Padrões e Desenho de Software

UA.DETI.PDS – 2022/23

José Luis Oliveira

Objetivos

- Compreender os âmbitos das abstrações: princípios de desenho, padrões de software e estilos de arquitetura de software
- Analisar um projeto de software de um sistema (ou parte) e identificar de forma crítica as opções de desenho e o seu possível impacto no desempenho
- Propor e defender uma solução para um projeto de software baseada nas boas práticas, princípios e padrões abordados na unidade curricular
- Rever código com suporte em ferramentas e em regras de boas práticas
- Identificar soluções expeditas / plataformas que possam ser alternativa à implementação de raiz de uma dada solução de parte de um sistema

Programa

- **Princípios de desenho de software**
  - Princípios e orientações (e.g. GRASP)
  - Impacto destes no processo de desenho de um sistema nomeadamente ao nível flexibilidade, de facilidade de manutenção e reutilização de componentes.

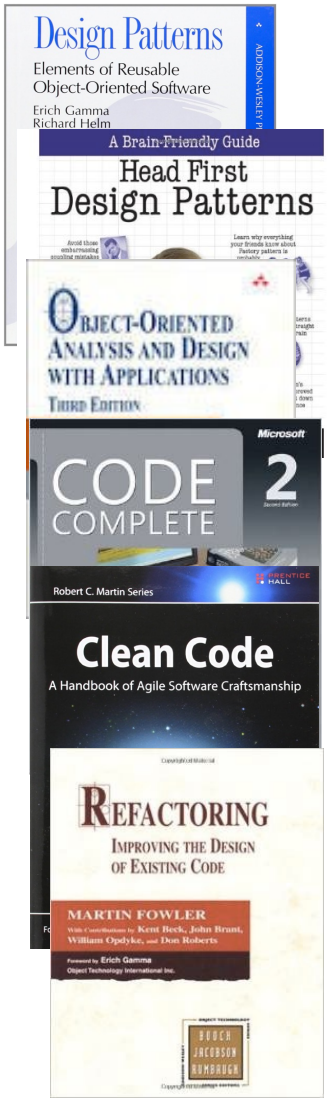
- **Revisão e melhoria do código**
  - Métricas sobre código
  - Revisão de código, "Bad smells" e reengenharia
  - Anti-padrões frequentes

Programa (cont.)

- **Padrões de desenho de software**
  - Apresentação dos principais padrões de desenho (e.g. GoF)
  - Identificação e estudo das características de padrões de desenho em casos concretos

- **Estilos de arquitetura de software**
  - Estilos de arquitetura estruturais e de interação/concorrência
  - Identificação e estudo das características de padrões de desenho em casos concretos
  - Introdução a padrões de concorrência: Sistemas distribuídos como um conjunto de componentes que cooperam entre si

Bibliografia principal



- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994, **Design Patterns: Elements of Reusable Object-Oriented Software**, Addison-Wesley Professional).
- E. Freeman, K. Sierra, B. Bates (2004), **Head first design patterns**, O'Reilly.
- Craig Larman (2004), **Applying UML and patterns : an introduction to object - oriented analysis and design and iterative development**, Prentice Hall PTR.
- Steve McConnell (2004), **Code Complete: A Practical Handbook of Software Construction**, Microsoft Press.
- Robert C. Martin (2008), **Clean code: a handbook of agile software craftmanship**. Prentice Hall.
- Martin Fowler, et al (2000). **Refactoring: Improving the Design of Existing Code**, Addison-Wesley.

Avaliação

- A avaliação da disciplina será discreta, sem avaliação na época normal:
- **Avaliação Teórico-Prática (ATP) - 40%**
  - ATP1 (20%): 12/04/2023
  - ATP2 (20%): 31/05/2023
- **Avaliação Prática (AP) - 60%**
  - AC (15%): correspondentes ao desempenho nas aulas, realização de guiões e trabalhos individuais
  - EP (45%): Exame prático, 31.05.2023

- A nota mínima para cada uma das duas componentes (Teórica e Prática) é de 7 valores.

Avaliação (cont.)

- Modelo de funcionamento das aulas práticas
  - Nas aulas terão de usar um **portátil pessoal** com o software necessário para cada módulo.
  - É importante a **assiduidade**, a **preparação** prévia, a discussão durante a aula, a **entrega** de todos os guiões.
  - **Entrega semanal** de trabalhos, até 48h após a aula prática

ECTS

- Escolaridade (T/TP/P): 0/2/2 - ECTS: 6

- O número de créditos ECTS indica o número de horas expectável que devem estudar para esta disciplina.
  - 1 ECTS = 25-30 horas de estudo.
  - 6 ECTS = 150-180 horas de estudo.
- Num semestre com 15 semanas devem estudar pelo menos 10 horas por semana.
- Estas horas incluem: aulas presenciais, leitura de livros, resolução de exercícios, estudo para testes e exames, etc.

Recursos

- elearning.ua.pt
  - Slides TP
  - Guiões Práticos
  - Entregas
  - Fóruns
  - Informações e resultados
- Links
  - http://sourcemaking.com/
  - https://refactoring.guru/design-patterns
  - http://www.tutorialspoint.com/design_pattern/
  - http://www.oodesign.com/
  - …

Docentes

- José Luis Oliveira (jlo@ua.pt) – TP, P2, P5
- Carlos Costa (carlos.costa@ua.pt) – P3, P4
- José Moreira (jose.moreira@ua.pt) – P1
- Rafael Direito (rafael.neves.direito@ua.pt)
- As OTs funcionarão por *marcação*.
  - *Por favor envie email para o docente até às 12h do dia anterior à OT que pretende agendar.*

Bons estudos e bom semestre!

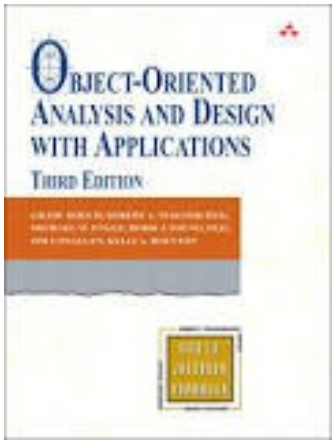
Introduction to Software Design

UA.DETI.PDS - 2022/23

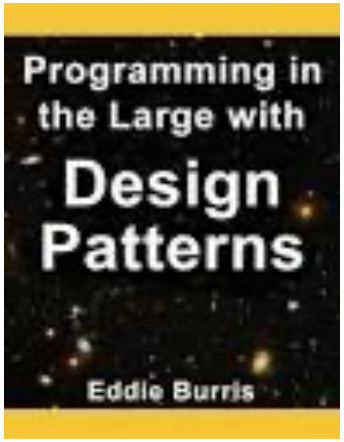
José Luis Oliveira

Resources & Credits

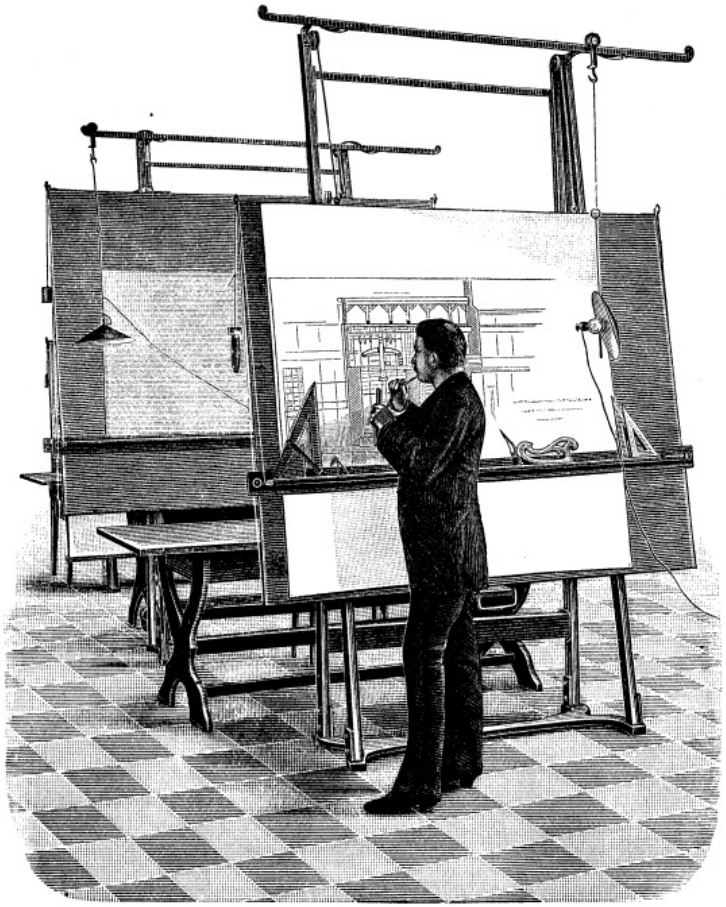
- **Object-Oriented Analysis and Design with Applications**
  Grady Booch, Robert A. Maksimchuk, Michael W. Engle, Bobbi J. Young, Jim Conallen, Kelli A. Houston
  Addison-Wesley Professional; 3rd edition



- **Programming in the Large with Design Patterns**
  Eddie Burris
  Pretty Print Press



Design



"You can use an eraser on the drafting table or a sledgehammer on the construction site.“

--Frank Lloyd Wright

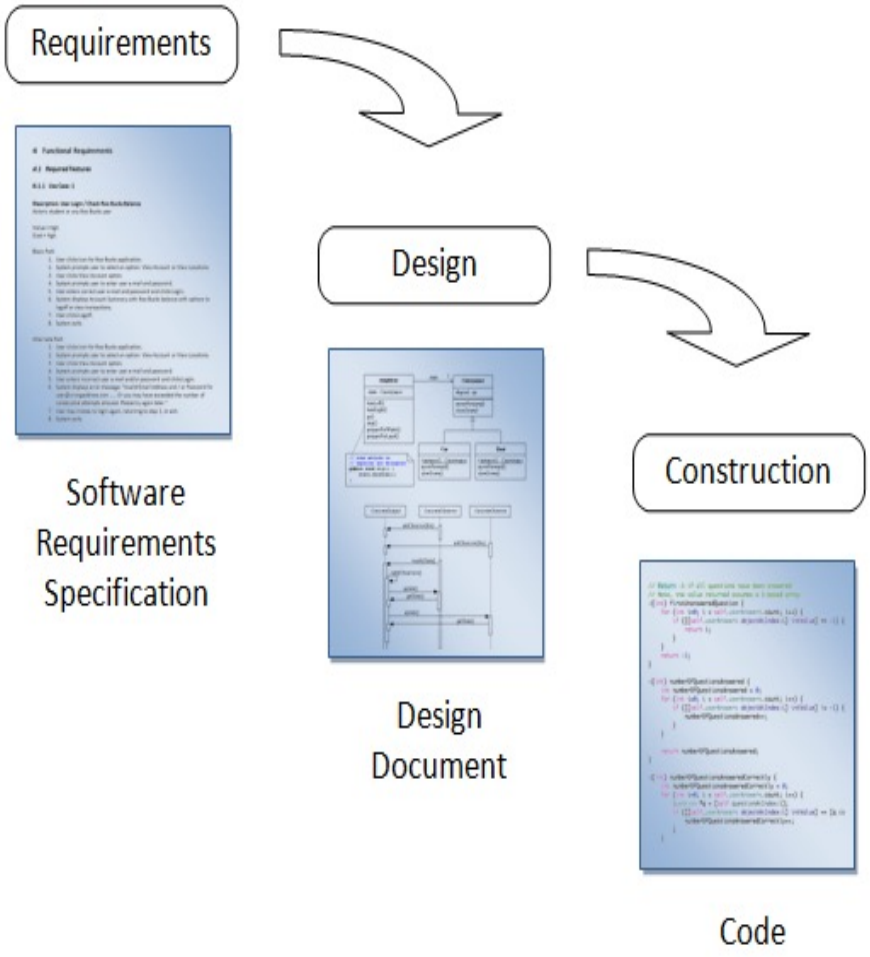
Design is a Universal Activity

- Any product that is an aggregate of more primitive elements, can benefit from the activity of design.

Building Design	Landscape Design	User Interface Design	Software Design
			
Doors, windows, plumbing fixtures, … Wood, steel, concrete, glass, …	Trees, flowers, grass, rocks, mulch, …	Tree view, table view, File chooser, … Buttons, labels, text boxes, …	Classes, procedures, functions, … Data declaration, expressions, control flow statements, …

What is Software Design?

- Design bridges the gap
  - between knowing what is needed (software requirements specification)
  - to entering the code that makes it work (the construction phase).

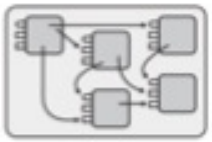


What is Software Design?

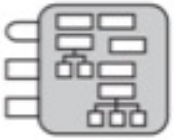
- Design is needed at several different levels of detail in a system:

– system

– subsystems or packages: user interface, data storage, application-level classes, graphics . . .



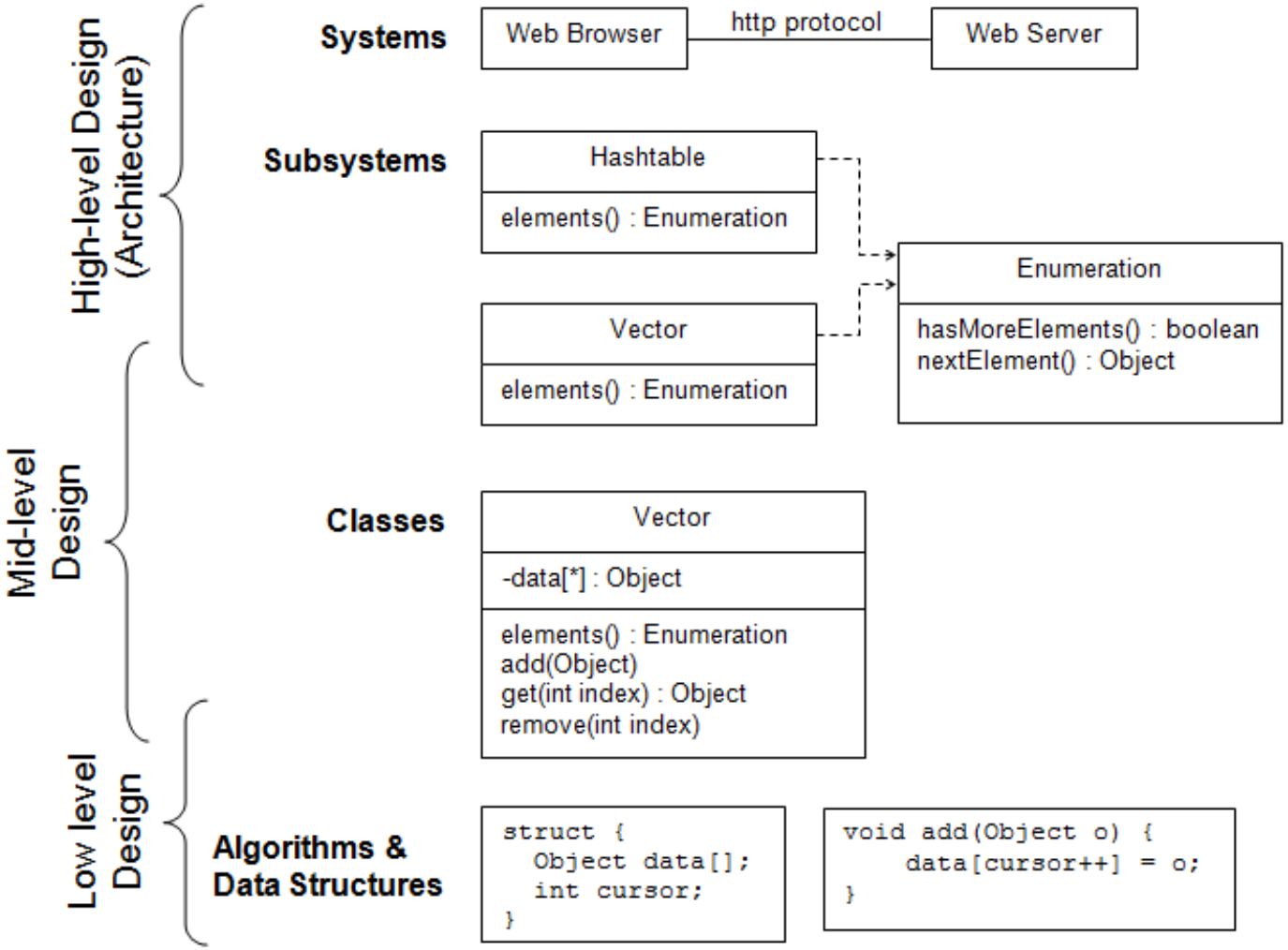
– classes within packages, class relationships, interface of each class, public methods



– attributes, private methods, inner classes . . .

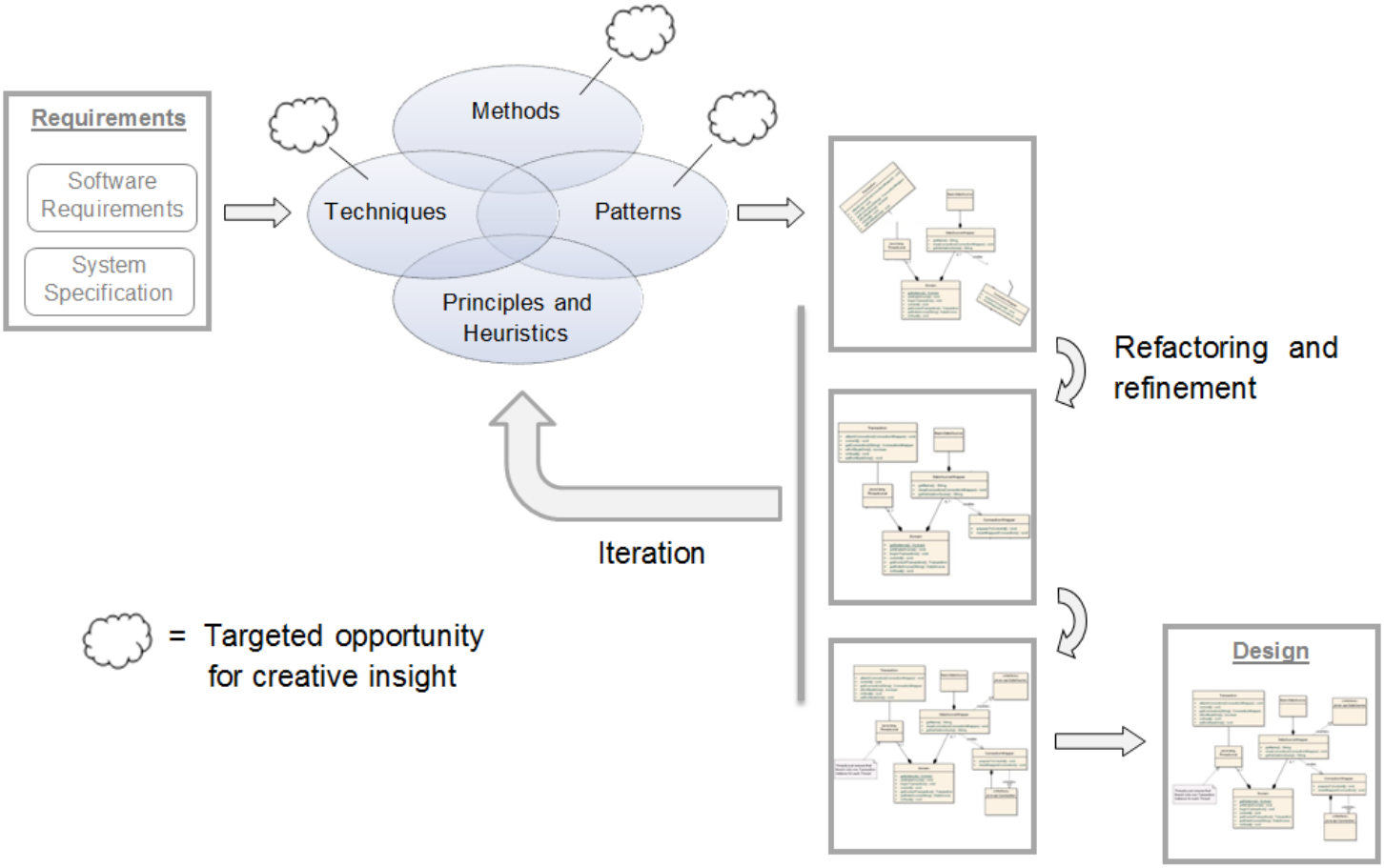
– source code implementing methods

Design Occurs at Different Levels



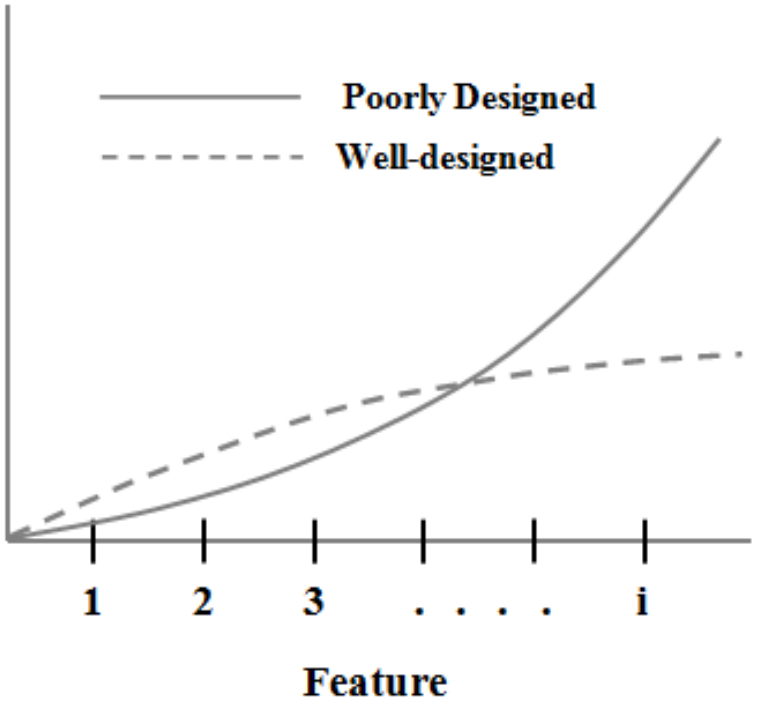
Importance of Software Design

- The design process can be made more systematic and predictable through the application of methods, techniques and patterns, all applied according to principles and heuristics.



Importance of Managing Complexity

- Poorly designed programs are difficult to understand and modify.
- The larger the program, the more pronounced are the consequences of poor design.



Two Types of Complexity in Software

- **Essential complexities**
  - complexities that are inherent in the problem.
- **Accidental/incidental complexities**
  - complexities that are artifacts of the solution.

- The total amount of complexity in a software solution is:
  - Essential Complexities + Accidental complexities

- The primary purpose of design is to control complexity
  - Goal: manage essential complexity while avoiding the introduction of additional accidental complexities

Dealing with Software Complexity

- Modularity – subdivide the solution into smaller easier to manage components. (divide and conquer)
- Abstraction – use abstractions to suppress details in places where they are unnecessary.
- Information Hiding – hide details and complexity behind simple interfaces
- Inheritance – general components may be reused to define more specific elements.
- Composition – reuse of other components to build a new solution

Design is a wicked problem

- A wicked problem is one that can only be clearly defined by solving it.

"TEX would have been a complete failure if I had merely specified it and not participated fully in its initial implementation. The process of implementation constantly led me to unanticipated questions and to new insights about how the original specifications could be improved."

Donald Knuth

Characteristics of Software Design

- **Non-deterministic**
  - No two designers or design processes are likely to produce the same output.
- **Heuristic**
  - Design techniques tend to rely on heuristics and rules-of-thumb rather than repeatable processes.
- **Emergent**
  - The final design evolves from experience and feedback. Design is an iterative and incremental process where a complex system arises out of relatively simple interactions.

A Generic Design Process

- Understand the problem (software requirements).
- Construct a "black-box" model of solution (system specification).
  - System specifications are typically represented with use cases (especially when doing OOD).
- Look for existing solutions (e.g., architecture and design patterns) that cover some or all of the software design problems identified.
- Consider building prototypes
- Document and review design
- Iterate over solution (Refactor)
  - Evolve the design until it meets functional requirements and maximizes non-functional requirements

Inputs to the design process

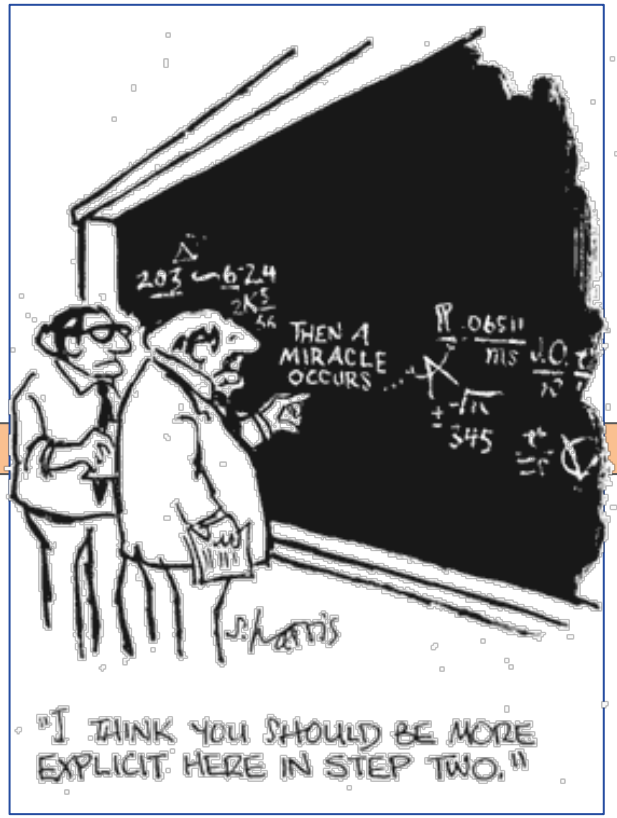
- User requirements and system specification
  - including any constraints on design and implementation options
- Domain knowledge
  - For example, if it's a healthcare application the designer will need some knowledge of healthcare terms and concepts.
- Implementation knowledge
  - capabilities and limitations of eventual execution environment

Desirable Internal Design Characteristics

- **Minimal complexity** – Keep it simple. Maybe you don't need high levels of generality.
- **Loose coupling** – minimize dependencies between modules
- **Ease of maintenance** – Your code will be read more often then it is written.
- **Extensibility** – Design for today but with an eye toward the future. Note, this characteristic can be in conflict with "minimize complexity". Engineering is about balancing conflicting objectives.
- **Reusability** – reuse is a hallmark of a mature engineering discipline
- **Portability** – works or can easily be made to work in other environments
- **High fan-in** on a few utility-type modules and low-to-medium fan-out on all modules. High fan-out is typically associated with high complexity.
- **Leanness** – when in doubt, leave it out. The cost of adding another line of code is much more than the few minutes it takes to type.
- **Stratification** – Layered. Even if the whole system doesn't follow the layered architecture style, individual components can.
- **Standard techniques** – sometimes it's good to be a conformist! Boring is good. Production code is not the place to try out experimental techniques.

