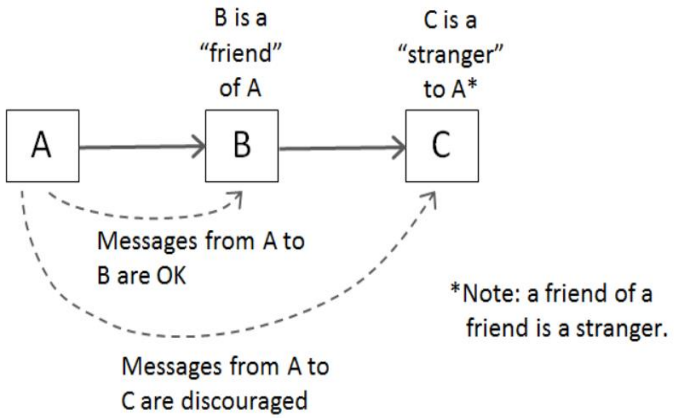
vs

```python
class Car:
    def __init__(self, engine: IEngine):
        self.engine = engine

    def accelerate(self):
        self.engine.do_engine_stuff()
```

Car og engine útfærlsan eru decoupluð frá hvort öðru ✓

Law of Demeter

- Líka þekkt sem *principle of least knowledge*
  - Klasi ætti að hafa eins litlar upplýsingar um aðra klasa
  - Klasi ætti að hafa upplýsingar um fáa klasa
- don‘t talk to strangers
  - Hver klasi ætti bara að tala við nánustu vini sína
- Látum fall *f* tilheyra hlut *H:*
  - Fall *f* ætti aðeins að nota föll af eftirfarandi**:**
    - *H*
    - Hlut sem *H* á tilvik af
    - Hlut sem er inntak af *f*
    - Hlut sem er búinn til af *f*
  - Fall *f* ætti því t.d. ekki að kalla á hlut *b* sem skilast af falli sem *f* **má** kall á
- Kostir:
  - Stuðlar að *Loose coupling*
  - Auðveldara að maintain-a
  - Auðveldara að test-a
- Gallar:
  - Leyðir til meira overhead því oft þarf að búa til wrapper föll t.d. af myndinn að ofan: ef C er með fall f_c þá þarf B möguleg að búa til sérstakt wrapper fall sem kallar á f_c svo A megi kalla á virknina í f_c



Law of Demeter dæmi 1

- Violation af Law of Demeter `obj.getX().getY().getZ().doSomething()`

- Ekki brot af Law of Demeter `obj.doSomething()`

Law of Demeter dæmi 2

Er þetta brot á Law of Demeter?

```
Options opts = ctxt.getOptions();
File scratchDir = opts.getScratchDir();
final String outputDir = scratchDir.getAbsolutePath();
```

Dæmi úr clean code eftir Robert C. Martin

En ef það er skrifað svona?

```
final String outputDir = ctxt.options.scratchDir.absolutePath;
```

Law of Demeter dæmi 2, svar

- Ef Data klasar
  - í raun engin hættuleg coupling –> Ekki brot

```python
@dataclass
class ScratchDir:
    absolute_path: str
```

```python
@dataclass
class Options:
    scratch_dir: ScratchDir
```

```python
@dataclass
class Context:
    options: Options
```

```python
cntxt = Context()
cntxt.options.scratch_dir.absolute_path
```

- Ef klasar með virkni
  - Alvöru coupling -> Brot á Law of Demeter
  - Hvað er þá fixið?

Law of Demeter dæmi 2, svar 2

- Ef þetta eru klasar með virkni hvað er þá besta lausnin?

```
ctxt.getAbsolutePathOfScratchDirectoryOption();
```

- Eða kannski

```
ctx.getScratchDirectoryOption().getAbsolutePath()
```

- En ef við erum með þetta context?:

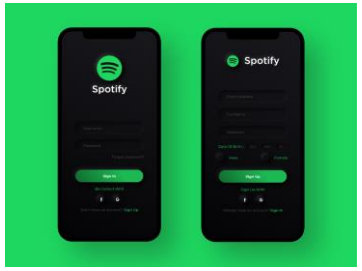
```
String outFile = outputDir + "/" + className.replace('.', '/') + ".class";
FileOutputStream fout = new FileOutputStream(outFile);
BufferedOutputStream bos = new BufferedOutputStream(fout);
```

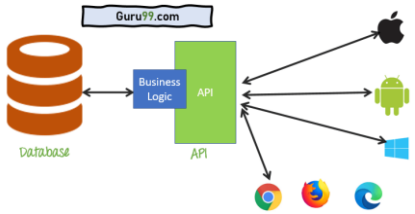
Þá er þetta eflaust betri lausn:

```
BufferedOutputStream bos = ctxt.createScratchFileStream(classFileName);
```

Program to an interface not implementation

- Fyrst, hvað er interface?
  - Almenn skilgreining: eitthvað *entity* sem tengir saman tvö mismunandi kerfi





API(Application Programming Interface)

```
public interface IFile
{
    void ReadFile();
    void WriteFile(string text);
}
```

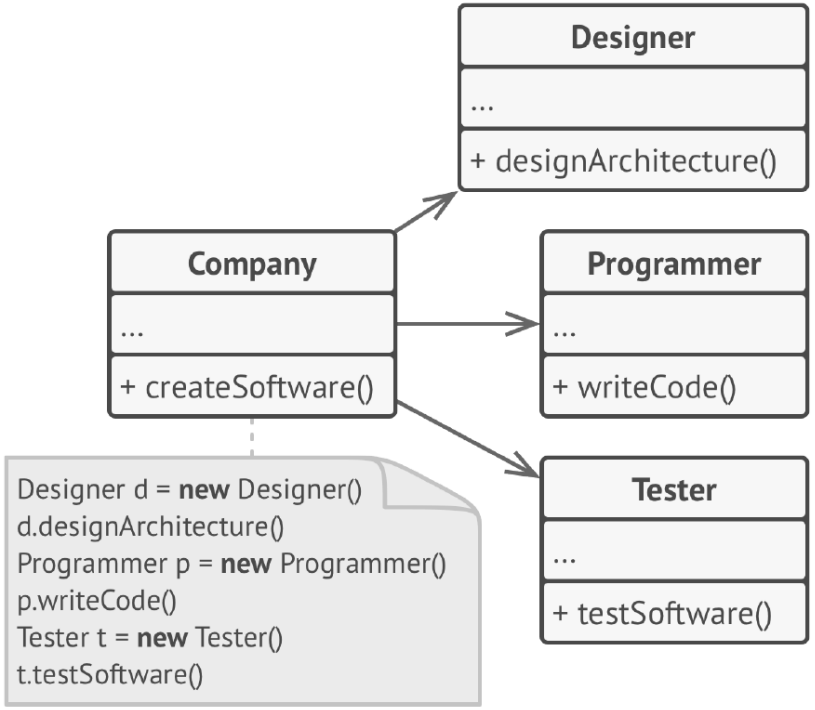
Explicit interface í kóða (C#)

```
class IFile(ABC):
    @abstractmethod
    def ReadFile(self) -> None: pass

    @abstractmethod
    def WriteFile(self, text: str) -> None: pass
```

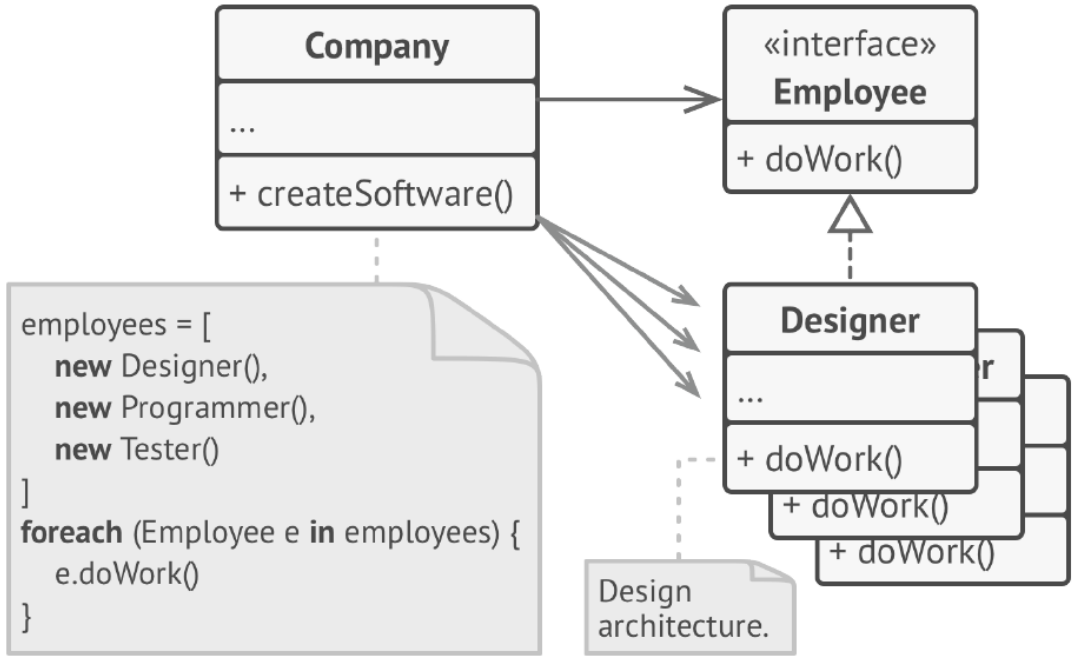
Explicit interface í kóða (python)

  - Interface í kóða
    - Explicit interface
      - Virkar sem samningur sem concrete klasar þurfa að útfæra
    - Implicit Interface
      - Interface getur líka talist sem public föll/property í concrete klösum

Interface dæmi 1



BEFORE: all classes are tightly coupled.



BETTER: polymorphism helped us simplify the code, but the rest of the `Company` *class still depends on the concrete employee classes.*

Dæmi úr dive into design patterns

Interface dæmi 2

```python
class Cat:
    def make_noise(self):
        print("rawr")


class Car:
    def make_noise(self):
        print("vroom vroom")


noise_makers = [Cat(), Car()]
for object in noise_makers:
    object.make_noise()
```

implicit interface (python)

```python
from abc import ABC, abstractmethod
from typing import List


class NoiseMakerInterface(ABC):
    @abstractmethod
    def make_noise(self):
        pass


class Cat(NoiseMakerInterface):
    def make_noise(self):
        print("rawr")


class Car(NoiceMakerInterface):
    def make_noise(self):
        print("vroom vroom")


noise_makers = [Cat(), Car()]  # type: List[NoiceMakerInterface]
for object in noise_makers:
    object.make_noise()
```

explicit interface (python)

```csharp
public interface NoiseMakerInterface
{
    public void MakeNoise();
}

public class Cat : NoiseMakerInterface
{
    public void MakeNoise()
    {
        Console.WriteLine(GetNoise());
    }

    private string GetNoise()
    {
        return "rawr";
    }
}

public class Car : NoiseMakerInterface
{
    public void MakeNoise()
    {
        Console.WriteLine(GetNoise());
    }

    private string GetNoise()
    {
        return "Vroom Vroom";
    }
}
```

```csharp
var noiseMakers = new List<NoiseMakerInterface> {
            new Cat(),
            new Car()
        };
noiseMakers.ForEach(noiseMaker => noiseMaker.MakeNoise());
```

Interface í C#

```csharp
public abstract class NoiseMaker
{
    protected abstract string noise { get; }

    public void MakeNoise()
    {
        Console.WriteLine(noise);
    }
}

public class Cat : NoiseMaker
{
    protected override string noise => "rawr";
}

public class Car : NoiseMaker
{
    protected override string noise => “Vroom Vroom";
}
```

```csharp
var noiseMakers = new List<NoiseMaker> {
    new Cat(),
    new Car()
};
noiseMakers.ForEach(noiseMaker => noiseMaker.MakeNoise());
```

Abstract í C#

Program to an interface not an implementation frh.

