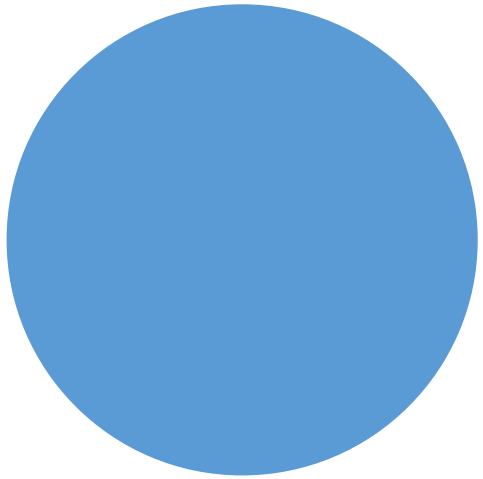
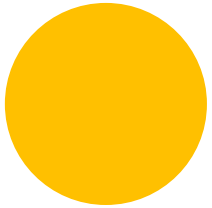



Programming I

Course 13

Introduction to programming

What we talked about?

- Object Oriented Analyze/Design/Programming

- How to represent classes?
  - Code
  - Graphical notation
  - Informal notation

What we talk today?

- Object Oriented Principles
  - SOLID
  - GRASP

What can help to design an application?

- **Experience** and common sense
- Using OO **principles**
- Design patterns [not object of this course]

OO Principles

- Principle
  - Is a principle or basic technique that can be applied to design or write easy to maintain, flexible and extensible code

- Principiile OOD – SOLID
  - SRP - Single-responsiblity principle
  - OCP - Open-closed principle
  - LSP - Liskov substitution principle
  - ISG - Interface segregation principle
  - DRY - Dependency Inversion Principle

SRP - Single-responsiblity principle

- Every object in the system should have a single responsibility, and all the object's services should be focused in carrying out that single responsibility.
- ONLY one reason to change something!
- Code will be simpler and easier to maintain.
- Example
  - Container and Iterator (Container manages objects; Iterator traverses the container)
- How to spot multiple responsibilities?
  - Forming sentences ending in itself.

SRP - Single-responsiblity principle

- Every object in the system should have a single responsibility, and all the object's services should be focused in carrying out that single responsibility.
- How to spot multiple responsibilities?
  - Forming sentences ending in itself.

Automobile
+ start() +stop() +change_tires() +drive() +check_oil() +change_oil()

The Automobile can start itself.
The Automobile can stop itself.
The Automobile can change tires itself.
The Automobile can drive itself.
The Automobile can check oil itself.
The Automobile can change oil itself.

Automobile
+ start() +stop() +drive() +check_oil()

Mechanic
+change_tires() +change_oil()

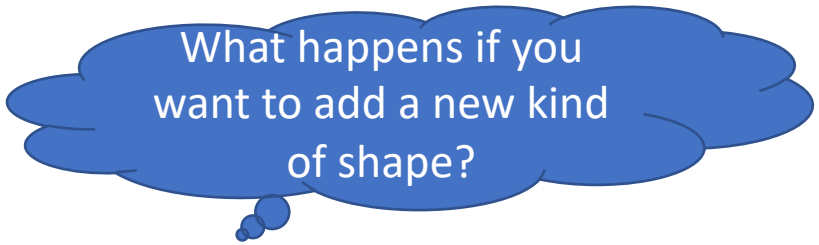
OCP - Open-closed principle

- OCP – Classes should be open for extension and closed for modification
- Allowing change, but without modifying existing code => flexibility.
- Use inheritance to extend/change existing working code and don’t touch working code.
- OCP can also be achieved using composition.