Software Design methods

Requirements



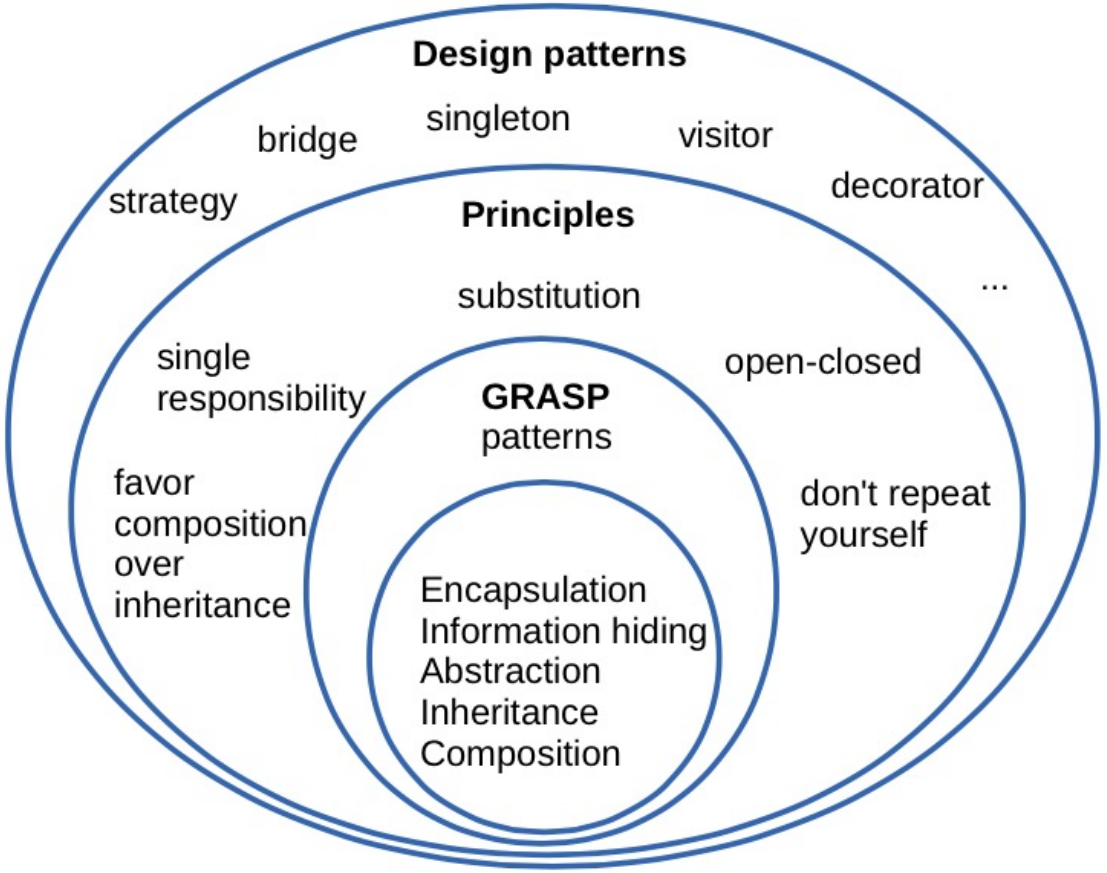
Software Design

Patterns

- A design pattern is a reusable solution to a commonly occurring design problem
- Design patterns are adapted for the unique characteristics of the problem
- Just as there are levels of design, there are levels of design patterns:
  - Architecture Styles/Patterns
  - Design Patterns
  - Programming Idioms

What next? O-O Software Design

- There's no a methodology to get the best object-oriented design, but there are principles, patterns, heuristics.



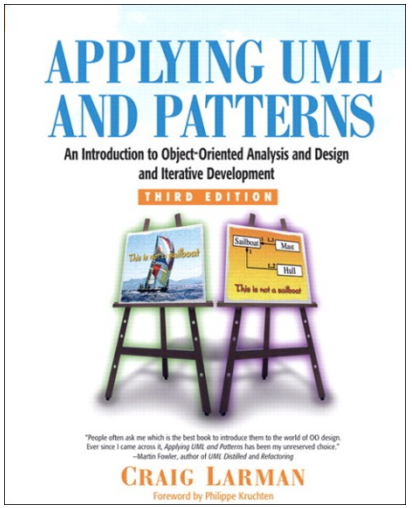
GRASP Principles

UA.DETI.PDS - 2022/23

José Luis Oliveira

Resources & Credits

- **Applying UML and Patterns**
  Craig Larman
  Chapters 16 & 22



- GRASP – General Responsibility Assignment Software Patterns Explained
  - http://www.kamilgrzybek.com/design/grasp-explained/

GRASP

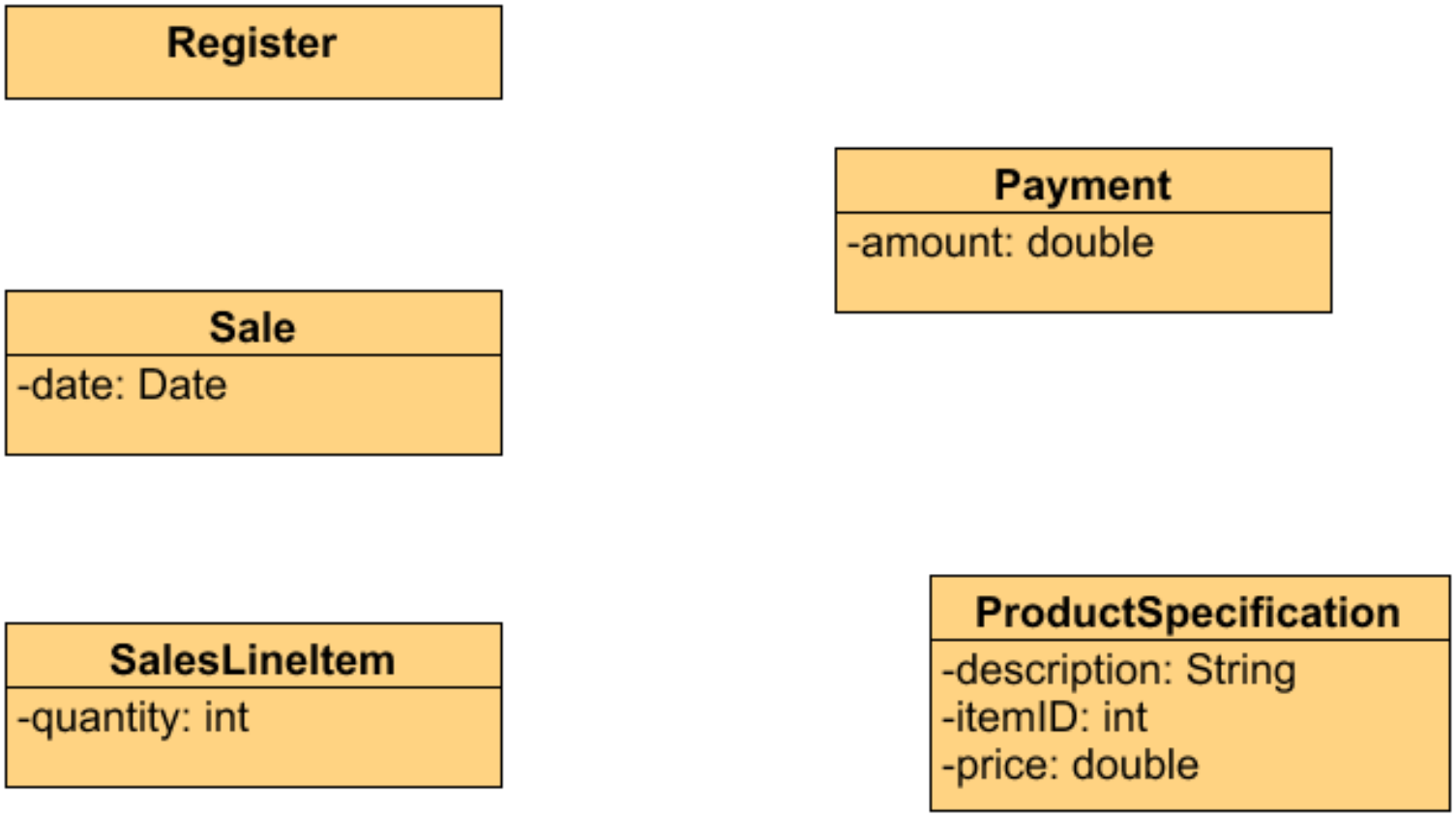
- General Responsibility Assignment Software Patterns
  - Name chosen to suggest the importance of grasping fundamental principles to successfully design object-oriented software
- Describe fundamental principles of object design and responsibility
- For instance …
  - You want to assign a responsibility to a class
  - You want to avoid or minimize additional dependencies
  - You want to maximise cohesion and minimise coupling
  - You want to increase reuse and decrease maintenance
  - You want to maximise understandability
  - …..etc.

Conducting example 1 - POS

- Point of Sale / Point of Sale Terminal:
  - Application for a shop, restaurant, etc. that registers sales.
  - Each sale is of one or more items of one or more product types, and happens at a certain date.

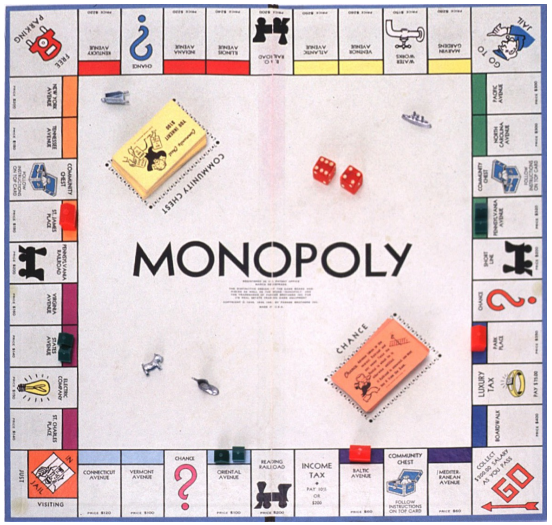
  - A product has a specification including a description, unitary price and identifier.
  - The application also registers payments (say, in cash) associated to sales.
  - A payment is for a certain amount, equal or greater that the total of the sale.

POS - A simple model



Conducting example 2: Monopoly

Die	MonopolyGame	Board
Player	Piece	Square



GRASP principles

- Creator
- Information Expert
- Low Coupling
- High Cohesion
- Controller
- Polymorphism
- Pure Fabrication
- Indirection
- Protected Variations

GRASP principles

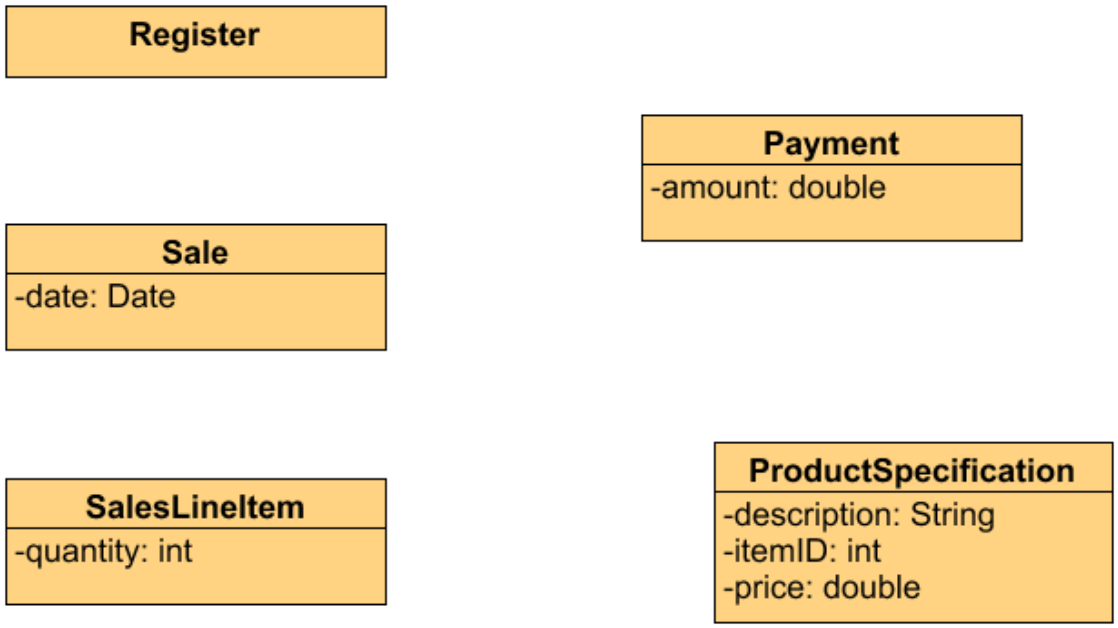
- Creator
- Information Expert
- Low Coupling
- High Cohesion
- Controller
- Polymorphism
- Pure Fabrication
- Indirection
- Protected Variations

Creator

- Name: Creator
- Problem: Who creates an instance of A?
- Solution: Assign class B the responsibility to create an instance of class A if one of these is true (the more the better):
  - B contains or aggregates A (in a collection)
  - B records A
  - B closely uses A
  - B has the initializing data for A

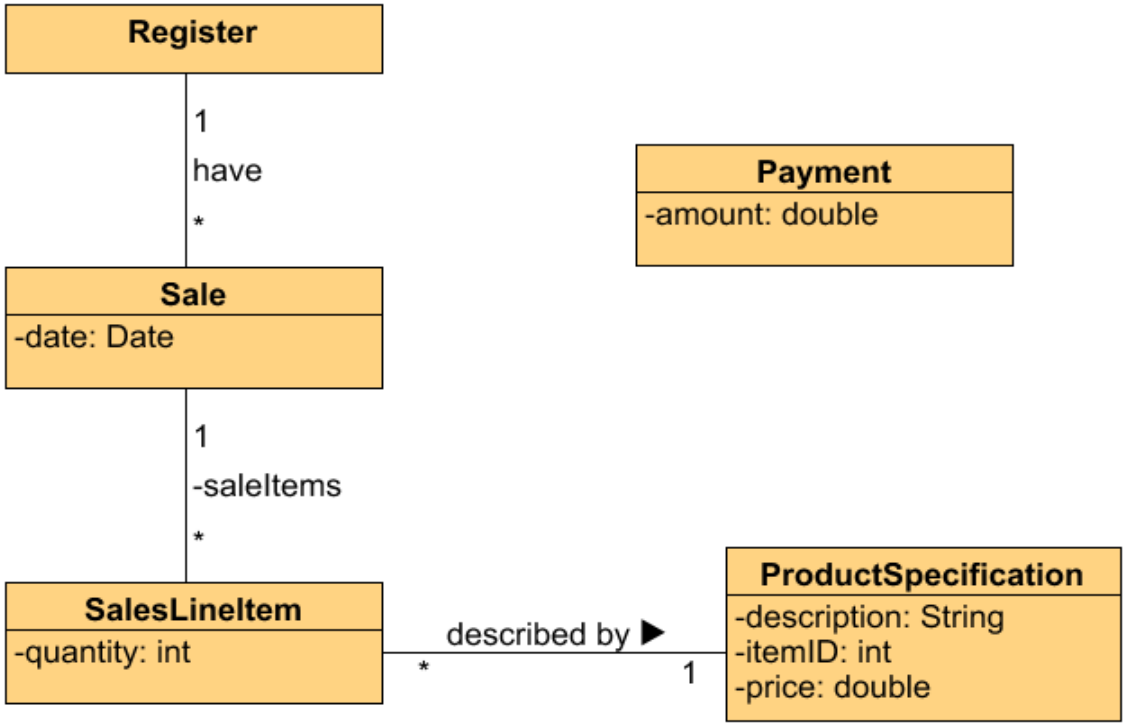
Creator: example

- Who is responsible for creating SalesLineItem objects, from an itemID and a quantity?



Creator: example

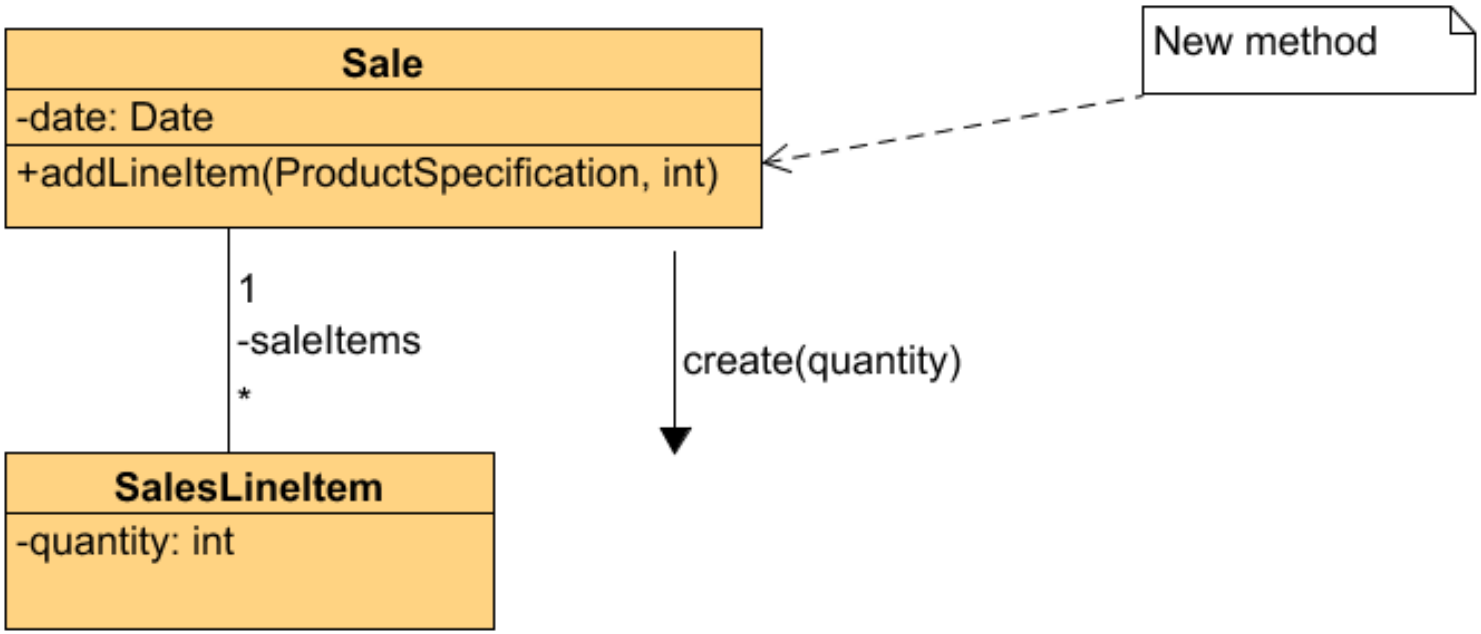
- Who is responsible for creating SalesLineItem objects, from an itemID and a quantity?
- Look for a class that aggregates or contains SalesLineItem objects.



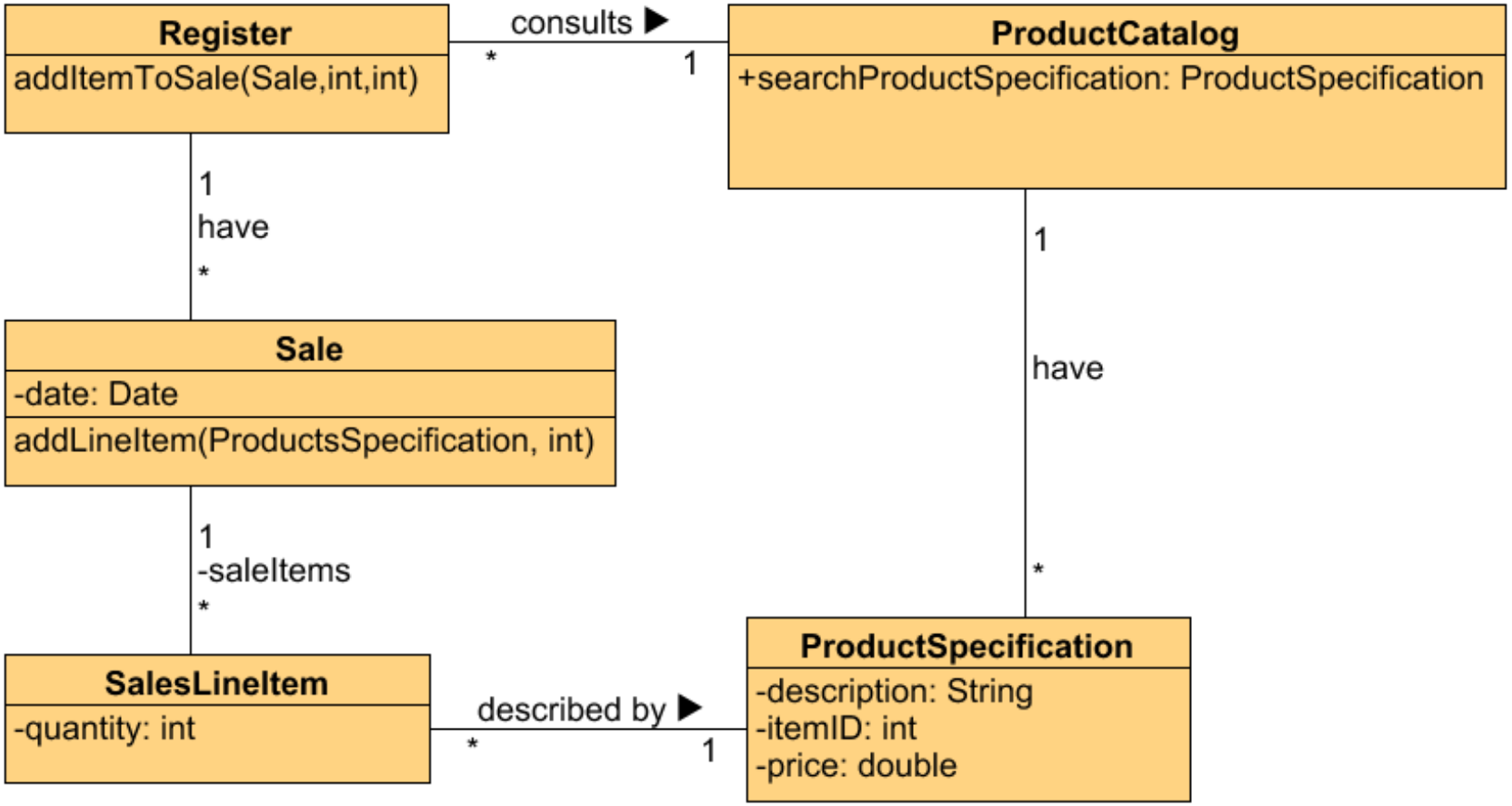
Creator: example

- Creator pattern suggests Sale.

- Collaboration diagram is

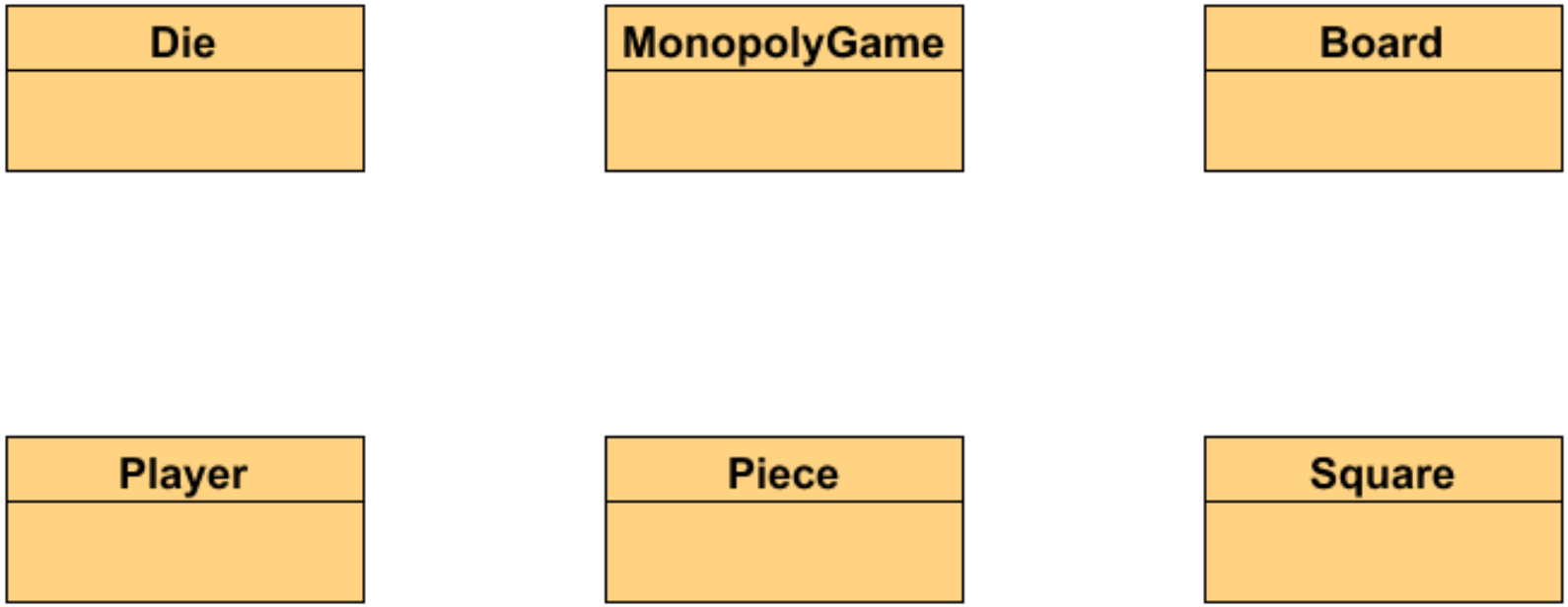


Creator: example



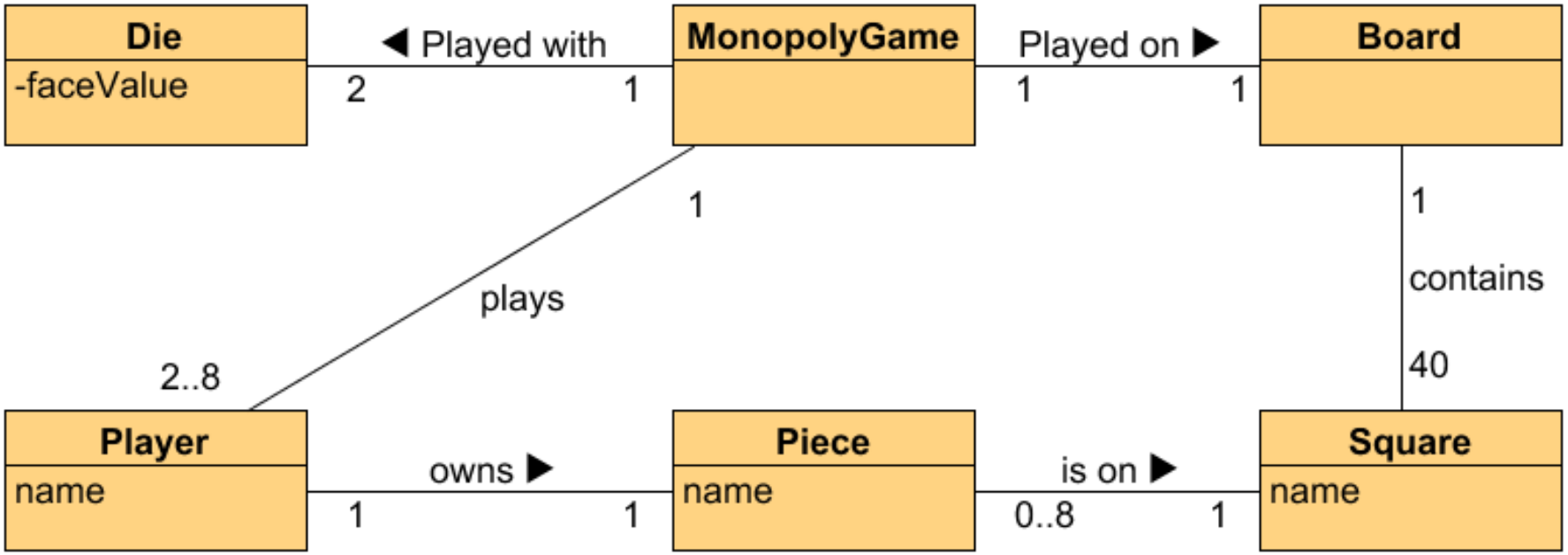
Creator: another example

- Who creates what?