- Hvað þýðir það?
  - Fókus á hvað klasi gerir en ekki hvernig
  - Klasi talar við klasa í gegnum interfaces (explicit eða implicit)
- Kostir
  - Loose coupling
- Interface per Implementation?
  - Umdeilt, sumir eru harðir á því aðrir ekki
  - Oft á eitthver annar interface-ið (sjá Dependency Inversion Principle)
  - Oft breytist klasi sjaldan og þarf ekki explicit interface (t.d. String í java)
  - Getur verið túlkað sem endurtekt

Encapsulate what varies

- Kröfur breytast
  - „The only constant thing about requirements is that they change”
  - Hugbúnaður ætti auðveldlega að geta aðlagað sig að breytingum
  - Í fullkomnum heimi ætti kóði að breytast á einum stað
- Encapsulation what varies
  - Einangraðu þann kóða sem breytist oft
  - Aðskildu parta sem munu breytast hvor í sínu lagi
  - Einangraðu parta saman sem munu breytast saman
  - Minnkar snertiflöt kóðans við breytingar
- Kostir af encapsualte what varies
  - Modularity
  - Extensibility
  - Readability
  - Maintainability

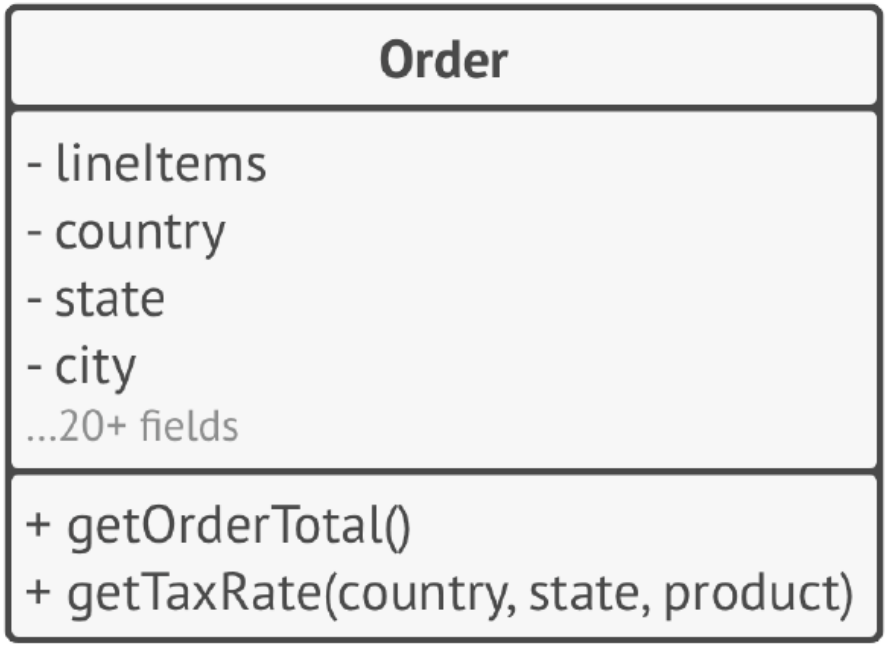
Encapsulate what varies dæmi 1


```python
# ❌ This is hard to understand and subject to change.
def checkout_book(customer, book):
    if (
        customer and
        customer.fine <= 0.0 and
        customer.card and
        customer.card.expiration is None and
        book and not book.is_checked_out
    ):
        customer.books.append(book)
        book.is_checkd_out = True
    return customer
```

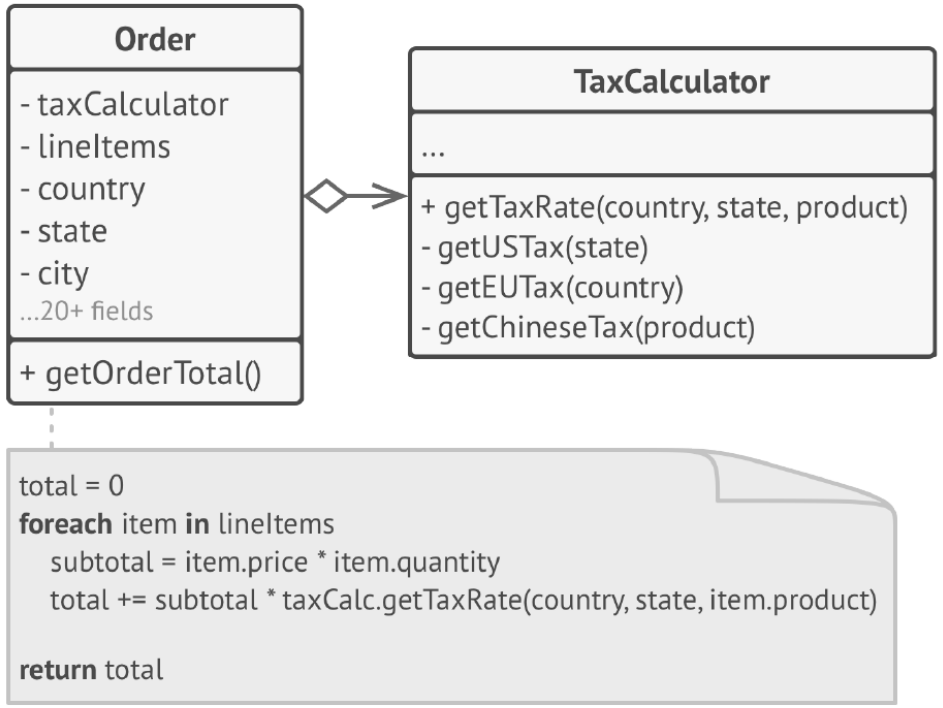
```python
# ✅ This is easy to read and won't change even if the checkout requirements vary.
def checkout_book(customer, book):
    if (customer.can_check_out(book)):
        customer.checkout(book)

    return customer
```

https://alexkondov.com/encapsulate-what-varies/

Encapsulate what varies dæmi 2

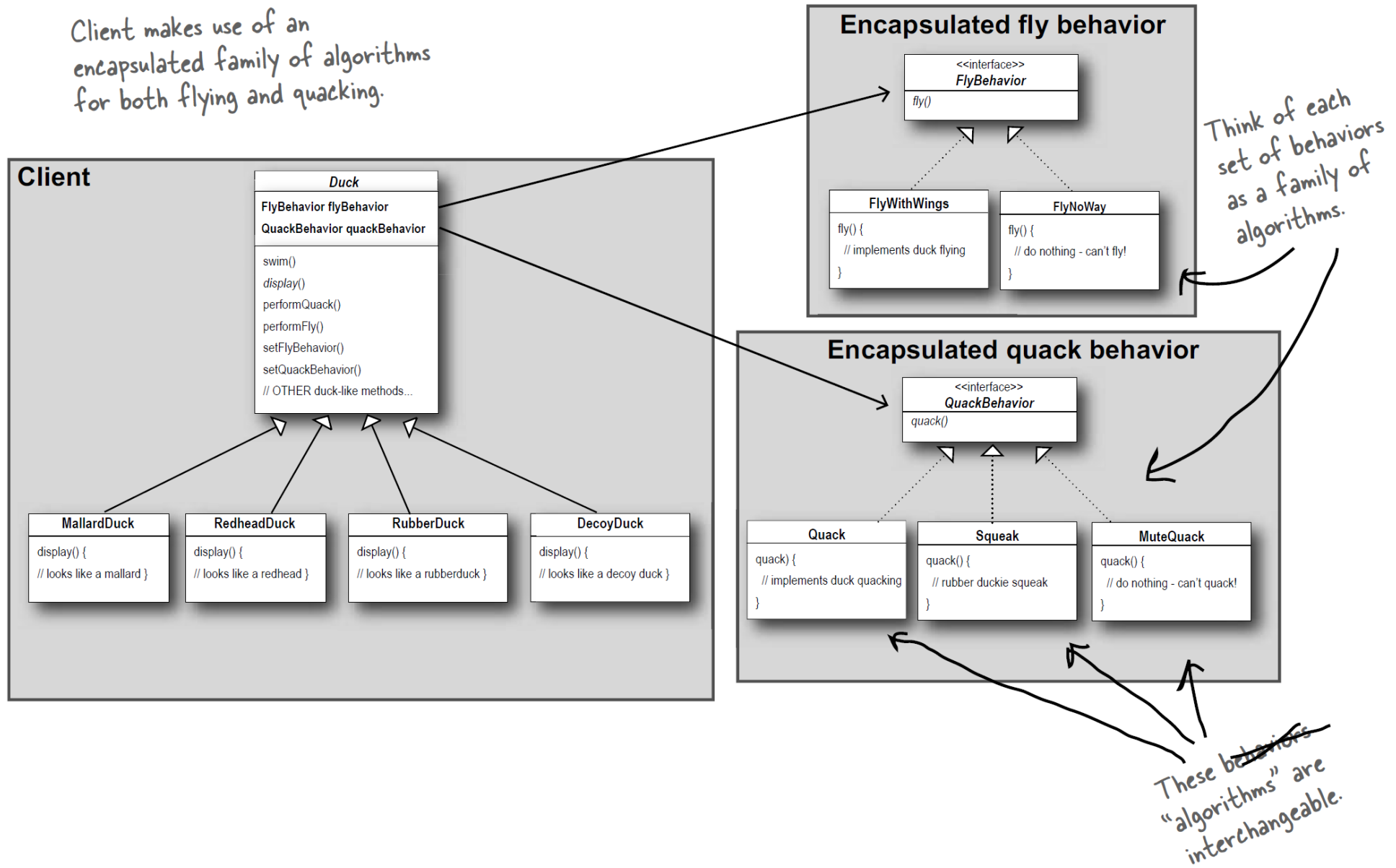


BEFORE: calculating tax in `Order` *class.*



AFTER: tax calculation is hidden from the order class.

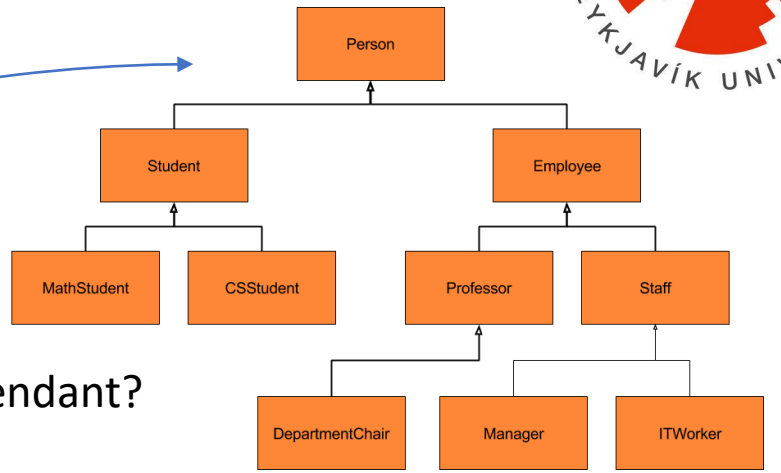
Encapsulate what varies dæmi 3



Inheritance, ~~the good~~, the bad and the ugly

- Inheritance oft ofnotað og notað rangt

- Inheritance er ekki eina leiðin til að extenda klasa



- Couple-ar saman ancestors og descendants
  - Stór hierarchy eru stíf og ósveigjanleg
  - Breytingar geta orðið erfiðar og brothættar
  - Geturðu verið viss um að breyting í ancestor klasa sé ekki að brjóta descendant?
  - Gerir djúpar inheritance keðjur erfitt að debug-a og maintain-a

- Ætti að vera notað fyrir frekari sérhæfingu ekki bara til að erfa virkni
  - *„Dogs don't inherit from Trees just because you want a bark() method "*
  - Is-a samband
  - Liskov Substitution Principle

- Oft ekki hentugt fyrir klasa sem breytast mikið
  - Til dæmis klasa sem tengjast domain model-inu (raunveruleikinn á til með að breytast á óútreiknanlega hætti)

Inheritance Brýtur Encapsulation

- Sumir segja „Make all data private, not protected “
- Virknin í ancestor klasa er ekki lengur eingöngu hjúpuð honum
- Descendant getur teygt sig í virkni í grandparent klasa þegar hann ætti bara að vita af parent

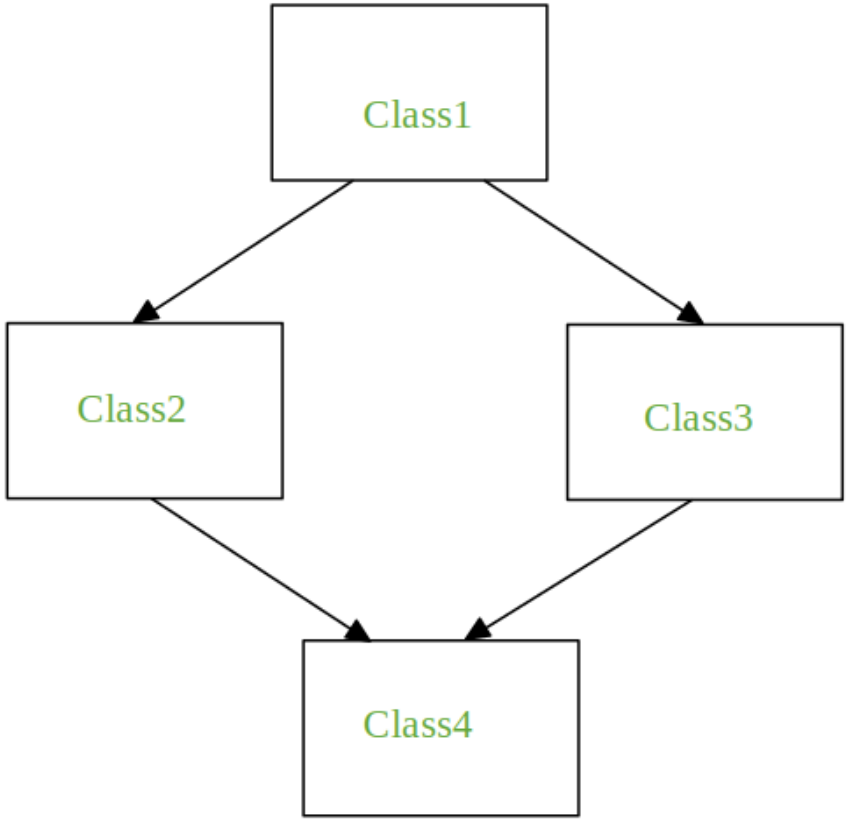
Multiple Inheritance Vandamál

- Ekki stutt af mörgum tungumálum
- Multiple inheritance oft misnotað
  - Oft notað til að erfa virkni en ekki til að sérhæfa
  - t.d. Pegasus erfir hest **og** fugl en pegasus er ekki fugl