OCP - Open-closed principle

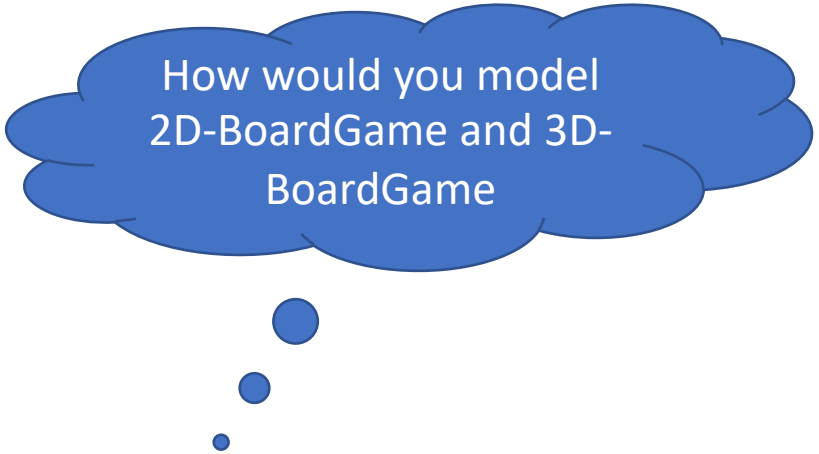
```
class Shape(object):
    def __init__(self, type):
        self.type = type

    def draw(self):
        if self.type == "Circle":
            print("Draw Circle")
        if self.type =="Rectangle":
            print("Draw Rectangle")

C = Shape("Circle")
```

```
class Shape(Object):
    def __init__(self):
        pass

class Circle(Shape):
    def draw(self):
        print("Draw Circle")

class Rectangle(Shape):
    def draw(self):
        print("Draw Rectangle")
```