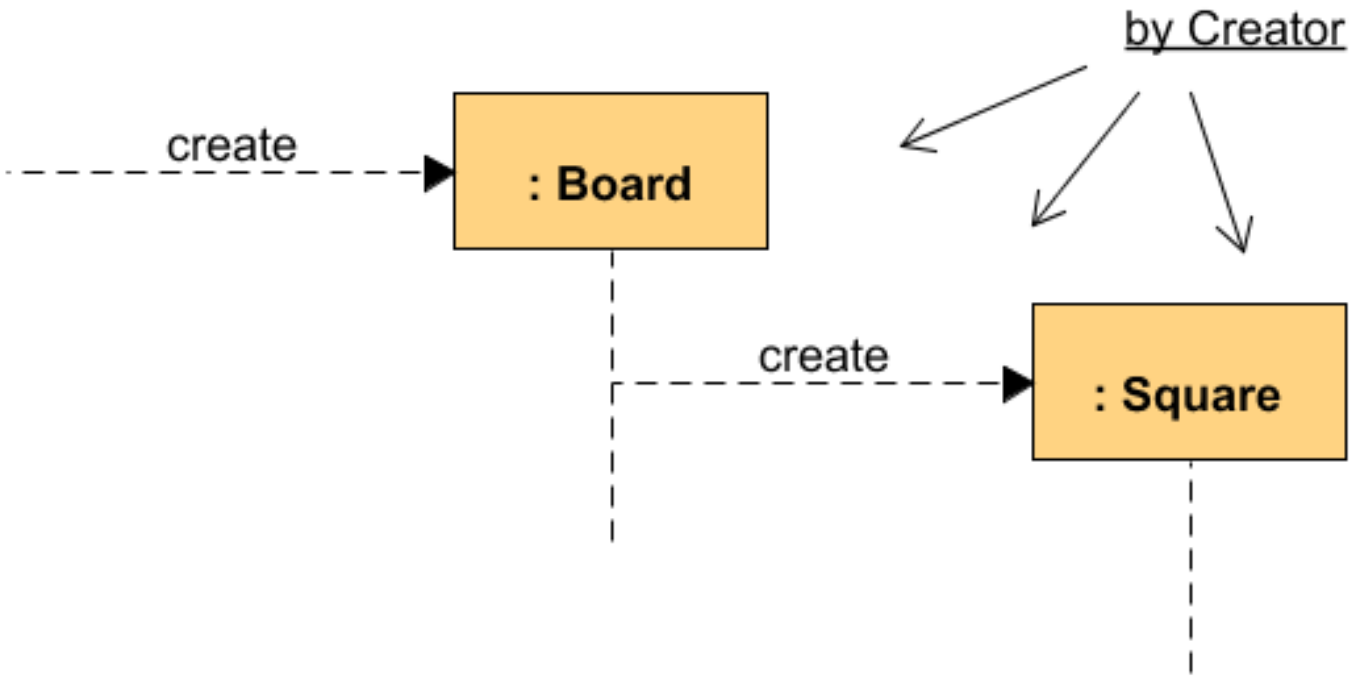
Creator: another example

- Who creates the Squares?



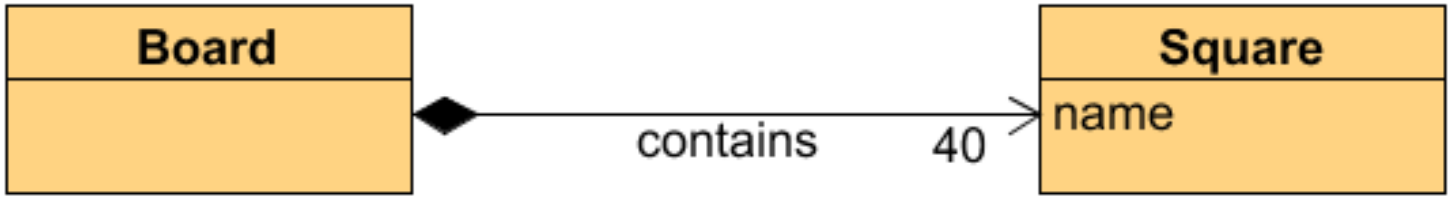
Creator pattern

- How does Create pattern lead to this partial Sequence diagram?



Creator pattern

- How does Create pattern develop this design class diagram?



- Board has a composite aggregation relationship with Square
  - I.e., Board contains a collection of Squares

Discussion of Creator pattern

- Promotes low coupling by making instances of a class responsible for creating objects they need to reference

- Connect an object to its creator when:
  - Aggregator aggregates Part
  - Container contains Content
  - Recorder records
  - Initializing data passed in during creation

Contraindications or caveats

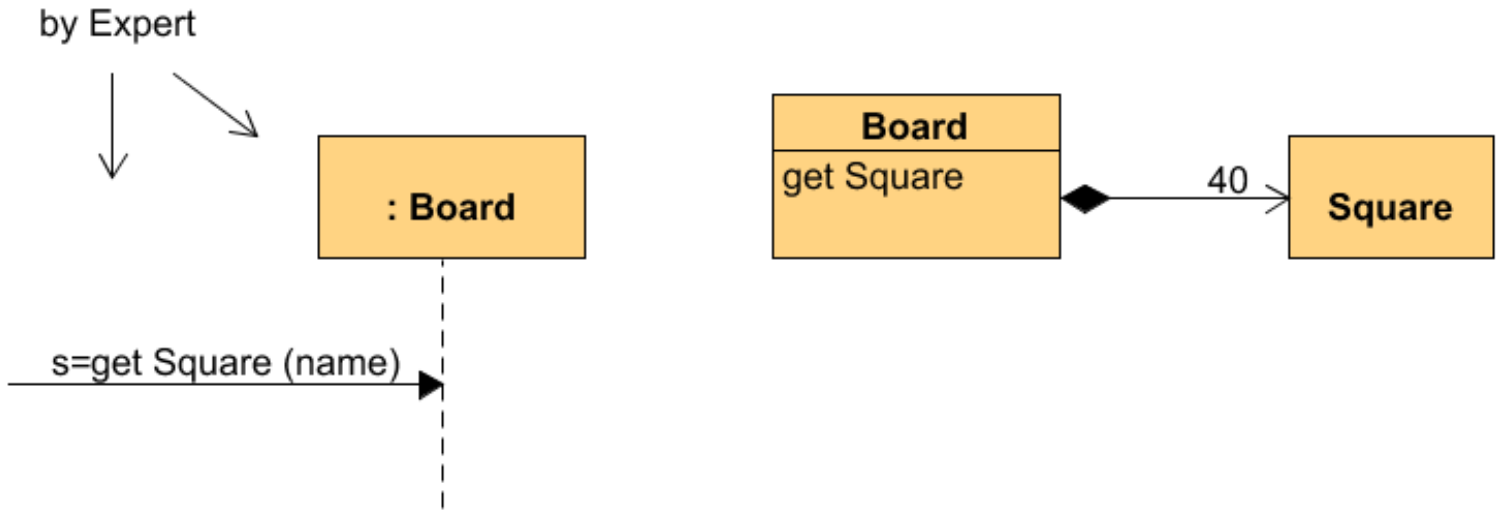
- Creation may require significant complexity:
  - recycling instances for performance reasons
  - conditionally creating instances from a family of similar classes
- In these instances, other patterns are available…
  - We'll learn about Factory and other patterns later…

GRASP

- Creator
- Information Expert
- Low Coupling
- High Cohesion
- Controller
- Polymorphism
- Pure Fabrication
- Indirection
- Protected Variations

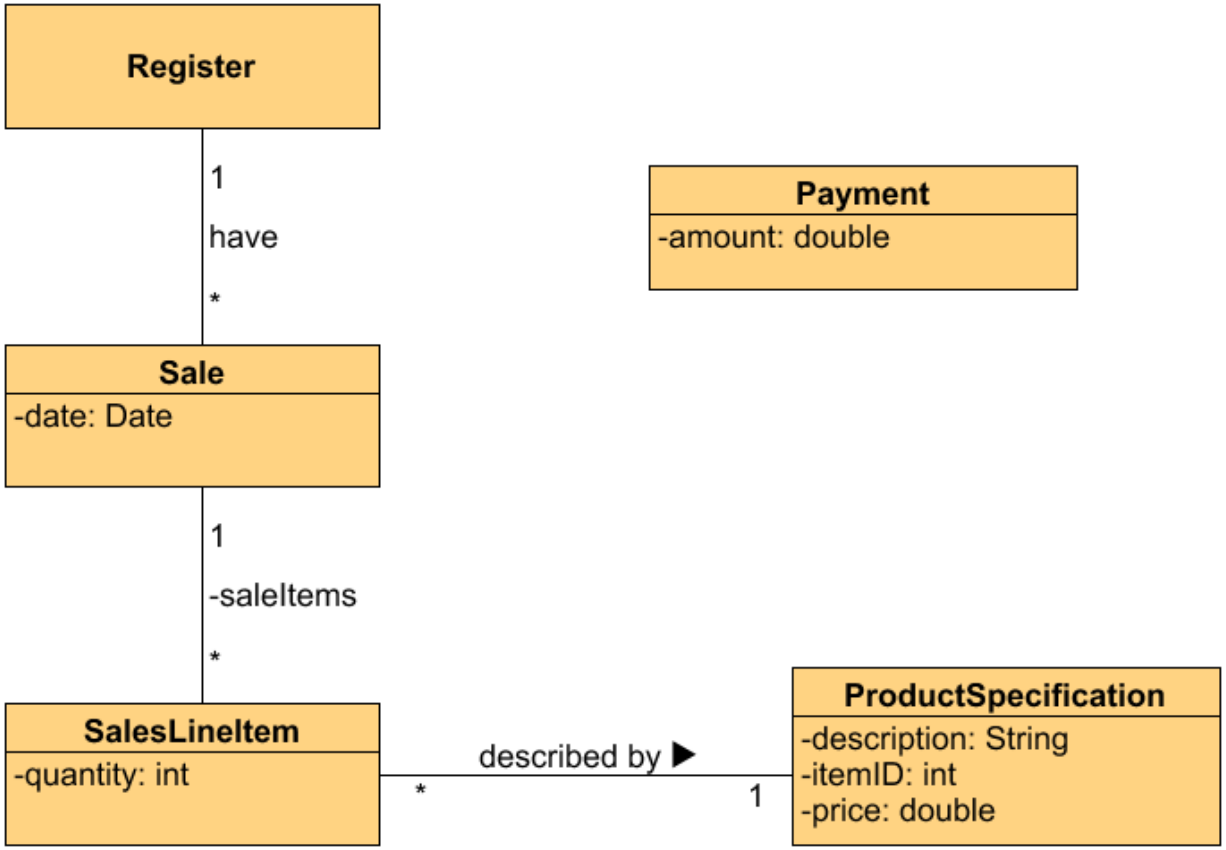
Information Expert principle

- Name: Information Expert
- Problem: How to assign responsibilities to objects?
- Solution: Assign responsibility to the class that has the information needed to fulfill it?
- E.g., *Board* information needed to get a *Square*



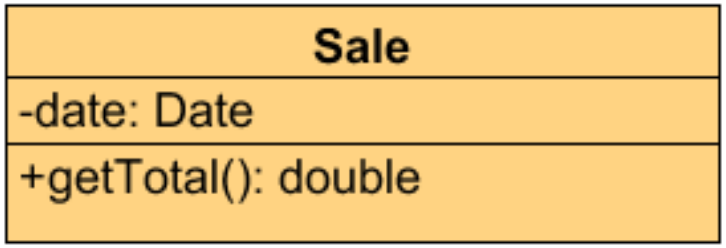
Information Expert: another example

- Who is responsible for knowing the grand total of a sale in a typical Point of Sale application?

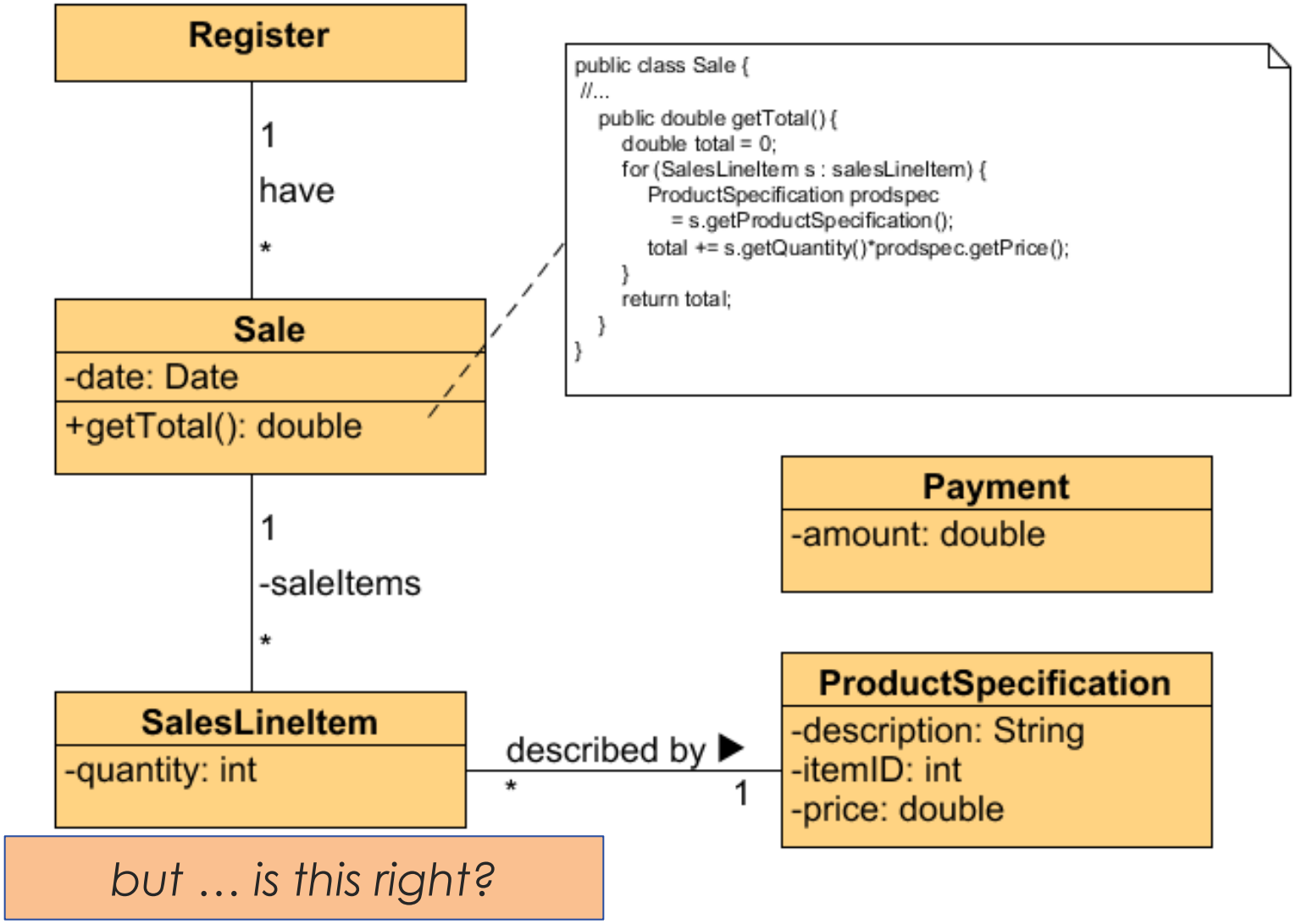


Information Expert: example

- Need all *SalesLineItem* instances and their subtotals. Only *Sale* knows this, so *Sale* is the information expert.
- Hence

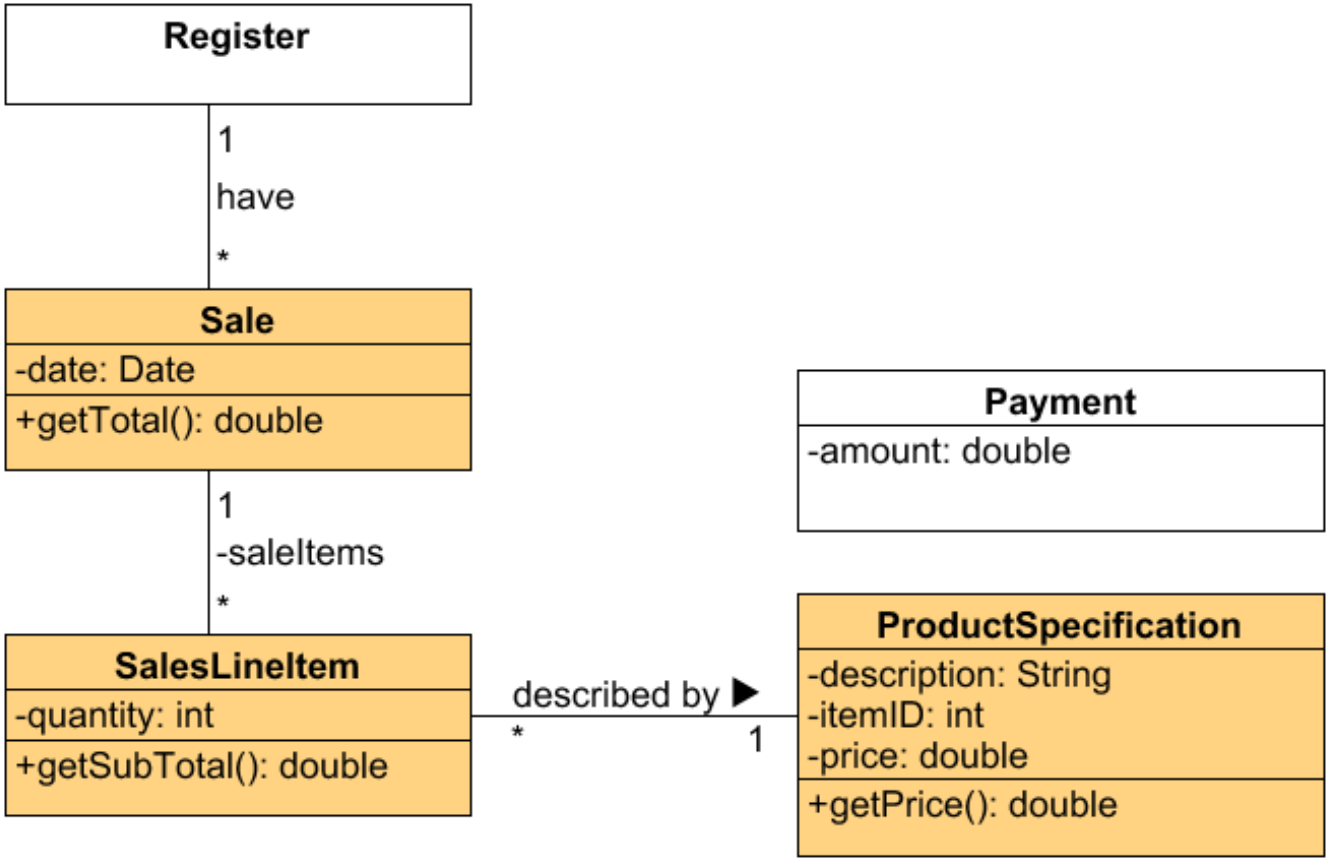


Information Expert: example



Information Expert: example

- But subtotals are needed for each line item.
  - By Expert, *SalesLineItem* is expert, knows quantity and has association with *ProductSpecification* which knows price.



Information Expert: example

```
class Register {
   List<Sale> sales = new ArrayList<>();
   //...
   public void addItemToSale(Sale sale, int itemID, int quantity) {
        ProductSpecification prodSpec =
                ProductCatalog.searchproductSpecification(itemID);
     sale.addLineItem(prodSpec, quantity);
   }
}

class Sale {
   List<SalesLineItem> salesLineItem = new ArrayList<>();
   //...
   public void addLineItem(ProductSpecification prodSpec, int quantity) {
        salesLineItem.add(new SalesLineItem(prodSpec, quantity);
   }
}
```

Information Expert: example

- Hence responsibilities assign to the 3 classes.

Class	Responsibility
Sale	knows sale total
SalesLineItem	knows line item subtotal
ProductSpecification	knows product price

- Fulfillment of a responsibility may require information spread across different classes, each expert on its own data.
  - Real world analogy: workers in a business, bureaucracy, military. "Don't do anything you can push off to someone else".

Benefits and Contraindications

- Facilitates information encapsulation
  - Classes use their own info to fulfill tasks - highly cohesive classes
  - Code easier to understand just by reading it
- Promotes low coupling
  - *Sale* doesn't depend on *ProductSpecification*

But:

- Can cause a class to become excessively complex
  - e.g. who is responsible to save *Sale* in a database? Sale is the information expert, but with this decision, then each class has its own services to save itself in a database.
  - This needs another kind of separation – domain and persistence

GRASP

- Creator
- Information Expert
- Low Coupling
- High Cohesion
- Controller
- Polymorphism
- Pure Fabrication
- Indirection
- Protected Variations

Low Coupling pattern

- Name: Low Coupling
- Problem: How to reduce the impact of change and encourage reuse?
- Solution: Assign a responsibility so that coupling (linking classes) remains low. Try to avoid one class to have to know about many others.
  - changes are localised
  - easier to understand
  - easier to reuse

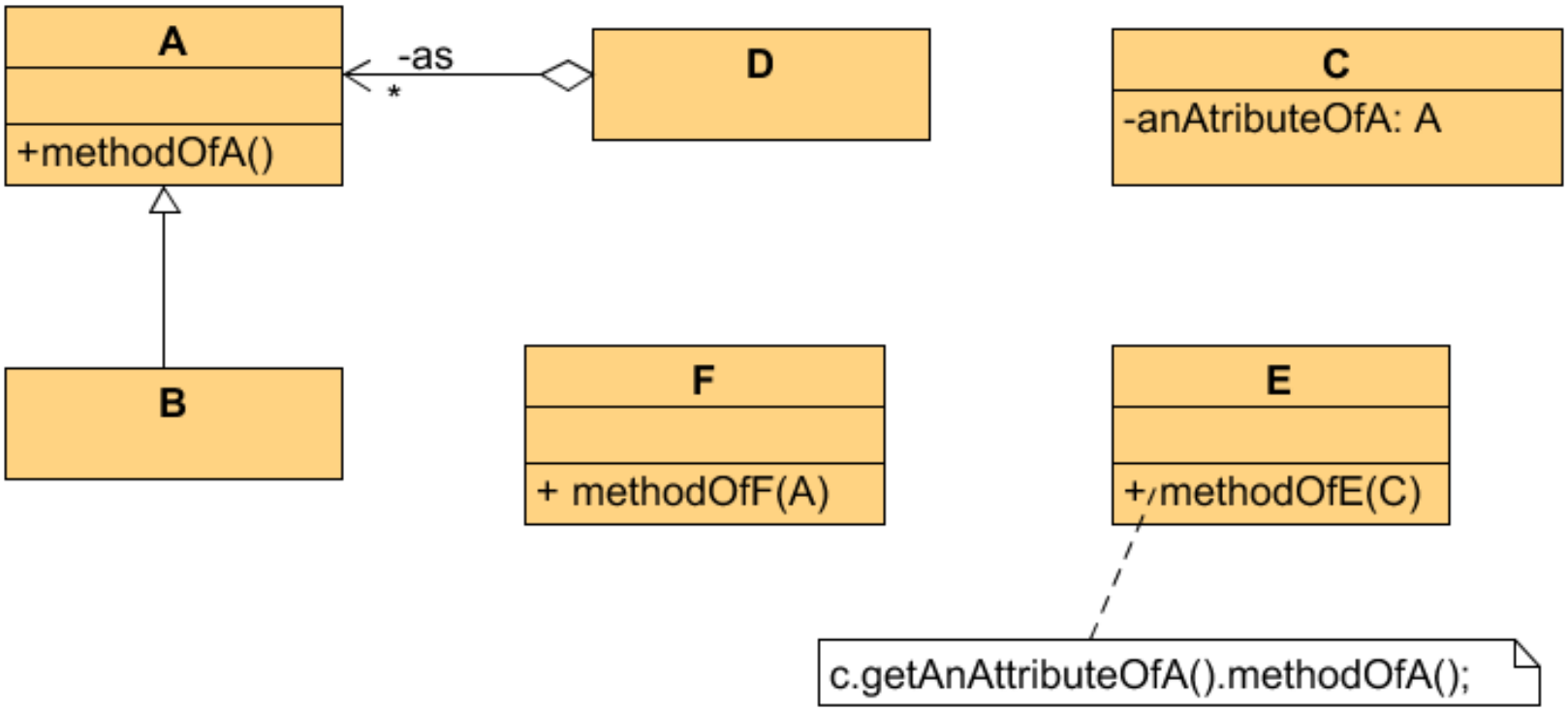
Low Coupling pattern

- Coupling measures of how strongly a class is connected, depends, relies on, or has knowledge of objects of other classes.
- 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.