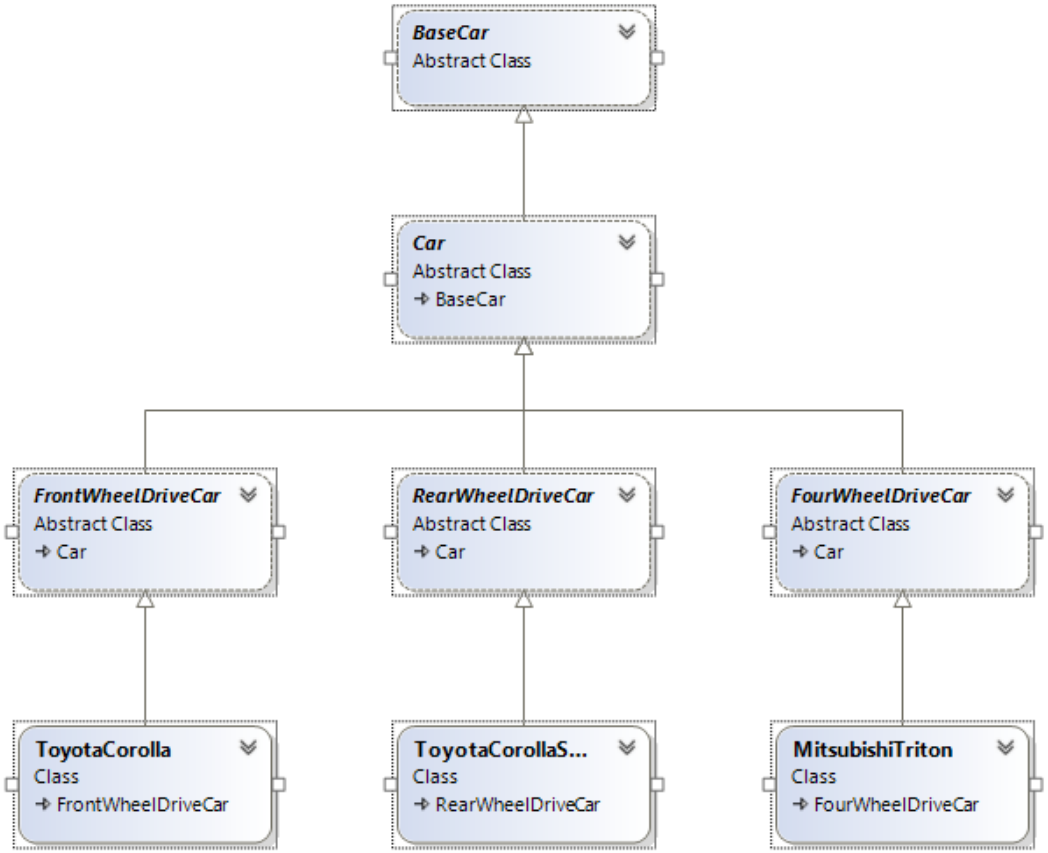
- The diamond problem
  - https://en.wikipedia.org/wiki/Multiple_inheritance#The_diamond_problem

The Diamond Problem

```
class Class1:
    def m(self):
        print("In Class1")


class Class2(Class1):
    def m(self):
        print("In Class2")


class Class3(Class1):
    def m(self):
        print("In Class3")


class Class4(Class2, Class3):
    pass


obj = Class4()
obj.m()  # prints In Class2
```



Stór hierarchy verða inflexible og stíf



https://codingdelight.com/2014/01/16/favor-composition-over-inheritance-part-1/

Favor Composition Over Inheritance

- Hvað er Composition?
  - „has-a“ samband
  - Byggir upp klasa út frá öðrum klösum
  - Yfirleitt gert með constructor inntökum *(oft með dependency injection)*
- Af hverju að nota composition yfir inheritance?
  - Inheritance hefur vandamál
  - Leyfir þér að breyta virkni *„at run-time“*
  - Meira flexibility
    - byggir klasa með litlum einingum, auðvelt að skipta út / bæta við / fjarlægja
  - Notar það sem þú þarft (ólíkt inheritance þar sem allt er erft)
  - Brýtur ekki Encapsulation ef gert rétt
    - ekki láta hlutina sem compose-a klasann vera public
  - Testable
  - Breytingar verða einangraðar
  - **Decoupling**

Hvað með Inheritance þá?

- Favor Composition ≠ Aldrei Inheritance
- Mörg hönnunarmynstur krefjast inheritance
- Inheritance getur sett skiljanlegan structure á kóðann
- Abstract klasar eru t.d. oft gagnlegir
- Gott dæmi um inheritance er custom exceptions
- Mixins

Composition vs Inheritance dæmi 1

```python
@dataclass
class Student(ABC):
    name: str
    age: int
    courses: List[str] = field(default_factory=lambda: [])

    @abstractmethod
    def _is_course_allowed(self, course: str) -> bool: pass

    def add_cousre(self, course: str) -> None:
        if (self._is_course_allowed(course)):
            self.courses.append(course)
        else:
            raise Exception('Course is not allowed')


class ComputerScienceStudent(Student):
    def _is_course_allowed(self, course: str) -> bool:
        return course in ['T-302-HONN', 'SC-T-111-PROG']


class EngineeringStudent(Student):
    def _is_course_allowed(self, course: str) -> bool:
        return course in ['SE-SE-T-102-EDL1', 'SE-SE-T-101-STA1']
```

Inheritance

```python
class Department(Enum):
    COMPUTER_SCIENCE_DEPARTMENT = 0,
    ENGINEERING_DEPARTMENT = 1


class Programme(Enum):
    SOFTWARE_ENGINEERING = 0,
    COMPUTER_SCIENCE = 1


class DepartmentCourseService:
    def get_department_courses(self, departments: List[Department]) -> List[str]:
        courses = []

        if (Department.COMPUTER_SCIENCE_DEPARTMENT in departments):
            courses += ['T-302-HONN', 'SC-T-111-PROG']
        if (Department.ENGINEERING_DEPARTMENT in departments):
            courses += ['SE-SE-T-102-EDL1', 'SE-SE-T-101-STA1']

        return courses


@dataclass
class CourseValidator:
    __department_course_service: DepartmentCourseService

    def __get_allowed_courses_for_programme(self, programme: Programme):
        if programme == Programme.COMPUTER_SCIENCE:
            return self.__department_course_service.get_department_courses([Department.COMPUTER_SCIENCE_DEPARTMENT])
        elif programme == Programme.SOFTWARE_ENGINEERING:
            return self.__department_course_service.get_department_courses([Department.COMPUTER_SCIENCE_DEPARTMENT, Department.ENGINEERING_D
EPARTMENT])
        return []

    def is_course_valid_for_programme(self, course: str, programme: Programme):
        allowed_courses = self.__get_allowed_courses_for_programme(programme)
        return course in allowed_courses


@dataclass
class Student(ABC):
    __course_validator: CourseValidator
    programme: Programme
    name: str
    age: int
    courses: List[str] = field(default_factory=lambda: [])

    def add_cousre(self, course: str) -> None:
        if (self.__course_validator.is_course_valid_for_programme(course, self.programme)):
            self.courses.append(course)
        else:
            raise Exception('Course is not allowed')
```

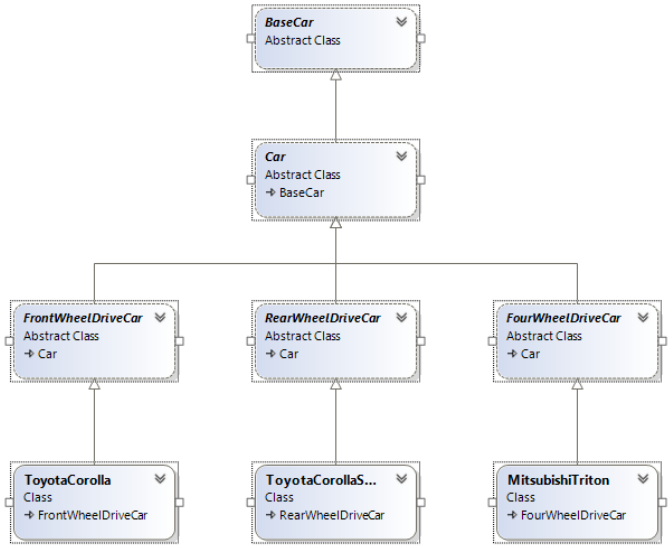
Composition

Composition vs Inheritance dæmi 2
(inheritance partur)

```python
@dataclass
class BaseCar(ABC):
    front_left: Wheel
    front_right: Wheel
    rear_left: Wheel
    rear_right: Wheel

    @property
    @abstractmethod
    def manufacturer(self) -> str: pass

    @abstractmethod
    def turn_left(self, degrees: float) -> None: pass

    @abstractmethod
    def turn_right(self, degrees: float) -> None: pass

    @abstractmethod
    def accelerate(self, kms_per_hour: float) -> None: pass


class Car(BaseCar):
    def __init__(self):
        super().__init__(
            front_left=Wheel(),
            front_right=Wheel(),
            rear_left=Wheel(),
            rear_right=Wheel())

    def turn_left(self, degrees: int) -> None:
        self.front_left.turn_left(degrees)
        self.front_right.turn_left(degrees)

    def turn_right(self, degrees: float) -> None:
        self.front_left.turn_right(degrees)
        self.front_right.turn_right(degrees)

    def accelerate(self, kms_per_hour: float) -> None:
        self.front_left.rotate(kms_per_hour)
        self.front_right.rotate(kms_per_hour)
```

```python
class FrontWheelDriveCar(Car):
    def accelerate(self, kms_per_hour: float) -> None:
        self.front_left.rotate(kms_per_hour)
        self.front_right.rotate(kms_per_hour)


class RearWheelDriveCar(Car):
    def accelerate(self, kms_per_hour: float) -> None:
        self.rear_left.rotate(kms_per_hour)
        self.rear_right.rotate(kms_per_hour)


class AllWheelDriveCar(Car):
    def accelerate(self, kms_per_hour: float) -> None:
        self.front_left.rotate(kms_per_hour)
        self.front_right.rotate(kms_per_hour)
        self.rear_left.rotate(kms_per_hour)
        self.rear_right.rotate(kms_per_hour)
```

```python
class ToyotaCorolla(FrontWheelDriveCar):
    def manufacturer(self) -> str:
        return "Toyota"


class ToyotaCorollaSports(RearWheelDriveCar):
    def manufacturer(self) -> str:
        return "Toyota"


class MitsubishiTitan(AllWheelDriveCar):
    def manufacturer(self) -> str:
        return "Mitsubishi“
```



Composition vs Inheritance dæmi 2
(composition partur)


```python
@dataclass
class Car:
    front_left: Wheel
    front_right: Wheel
    rear_left: Wheel
    rear_right: Wheel
    steering: FrontSteering
    driving: IDriving
    manufacturer: IManufacturer
    name: str
```

```python
class IManufacturer(ABC):
    @property
    @abstractmethod
    def name(self) -> str: pass


class Toyota(IManufacturer):
    def name(self) -> str:
        return "Toyota"


class Mitsubishi(IManufacturer):
    def name(self) -> str:
        return "Mitsubishi"
```