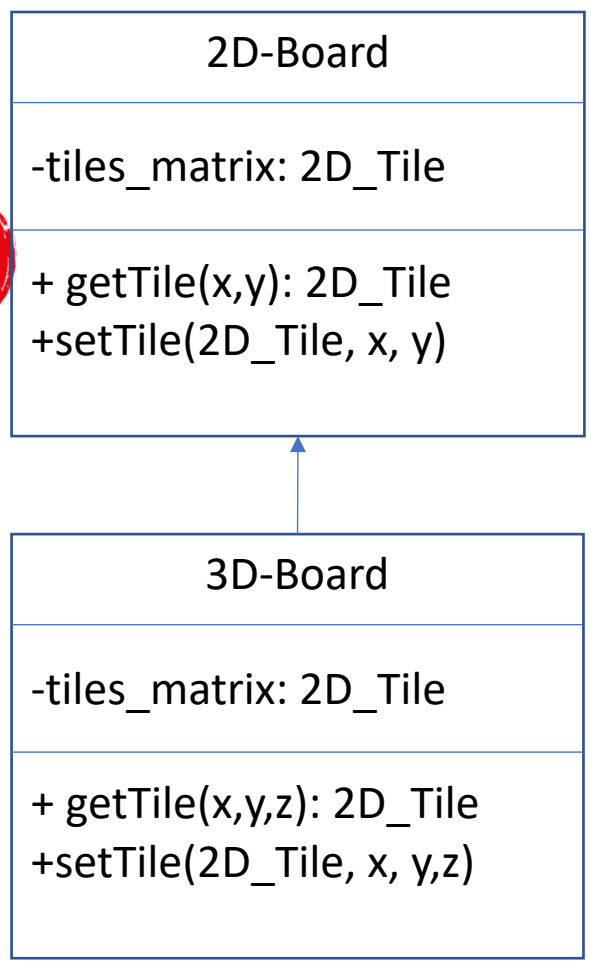


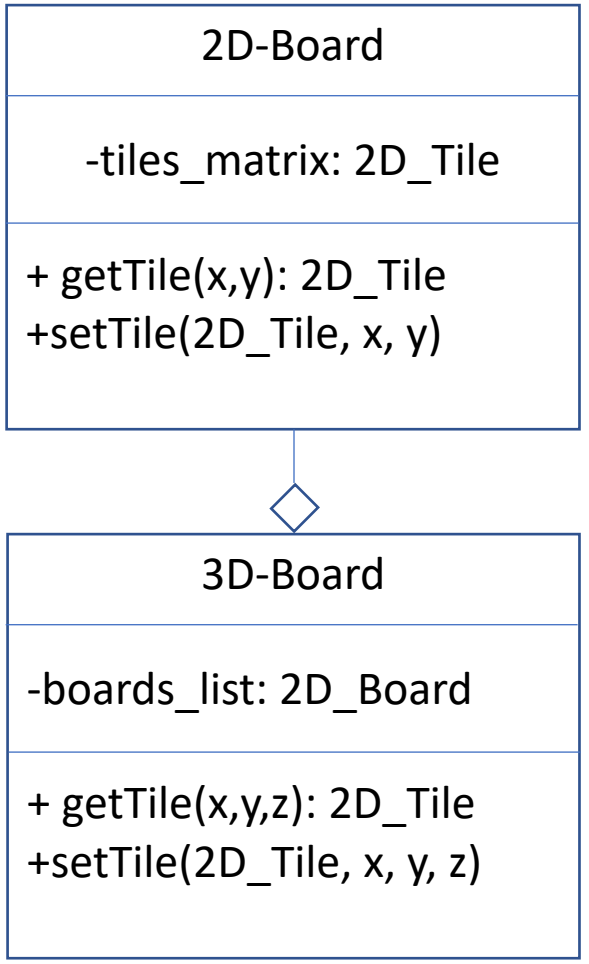
LSP - Liskov substitution principle

- Subtypes must be substitutable for their base types.
- Well-designed class hierarchies
- Subtypes must be substitutable for their base class without things going wrong.



LSP - Liskov substitution principle



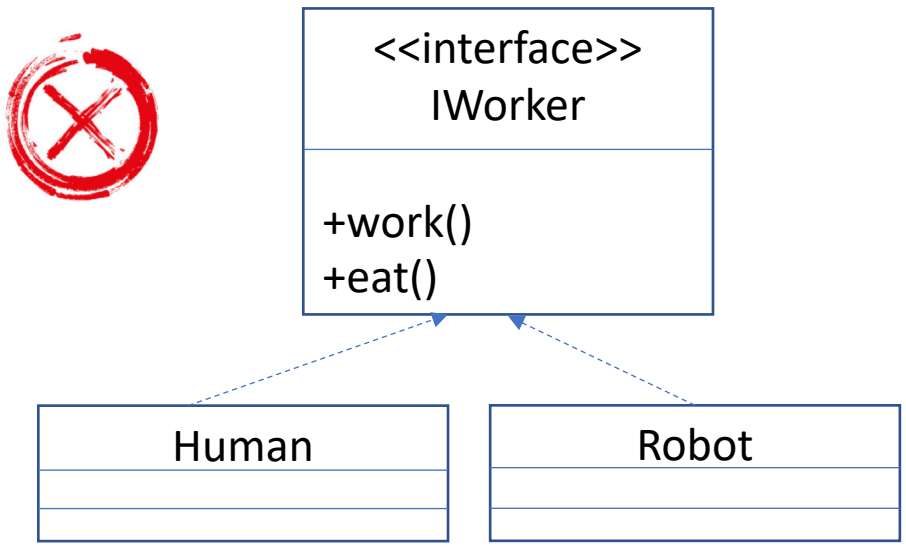


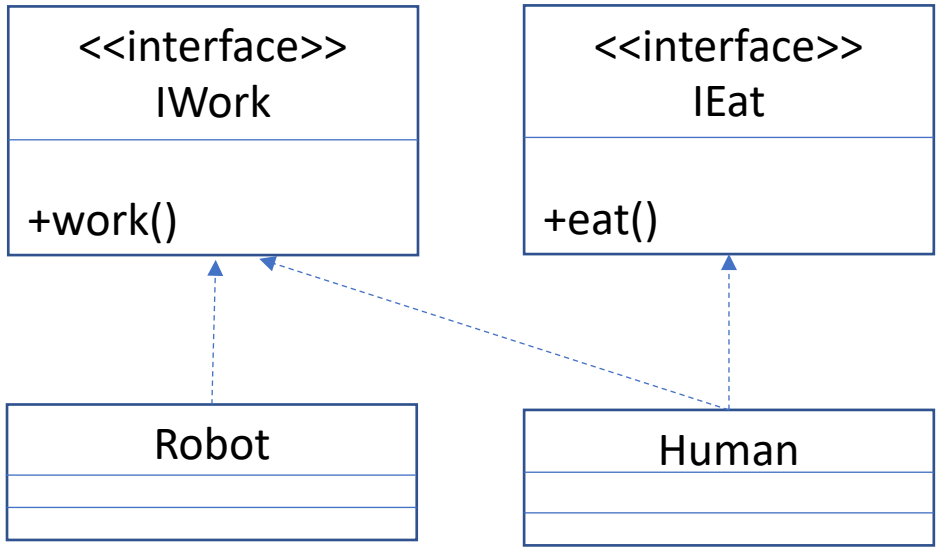

```
board = 3D-Board()
board.getTile(4,5) // does not make sense of 3D board
```

```
board = 3D-Board()
board.getTile(1,4,5)
```