Entities coupling

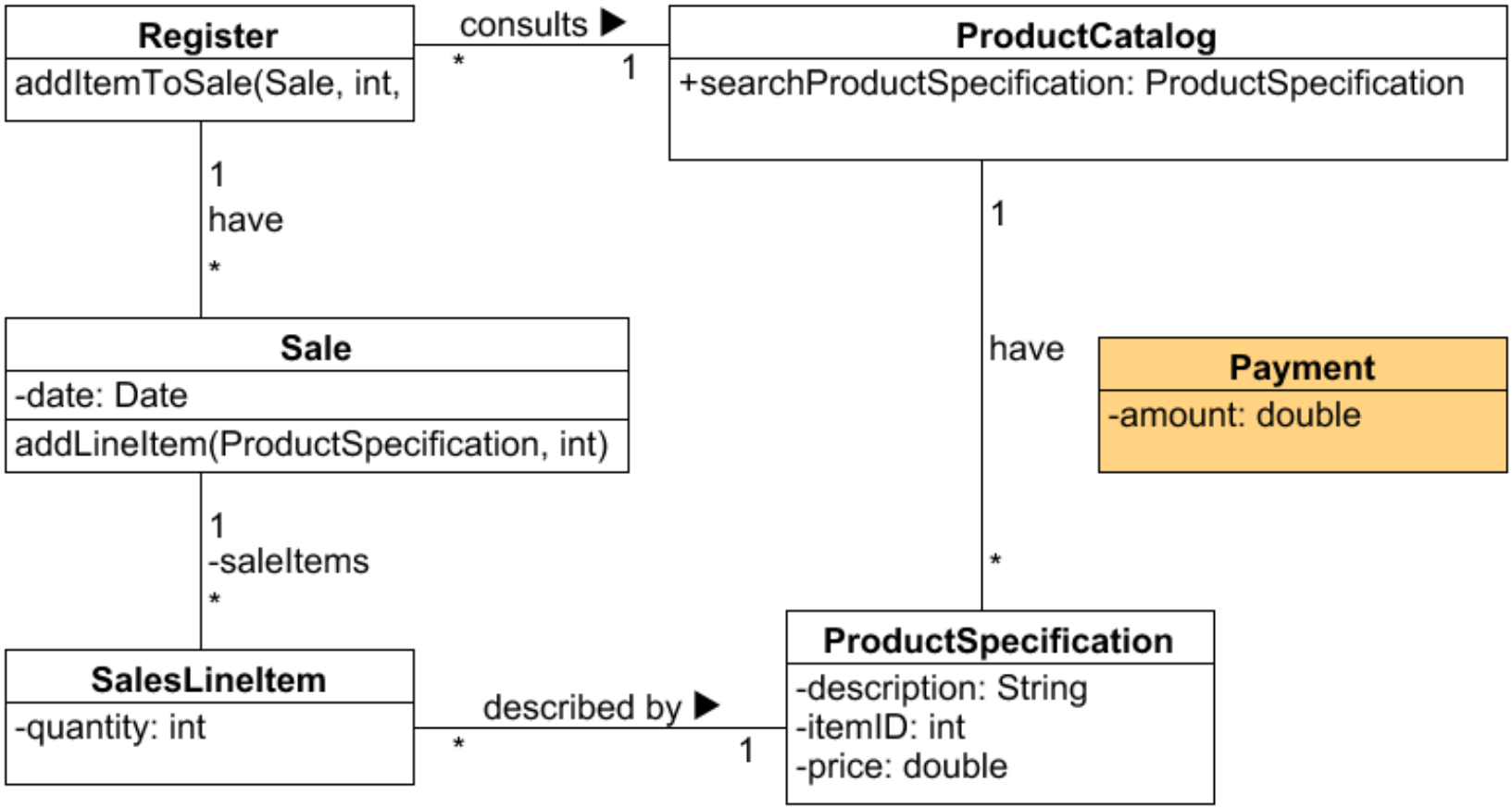
- In object-oriented languages, common forms of coupling from TypeX to TypeY include:
  - TypeX has an attribute (data member or instance variable) that refers to a TypeY instance, or TypeY itself.
  - TypeX has a method which references an instance of TypeY, or TypeY itself, by any means. These typically include a parameter or local variable of type TypeY, or the object returned from a message being an instance of TypeY.
  - TypeX is a direct or indirect subclass of TypeY.
  - TypeY is an interface, and TypeX implements that interface.

Entities coupling



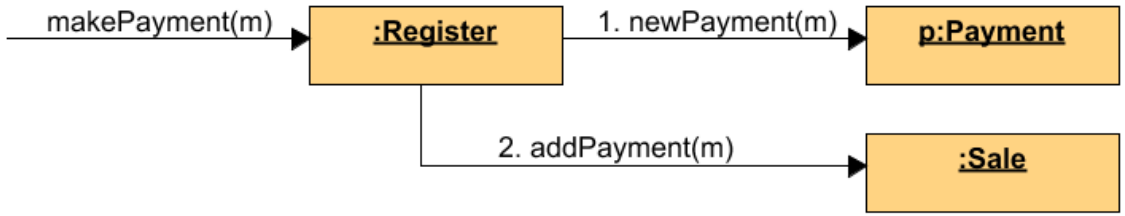
Low Coupling: example

- Who has responsibility to create a payment?

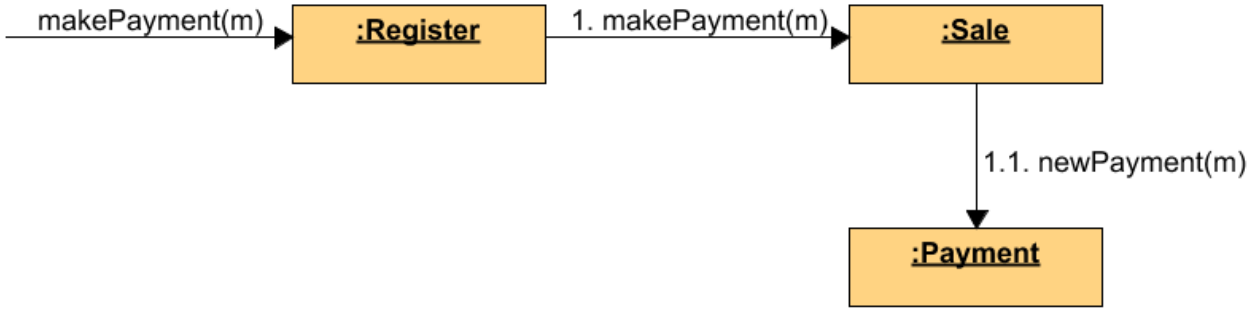


Low Coupling: example

- Two possibilities:
  - Register



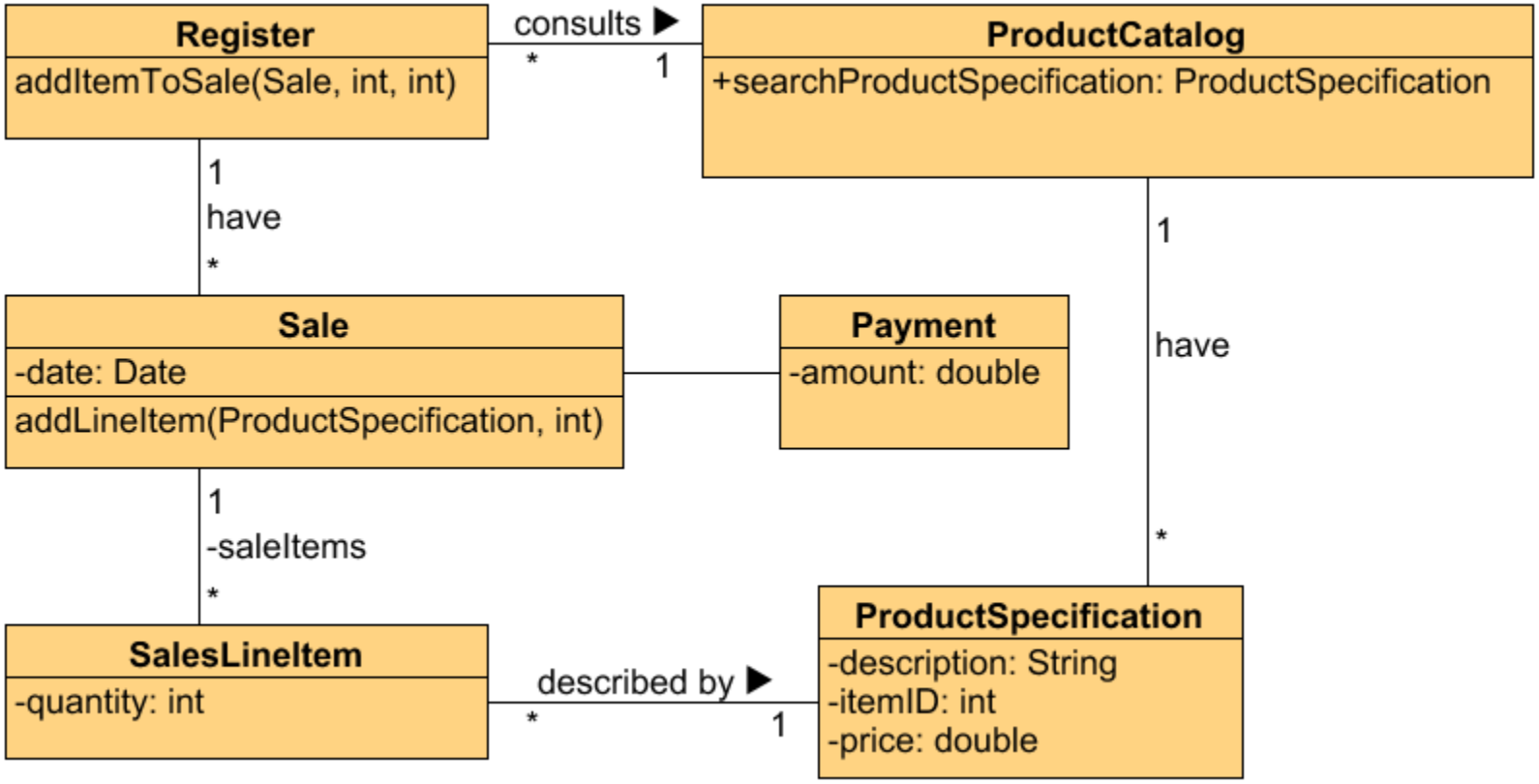
  - Sale



  - Low coupling suggests *Sale* because *Sale* must be coupled to *Payment* anyway (*Sale* knows its *total*).

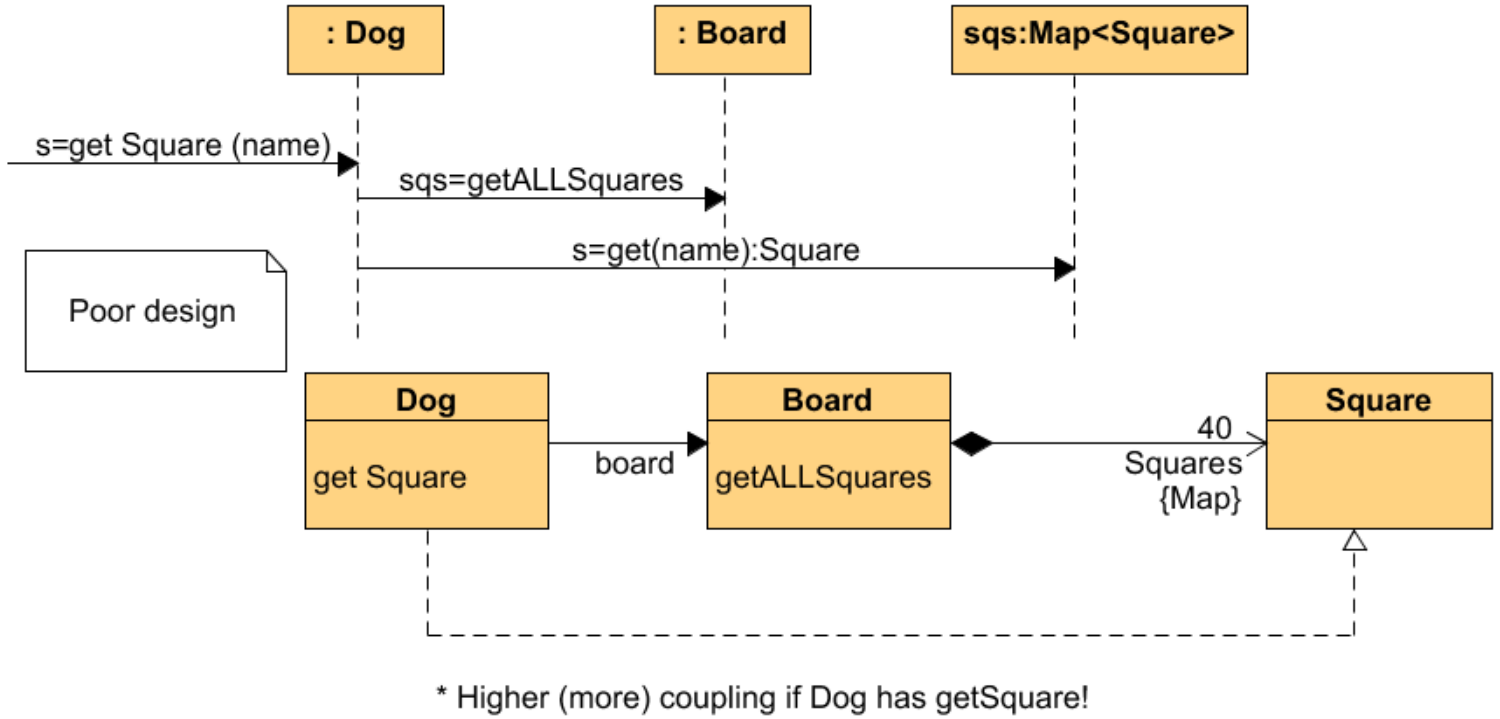
Low Coupling: example

- Who should own the method *getBalance()* that computes payment amount - total of sale?



Low Coupling: monopoly

- Why does the following design violate Low Coupling?



* Higher (more) coupling if Dog has getSquare!

  - Why is a better idea to leave getSquare responsibility in Board?

Benefits & Contraindications

- Understandability: Classes are easier to understand in isolation
- Maintainability: Classes aren't affected by changes in other components
- Reusability: easier to grab hold of classes

But:

- A higher coupling to stable classes is not a big issue
  - e.g., libraries and well-tested classes

GRASP

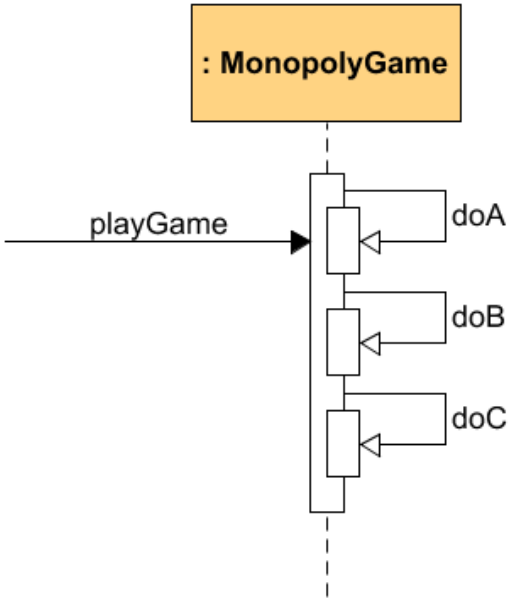
- Creator
- Information Expert
- Low Coupling
- High Cohesion
- Controller
- Polymorphism
- Pure Fabrication
- Indirection
- Protected Variations

High Cohesion pattern

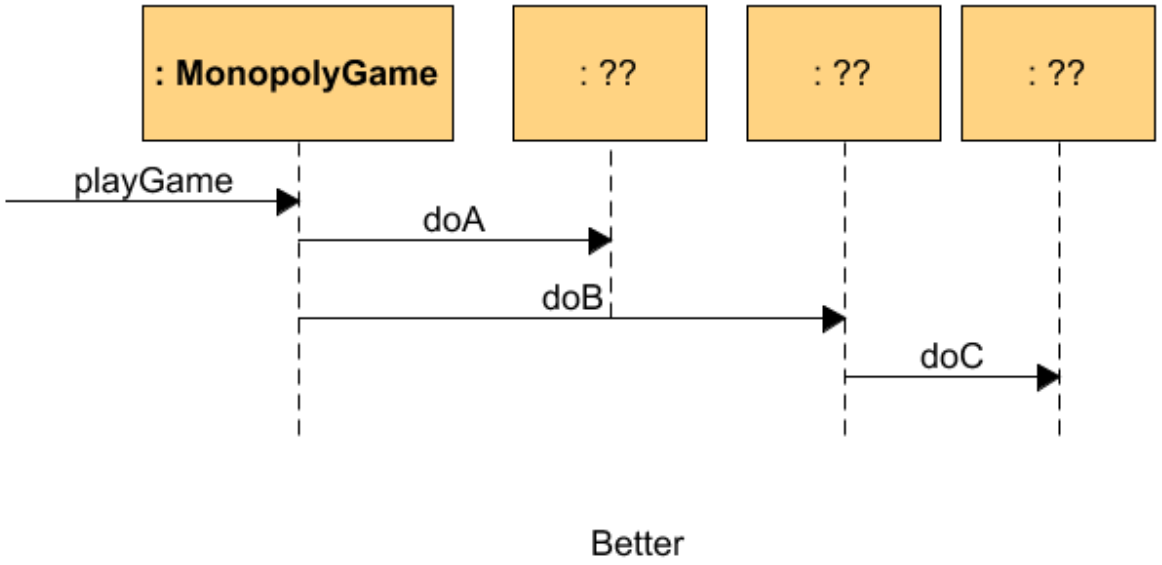
- Cohesion measures how strongly related and focused are the responsibilities of an element
- Name: High Cohesion
- Problem: How to keep classes focused and manageable?
- Solution: Assign responsibility so that cohesion remains high.

High Cohesion

- How does the design on right promote high cohesion?



Poor (Low) Cohesion in the Monopoly Game object

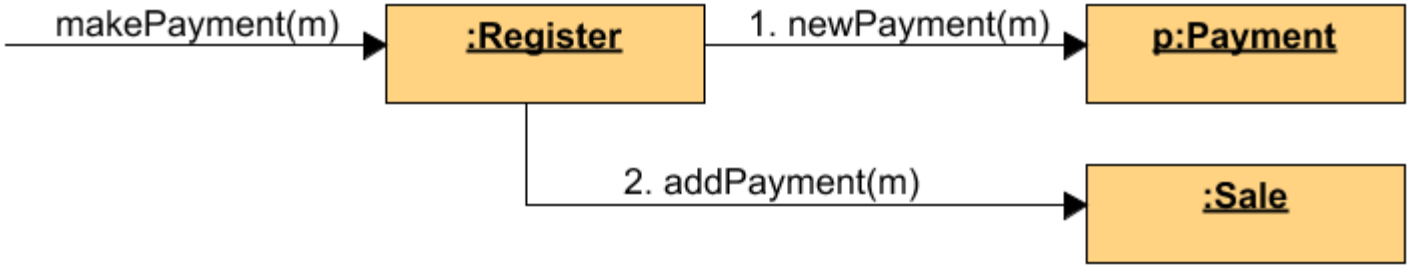


Better

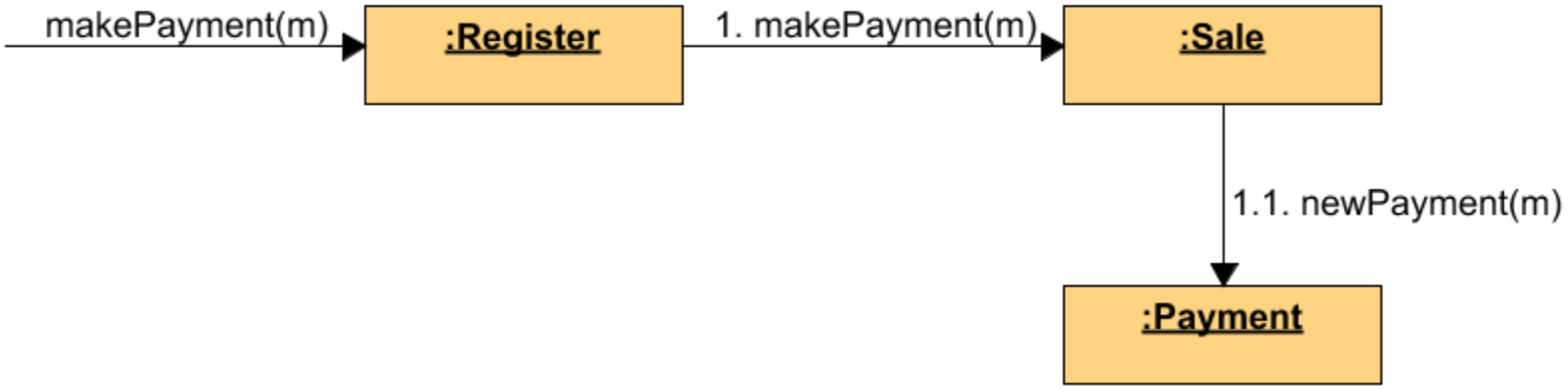
- Delegate responsibility & coordinate work

High Cohesion

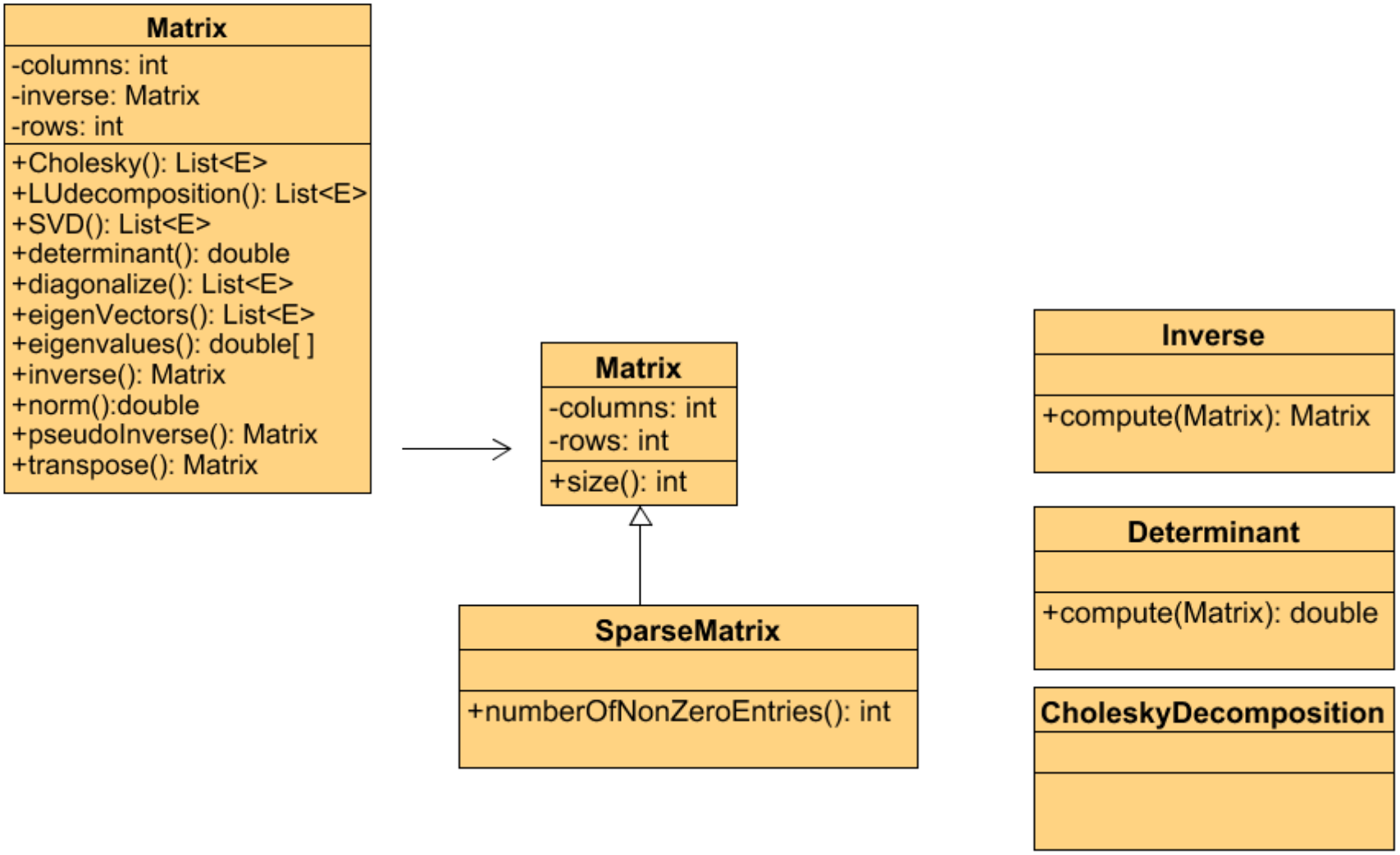
- *Register* would take on more and more responsibilities and become less cohesive.



- Giving responsibility to *Sale* supports higher cohesion in *Register*, as well as low coupling.