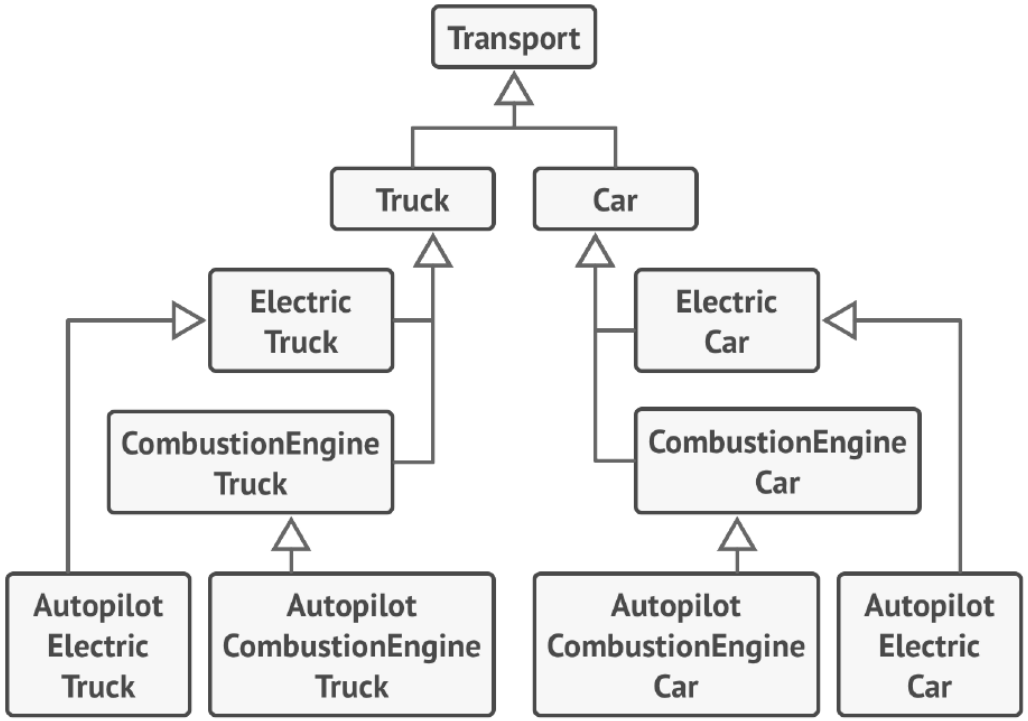
```python
class IDriving(ABC):
    @abstractmethod
    def accelerate(self, kms_per_hour: float) -> None: pass


class TwoWheelDrive(IDriving):
    def __init__(self, left: Wheel, right: Wheel) -> None:
        self.__left = left
        self.__right = right

    def accelerate(self, kms_per_hour: float) -> None:
        self.__left.rotate(kms_per_hour)
        self.__right.rotate(kms_per_hour)


class AllWheelDrive(IDriving):
    def __init__(self,
                 front_left: Wheel,
                 front_right: Wheel,
                 rear_left: Wheel,
                 rear_right: Wheel) -> None:
        self.__front_left = front_left
        self.__front_right = front_right
        self.__rear_left = rear_left
        self.__rear_right = rear_right

    def accelerate(self, kms_per_hour: float) -> None:
        self.__front_left.rotate(kms_per_hour)
        self.__front_right.rotate(kms_per_hour)
        self.__rear_left.rotate(kms_per_hour)
        self.__rear_right.rotate(kms_per_hour)
```

```python
class FrontSteering:
    def __init__(self, front_left: Wheel, front_right: Wheel) -> None:
        self.__front_left = front_left
        self.__front_right = front_right

    def turn_left(self, degrees: float) -> None:
        self.__front_left.turn_left(degrees)
        self.__front_right.turn_left(degrees)

    def turn_right(self, degrees: float) -> None:
        self.__front_left.turn_right(degrees)
        self.__front_right.turn_right(degrees)
```

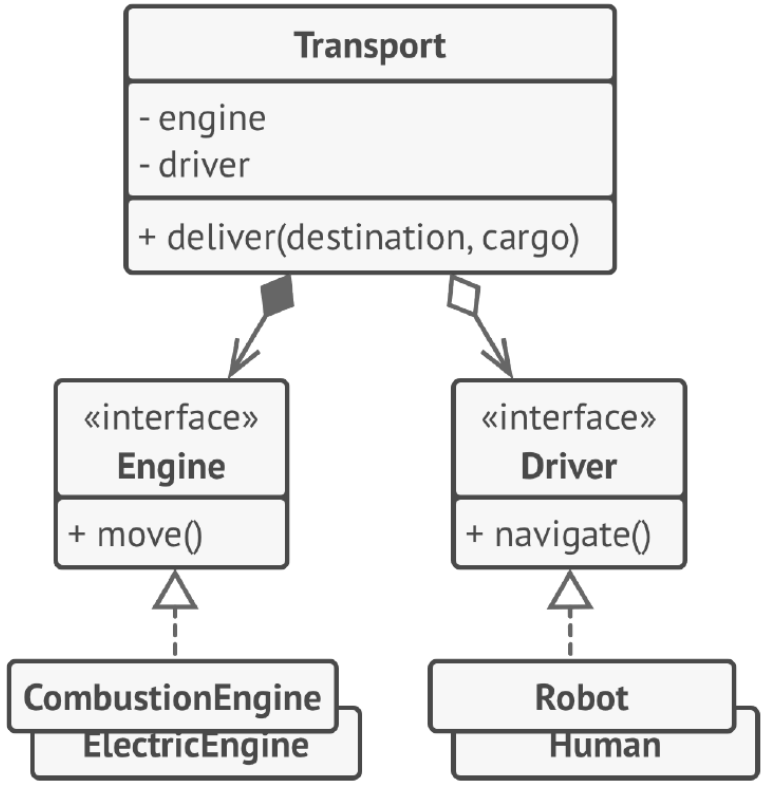
Composition vs Inheritance dæmi 2
(composition partur frh)

```
class CarFactory():
    def create_toyota_corolla(self) -> Car:
        front_right = Wheel()
        front_left = Wheel()

        return Car(front_left=front_left, front_right=front_right,
                   rear_left=Wheel(), rear_right=Wheel(),
                   steering=FrontSteering(front_left, front_right),
                   driving=TwoWheelDrive(front_left, front_right),
                   manufacturer=Toyota(),
                   name="toyota corolla"
                   )

    def create_toyota_corolla_sport(self) -> Car:
        front_right = Wheel()
        front_left = Wheel()
        rear_right = Wheel()
        rear_left = Wheel()

        return Car(
            front_left=rear_right, front_right=rear_left,
            rear_left=rear_right, rear_right=rear_left,
            steering=FrontSteering(front_left, front_right),
            driving=TwoWheelDrive(rear_left, rear_right),
            manufacturer=Toyota(),
            name="toyota corolla sport"
        )

    def mitsubishi_titan(self) -> Car:
        front_right = Wheel()
        front_left = Wheel()
        rear_right = Wheel()
        rear_left = Wheel()

        return Car(front_left=rear_right, front_right=rear_left,
                   rear_left=rear_right, rear_right=rear_left,
                   steering=FrontSteering(front_left, front_right),
                   driving=AllWheelDrive(
                       front_right, front_right, rear_left, rear_left),
                   manufacturer=Mitsubishi(),
                   name="Mitsubishi titan"
                   )
```

Composition vs Inheritance dæmi 3



INHERITANCE: extending a class in several dimensions (cargo type × engine type × navigation type) may lead to a combinatorial explosion of subclasses.



COMPOSITION: different "dimensions" of functionality extracted to their own class hierarchies.

Dæmi úr dive into design patterns

Mixins

- Inheritance sem er ekki „is-a“ samband
- Er notað til að gefa einfaldar virknir sem eru ólíklegar til að breytast eða valda vandamálum
- Hugmyndafræðilegur munur á Mixin og inheritance í python (útfært eins) enn í tungumálum eins og Dart er Mixin byggt inn.
- Getur auðvitað haft sömu vandamál og inheritance ef ekki notað rétt
- inheritance keðjan ætti yfirleitt bara að vera parent-child (í sumum tungumálum byggt inn þannig)
- Eru ekki ætluð til að vera initialized independently
- Mixins eru mögulega eina “góða” dæmið um multiple inheritance
- Lendir ekki í diamond problem því Mixins ættu að vera independent frá hvort öðrum

```python
class AsDictionaryMixin:
    def to_dict(self):
        return {
            prop: self._represent(value)
            for prop, value in self.__dict__.items()
            if not self._is_internal(prop)
        }

    def _represent(self, value):
        if isinstance(value, object):
            if hasattr(value, 'to_dict'):
                return value.to_dict()
            else:
                return str(value)
        else:
            return value

    def _is_internal(self, prop):
        return prop.startswith('_')
```

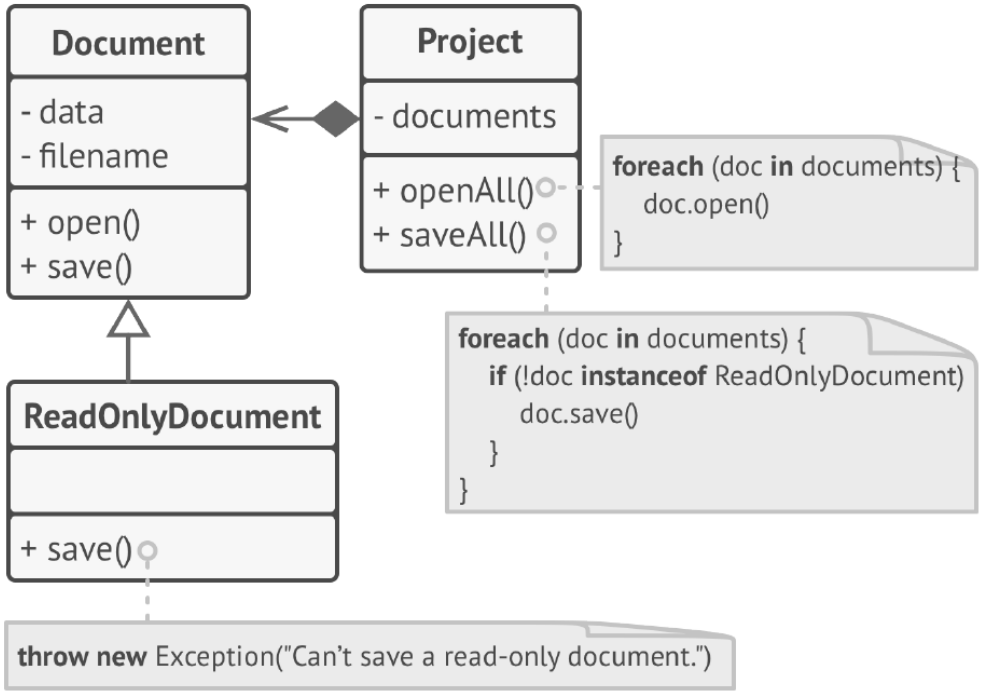
```python
class Employee(AsDictionaryMixin):
    def __init__(self, name, age) -> None:
        self.name = name
        self.age = age
```

```python
employee = Employee("namey", 22)
print(employee.to_dict())  # outputs: {'name': 'namey', 'age': '22'}
```

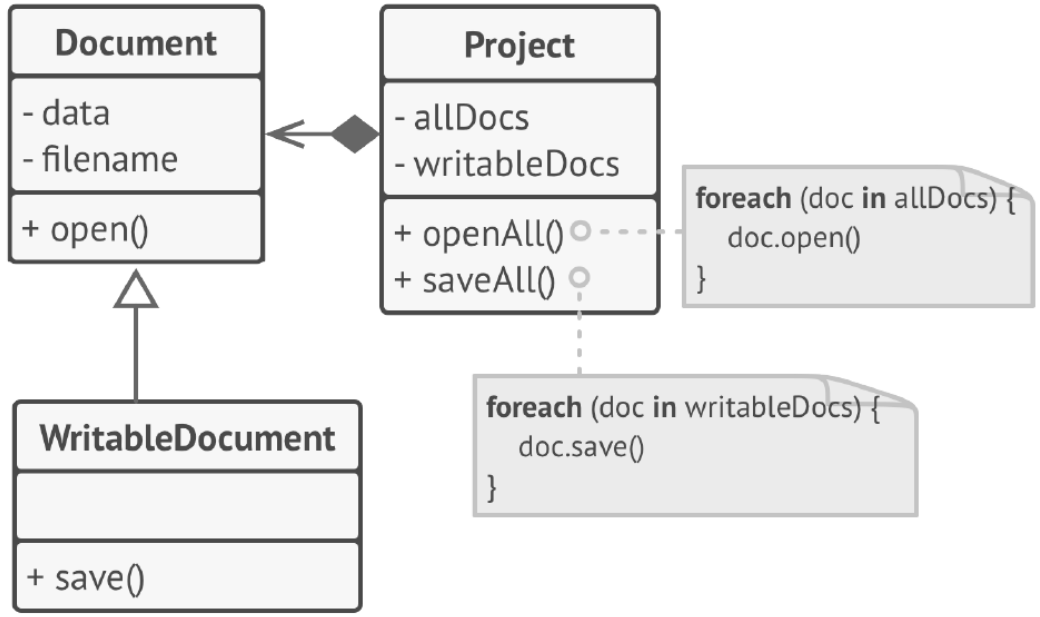
Liskov Substitution Principle

- Eitt af SOLID principle-unum
- Formleg skilgreining
  - *„ Let Φ(x) be a property provable about objects x of type T. Then Φ(y) should be true for objects y of type S where S is a subtype of T.“*
  - *„If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T“*
- Óformleg skilgreining
  - „Suptypes must be substituable for their base types“
  - LSP segir að ef við skiptum út ancestor hlut út fyrir descendant hlut þá ætti forritið ennþá að virka
  - Á við um bæði interface erfðir og implementation erfðir
- Supklasi á að vera specialization af superklasa
  - Klasi ætti ekki að erfa frá öðrum klasa nema til að sérhæfa
  - uppfyllir„is-a“ sambandið (is-a er í raun of víðtækt hugtak, „is-a“ ætti frekar að vera „ substituable“, sjá rectangle vs square dæmið)
- Bæði semantic og syntactic samningar
  - Semantic
    - Merkingafræðilegur
    - Ekki reglur sem þú getur inforce-að í kóða með types og þess háttar
    - Implicit samningur
    - t.d. Draw fall á að teikna ekki skrifa
  - Syntactic
    - Setningafræðilegur
    - Reglur sem er hægt að force með syntax-inum í forritunarmálinu
    - Explicit samningur
    - t.d. Að get_tax fall skili tölu
  - Segir að það eigi að vera semantic samningur á milli ancestor klasa og descendant klasa en ekki bara syntactic samningur
  - Descendant ætti að haga sér eins og ancestor klasi
- Dæmi um brot:
  - Descendant kastar villu í falli sem ancestor klasi gerir ekki
  - Descendant breytir semantic af falli í ancestor klasa

LSP dæmi 1

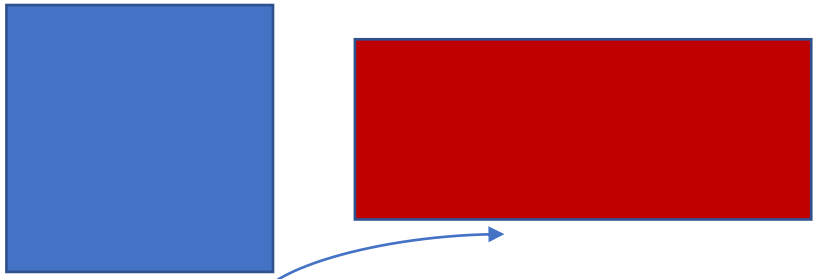


BEFORE: saving doesn't make sense in a read-only document, so the subclass tries to solve it by resetting the base behavior in the overridden method.