ISG - Interface segregation principle

- Clients should not be forced to depend on methods they do not use
- Keep interfaces small, cohesive, and focused
- Whenever possible, let the client define the interface





DRY - Dependency Inversion Principle

- High-level modules should not depend on low-level modules.
  - Both should depend on abstractions
- Abstractions should not depend on details.
  - Details should depend upon abstractions
- Detail should be dependent on Policy.
  - This means that you should have the Policy define and own the abstraction that the detail implements

DRY - Dependency Inversion Principle

```
class Worker(object):
   def work(self):
      print("… working")

class Manager(object):
  def __init__(self, worker):
     if not isinstance(worker, Worker):
        raise TypeError("Unexpectd Type")
     self.worker = worker
  def manage(self):
    self.worker.work()

class SuperWorker(object):
   def work(self):
        print("… working much more")
```

```
class IWorker(self):
  def work(self):
    pass
class Worker(IWorker):
  def work(self):
      print("… working")
class SuperWorker(IWorker):
  def work(self):
      print("… working much more ")

class Manager(object):
  def __init__(self, worker):
      if not isinstance(worker, IWorker):
         raise TypeError("Unexpectd Type")
     self.worker = worker
  def manage(self):
    self.worker.work()
```

GRASP

- GRASP
  - General
  - Responsibilities
  - Assignment
  - Software
  - Patterns (Principles)

- Describe fundamental principles of object design and responsibility

- Name chosen to suggest the importance of **grasp**ing fundamental principles to successfully design object-oriented software

GRASP Patterns

- Pattern
  - a named and well-known problem/solution pair that can be applied in new contexts, with advice on how to apply it in new situations and discussion of its trade-offs, implementations, variations, etc.

- A pattern is characterized by
  - A name
  - A problem it tries to solve
  - A solution

Patterns in engineering

- *How do other engineers find and use patterns?*
  - Mature engineering disciplines have handbooks describing successful solutions to known problems
  - Automobile designers don't design cars from scratch using the laws of physics
  - Instead, they reuse standard designs with successful track records, learning from experience
  - *Should software engineers make use of patterns? Why?*

- Developing software from scratch is also expensive
  - Patterns support reuse of software architecture design

GRASP Patterns

1. Information Expert
   - assign responsibilities to class with knowledge
2. Creator
   - knows the necessary details to create
3. Low Coupling
   - reduce connectivity
4. Controller
   - use cases or system based classes
5. High Cohesion
   - does related things
6. Polymorphism
   - behaviour depends on the type
7. Pure Fabrication
   - class based in software world
8. Indirection
   - avoid direct coupling with an intermediary
9. Protected Variations
   - information hiding - open/close

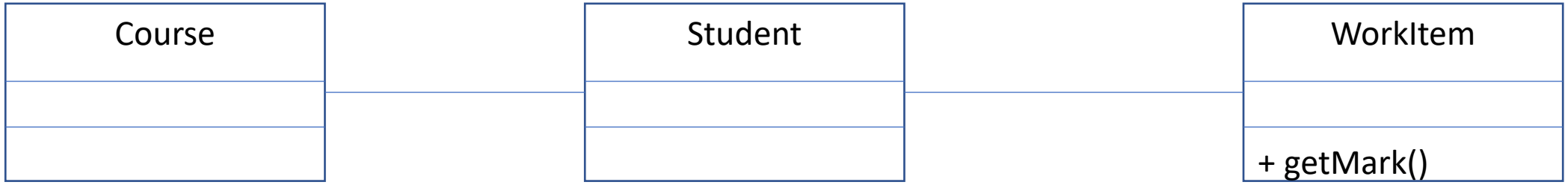
Information Expert Pattern

- Problem
  - What is a general principle for assigning responsibilities/functions to objects?