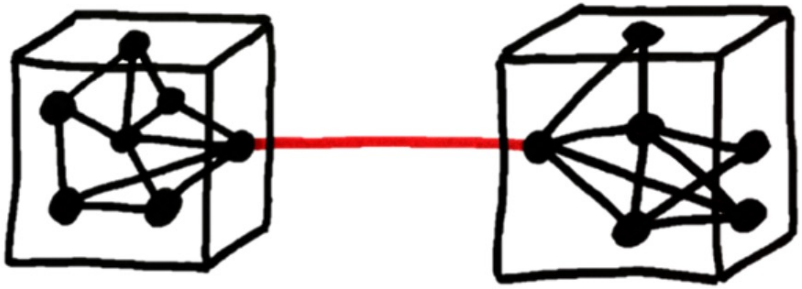


High Cohesion



Benefits & Contraindications

- Understandability, maintainability
- Complements Low Coupling



But:

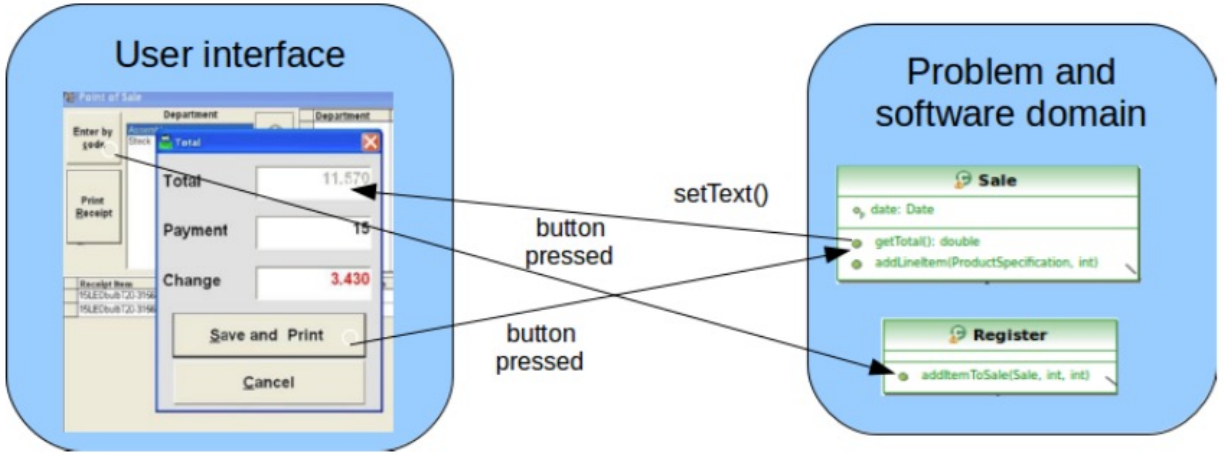
- Sometimes desirable to create less cohesive server objects
  - that provide an interface for many operations, due to performance needs associated with remote objects and remote communication

GRASP

- Creator
- Information Expert
- Low Coupling
- High Cohesion
- Controller
- Polymorphism
- Pure Fabrication
- Indirection
- Protected Variations

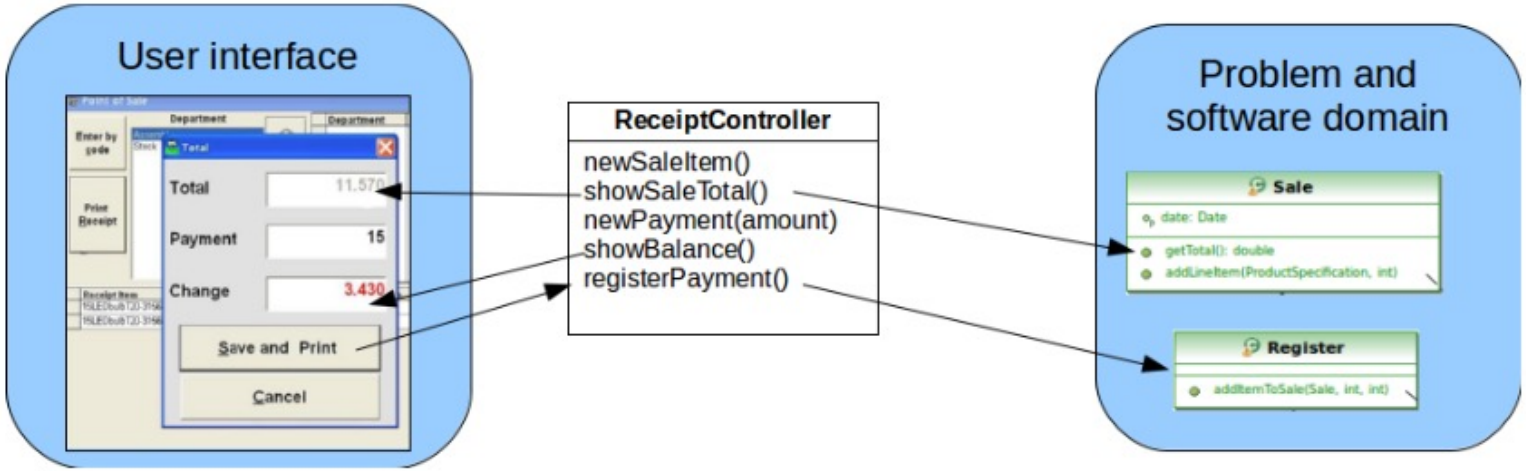
Controller

- Name: Controller
  - *(more on Model-View-Controller architecture)*
- Problem: Who should be responsible for UI events?



Controller

- Solution:
  - If a program receive events from external sources other than its GUI, add an event class to decouple the event source(s) from the objects that actually handle the events.

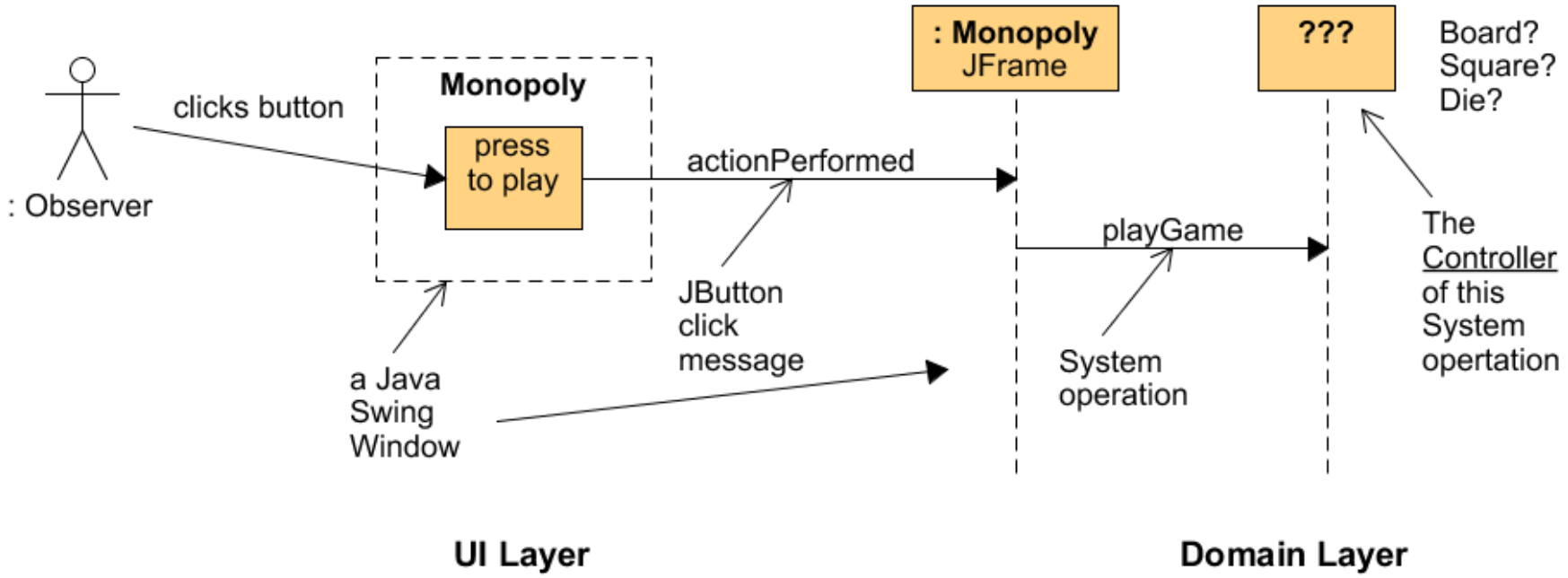


Controller

- Assign the responsibility for handling a system event message to a class representing one of these choices:
  1. The business or overall "system"(a façade controller).
  2. An artificial class, Pure Fabrication representing the use case (a use case controller).

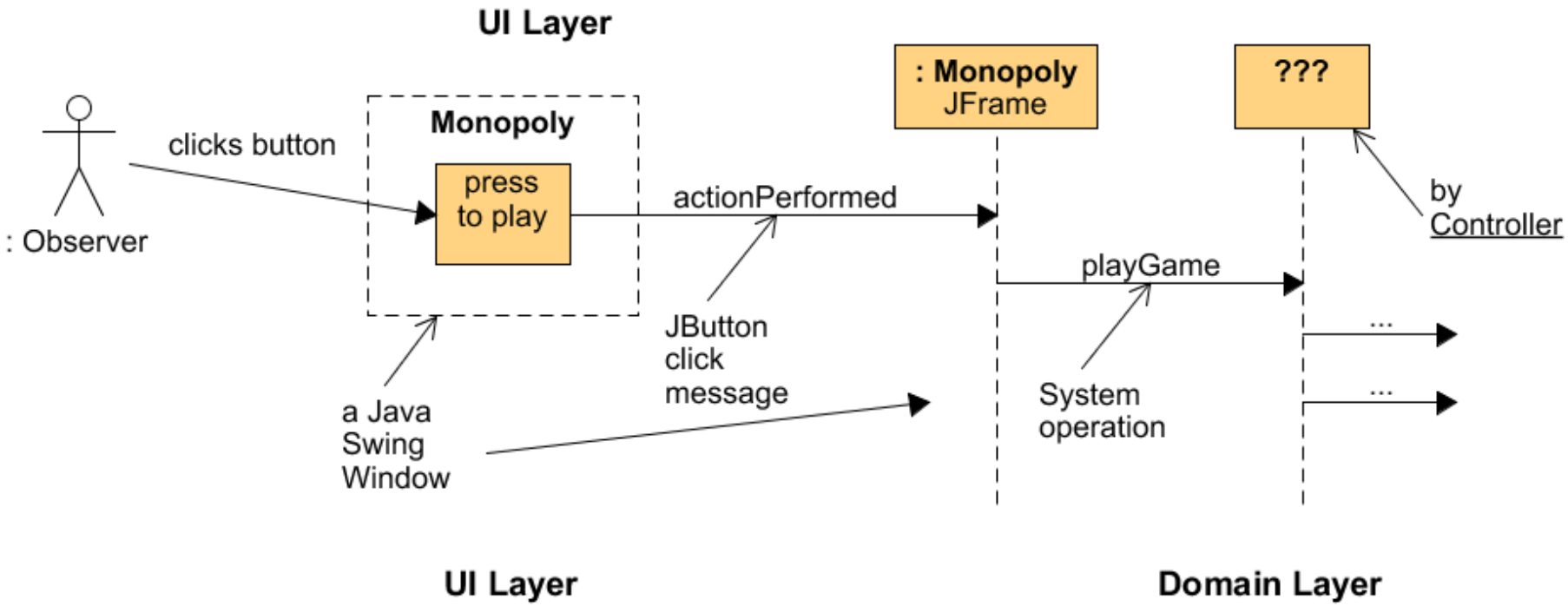
Controller

- Who is the controller of playGame operation?



Controller

- Separation between logical and UI views



Benefits & Contraindications

- Increased potential for reuse
  - Using a controller object keeps external event sources and internal event handlers independent of each other's type and behaviour.
  - either the UI classes or the problem/software domain classes can change without affecting the other side.
- Controller just forwards
  - event handling requests
  - output requests
- Reason about the states of the use case
  - Ensure that the system operations occurs in legal sequence, or to be able to reason about the current state of activity and operations within the use case.

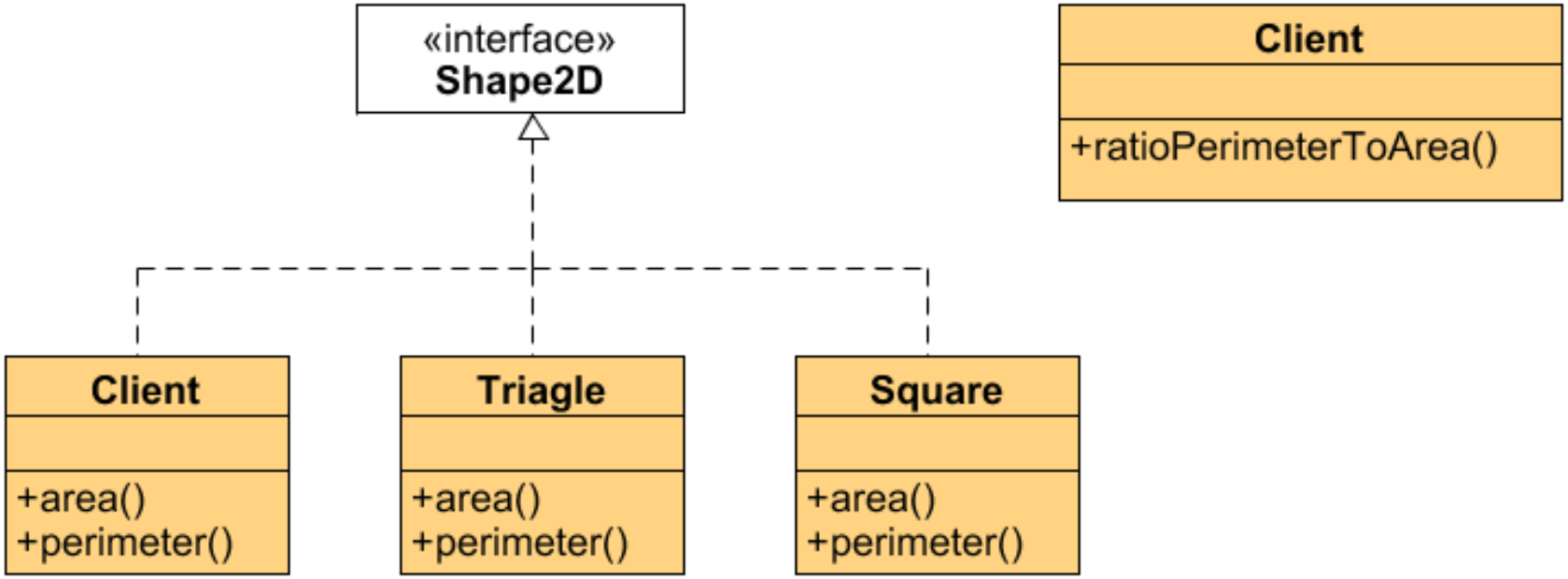
GRASP

- Creator
- Information Expert
- Low Coupling
- High Cohesion
- Controller
- Polymorphism
- Pure Fabrication
- Indirection
- Protected Variations

Polymorphism

- Problem:
  - How to handle behavior based on type (i.e., class) but not with an if-then-else or switch statement involving the class name or a tag attribute?
- Solution:
  - When alternate behaviours are selected based on the type of an object, use polymorphic method call to select the behaviour, rather than using if statement to test the type.
  - Polymorphic methods: giving the same name to (different) services in different classes. Services are implemented by methods.

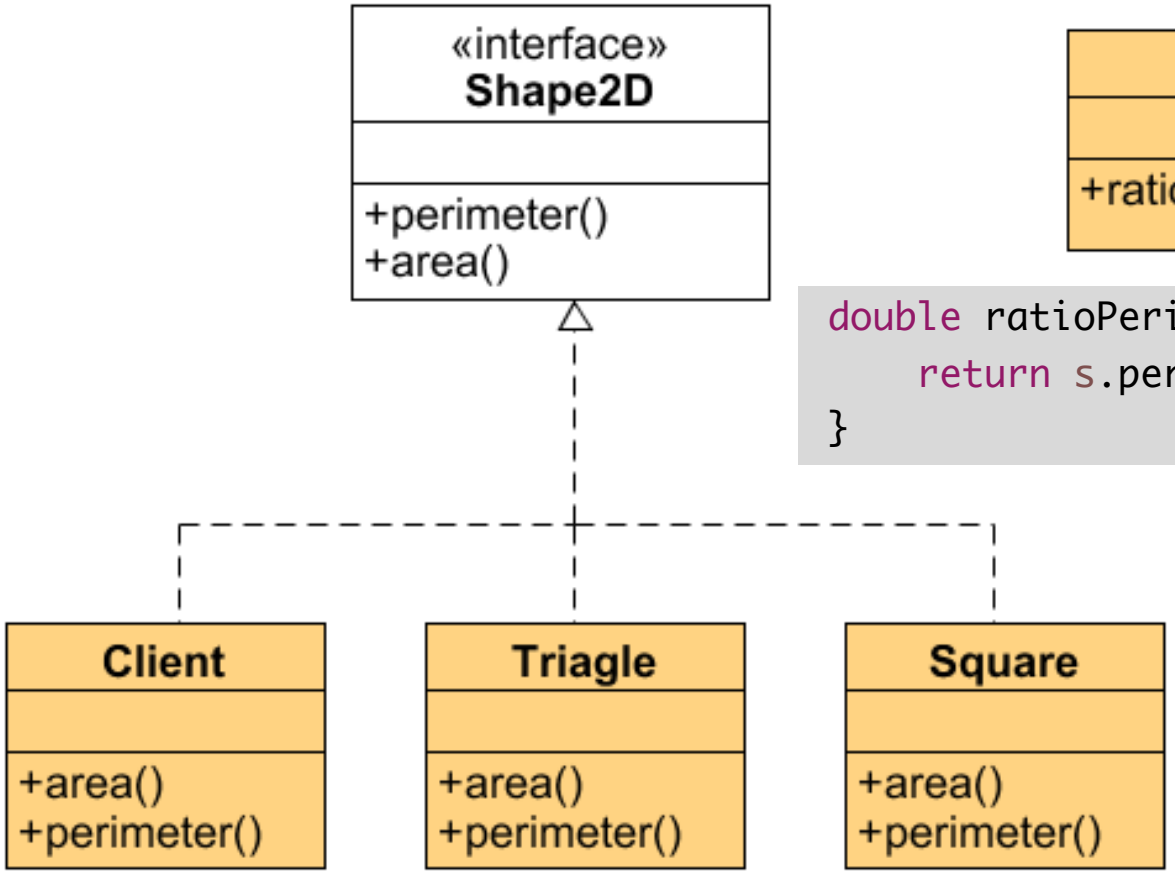
Example


«interface»
Shape2D
Client
+ratioPerimeterToArea()
Client
+area()
+perimeter()
Triagle
+area()
+perimeter()
Square
+area()
+perimeter()

Example

```
double ratioPerimeterToArea(Shape2D s) {
   double ratio = 0.0;
   if (s instanceof Triangle) {
     // or String name = s.getClass().getName();
     // if (name=="Triangle") {
        Triangle t = (Triangle) s;
        ratio = t.perimeter()/t.area();
    } else if (s instanceof Circle) {
        Circle c = (Circle) s;
        ratio = c.perimeter()/c.area();
    } else if (s instanceof Square) {
        Square sq = (Square) s;
        ratio = sq.perimeter()/sq.area();
    }
    return ratio;
}
```

Example - Polymorphism



Client

+ratioPerimeterToArea()

```
double ratioPerimeterToArea(Shape2D s) {
    return s.perimeter()/s.area();
}
```

Benefits & Contraindications

- Easier and more reliable than using explicit selection logic.
- Easier to add additional behaviours later on.

- If polymorphism is not used, and instead the code tests the type of the object, then that section of code will grow as more types are added to the system.
  - This section of code becomes more coupled (i.e., it knows about more types) and less cohesive (i.e., it is doing too much).

GRASP

- Creator
- Information Expert
- Low Coupling
- High Cohesion
- Controller
- Polymorphism
- Pure Fabrication
- Indirection
- Protected Variations

Pure Fabrication

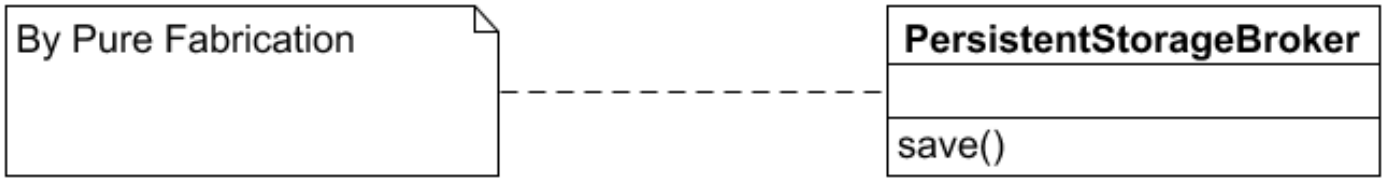
- Problem:
  - What object should have a responsibility when no class of the problem domain may take it without violating High Cohesion and Low Coupling?
  - Not all responsibilities fit into domain classes, like persistence, network communications, user interaction, etc.

- Solution:
  - Assign a highly cohesive set of responsibilities to an artificial class that does not represent anything in the problem domain.

Example

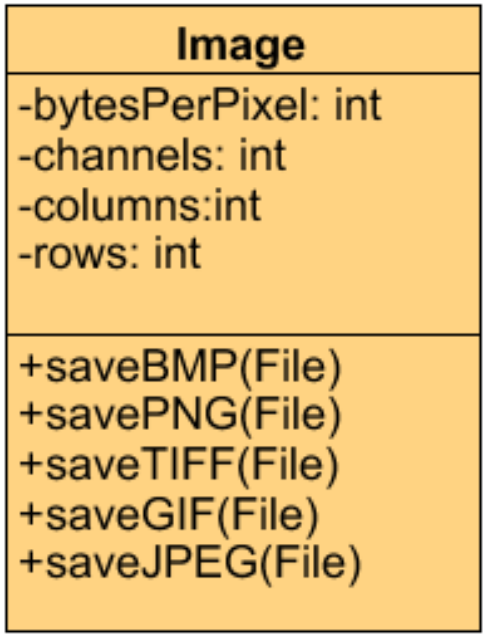
- Suppose, in the point of sale example, that we need to save Sale instances in a relational database.
  - By Expert, there is some justification to assign this responsibility to Sale class.
- However..
  - The task requires a relatively large number of supporting database-oriented operations and the Sale class becomes not cohesive.
  - The sale class has to be coupled to the relational database increasing its coupling.
  - Saving objects in a relational database is a very general task for which many classes need support. Placing these responsibilities in the Sale class suggests there is going to be poor reuse.

Pure Fabrication: example

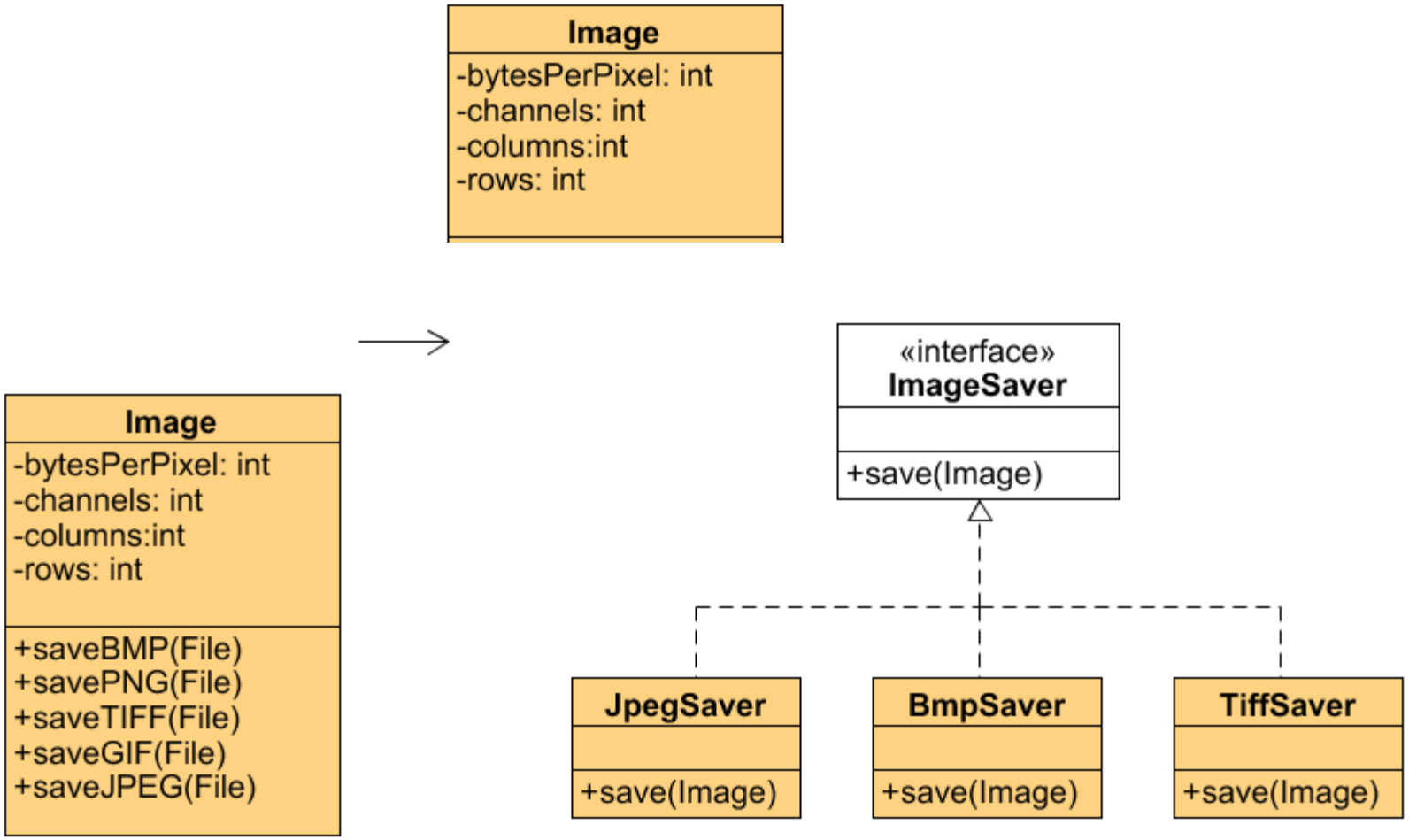


- The Sale remains well design, with high cohesion and low coupling
- The PersistentStorageBroker class is itself relatively cohesive
- The PersistentStorageBroker class is a very generic and reusable object

Pure Fabrication: Another example



Pure Fabrication: Another example



Benefits & Contraindications

- High cohesion is supported because responsibilities are factored into a class that only focuses on a very specific set of related tasks.
- Reuse potential may be increased because of the presence of fine grained Pure Fabrication classes.

GRASP

- Creator
- Information Expert
- Low Coupling
- High Cohesion
- Controller
- Polymorphism
- Pure Fabrication
- Indirection
- Protected Variations

Indirection

- Problem:
  - How to avoid direct coupling?
  - How to de-couple objects so that Low coupling is supported, and reuse potential remains high?
- Solution:
  - Assign the responsibility to an intermediate object to mediate between other components or services, so that they are not directly coupled.

Indirection

- Indirection can be categorized into the following main groups:
  - Behavioural Extension.
  - Interface Modification.
  - Technology Encapsulation.
  - Complexity Encapsulation.

Example: PersistentStorageBroker

- The Pure fabrication example
  - de-coupling the Sale from the relational database services through the introduction of a PersistentStorageBroker is also an example of assigning responsibilities to support Indirection.
  - The PersistentStorageBroker acts as a intermediary between the Sale and database

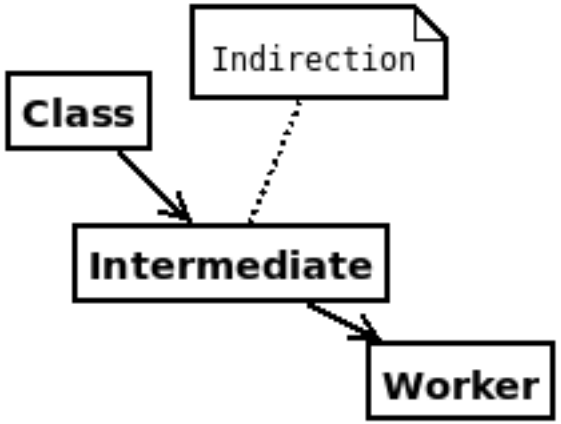
Indirection: example

- Assume that :
  - A point-of-sale terminal application needs to setup a specific communication channel in order to transmit credit payment request
  - The operating system provides a low-level function call API for doing so.