AFTER: the problem is solved after making the read-only document class the base class of the hierarchy.

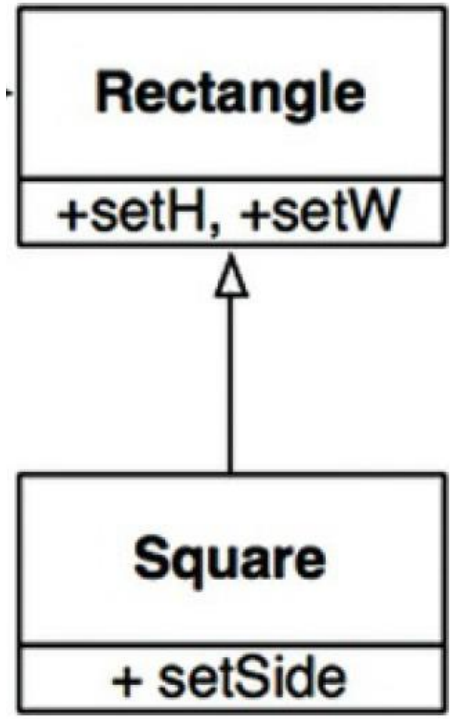
LSP ekki uppfyllt, dæmi 2



- Ferningur(Square) er ekki Rétthyrningur(Rectangle) því hæðin og breiddin á rétthyrning þarf ekki endilega að vera sú sama eins og í ferning

```python
class Rectangle:
    def set_width(self, width: float):
        self._width = width

    def set_height(self, height: float):
        self._height = height

    def area(self):
        return self._width * self._height


class Square(Rectangle):
    def set_width(self, width: float):
        self._width = width
        self._height = width

    def set_height(self, height: float):
        self._height = height
        self._width = height
```

```python
rectangle = ...  # annað hvort square eða rectangle
rectangle.set_width(5)
rectangle.set_height(2)

# þessi typecheck eru code smell og merki um að LSP er brotið
if type(rectangle) is Rectangle:
    assert rectangle.area() == 10
elif type(rectangle) is Square:
    assert rectangle.area() == 4
```



LSP ekki uppfyllt, dæmi 3

```
class Cat:
    def scratch(self):
        print("scratchy scratchy")


class DeclawedCat(Cat):
    def scratch(self):
        raise NotImplementedError('cat is declawed')
```

LSP uppfyllt, dæmi 4

```python
class Dog:
  def __init__(self, name: str) -> None:
      self._name = name

  def bark(self):
    print(f'My name is {self._name}')


class GoldenRetriever(Dog):
  def retrieve_gold(self):
    return "gold"

  def bark(self):
    super().bark()
    print("and I am a Golden Retriver")
```

SOLID Principles

- Single Responsibility Principle (SRP)
- Open-Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

Single Responsibility Principle (SRP)

- ***„A class should have only one reason to change“***
  - Ef þú getur fundið fleiri en eina ástæðu fyrir klasa til að breytast þá hefur hann fleiri en eitt *responsibility*
- Leyðir til decoupled design
  - Ef klasi er með fleiri en eitt responsibility þá verða þessi responsibility tightly coupled
- Gerir klasa meira resuable
  - Klasi sem fylgir SRP er yfirleitt lítill og hefur skýr mörk og er því auðvelt að plug-a honum inn í aðra klasa
- Stuðlar að encapsulate what varies
  - Að setja sambærilega þætti í sama klasa gerir líklegra að þú munt bara þurfta að breyta þeim klasa þegar sú virkni breytist
- Afstætt hvort klasi fylgir SRP eða ekki
  - Þú getur farið ofar og ofar í “abstraction keðjunni” og sagt að klasinn uppfyllir SRP því hann er bara single responsibility fyrir ákveðnu abstraction-i
  - T.d. EmailService sem sendir **og** semur email
  - **Fer eftir ástæðu til að breytast**

Cohesion

- Cohesion er *skyldleiki* falla /virknis í klasa/ module-a / kerfa
- T.d. því fleiri klasa breytur sem fall í klasa notar því meira *cohesive* er fallið við virkni klasans
- Ef klasi er með hátt cohesion þá eru góðar líkur að hann fylgi SRP því þá hanga föllin saman *“as a logical whole”*
- Við viljum hafa hátt cohesion

Dæmi um brot á SRP

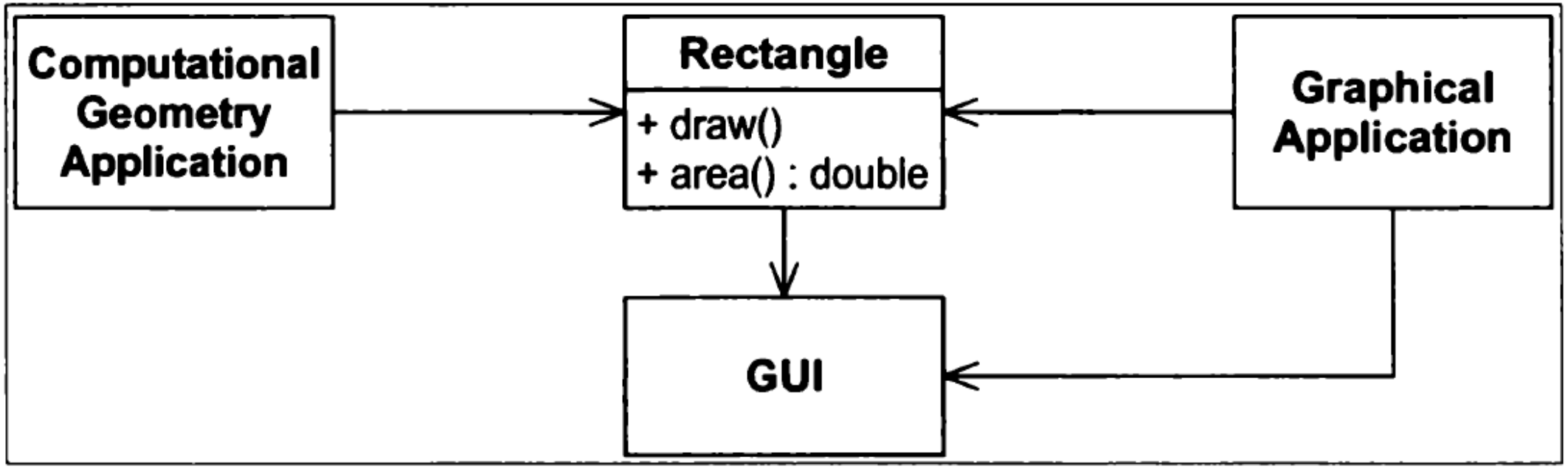


Figure 8-1 More than one responsibility

- Rectangle er með tvö responsibility
  - Represent-a rúmfræðilegu eiginleika rétthyrnings
  - Teikna rétthyrningin

Dæmi frá Agile Software Development, principles, patterns and practices, Robert C. Martin

Dæmi um brot á SRP frh.

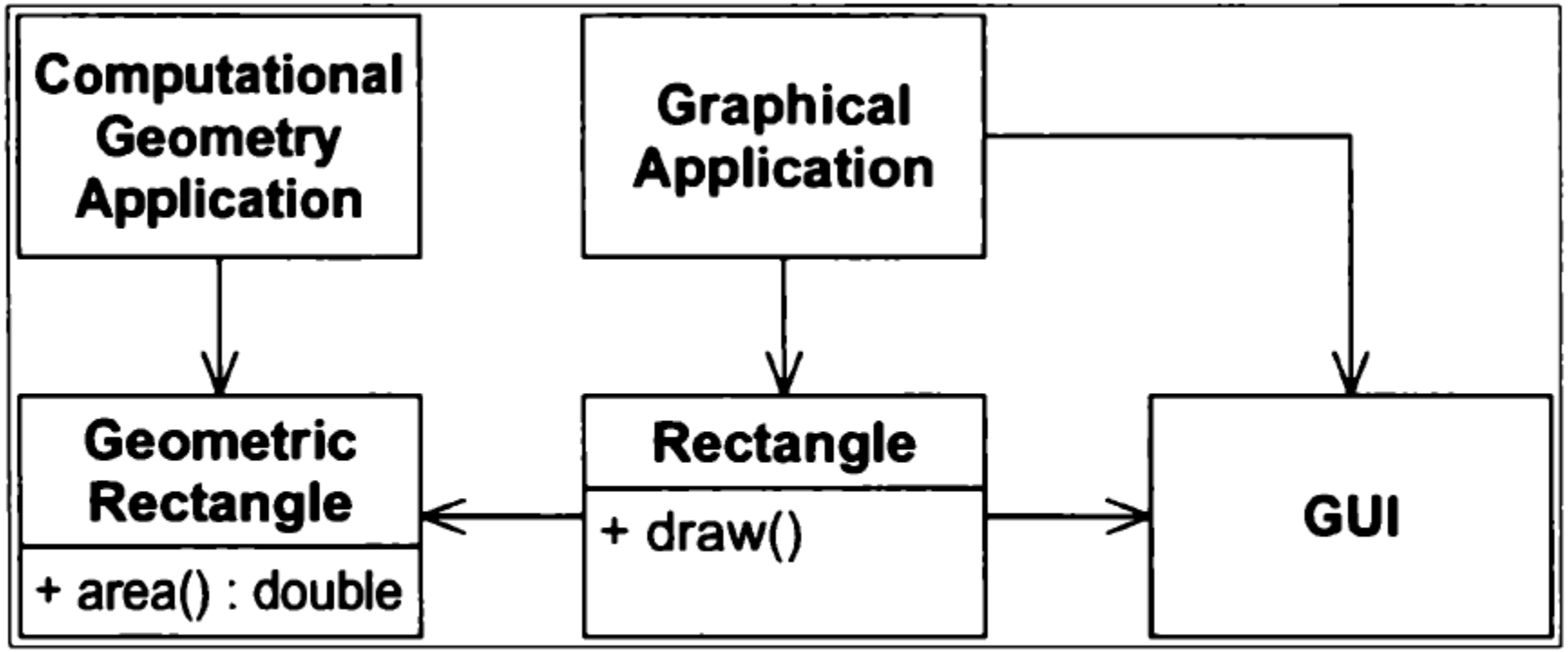


Figure 8-2 Separated Responsibilities

Dæmi 2 um brot á SRP

Listing 8-1

Modem.java -- SRP Violation

```
interface Modem
{
  public void dial(String pno);
  public void hangup();
  public void send(char c);
  public char recv();
}
```

- Modem = „a device attached to a computer that enables the transfer of data to or from a computer through telephone lines. “
- Klasinn Modem lítur saklaus við fyrstu sýn en hefur tvö responsibility
  - Connection management (dial, hangup)
  - Data communication (send, recv)
- Ætti hann að vera skiptur í tvö klasa?

Dæmi 2 um brot á SRP frh.