- Solution
  - Assign a responsibility to the information expert, that is, the class that has the information necessary to fulfill the responsibility.

Information Expert Pattern

- Problem
  - Which class should determine the final mark the student receives in a course?
- Discussions
  - WorkItem?
    - The class can determine the value of an individual items, they can not determine the final mark.
  - Student?
    - The class should be assigned this responsibility since it knows about all of the work items (does not understand how the mark is calculated).
    - The class rely on the WorkItem class to determine the individual marks.



Read world analogy: who do you ask about X, you ask the expert who knows about X.

Information Expert Pattern

- The marking system can be modeled with the following domain classes: WorkItem, MarkingScheme, Student and Course.

- Consider the following responsibilities:
  - the calculation of the final grade for a student,
  - editing a working item,
  - the report of all the grades in a class,
  - a list of all the student name and numbers in the class,

- Using the Expert design pattern, decide which class if possible of the domain class should be assigned the given responsibility.

- If no domain class is possible, suggest a software class that should be responsible.

Creator pattern

- Problem
  - Who creates an instance of A?

- Solution
  - Assign B the responsibility to create an instance of class A if one of the following is true
    - B contains or aggregates A objects (in a collection)
    - B records instances of A objects
    - B closely uses A objects
    - B has the initializing data that will be passed to A when it is created.

Creator pattern

- The marking system can be modeled with the following domain classes: WorkItem, MarkingScheme, Student and Course.

- Problems
  - How is responsible for MarkingScheme creation?
  - How is responsible for WorkItem creation?

Low Coupling

- Problem
  - How to support low dependency, low change impact, increased reuse?
- Solution
  - Assign a responsibility so coupling is low.
- Coupling
  - a measure of how strongly one element is connected to, has knowledge of, or relies on other elements