- How to?
  - A class called CreditAuthorizationService is responsible for talking to the communication equipment

Benefits & Contraindications

- Low coupling
- Promotes reusability



GRASP

- Creator
- Information Expert
- Low Coupling
- High Cohesion
- Controller
- Polymorphism
- Pure Fabrication
- Indirection
- Protected Variations

Protected Variations

- Problem:
  - How to design objects, subsystems, and systems so that variations or instabilities in some elements do not have an undesirable impact on other elements?
- Solution:
  - Identify points of predicted variation or instability, assign responsibilities to create a stable interface around them.

- Protected Variations is a fundamental design principle which is the foundation for many design patterns

Protected Variations

- Mechanisms motivated by Protected Variations:
  - Core PV mechanisms: data encapsulation, interfaces, polymorphism, indirection, standards
  - Data-driven designs: style sheets, property files, other mechanisms for reading in configuration data at run time
  - Service lookup including naming services (Java's JNDI) or traders (Java's Jini or UDDI for web services)
  - Interpreter-driven designs
  - Reflective or meta-level designs
  - Uniform access – language support for uniform access to methods and data
  - Liskov Substitution Principle (LSP) - *more following*
  - Structure-hiding designs (Law of Demeter – "don't talk to strangers") - *more following*

Liskov Substitution Principle (LSP)

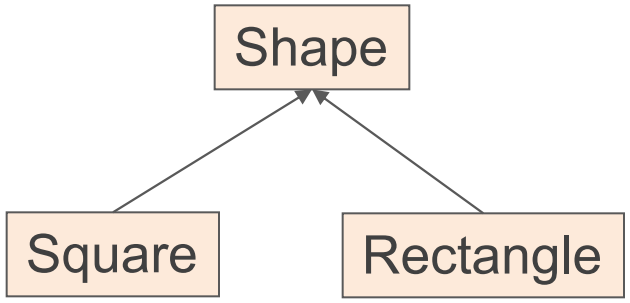
- Due to Barbara Liskov, Turing Award 2008
- LSP: a subclass B of A should be substitutable for superclass A, i.e., B should be a true subtype of A
- Reasoning at the specification level
  - B should not remove methods from A
  - For each B.m, which "substitutes" A.m, B.m's specification is stronger than A.m's specification
  - Client: A a; … a.m(int x,int y);
  - Call a.m can bind to B's m and B's m should not surprise client

Classic Example

- Every Square is-a Rectangle?
- Thus,
  - class Square extends Rectangle { … }
- But is a Square a true subtype of Rectangle?
  - In other words, is Square substitutable for Rectangle in clients expecting a Rectangle?

Every Square is-a Rectangle?

- Square is not a true subtype of Rectangle
  - Rectangles are expected to have height and width that can change independently
  - Squares violate that expectation. Surprise clients
- And the opposite? Is Rectangle a true subtype of Square?
  - No. Squares are expected to have equal height and width. Rectangles violate this expectation
- One solution:
  - make them unrelated



Law of Demeter (Don't talk to strangers)

- Problem:
  - How to avoid knowing about the structure of indirect objects?
- Solution:
  - If two classes have no other reason to be directly aware of each other or otherwise coupled, then the two classes should not directly interact.
    - e.g., in A don't do getB().getC().methodOfC()
  - Within a method, messages should only be sent to the following objects:
    - The this object (or self)
    - A parameter of the method
    - An attribute of self
    - An element of a collection which is an attribute of self
    - An object created within the method

Law of Demeter: Example

```
class Company {
    Collection<Department> departments = new ArrayList<>();
}
class Department {
    private Employee manager;
    public Employee getManager() {
        return manager;
}
class Employee {
    private double salary;
    public double getSalary() {
        return salary;

    }
}
```

Now *Company* needs to have the total of amount spend with Managers' salary. How?

Law of Demeter: Example

- Don't:

```
// within Company
for (Department dept : departments) {
     System.out.println( dept.getManager().getSalary() );
     // now Company depends on Employee
}
```

- Do:

```
class Department { //...
    double getManagerSalary() {
        return getManager().getSalary();
    }
}
// within Company
for (Department dept : departments) {
    System.out.println( dept.getManagerSalary() );
}
```

Benefits & Contraindications

- Keeps coupling between classes low and makes a design more robust
- Adds a small amount of overhead in the form of indirect method calls

Others - SOLID principles

- Single responsibility
  - "*every class should have a single responsibility, and that responsibility should be entirely encapsulated by the class*" (Robert Martin)
- Open/closed (OCP)
  - "*software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification*" (Bertrand Meyer)
- Liskov substitution (LSP)
- Interface segregation
  - "*no client should be forced to depend on methods it does not use*" (similar to High Cohesion of GRASP)
- Dependency inversion
  - "High-level modules should not depend on low-level modules. Both should depend on abstractions"

Others principles / jargons

- Minimalism
  - Keep it simple, stupid (KISS)
  - Worse is better (Less is more)
  - You aren't gonna need it (YAGNI)
  - Principle of good enough (POGE)
  - Quick-and-dirty
- Don't repeat yourself (DRY)
  - *Cut and paste of code is evil.*
- Inversion of control (IoC)

- … and many others

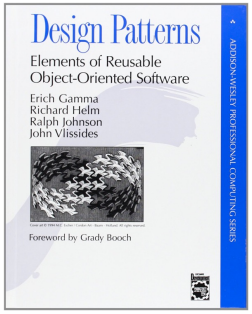
Summary

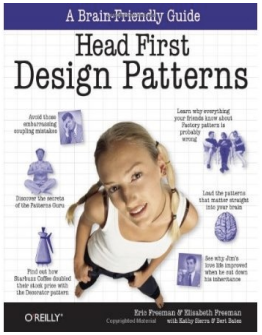
- Skillful assignment of responsibilities is extremely important in object-oriented design
- Patterns are named problem/solution pairs that codify good advice and principles related to assignment of responsibilities
- GRASP identifies several principles:
  - Creator, Information Expert, Controller, Low Coupling, High Cohesion, Polymorphism, Pure Fabrication, Indirection, Protected Variations

Design Patterns
General concepts

UA.DETI.PDS
José Luis Oliveira

Resources





- *Design patterns: elements of reusable object oriented software*. E. Gamma, R. Helm, R. Johnson, J. Vlissides. Addison Wesley, 1994.
- *Head first design patterns*. E. Freeman, E. Freeman, K. Sierra, B. Bates. O'Reilly, 2004.

- Also based on:
  - Object-Oriented Software Engineering, Glenn D. Blank, http://www.cse.lehigh.edu/~glennb/oose/oose.htm
  - Software Design, Joan Serrat, http://www.cvc.uab.es/shared/teach/a21291/web/

What are patterns?

- Principles and solutions codified in a structured format describing a problem and a solution
- A named problem/solution pair that can be applied in new contexts
- It is advice from previous designers to help designers in new situations
- The idea behind design patterns is simple:
  - Write down and catalog common interactions between objects that programmers have frequently found useful.
- Result:
  - Facilitate reuse of object-oriented code between projects and between programmers.

Some definitions of design patterns

- "Design patterns constitute a set of rules describing how to accomplish certain tasks in the realm of software development." (Pree, 1994)
- "Design patterns focus more on reuse of recurring architectural design themes, while frameworks focus on detailed design… and implementation." (Coplien & Schmidt, 1995).
- "A pattern addresses a recurring design problem that arises in specific design situations and presents a solution to it" (Buschmann, et. al. 1996)
- "Patterns identify and specify abstractions that are above the level of single classes and instances, or of components." (Gamma, et al., 1993)

Characteristics of Good patterns

- It solves a problem
- It is a proven concept
- The solution isn't obvious
- It describes a relationship
- The pattern has a significant human component

Types of patterns

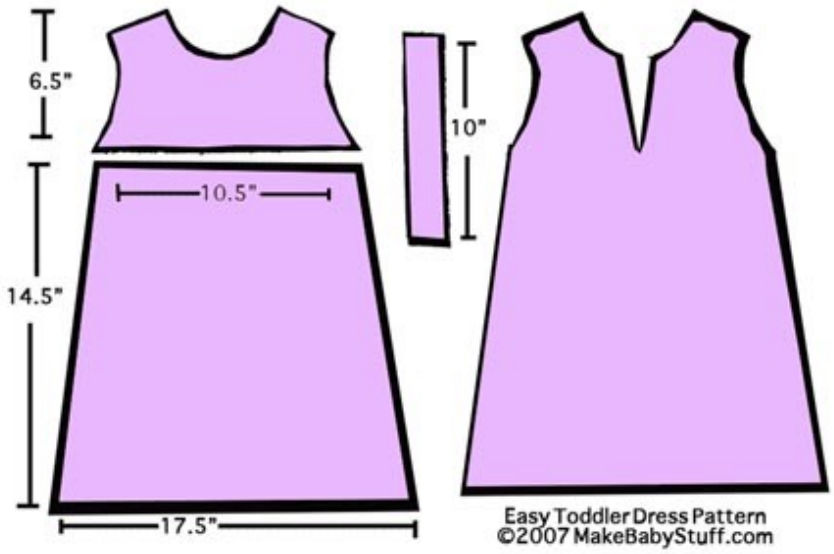
- Architectural Patterns
  - Expresses a fundamental structural organization or schema for software systems.
- Design Patterns
  - Provides a scheme for refining the subsystems or components of a software system, or the relationships between them.
- Idioms
  - An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language.

Design patterns in architecture

- A pattern is a recurring solution to a standard problem, in a context.
- Christopher Alexander, professor of architecture…
  - Why is what a prof of architecture says relevant to software?
  - "A pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."

Design and dress patterns

- Jim Coplein, a software engineer:
  - "I like to relate this definition to dress patterns …
  - I could tell you how to make a dress by specifying the route of a scissors through a piece of cloth in terms of angles and lengths of cut. Or, I could give you a pattern. Reading the specification, you would have no idea what was being built or if you had built the right thing when you were finished. The pattern foreshadows the product: it is the rule for making the thing, but it is also, in many respects, the thing itself."



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

Gang of Four (GoF) Patterns

- Eric Gamma and colleagues published in 1995 the influential book Design patterns: *Elements of Reusable Object-Oriented Software*.
- Has a catalogue of 23 patterns. For each one, a template is followed:
  - Name
  - Intent : what it does and advantages 1–2 sentences
  - Motivation : example
  - Structure : template class diagram
  - Applicability : when to use it
  - Consequences : advantages and shortcomings
  - Implementation discussion, C++ sample code

Naming Patterns – important!

- Patterns have suggestive names:
  - Arched Columns Pattern, Easy Toddler Dress Pattern, etc.
- Why is naming a pattern or principle helpful?
  - It supports chunking and incorporating that concept into our understanding and memory
  - It facilitates communication

GoF Patterns

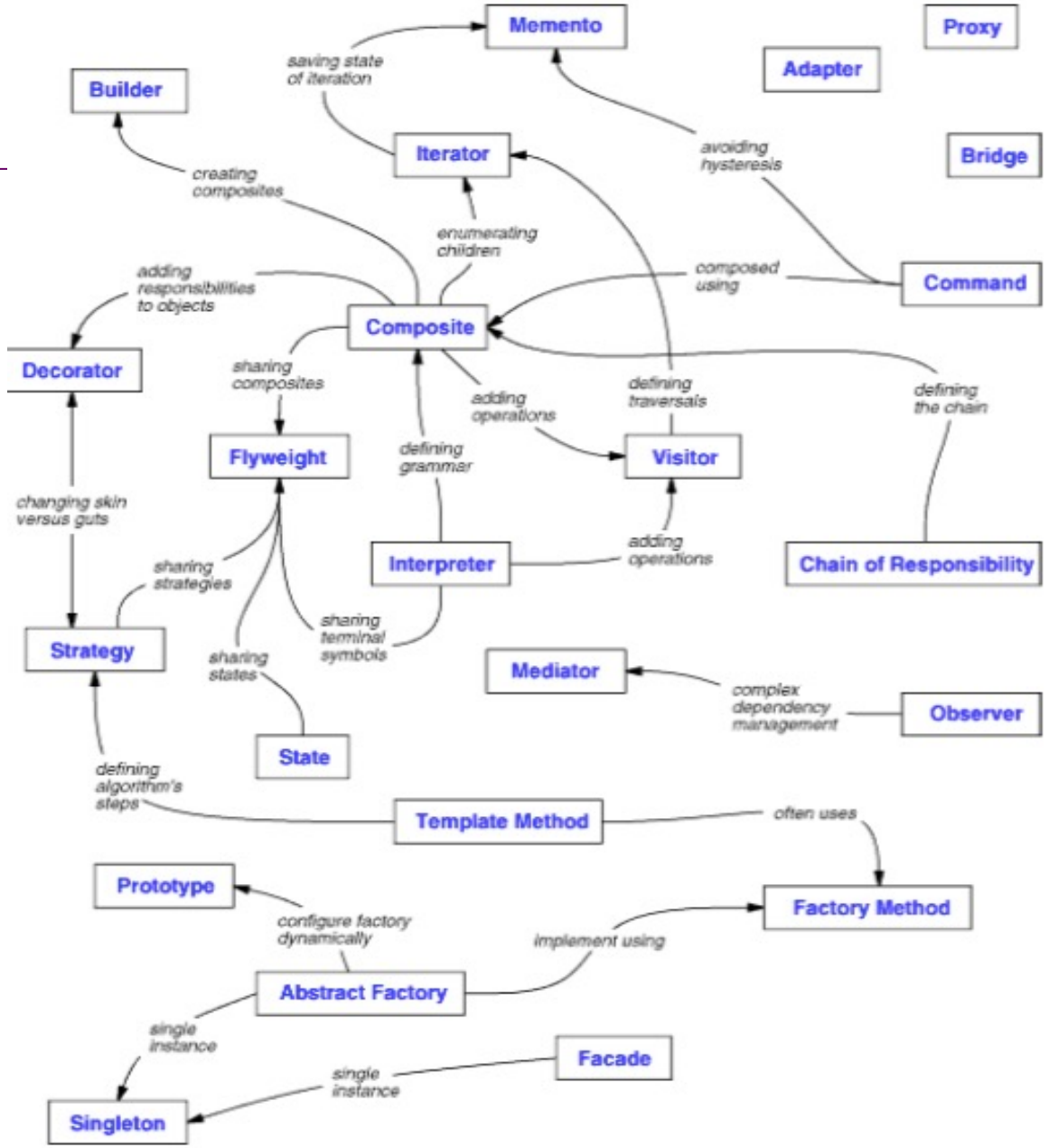
- Gamma et al. classify patterns into 3 groups:

- Creational
  - patterns concern the process of object creation
- Structural
  - patterns deal with the composition of classes or objects
- Behavioral
  - patterns characterize the ways in which classes or objects interact and distribute responsibilities

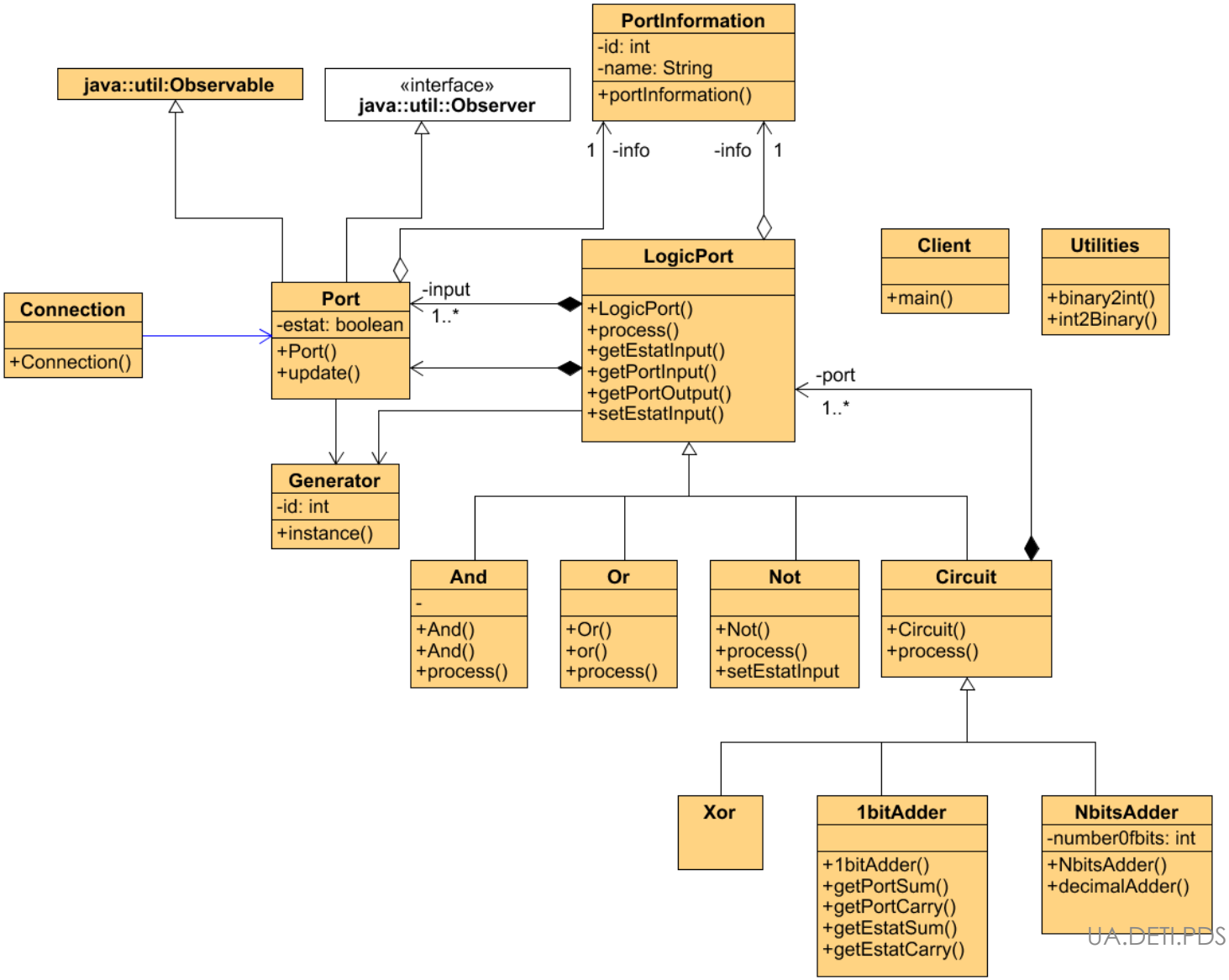
GoF Patterns

By Purpose				
		Creational	Structural	Behavioral
By Scope	Class	• Factory Method	• Adapter (class)	• Interpreter • Template Method
	Object	• Abstract Factory • Builder • Prototype • Singleton	• Adapter (object) • Bridge • Composite • Decorator • Façade • Flyweight • Proxy	• Chain of Responsibility • Command • Iterator • Mediator • Memento • Observer • State • Strategy • Visitor

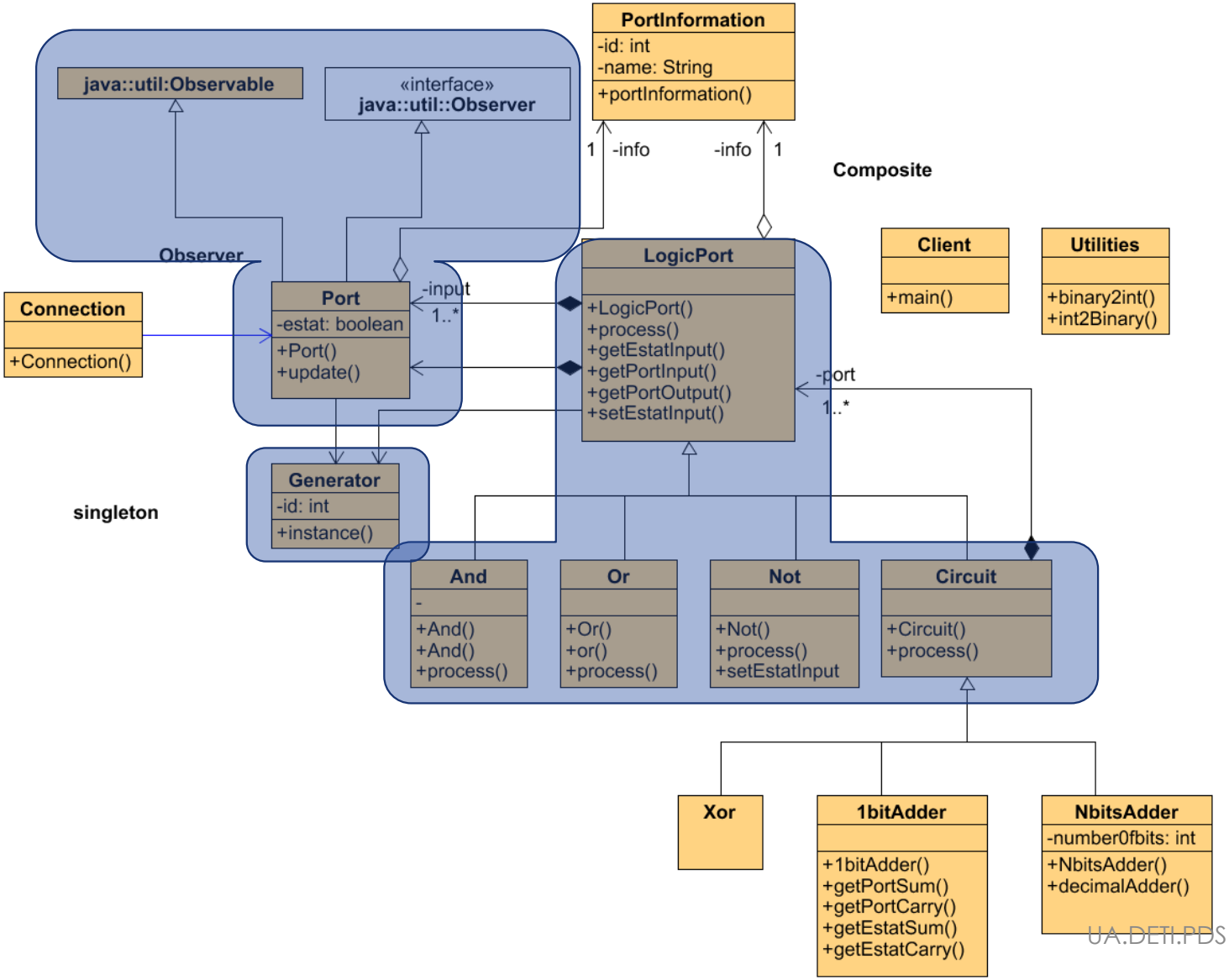
Relationships



Why patterns?



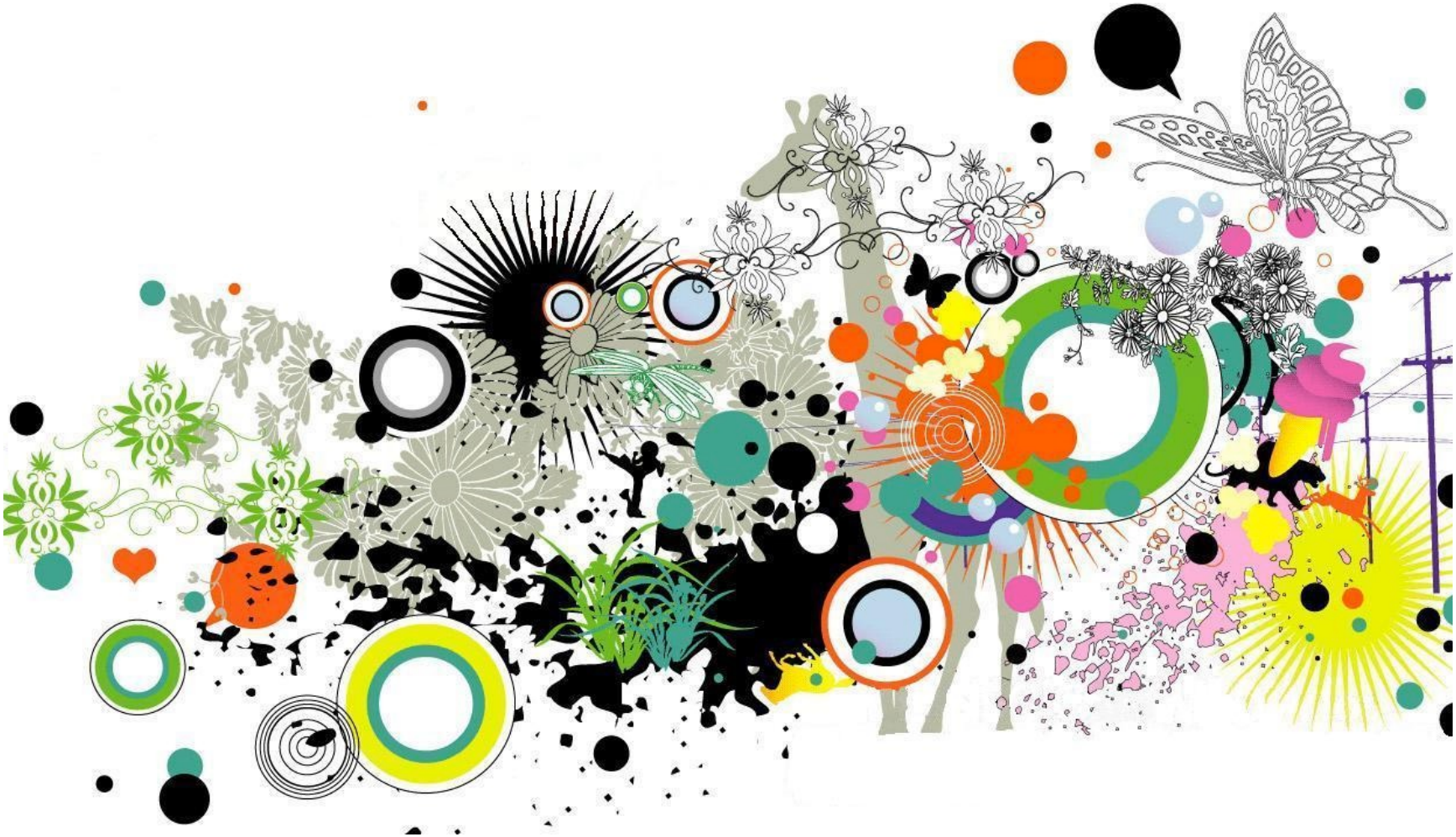
Why patterns?



Why patterns?