- Ætti Modem klasinn að vera splitaður?
  - Ef virknin breytist saman **þá nei**
  - Ef virknin mun breytast í sitthvoru lagi **þá já**

  - Annar möguleiki væri að fara milliveginn og splitta interface-inu

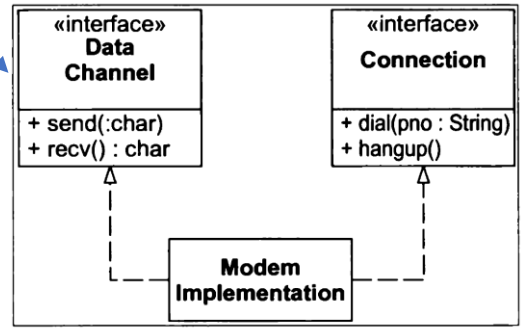
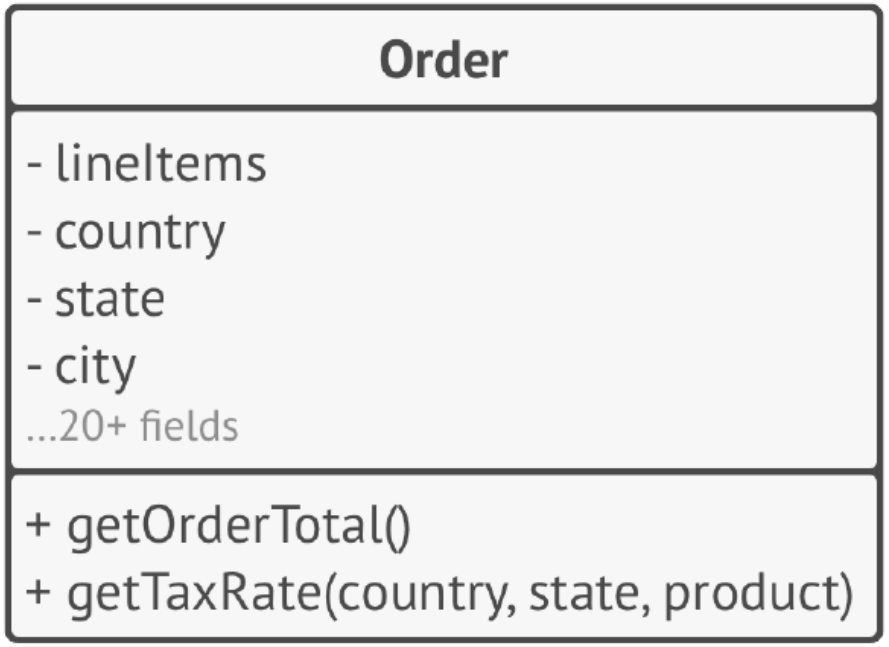
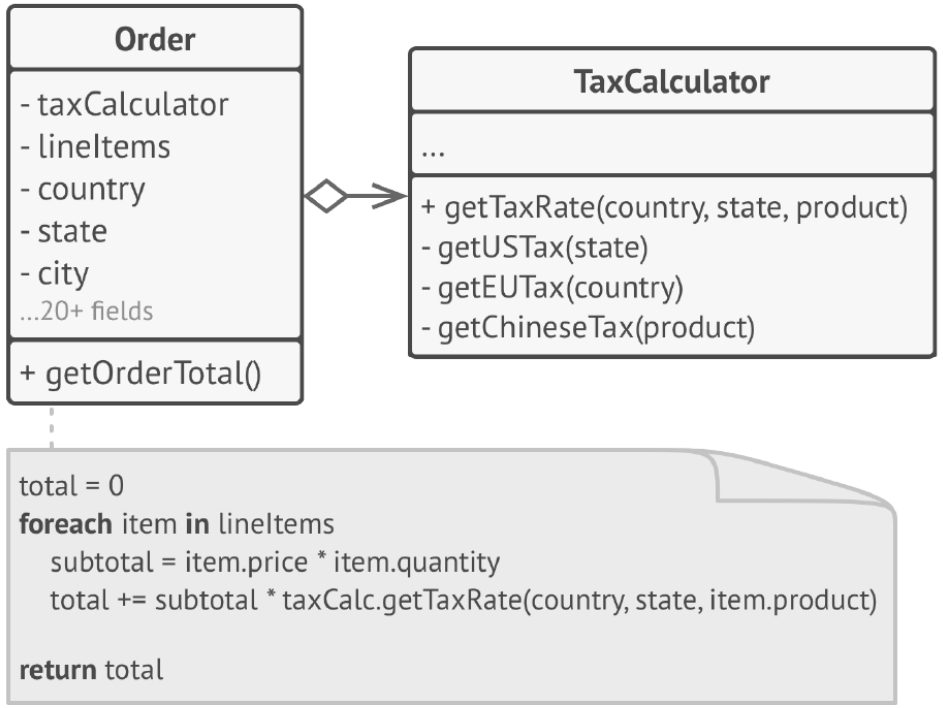


Figure 8-3 Separated Modem Interface

SRP dæmi 3



BEFORE: calculating tax in `Order` *class.*



AFTER: tax calculation is hidden from the order class.

Cohesion dæmi

Listing 10-4

`Stack.java` A cohesive class.

```
public class Stack {
  private int topOfStack = 0;
  List<Integer> elements = new LinkedList<Integer>();

  public int size() {
    return topOfStack;
  }

  public void push(int element) {
    topOfStack++;
    elements.add(element);
  }

  public int pop() throws PoppedWhenEmpty {
    if (topOfStack == 0)
      throw new PoppedWhenEmpty();
    int element = elements.get(--topOfStack);
    elements.remove(topOfStack);
    return element;
  }
}
```

- Klasinn hefur hátt cohesion, af þremur föllunum þá er bara size sem notar ekki bæði topOfStack og elements property-in

Dæmi tekið úr Clean Code, Robert C. Martin

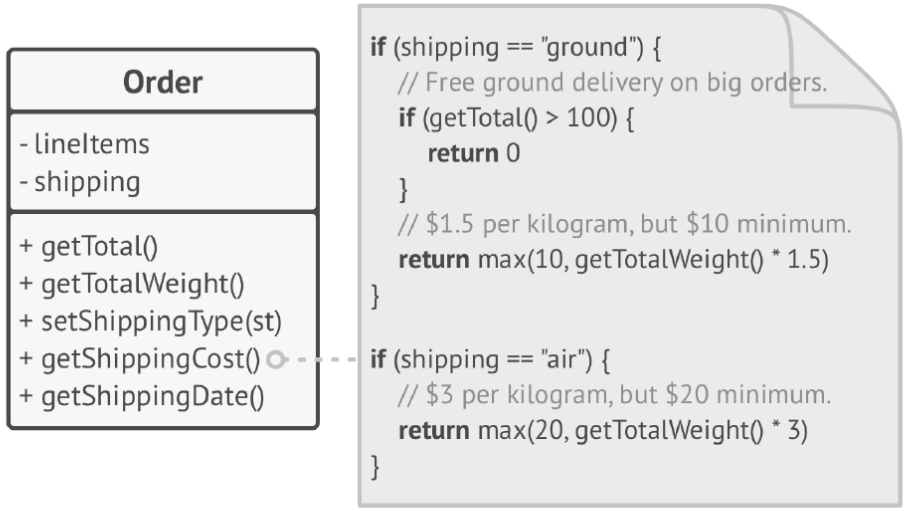
Open-Closed Principle

- *„Software entities should be open for extension but closed for modification“*
  - Breytingar brjóta, viljum forðast breytingar á klösum með því að extend-a klasa og virkni í staðinn
- Nýr kóði ekki breyttur kóði
  - Breyttar/nýjar kröfur koma fram í ljósi nýs kóða ekki breyttan kóða
  - Komið í verk með *Abstractions*
  - Dæmi um hönnunarmynstur sem gera þetta
    - Strategy Pattern
    - Template method pattern
    - Decorator Pattern
    - Abstract Factory Pattern
    - Observer Pattern
    - O.fl.
- Verður að velja fyrir hvaða breytingum klasinn er lokaður
  - Munt aldrei geta lokað klasa fyrir öllum breytinum
  - Verður að velja hvaða breytingar þú vilt loka klasanum fyrir
  - Verður að sjá fyrir hvaða breytingar eru líklegar til að eiga sér stað (eða rule of three)
- Getur ekki fylgt OCP alltaf
  - Að fylgja OCP er dýrt, það tekur tíma að gera góð abstraction
  - Vilt bara reyna að fylgja OCP þar sem svipaðar breytingar munu vera algengar
  - Abstraction-in munu oft gera hönnunina flóknari
    - KISS
    - YAGNI
    - Rule of three

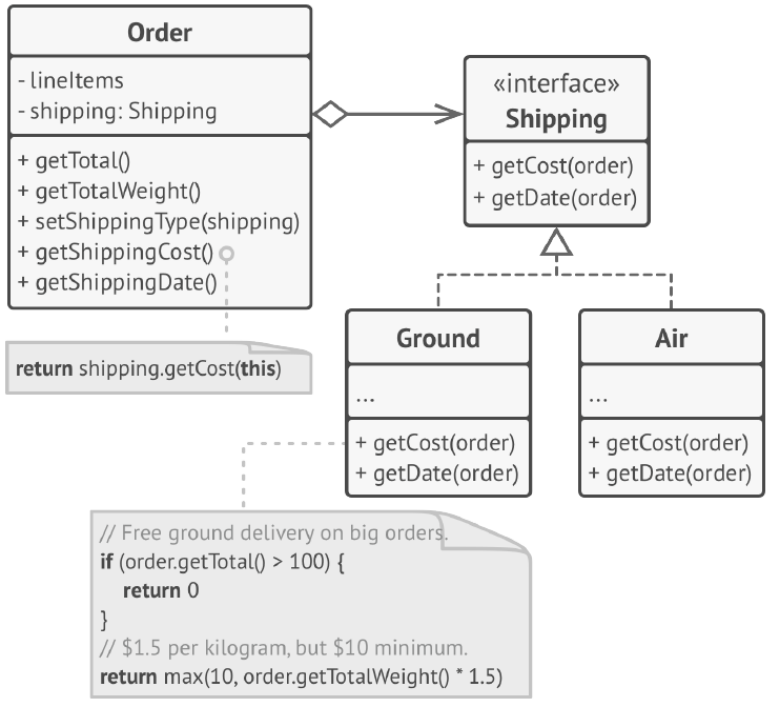
Come on in; we're *open*. Feel free to extend our classes with any new behavior you like. If your needs or requirements change (and we know they will), just go ahead and make your own extensions.

Sorry, we're *closed*. That's right, we spent a lot of time getting this code correct and bug free, so we can't let you alter the existing code. It must remain closed to modification. If you don't like it, you can speak to the manager.

OCP dæmi 1



BEFORE: you have to change the `Order` class whenever you add a new shipping method to the app.



AFTER: adding a new shipping method doesn't require changing existing classes.

OCP dæmi 2

```python
class Shape(ABC):
    @abstractmethod
    def get_color(self) -> Color: pass


class Circle(Shape):
    def __init__(self, radius: float, center: Point) -> None:
        self.__radius = radius
        self.__center = center

    def get_color(self) -> Color:
        return Color.RED


class Square(Shape):
    def __init__(self, side_length: float, top_left: Point) -> None:
        self.__side_length = side_length
        self.__top_left = top_left

    def get_color(self) -> Color:
        return Color.RED


class ShapeRenderer:
    def draw_shapes(self, shapes: list[Shape]):
        for shape in shapes:
            if (type(shape) is Circle):
                self.__draw_circle(shape)
            elif (type(shape) is Square):
                self.__draw_square(shape)

    def __draw_circle(self, circle: Circle):
        print("drawing cicle:", circle.get_color())

    def __draw_square(self, square: Square):
        print("drawing square:", square.get_color())


if __name__ == '__main__':
    shape_renderer = ShapeRenderer()

    shapes = [Circle(10, Point(5, 3)), Square(12, Point(10, 2))]
    shape_renderer.draw_shapes(shapes)
```

Brýtur OCP

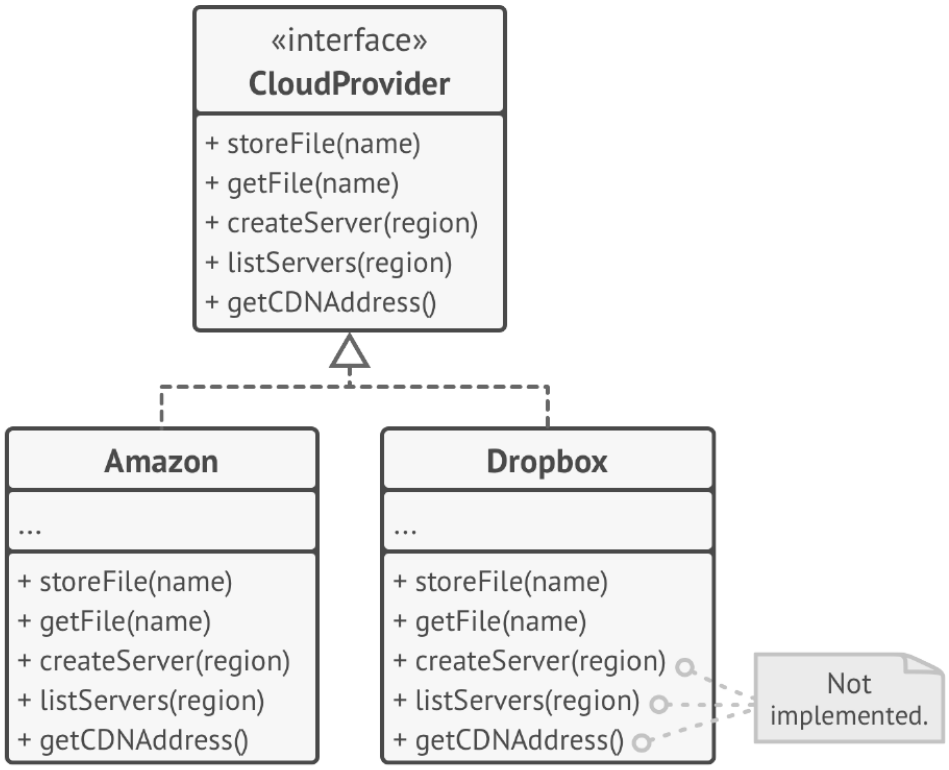
```python
class Shape(ABC):
    @abstractmethod
    def get_color(self) -> Color: pass

    @abstractmethod
    def draw(self): pass


class Circle(Shape):
    def __init__(self, radius: float, center: Point) -> None:
        self.__radius = radius
        self.__center = center

    def get_color(self) -> Color:
        return Color.RED

    def draw(self):
        print("drawing cicle:", self.get_color())


class Square(Shape):
    def __init__(self, side_length: float, top_left: Point) -> None:
        self.__side_length = side_length
        self.__top_left = top_left

    def get_color(self) -> Color:
        return Color.RED

    def draw(self):
        print("drawing square:", self.get_color())


class ShapeRenderer:
    def draw_shapes(self, shapes: list[Shape]):
        for shape in shapes:
            shape.draw()


if __name__ == '__main__':
    shape_renderer = ShapeRenderer()

    shapes = [Circle(10, Point(5, 3)), Square(12, Point(10, 2))]
    shape_renderer.draw_shapes(shapes)
```