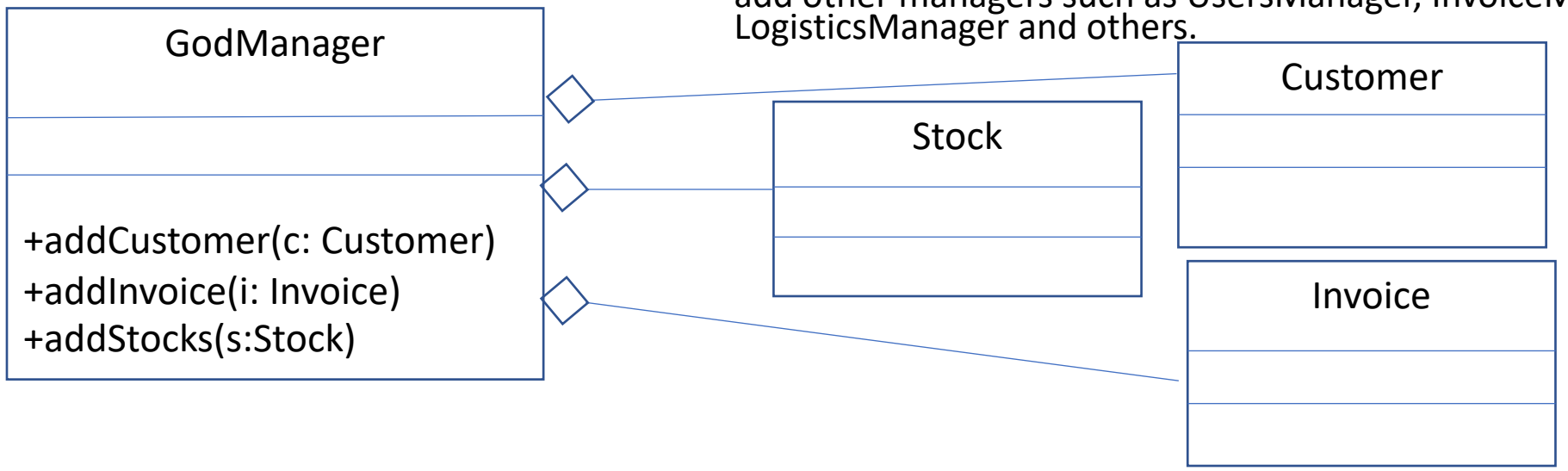
Low Coupling

- Coupling
  - a measure of how strongly one element is connected to, has knowledge of, or relies on other elements
- Classes with strong coupling
  - suffer from changes in related classes
  - are harder to understand and maintain
  - are more difficult to reuse
- But coupling is necessary if we want classes to exchange messages!
- The problem is too much of it and/or too unstable classes.

Low Coupling

- Manager include the logic for working with
  - customers
  - Invoices
  - Logistics
  - …

- Simply for everything.
  - "god objects" -> have too much responsibility -> create too many coupling
- The total number of references in the application is not that important, it's the number of references between objects what matters.
- Always try to make the class communicates with as few other classes as possible,
  - add other managers such as UsersManager, InvoiceManager, LogisticsManager and others.



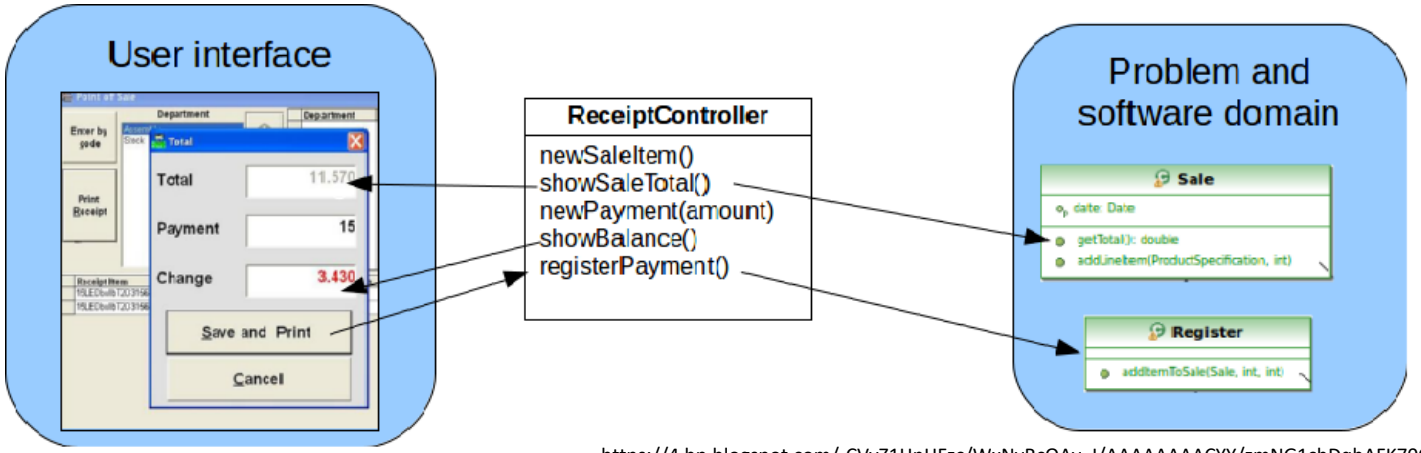
Controller

- Problem
  - Which first object beyond the User Interface (UI) layer receives and coordinates a system operations? (Who should be responsible for handling a system event?)

- Solution
  - Assign the responsibility for receiving and/or handling a system event to one of following choices:
    - Object that represents overall system, device or subsystem (*façade controller*)
    - Object that represents a use case scenario within which the system event occurs (a <UseCase>Handler)

Controller

- Controller classes provides the *glue* between the system events and software model.
- Entity, Boundary, and Control Objects
  - Entity objects are instances of domain classes.
  - Boundary objects represent the interaction between actors and the system
  - Control objects are in charge of realizing use cases.



https://4.bp.blogspot.com/-CVyZ1HnHFzc/WxNyRcQAv_I/AAAAAAAACYY/zmNG1chDqhAFK70OOz_cuyJdZX1-43jpgCLcBGAs/s1600/controler.png