- A novice chess player knows
  - the game rules
  - the value of all pieces
- A novice OO designer must know
  - inheritance, encapsulation, data abstraction . . .
  - UML notation
- A good chess player knows
  - tactics: occupy central cells, …
  - strategies: immobilize, win with two bishops, …
  - apertures, famous matches
- An expert designer knows
  - object oriented principles
  - examples of good designs
  - design patterns

More on this in the next weeks…

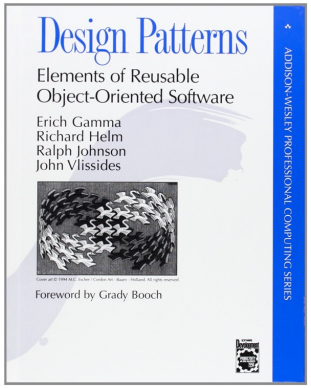


Design Patterns – Creational

UA.DETI.PDS
José Luis Oliveira

Resources

- Design Patterns – Elements of Reusable Object-Oriented Software; Gamma, et. al.



- *Design Patterns Explained Simply (sourcemaking.com)*

Creational patterns

Class

- Factory Method

Object

- Abstract Factory
- Builder
- Singleton
- Object Pool
- Prototype

Creational Patterns

- Problem: constructors in Java (and other OO languages) are inflexible
  - 1. Can't return a subtype of the type they belong to
  - 2. Always return a fresh new object, can't reuse
- "Factory" creational patterns present a solution to the first problem
  - Factory method, Factory object, Prototype
- "Sharing" creational patterns present a solution to the second problem
  - Singleton

Factory Method

Class

- Factory Method

Object

- Abstract Factory
- Builder
- Singleton
- Object Pool
- Prototype

Motivation

- Intent
  - The new operator is considered harmful.
  - Define a "virtual" constructor.
  - Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- Problem
  - A framework needs to standardize the architectural model for a range of applications, but allow for individual applications to define their own domain objects and provide for their instantiation.

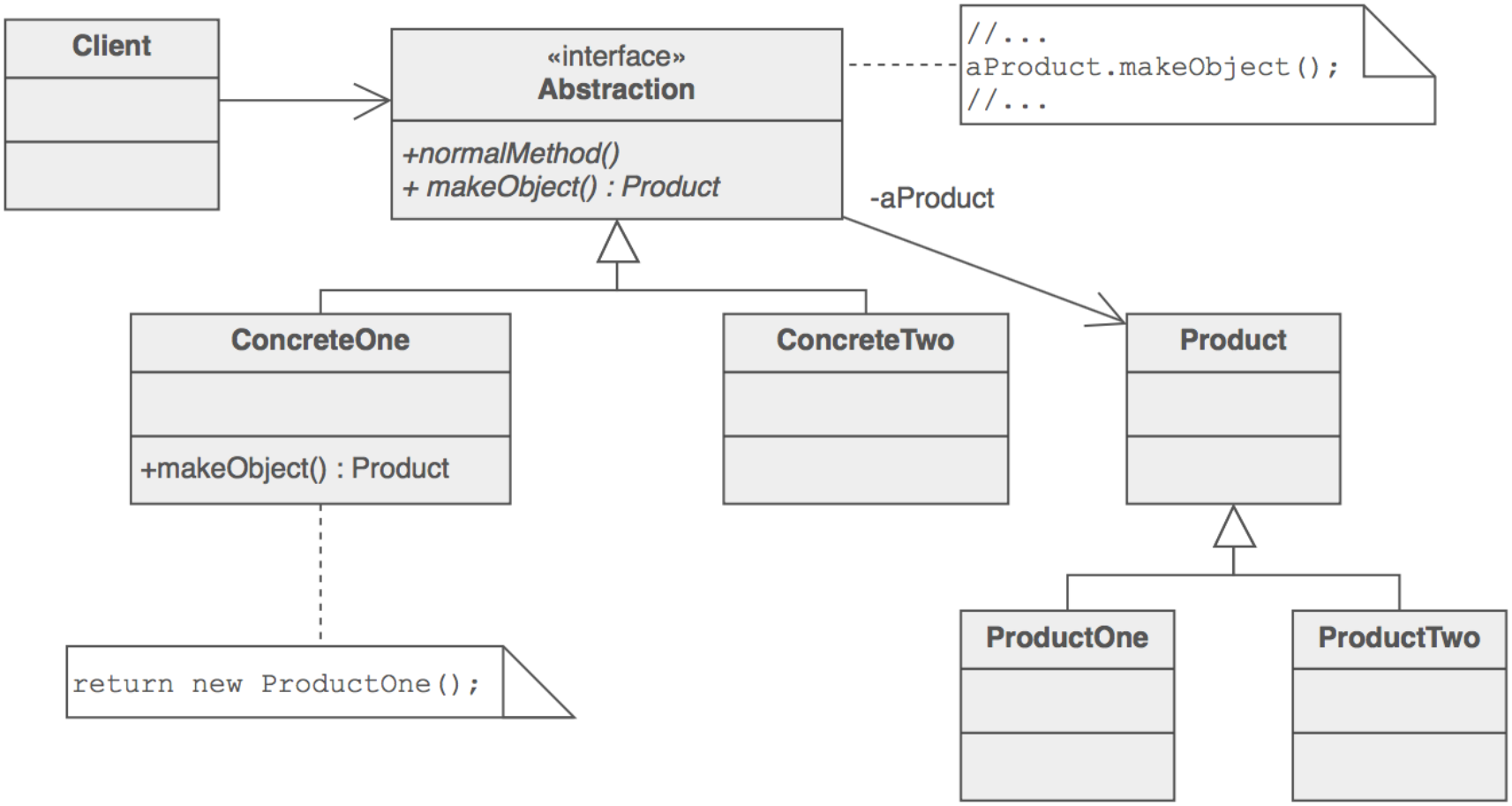
Solution (simple example)

```
public final class ComplexNumber {

        // Static factory method returns an object of this class.
    public static ComplexNumber valueOf(double aReal, double aImaginary) {
        return new ComplexNumber(aReal, aImaginary);
    }
        // Caller cannot see this private constructor.
    private ComplexNumber(double aReal, double aImaginary) {
        fReal = aReal;
        fImaginary = aImaginary;
    }
        //…
    private double fReal;
    private double fImaginary;
}
```

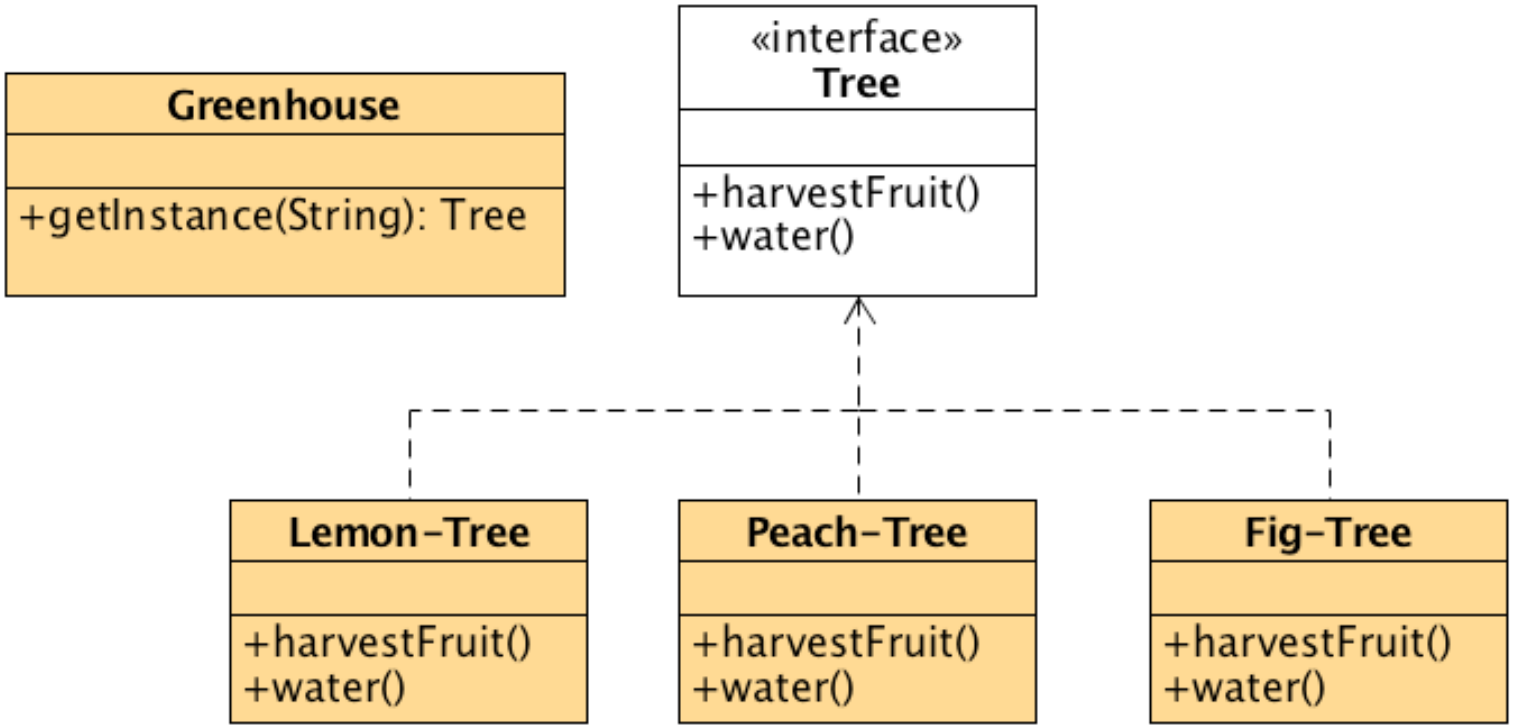
Solution

- An increasingly popular definition of factory method is a static method of a class that returns an object of that class' type.
  - But unlike a constructor, the actual object it returns might be an instance of a subclass.
  - Unlike a constructor, an existing object might be reused, instead of a new object created.
  - Unlike a constructor, factory methods can have different and more descriptive names
    - Color.make_RGB_color(float red, float green, float blue)
    - Color.make_HSB_color(float hue, float saturation, float brightness)

Structure


Client
«interface»
Abstraction
+normalMethod()
+ makeObject() : Product
//...
aProduct.makeObject();
//...
-aProduct
ConcreteOne
+makeObject() : Product
ConcreteTwo
Product
ProductOne
ProductTwo
return new ProductOne();

Example


Greenhouse
+getInstance(String): Tree
«interface»
Tree
+harvestFruit()
+water()
Lemon-Tree
+harvestFruit()
+water()
Peach-Tree
+harvestFruit()
+water()
Fig-Tree
+harvestFruit()
+water()

Example

```
interface Arvore {
    void regar();
    void colherFruta();
}

class Figueira implements Arvore {
    protected Figueira() {System.out.println("Figueira plantada."); }
    public void regar() { System.out.println("Figueira: Regar muito pouco"); }
    public void colherFruta() { System.out.println("Hum.. figos!"); }
}

class Pessegueiro implements Arvore {
    protected Pessegueiro() {System.out.println("Pessegueiro plantado."); }
    public void regar() { System.out.println("Pessegueiro: Regar normal"); }
    public void colherFruta() { System.out.println("Boa.. pessegos!"); }
}

class Limoeiro implements Arvore {
    protected Limoeiro() {System.out.println("Limoeiro plantada."); }
    public void regar() { System.out.println("Limoeiro: Regar pouco"); }
    public void colherFruta() { System.out.println("Ahh.. Caipirinha!"); }
}
```

Example

```
class Viveiro {
    public static Arvore factory(String pedido){
        if (pedido.equalsIgnoreCase("Figueira"))
            { return new Figueira(); }
        if (pedido.equalsIgnoreCase("Pessegueiro"))
            { return new Pessegueiro(); }
        if (pedido.equalsIgnoreCase("Limoeiro"))
            { return new Limoeiro(); }
        else
            throw new IllegalArgumentException(pedido +" não existente!");
    }
    //...
}
```

```
// or with Java Reflection
  public static Arvore factory2(String pedido) {
     Arvore arv = null;
     try {
        arv = (Arvore) Class.forName("Factory."+pedido).newInstance();
     } catch(Exception e) {
        throw new IllegalArgumentException(pedido +" nao existente!");
     }
     return arv;
  }
```

Example

```
public static void main(String[] args) {
    Arvore pomar[] = {
        Viveiro.factory("Figueira"),
        Viveiro.factory("Pessegueiro"),
        Viveiro.factory("Limoeiro")
    };
    for (Arvore a: pomar)
        a.regar();
    for (Arvore a: pomar)
        a.colherFruta();
}
```

```
Figueira plantada.
Pessegueiro plantado.
Limoeiro plantada.
Figueira: Regar muito pouco
Pessegueiro: Regar normal
Limoeiro: Regar pouco
Hum.. figos!
Boa.. pessegos!
Ahh.. Caipirinha!
```

Another Example

```
class Race {
   Race createRace() {
      Bicycle bike1 = new Bicycle();
      Bicycle bike2 = new Bicycle(); //...
   }
}
class TourDeFrance extends Race {
   Race createRace() {
      Bicycle bike1 = new RoadBicycle();
      Bicycle bike2 = new RoadBicycle(); //...
   }
}
class Cyclocross extends Race {
   Race createRace() {
      Bicycle bike1 = new MountainBicycle();
      Bicycle bike2 = new MountainBicycle(); //...
   }
}
```

Problem with this code:
Code duplication!

createRace is very similar among the 3 classes.

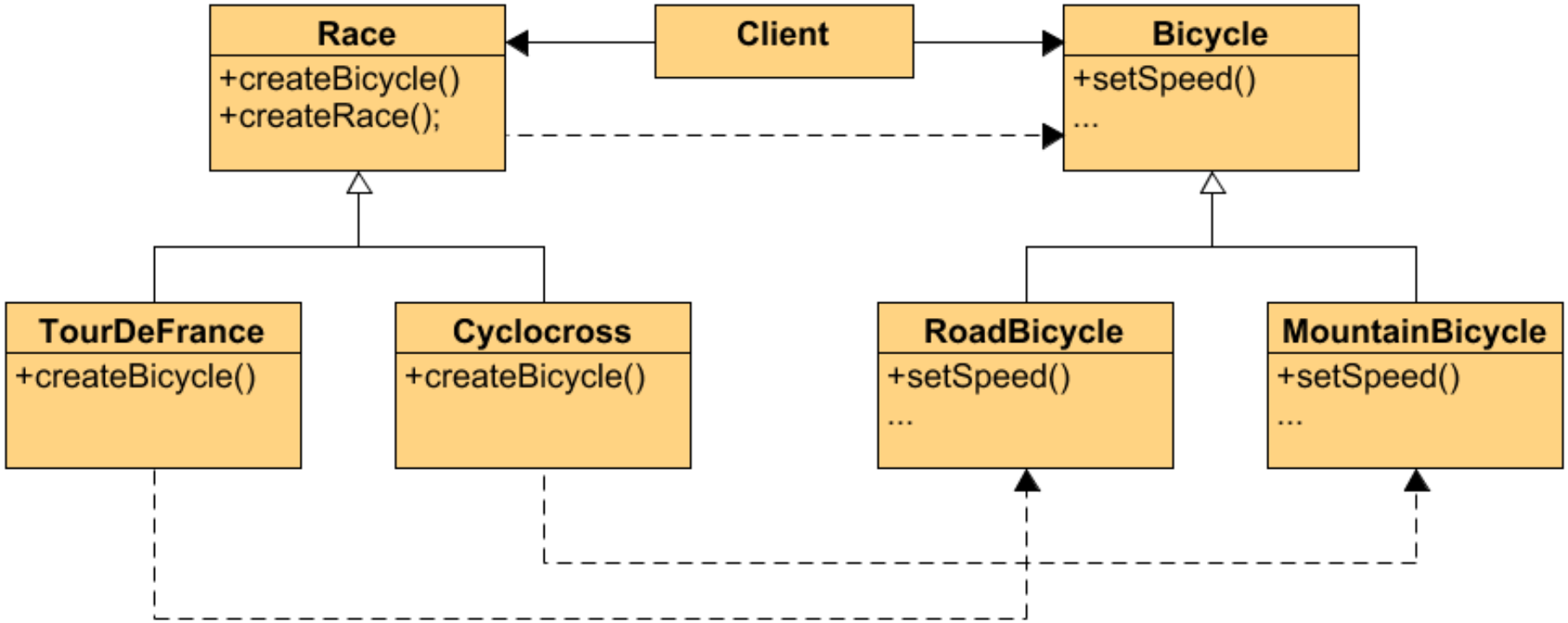
Why not have a single createRace in Race?

Using Factory Method

```
class Race {
   Bicycle createBicycle() {
      return new Bicycle();
   }
   Race createRace() {
      Bicycle bike1 = createBicycle();
      Bicycle bike2 = createBicycle(); //...
   }
}
class TourDeFrance extends Race {
   Bicycle createBicycle() {
      return new RoadBicycle();
   }
}
class Cyclocross extends Race {
   Bicycle createBicycle() {
      return new MountainBicycle();
   }
}
```

Parallel Hierarchies

- Can extend with new Races and Bikes with no modification (generally) to Client



Factory Methods in the JDK

- Calendar replaced Date (JDK1.0)
- DateFormat encapsulates knowledge on how to format a Date
  - Options: Just date? Just time? date+time?

```
Calendar td = Calendar.getInstance();
Date today = td.getTime();

DateFormat df1 = DateFormat.getDateInstance();
DateFormat df2 = DateFormat.getTimeInstance();
DateFormat df3 = DateFormat.getDateInstance(DateFormat.FULL);

System.out.println(df1.format(today));  // "9/jan/2015"
System.out.println(df2.format(today));  // "10:01:24"
System.out.println(df3.format(today));  // "Sexta-feira, 9 de Janeiro de 2015"
```

Check list

- If the constructor may lead to inconsistent objects, consider designing a factory method.
- Consider making all constructors private or protected.
- If you have an inheritance hierarchy that exercises polymorphism, consider adding a polymorphic creation capability by defining a static factory method in the base class.
- Consider designing an internal "object pool" that will allow objects to be reused instead of created from scratch.

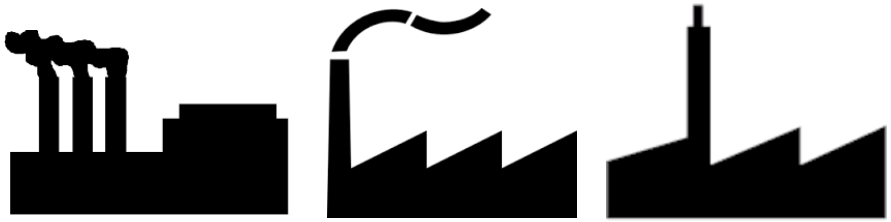
Abstract Factory

Class

- Factory Method

Object

- Abstract Factory
- Builder
- Singleton
- Object Pool
- Prototype

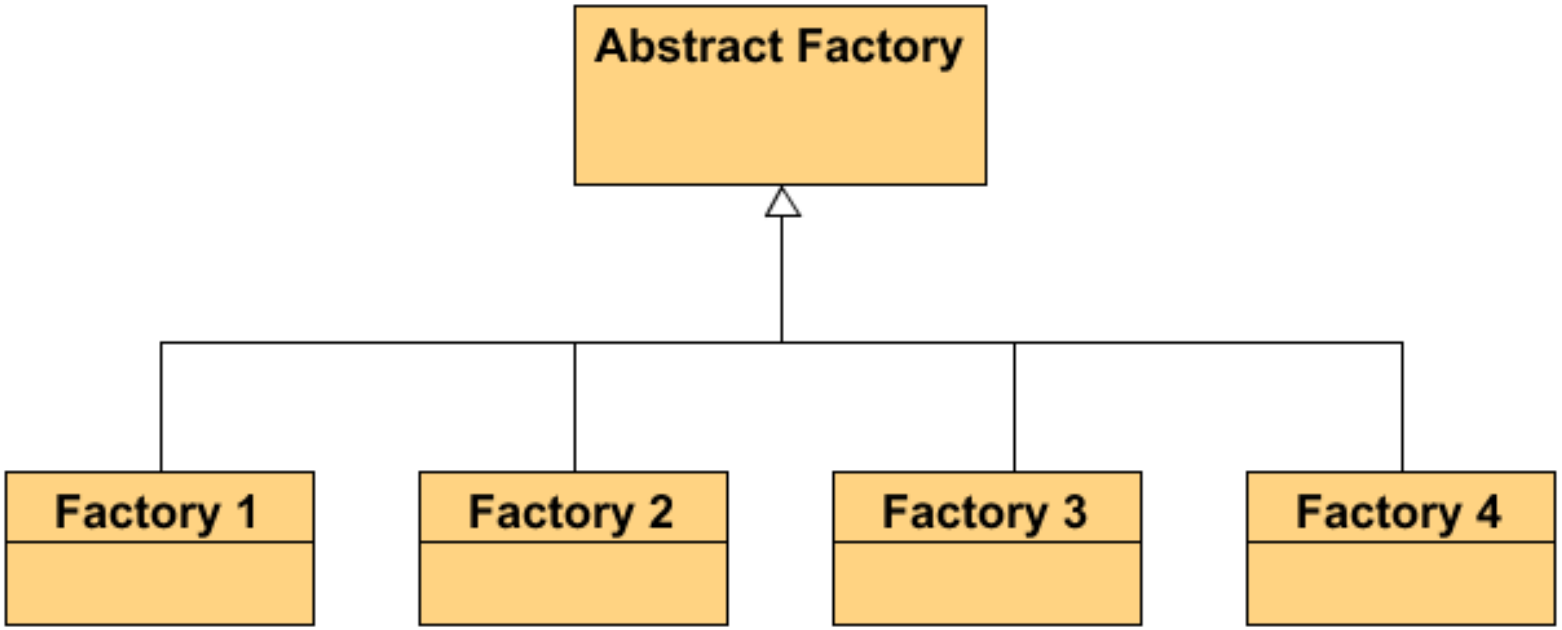


Motivation

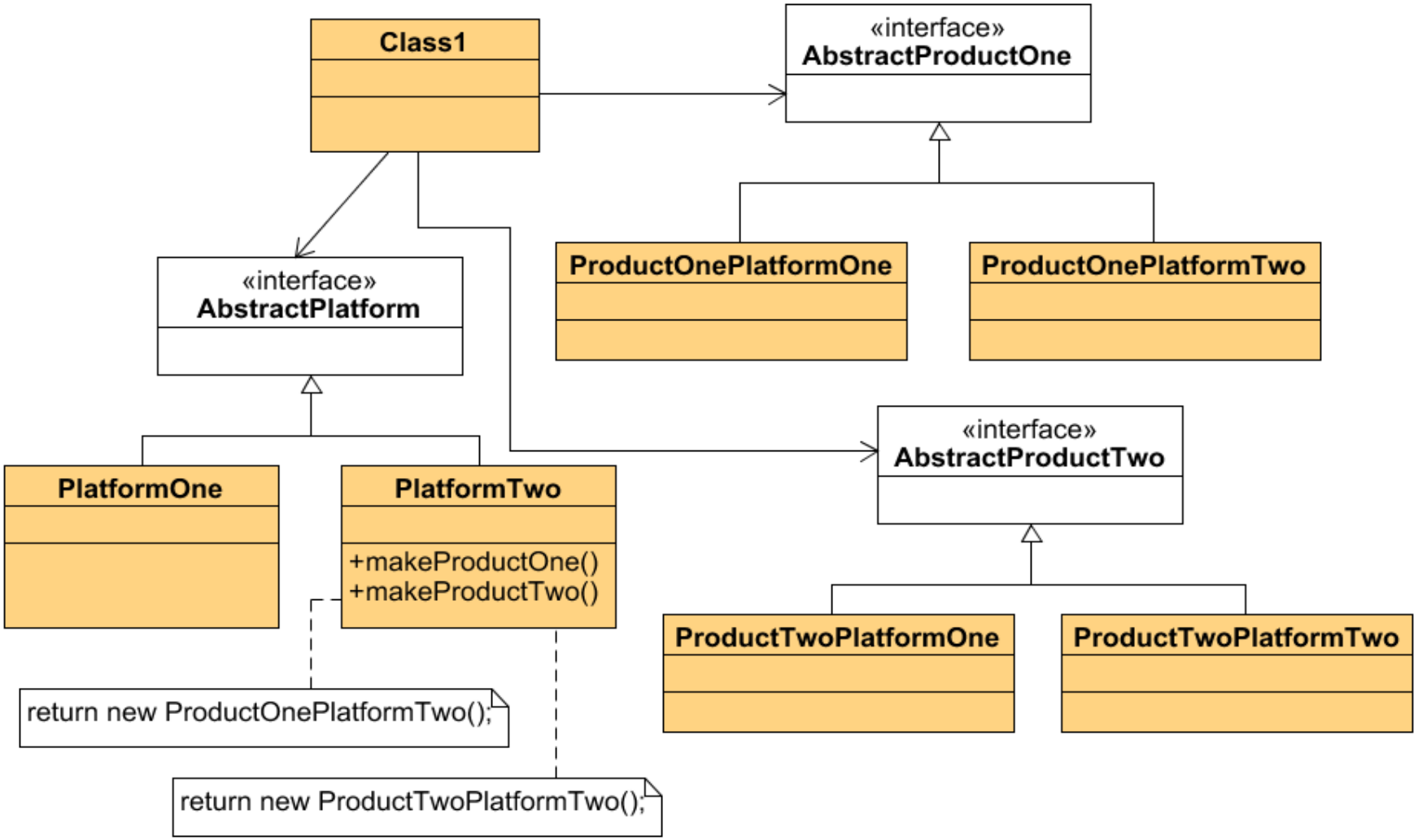
- Problem
  - If an application is to be portable, it needs to encapsulate platform dependencies.
  - These "platforms" might include a windowing system, operating system, database, etc.
- Intent
  - The *new* operator is considered harmful.
  - Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
  - A hierarchy that encapsulates many possible "platforms", and the construction of a suite of "products".