Fylgir OCP

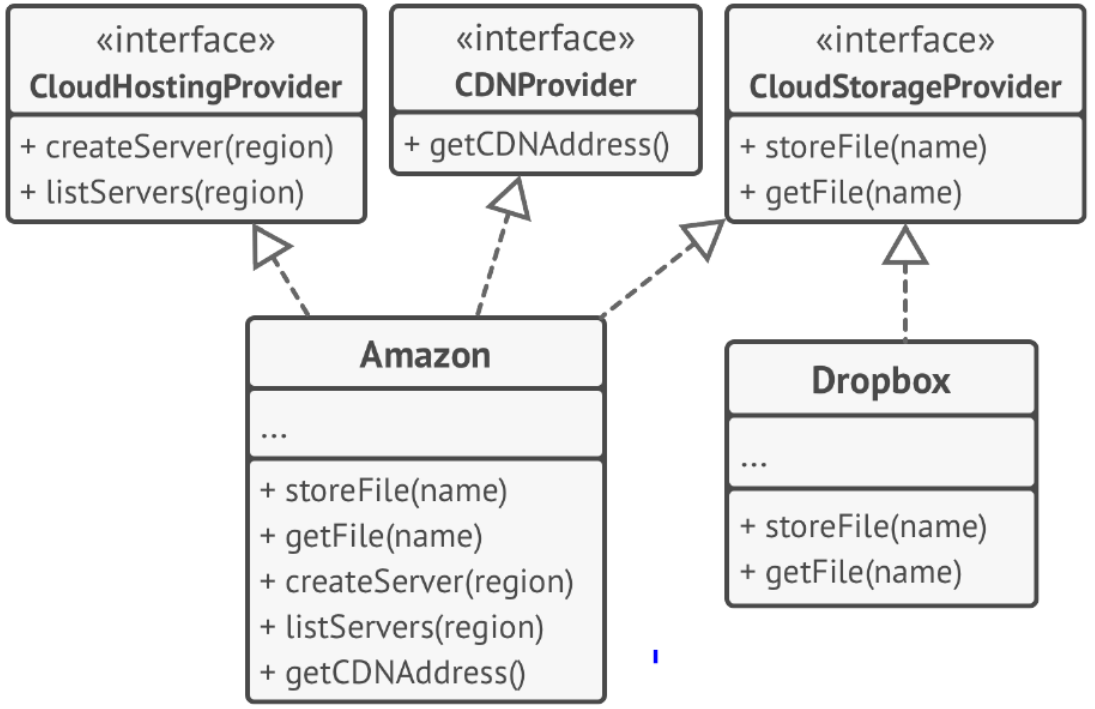
Interface-Segregation Principle

- „Clients should not be forced to depend upon methods that they do not use“
  - Client-ar þurfa oft bara hlutmengi af föllum í interface-i
  - Stór interface er því oft hægt að skipta upp
  - Client-ar depend-a bara á þau interface sem þau þurfa
  - Annars getur myndast ótengd coupling á milli client-a
- Einn concrete klasi eða margir
  - Einn concrete klasi getur ennþá séð um að útfæra öll interface-in
  - Getur verið brotið upp í klasa per interface
  - Client-inn mun ekki vita af því
  - Oft er brot á ISP merki um brot á SRP, ef það er tilfellið er örugglega betra að brjóta virknina upp í marga concrete klasa

ISP dæmi 1



BEFORE: not all clients can satisfy the requirements of the bloated interface.



AFTER: one bloated interface is broken down into a set of more granular interfaces.

ISP dæmi 2

```python
class Shape(ABC):
    @abstractmethod
    def calculate_area(self) -> float: pass

    @abstractmethod
    def calculate_volume(self) -> float: pass


class Sphere(Shape):
    def __init__(self, radius: float) -> None:
        self.__radius = radius

    def calculate_area(self) -> float:
        return 4 * pi * self.__radius ** 2

    def calculate_volume(self) -> float:
        return (4 / 3) * pi * self.__radius ** 3


class Circle(Shape):
    def __init__(self, radius: float) -> None:
        self.__radius = radius

    def calculate_area(self) -> float:
        return pi * self.__radius**2

    def calculate_volume(self) -> float:
        raise NotImplementedError('Method does not apply for circles')
```

Brot á ISP

```python
class TwoDShape(ABC):
    @abstractmethod
    def calculate_area(self) -> float: pass


class ThreeDShape(TwoDShape):
    @abstractmethod
    def calculate_volume(self) -> float: pass


class Sphere(ThreeDShape):
    def __init__(self, radius: float) -> None:
        self.__radius = radius

    def calculate_area(self) -> float:
        return 4 * pi * self.__radius ** 2

    def calculate_volume(self) -> float:
        return (4 / 3) * pi * self.__radius ** 3


class Circle(TwoDShape):
    def __init__(self, radius: float) -> None:
        self.__radius = radius

    def calculate_area(self) -> float:
        return pi * self.__radius**2
```

Ekki brot á ISP

ISP dæmi 3

```python
class IMessageService(ABC):
    def send_message(self) -> None: pass

    def receive_message(self) -> str: pass

    def compose_message(self) -> str: pass


class MessageService(IMessageService):
    def send_message(self) -> None:
        ...

    def receive_message(self) -> str:
        ...

    def compose_message(self) -> str:
        ...


class Client:
    def __init__(self, message_service: IMessageService) -> None:
        self.__message_service = message_service

    def get_messages(self) -> str:
        return self.__message_service.receive_message()
```

Brot á ISP

```python
class ISendMessageService(ABC):
    def send_message(self) -> None: pass


class IReceiveMessageService(ABC):
    def send_message(self) -> None: pass


class IComposeMessageService(ABC):
    def send_message(self) -> None: pass


class MessageService(
        ISendMessageService,
        IReceiveMessageService,
        IComposeMessageService):
    def send_message(self) -> None:
        ...

    def receive_message(self) -> str:
        ...

    def compose_message(self) -> str:
        ...


class Client:
    def __init__(self, message_service: IReceiveMessageService) -> None:
        self.__message_service = message_service

    def get_messages(self) -> str:
        return self.__message_service.receive_message()
```

Ekki brot á ISP

Dependency-Inversion Principle

-
- Skilgreining 1 (Réttari skilgreining)
  - a.) *„high-level modules should not depend on low-level modules. Both should depend on abstractions“*
    - Þegar high-level component-ar(HLC) depend-a á low-level component-a(LLC) þá geta breytingar í LLC haft bein áhrif á HLC
    - Það eru high-level componentarnir sem eiga að hafa áhrif á low-level component-ana, þeir skilgreina t.d. Buisiness logic-ina og knýja fram breytingar
    - Bæði high-level og low-level componentarnir ættu að depend-a á sama abstraction
    - Minnkar couplingu á milli high og low level component-a
  - b.) *„Abstractions should not depend on details. Details should depend on abstractions“*
    - Abstraction-ið á ekki að vita um concrete útfærsluna, concrete útfærlsan á að vita um, depend-a á og útfæra abstraction-ið
    - Ef detail-in breytast þá ætti abstraction-ið yfirleitt ekki að breytast
    - Ef abstraction-ið breytist þá mun detail-ið að öllum líkindum breytast
    - Higher level component eða þriðji aðili á interface-ið/abstraction-ið ekki lower-level componentinn (sjá *interface ownership inversion* fyrir neðan)

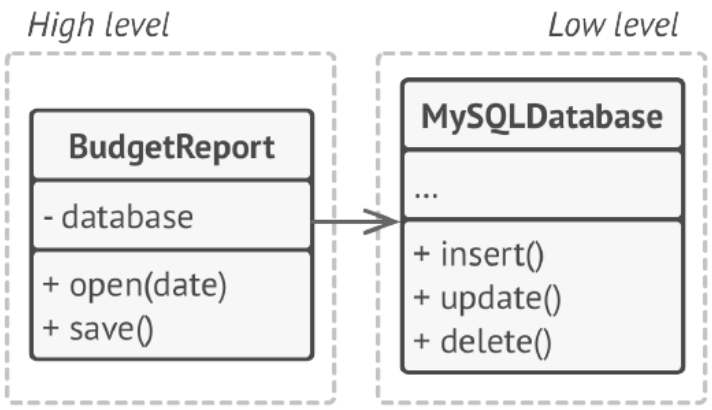
Dependency-Inversion Principle

- Skilgreining 2 (Naive en samt öflug)
  - *“Depend upon abstractions. Do not depend upon concrete classes.”*
    - Skilgreiningin sem *Head First Design Principles* notar
    - Einfaldari skilgreining, en segir ekki jafn mikið
    - Hljómar eins og “program to an interface not implementation”
      - Svipað en DIP gerir sterkari kröfur um abstractions
      - DIP segir að að high-level componentar dependa ekki á low level componenta heldur depend-a bæði á sama abstraction
      - DIP setur kröfu á abstraction “ownership-ið”, í DIP er það high-level componentinn eða þriðji aðili sem “á” abstraction-ið
    - Guidelines til að fylgja þessari skilgreiningu (Öll forrit munu auðvitað brjóta þessi guidelines en það er gott að hafa þau í huga þar sem við á)
      - Engin breyta ætti að að vera með beint reference á concrete klasa
        - Ættir ekki að instantiate-a breytu beint með constructor fyrir concrete hlut (new … í Java/C#)
      - Engin klasi ætti að erfa frá concrete klasa
        - Ef þú erfir frá concrete klasa þá ertu að depend-a á þann klasa, ættir frekar að erfa frá interface eða abstract klasa
      - Ekkert fall ætti að override-a útfært fall í base-klasa
        - Ef þú override-ar fall þá var base klasinn ekki gott abstraction til að byrja með.

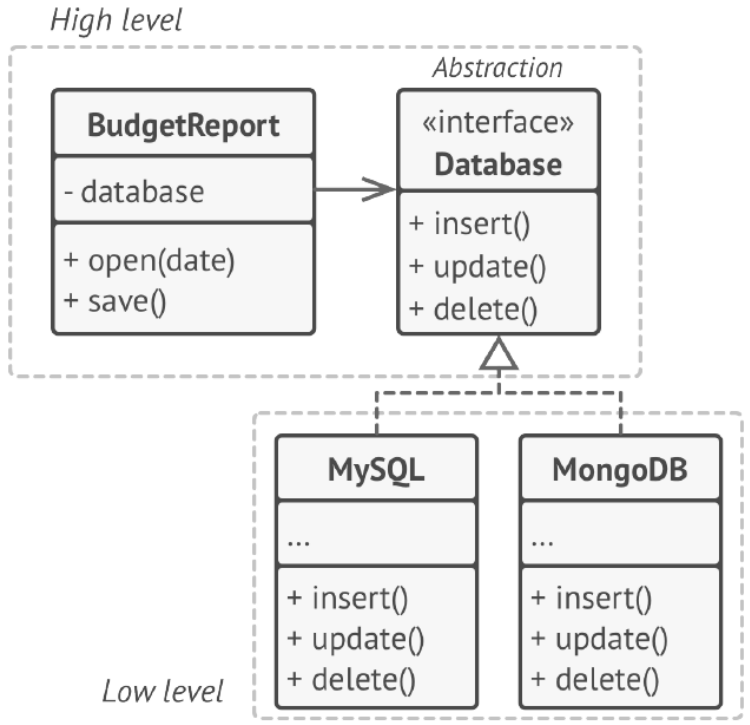
Hvað er Inversion-ið?