High Cohesion

- Problem
  - How to keep objects focused, understandable and manageable?
- Solution
  - Assign the responsibility so that cohesion remains high.
- Cohesion – a measure of how strongly related and focused the responsibilities of an element (class, subsystem, etc.) are

High Cohesion

- Degrees of cohesion
  - Very low cohesion
    - A class is solely responsible for many things in very different functional areas. If most programs are implemented in one class, then that class would have very low cohesion.
  - Low cohesion
    - A class has sole responsibility for a complex task in one functional area.
  - High cohesion
    - A class has moderate responsibilities in one functional area and collaborates with other classes to fulfill the task.
    - A real world analogy of low cohesion is a person that takes on too many unrelated responsibilities, especially ones that should properly be delegated to others

Polymorphism

- Problem
  - How to handle related but varying elements based on element type?
- Solution
  - Polymorphism guides us in deciding which object is responsible for handling those varying elements.
- Benefits
  - handling new variations will become easy.

Pure Fabrication

- Problem
  - What object should have the responsibility, when you do not want to violate High Cohesion and Low Coupling, or other goals, but solutions offered by Expert are not appropriate.

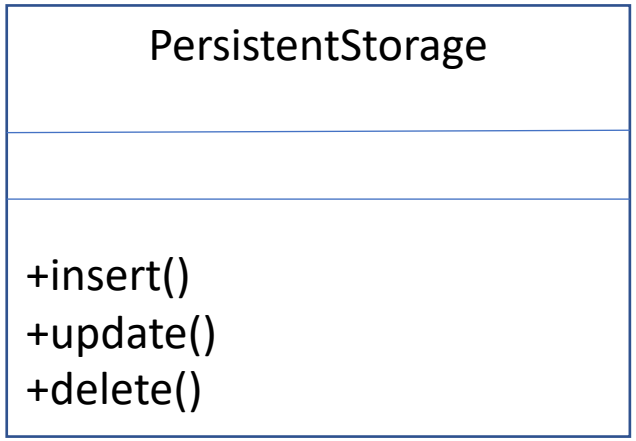
- Solution
  - Assign a highly cohesive set of responsibilities to an artificial or convenience class that does not represent a problem domain concept -- something made up, to support high cohesion, low coupling and reuse.

Pure Fabrication

- Pure Fabrication suggests to create a new class for these new responsibilities

- Example
  - Store the Course into a persistent format

- *PersistentStorage is a fabrication*
  - *it is made up from your imagination; it cannot be found in the Domain Model*
- Course remains well-designed - high cohesion, low coupling
- PersistentStorage class is relatively cohesive - sole purpose is to store/retrieve objects to/from a persistent system (database, files, …)