- Dependncy inversion
  - Inversion-ið er að núna depend-a high-level component-arnir ekki á low-level component-ana, í staðinn depend-a low-level componentarnir á higher level abstraction
- Interface ownership inversion
  - Inversion-ið er ekki bara hvað varðar dependency, inversion-ið er líka um hver á abstraction-ið, við hugsum oft um að low-level component-arnir eigi sín eigin interface en þegar DIP er fylgt þá er það client-inn/high-level componentarnir sem eiga yfirleitt interface-in eða eitthver þriðji aðili, low-level/service component-arnir útfæra síðan þau interface. Þetta er Hollywood principle-ið -> “you don’t call us we call you” (“you” verandi low-level service component og “we” verandi higher level component)

DIP dæmi 1

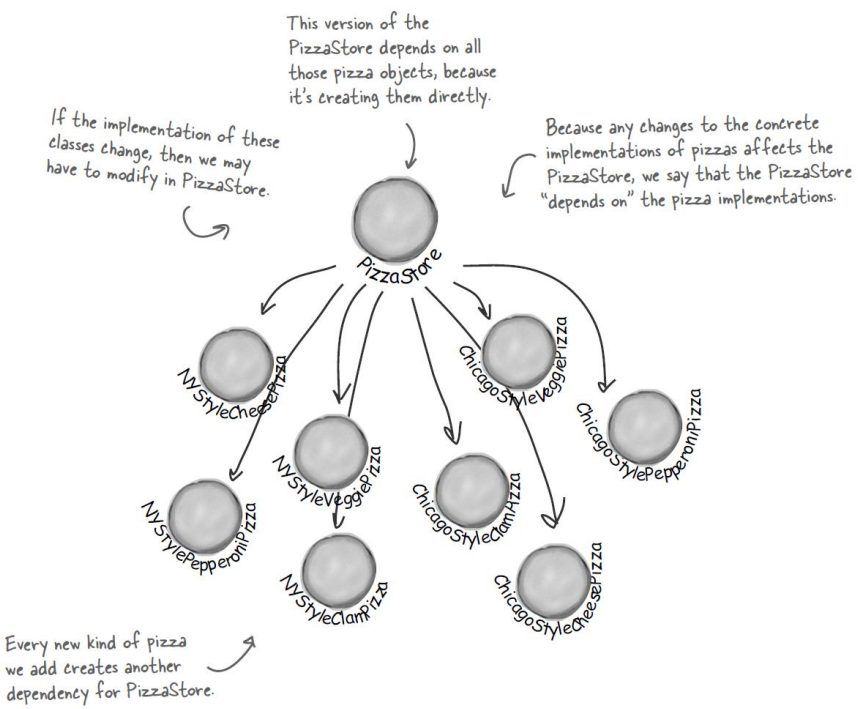


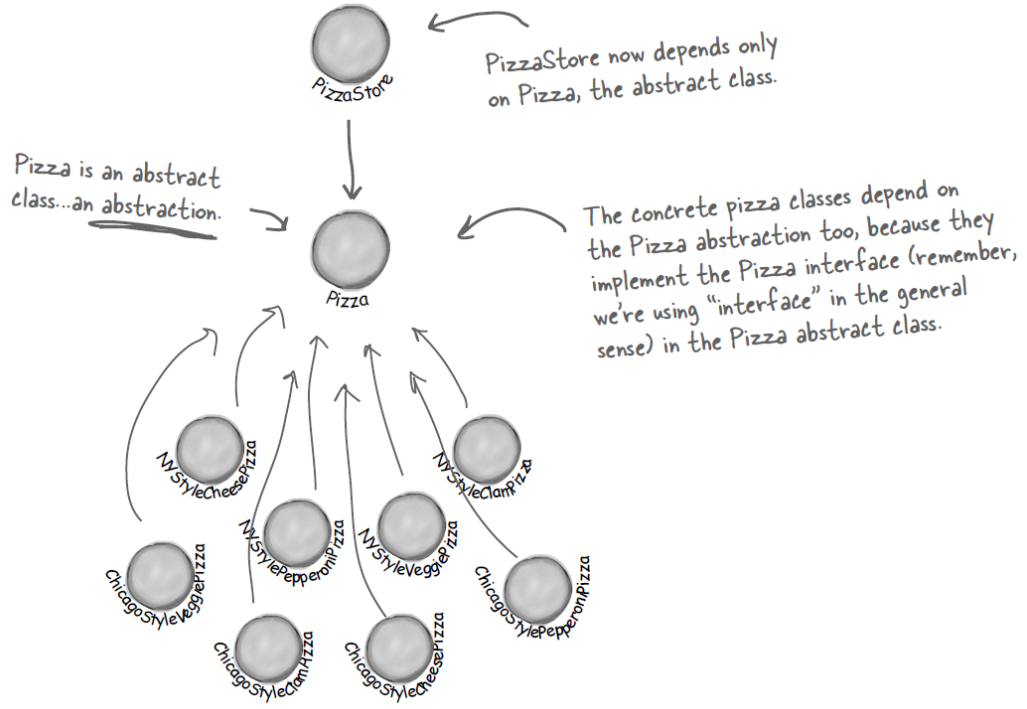
BEFORE: a high-level class depends on a low-level class.



AFTER: low-level classes depend on a high-level abstraction.

DIP dæmi 2





Dæmi úr Head First Design Patterns

„Inverision-ið“ hér er að það er ekki verið að depend-a á concrete implementation-in af pizza tegundum heldur eru það pizza tegundirnar sem eru að depend-a á Pizza abstract klasann

DIP dæmi 3

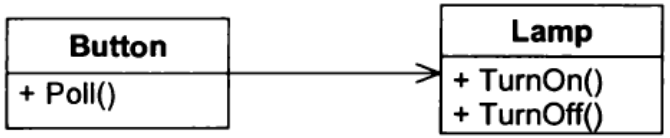


Figure 11-3 Naive Model of a Button and a Lamp

```
public class Button
{
  private Lamp itsLamp;
  public void poll()
  {
    if (/*some condition*/)
      itsLamp.turnOn();
  }
}
```

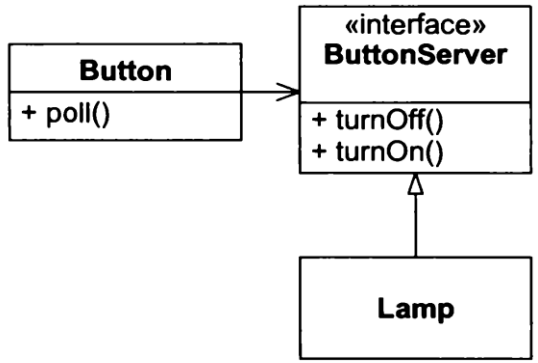


Figure 11-4 Dependency Inversion Applied to the Lamp

DIP dæmi 4

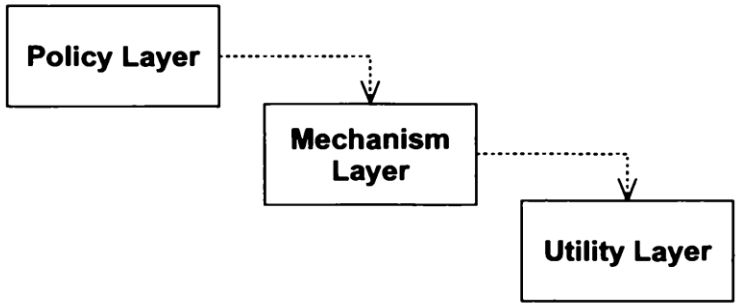


Figure 11-1 Naive layering scheme

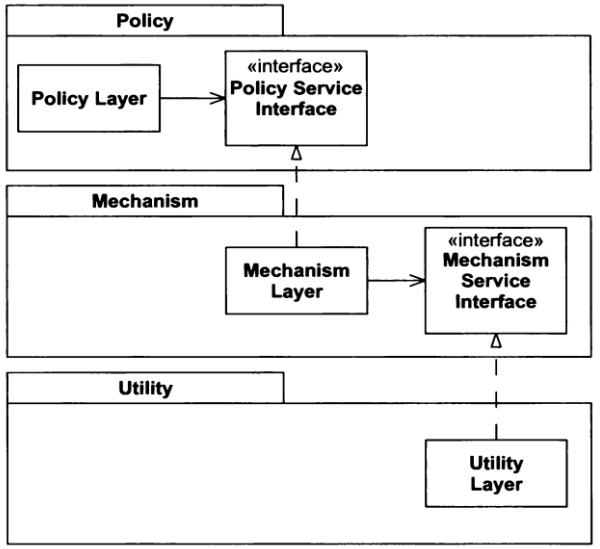
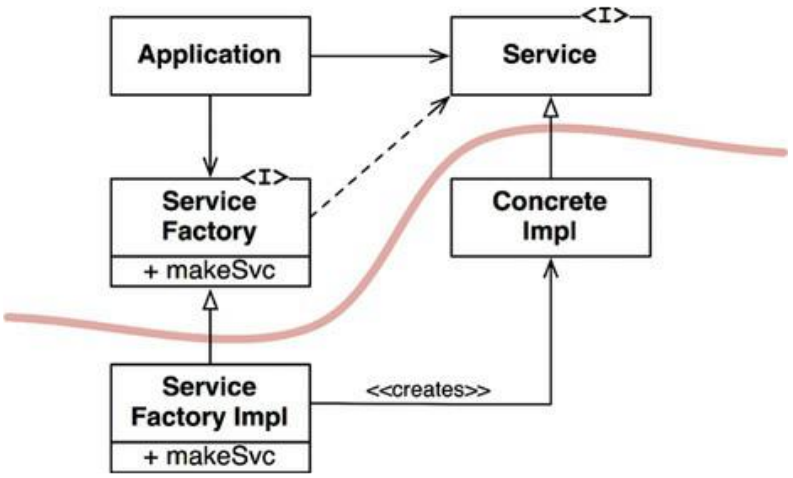


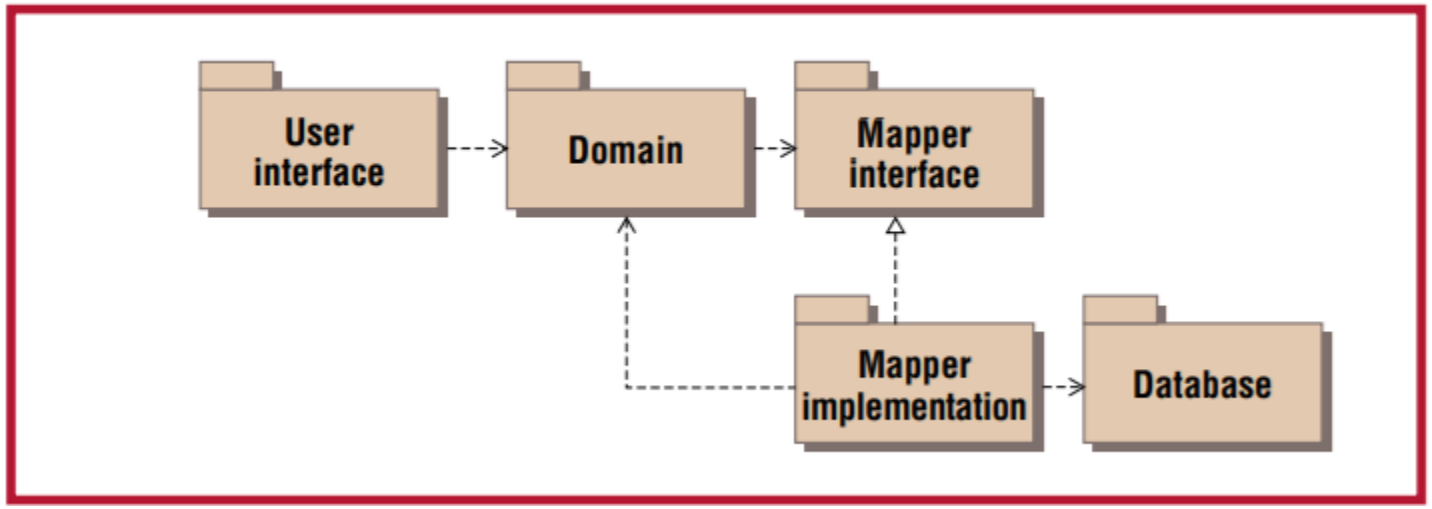
Figure 11-2 Inverted Layers

DIP dæmi 5



Línan er architectural boundary, „seperates the abstract from the concrete“

DIP dæmi 6



https://martinfowler.com/ieeeSoftware/coupling.pdf

Önnur speki

- Conway‘s Law
  - *“Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.”*
- Brook‘s Law
  - *“Adding manpower to a late project makes it later.”*
- Hofstadter’s Law
  - *“It always takes longer than you expect. (Even when you factor in Hofstadter’s law.)”*
- Linus’s Law
  - *“Given enough eyeballs, all bugs are shallow.”*
- Gall’s Law
  - *“A complex system that works has evolved from a simple system that worked. A complex system built from scratch won’t work.”*
- Eagleson’s Law
  - *“Any code of your own that you haven’t looked at for six or more months might as well have been written by someone else.”*
- Miller’s Law
  - *“To understand what another person is saying, you must assume that it is true and try to imagine what it could be true of.”*
- The 90-90 rule
  - *“The first 90 percent of the code accounts for the first 90 percent of the development time. The remaining 10 percent of the code accounts for the other 90 percent of the development time.”*
- Humphrey’s Law
  - *“For a new software system, the requirements will not be completely known until after the users have used it.”*
- North’s Law
  - *“Every decision is a trade off.*
- Occam’s Razor
  - *If we face two possible explanations which make the same predictions, the one based on the least number of unproven assumptions is preferable, until more evidence comes along.*
  - *“The simplest solution is almost always the best (simple meaning having few assumptions)”*