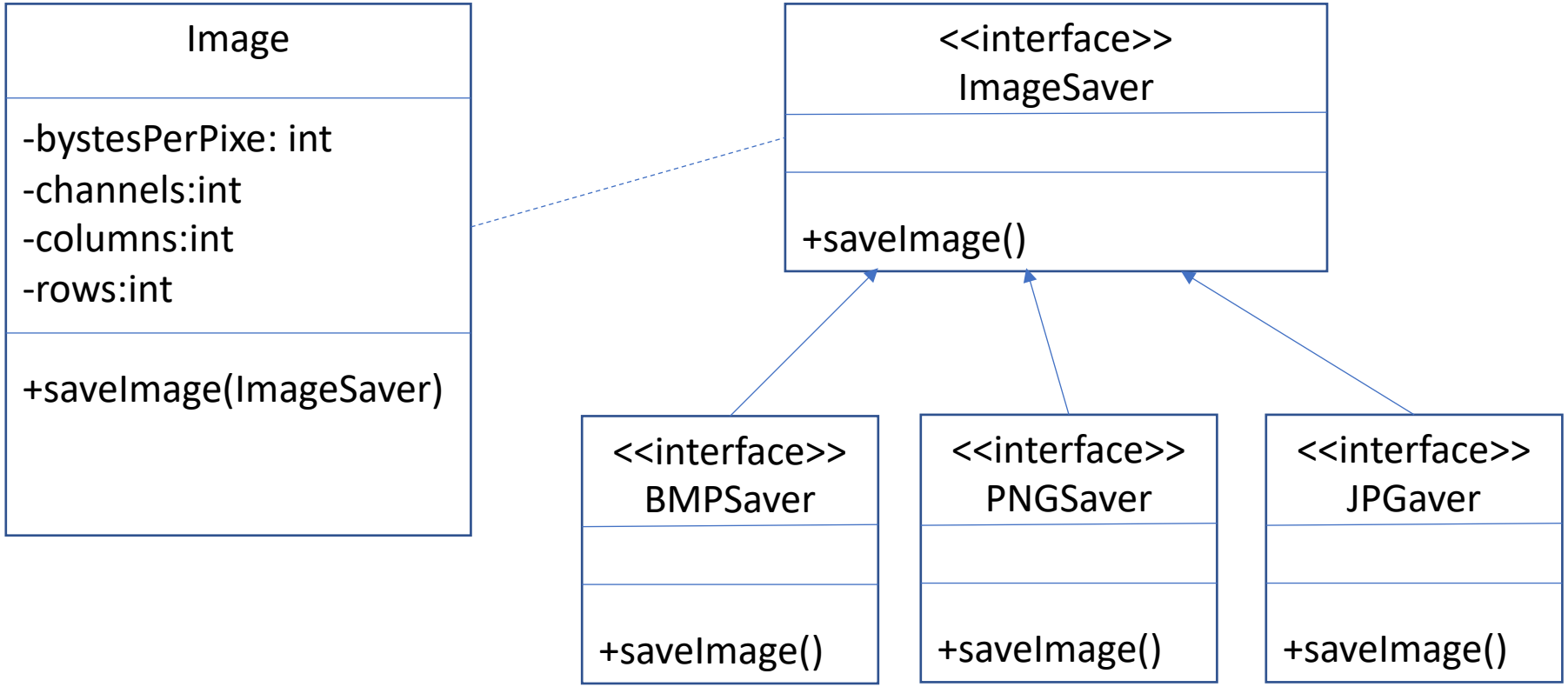


Pure Fabrication

Without Pure Fabrication

Image
-bystesPerPixe: int -channels:int -columns:int -rows:int
+saveBMPImage() +savePNGImage +saveJPEGImage()

Using Pure Fabrication



Indirectation

- Problem
  - How can we avoid a direct coupling between two or more elements?

- Solution
  - Indirection introduces an intermediate unit to communicate between the other units, so that the other units are not directly coupled.

Indirectation

- Problem
  - In a Sale System, there are multiple external third-party tax calculators that must be supported
  - Sale class is responsible to tax calculation
  - We want to keep the system independent from the varying external tax calculators

Protected Variation

- Problem
  - How to avoid impact of variations of some elements on the other elements.
- Solution
  - Provide a well defined interface so that the there will be no affect on other units.
  - Provides flexibility and protection from variations.
  - Provides more structured design.
- Example: polymorphism, data encapsulation, interfaces

Protected Variation. Examples

- Data encapsulation, interfaces, polymorphism, indirection, and standards are motivated by PV.
- Virtual machines are complex examples of indirection to achieve PV
- Service lookup: Clients are protected from variations in the location of services, using the stable interface of the lookup service.
- Uniform Access Principle
- …

Conclusions

- GRASP provides a map of considerations to provide strong guidance for an OO designer
- But at the same time, GRASP still leaves a lot of room to the designer and creating a good design is still an art!
- Taking a look at GRASP—and really *Applying UML and Patterns*—is a good bet for OO designers who know the basics of OOP but are still inexperienced

Bibligraphy

- Larman, Craig. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Third edition, Prentice Hall, 2005

- Wirfs-Brock, RebeccaandMcKean, Alan. *ObjectDesign: Roles, Responsibilities, and Collaborations*. Addison-Wesley Professional, 2002

- Evans, Eric. *Domain-DrivenDesign: Tackling Complexity in the Heartof Software*. Addison-Wesley Professional, 2003