Solution

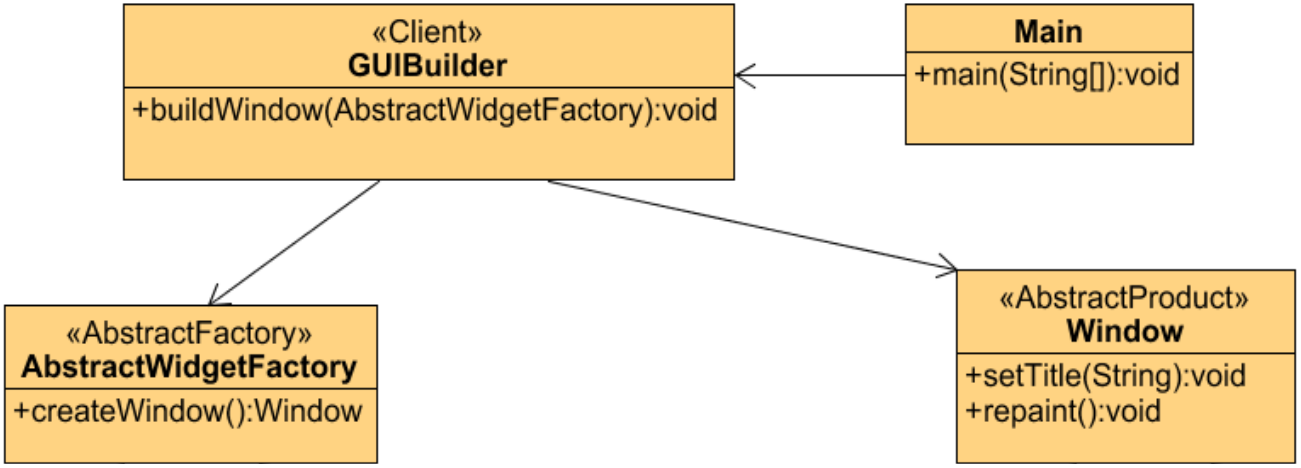
- The Abstract Factory defines a Factory Method per product



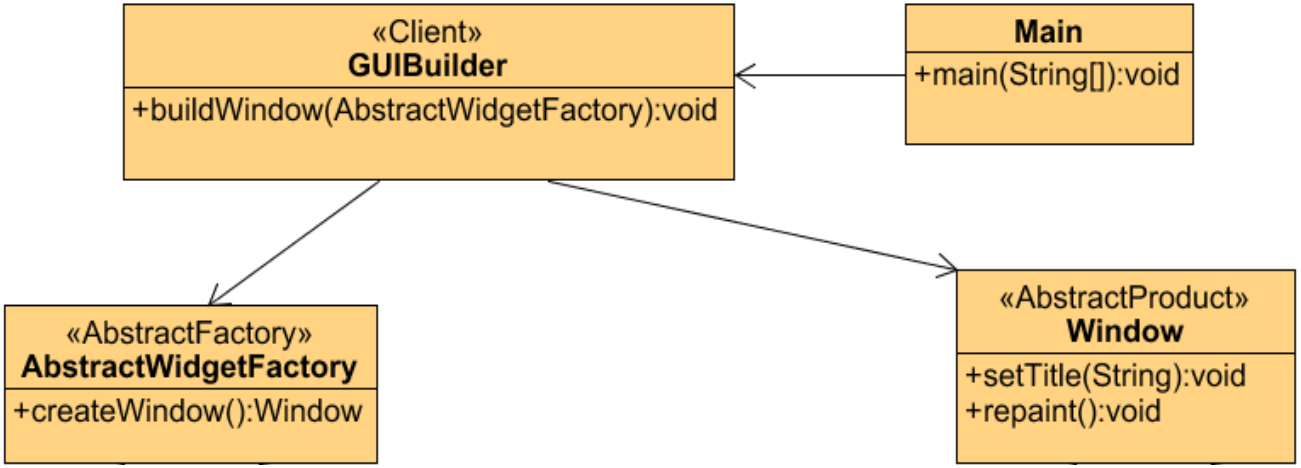
Structure


Class1
«interface»
AbstractProductOne
«interface»
AbstractPlatform
ProductOnePlatformOne
ProductOnePlatformTwo
«interface»
AbstractProductTwo
PlatformOne
PlatformTwo
+makeProductOne()
+makeProductTwo()
ProductTwoPlatformOne
ProductTwoPlatformTwo
return new ProductOnePlatformTwo();
return new ProductTwoPlatformTwo();

Example

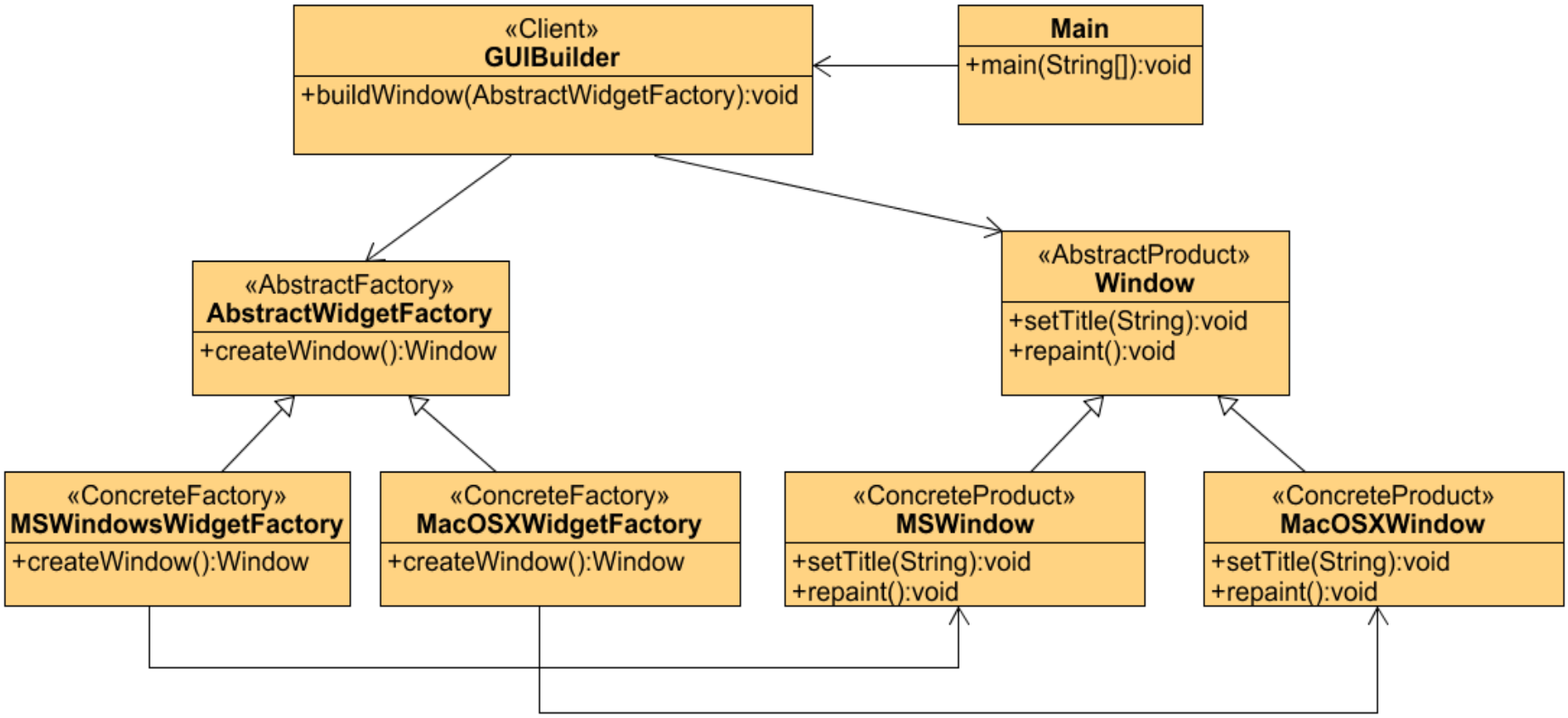

«Client»
GUIBuilder
+buildWindow(AbstractWidgetFactory):void
Main
+main(String[]):void
«AbstractFactory»
AbstractWidgetFactory
+createWindow():Window
«AbstractProduct»
Window
+setTitle(String):void
+repaint():void

Example




```
public class GUIBuilder {
    public void buildWindow(AbstractWidgetFactory widgetFactory) {
        Window window = widgetFactory.createWindow();
        window.setTitle("New Window");
    }
}
```

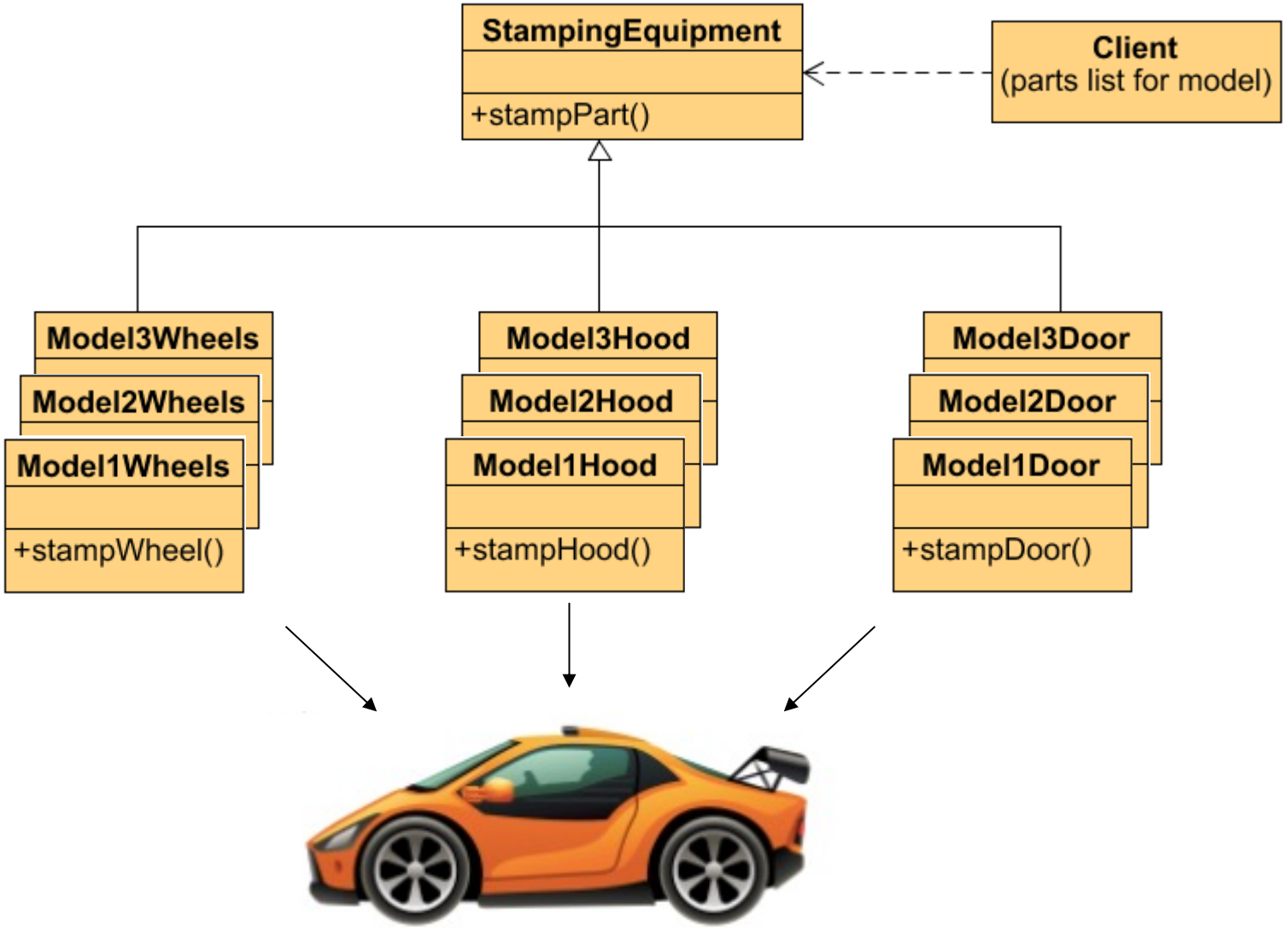
Example


«Client»
GUIBuilder
+buildWindow(AbstractWidgetFactory):void
Main
+main(String[]):void
«AbstractFactory»
AbstractWidgetFactory
+createWindow():Window
«AbstractProduct»
Window
+setTitle(String):void
+repaint():void
«ConcreteFactory»
MSWindowsWidgetFactory
+createWindow():Window
«ConcreteFactory»
MacOSXWidgetFactory
+createWindow():Window
«ConcreteProduct»
MSWindow
+setTitle(String):void
+repaint():void
«ConcreteProduct»
MacOSXWindow
+setTitle(String):void
+repaint():void

Example

```
public class MainTest {
   public static void main(String[] args) {
      GUIBuilder builder = new GUIBuilder();
      if (Platform.currentPlatform()=="MACOSX")
         builder.buildWindow(new MacOSXWidgetFactory());
      else if (Platform.currentPlatform()=="WIN")
         builder.buildWindow(new MsWindowsWidgetFactory());
      else //...
   }
}
public class GUIBuilder {
   public void buildWindow(AbstractWidgetFactory widgetFactory) {
      Window window = widgetFactory.createWindow();
      window.setTitle("New Window");
   }
}
```

Another example



Check list

- Decide if "platform independence" and creation services are the current source of pain.
- Map out a matrix of "platforms" versus "products".
- Define a factory interface that consists of a factory method per product.
- Define a factory derived class for each platform that encapsulates all references to the new operator.
- The client should retire all references to new, and use the factory methods to create the product objects.

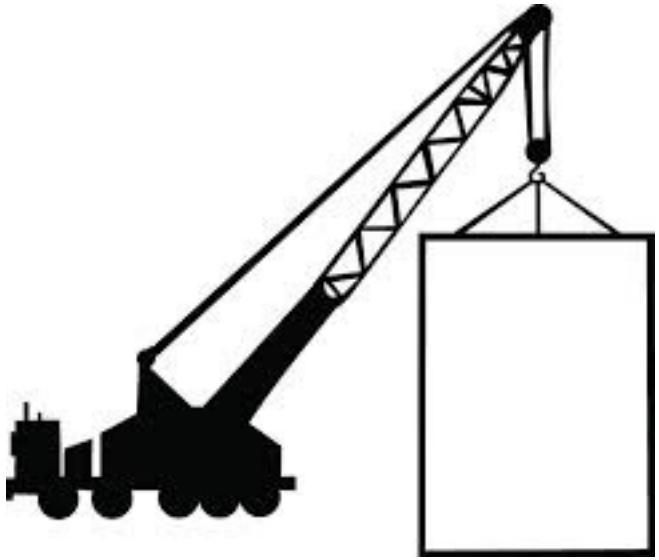
Builder

Class

- Factory Method

Object

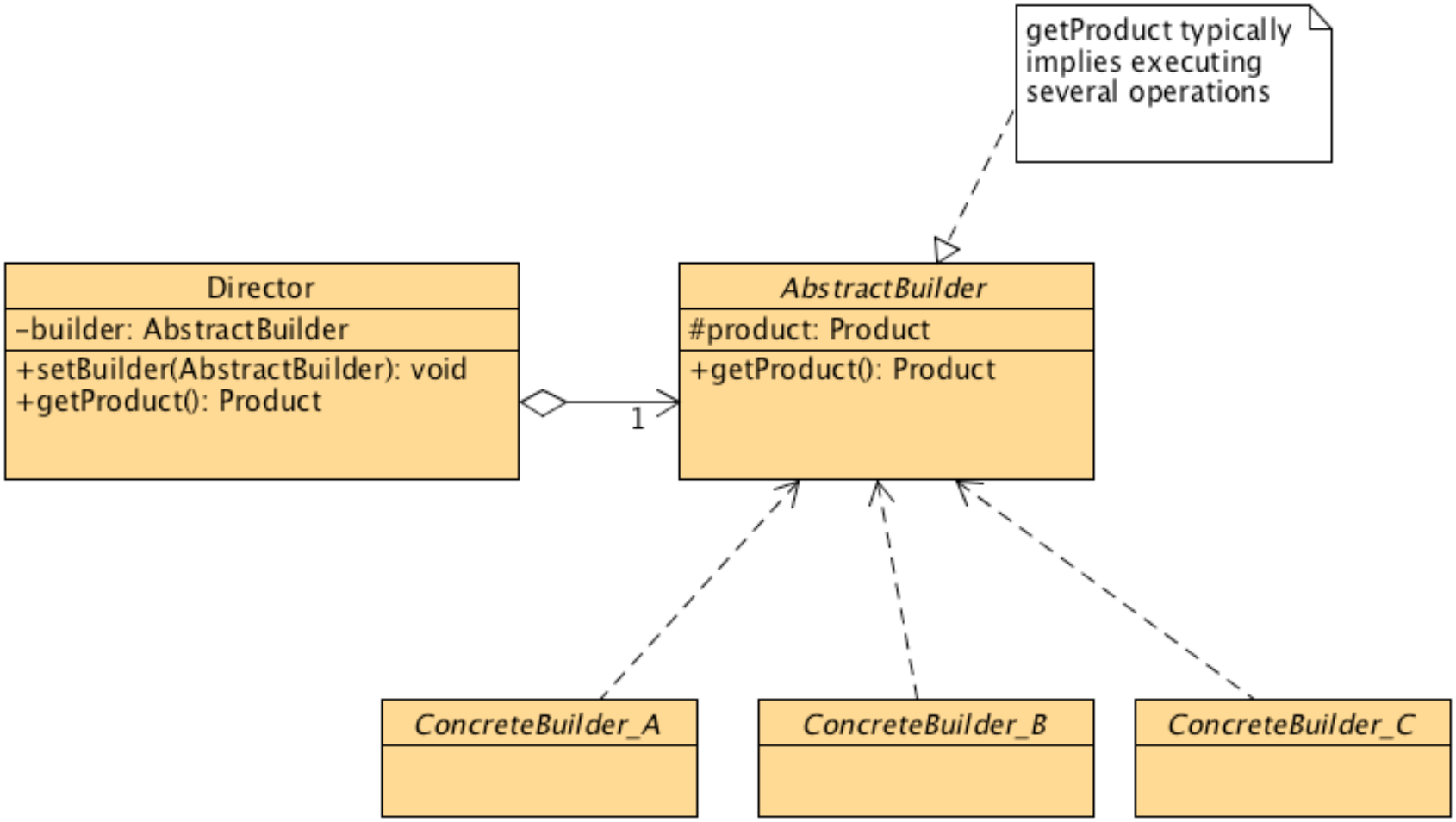
- Abstract Factory
- Builder
- Singleton
- Object Pool
- Prototype



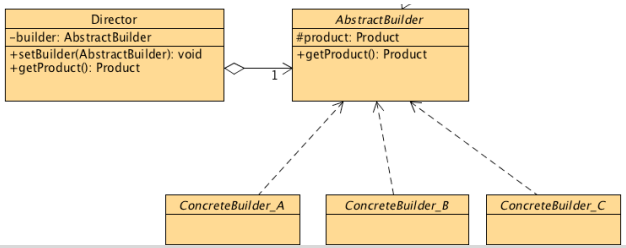
Motivation

- Intent
  - Separate the construction of a complex object from its representation so that the same construction process can create different representations.
  - Parse a complex representation, create one of several targets.
- Problem
  - An application needs to create the elements of a complex aggregate. The specification for the aggregate exists on secondary storage and one of many representations needs to be built in primary storage.

Structure

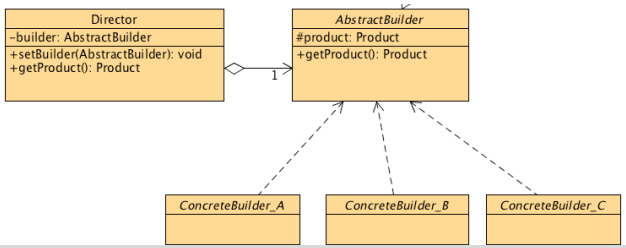

getProduct typically implies executing several operations
Director
-builder: AbstractBuilder
+setBuilder(AbstractBuilder): void
+getProduct(): Product
1
AbstractBuilder
#product: Product
+getProduct(): Product
ConcreteBuilder_A
ConcreteBuilder_B
ConcreteBuilder_C

Example (1)



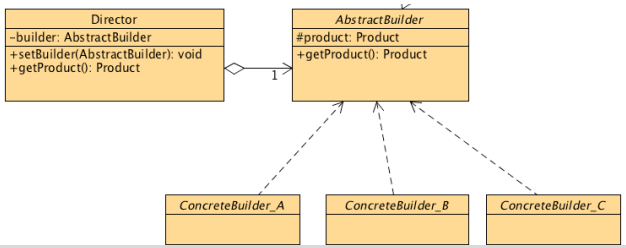

```
class Pizza {    /* "Product" */
  private String dough;
  private String sauce;
  private String topping;
  public void setDough(String dough)     { this.dough = dough; }
  public void setSauce(String sauce)     { this.sauce = sauce; }
  public void setTopping(String topping) { this.topping = topping; }
  public String toString() { /* .. */}
}

abstract class PizzaBuilder {   /* "Abstract Builder" */
  protected Pizza pizza = new Pizza();

  public Pizza getPizza() { return pizza; }

  public abstract void buildDough();
  public abstract void buildSauce();
  public abstract void buildTopping();
}
```

Example (2)



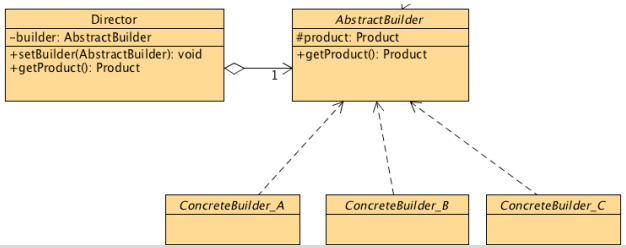

```
/* "ConcreteBuilder" */
class HawaiianPizzaBuilder extends PizzaBuilder {
  public void buildDough()   { pizza.setDough("cross"); }
  public void buildSauce()   { pizza.setSauce("mild"); }
  public void buildTopping() { pizza.setTopping("ham+pineapple"); }
}

/* "ConcreteBuilder" */
class SpicyPizzaBuilder extends PizzaBuilder {
  public void buildDough()   { pizza.setDough("pan baked"); }
  public void buildSauce()   { pizza.setSauce("hot"); }
  public void buildTopping() { pizza.setTopping("pepperoni+salami"); }
}
```

Example (3)




```
class Waiter {    /* "Director" */
    private PizzaBuilder pizzaBuilder;

    public void setPizzaBuilder(PizzaBuilder pb) {
        pizzaBuilder = pb;
    }
    public void constructPizza() {
        pizzaBuilder.buildDough();
        pizzaBuilder.buildSauce();
        pizzaBuilder.buildTopping();
    }
    public Pizza getPizza() {
        return pizzaBuilder.getPizza();
    }

}
```

Example (4)




```
/* A customer ordering a pizza. */
class BuilderExample {
  public static void main(String[] args) {
    Waiter waiter = new Waiter();

    waiter.setPizzaBuilder(new HawaiianPizzaBuilder());
    waiter.constructPizza();
    Pizza pizza = waiter.getPizza();
    System.out.println(pizza);

    waiter.setPizzaBuilder(new SpicyPizzaBuilder());
    waiter.constructPizza();
    pizza = waiter.getPizza();
    System.out.println(pizza);

  }
}
```

```
Pizza [dough=cross, sauce=mild, topping=ham+pineapple]
Pizza [dough=pan baked, sauce=hot, topping=pepperoni+salami]
```

Check list

- Decide if a common input and many possible representations (or outputs) is the problem at hand.
- Encapsulate the parsing of the common input in a Reader class (*the Director*).
- Design a standard protocol for creating all possible output representations. Capture the steps of this protocol in a Builder interface.
- Define a Builder derived class for each target representation.
- The client creates a Reader object and a Builder object, and registers the latter with the former.
- The client asks the Reader to "construct".
- The client asks the Builder to return the result.

Another example – slightly different

- Consider a builder when faced with many constructors
- Use a builder inner class

Another example

```
public class NutritionFacts {
     private final int servingSize;     // (mL) required
     private final int servings;        // (per container)
     private final int calories;        // optional
     private final int fat;             // (g) optional
     private final int sodium;          // (mg) optional
     private final int carbohydrate;    // (g)  optional

     public NutritionFacts(int servingSize, int servings,
                           int calories, int fat, int sodium,
                           int carbohydrate) {
       this.servingSize = servingSize;
       this.servings = servings;
       this.calories = calories;
       this.fat = fat;
       this.sodium = sodium;
       this.carbohydrate = carbohydrate;
     }
   }
```

What's wrong?

Example – more constructors

```
public NutritionFacts(int servingSize, int servings) {
  this(servingSize, servings, 0);
}

public NutritionFacts(int servingSize, int servings,
                      int calories) {
  this(servingSize, servings, calories, 0);
}

public NutritionFacts(int servingSize, int servings,
                      int calories, int fat) {
  this(servingSize, servings, calories, fat, 0);
}

public NutritionFacts(int servingSize, int servings,
                      int calories, int fat, int sodium) {
  this(servingSize, servings, calories, fat, sodium, 0);
}
```

Still wrong?

Example – with Builder (1)

```
public class NutritionFacts { // Builder Pattern
    private final int servingSize;
    private final int servings;
    private final int calories;
    private final int fat;
    private final int sodium;
    private final int carbohydrate;

    public static class Builder {
        // Required parameters
        private final int servingSize;
        private final int servings;
        // Optional parameters - initialized to default values
        private int calories = 0;
        private int fat = 0;
        private int carbohydrate = 0;
        private int sodium = 0;

        //...
```

Example – with Builder (2)

```
        public Builder(int servingSize, int servings) {
            this.servingSize = servingSize;  this.servings = servings;
        }
        public Builder calories(int val) {
            calories = val;
            return this;
        }
        public Builder fat(int val) {
            fat = val;
            return this;
        }
        public Builder carbohydrate(int val) {
            carbohydrate = val;
            return this;
        }
        public Builder sodium(int val) {
            sodium = val;
            return this;
        } //...
```

Example – with Builder (3)

```
        public NutritionFacts build() {
            return new NutritionFacts(this);
        }
    } // end of class Builder

    private NutritionFacts(Builder builder) {
        servingSize = builder.servingSize;
        servings = builder.servings;
        calories = builder.calories;
        fat = builder.fat;
        sodium = builder.sodium;
        carbohydrate = builder.carbohydrate;
    }
}
```

We can now use this static inner class as follows:

```
NutritionFacts sodaDrink = new NutritionFacts.Builder(240, 8).
      calories(100).sodium(35).carbohydrate(27).build();
```

Builders in the JDK

- All implementations of java.lang.Appendable are good example of use of Builder pattern in java.

```
public static void main(String[] args) {
    String data = new StringBuilder("Exemplo de builder_")
        .append(1)
        .append(true)
        .append("_para_fechar")
        .toString();
    System.out.println(data);
}
```

```
Exemplo de builder_1true_para_fechar
```

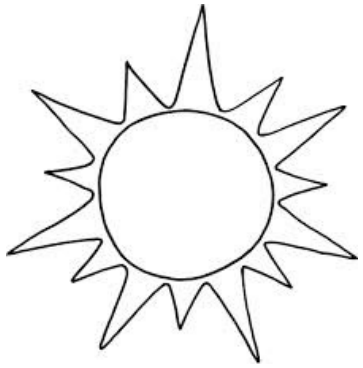
Singleton

Class

- Factory Method

Object

- Abstract Factory
- Builder
- Singleton
- Object Pool
- Prototype



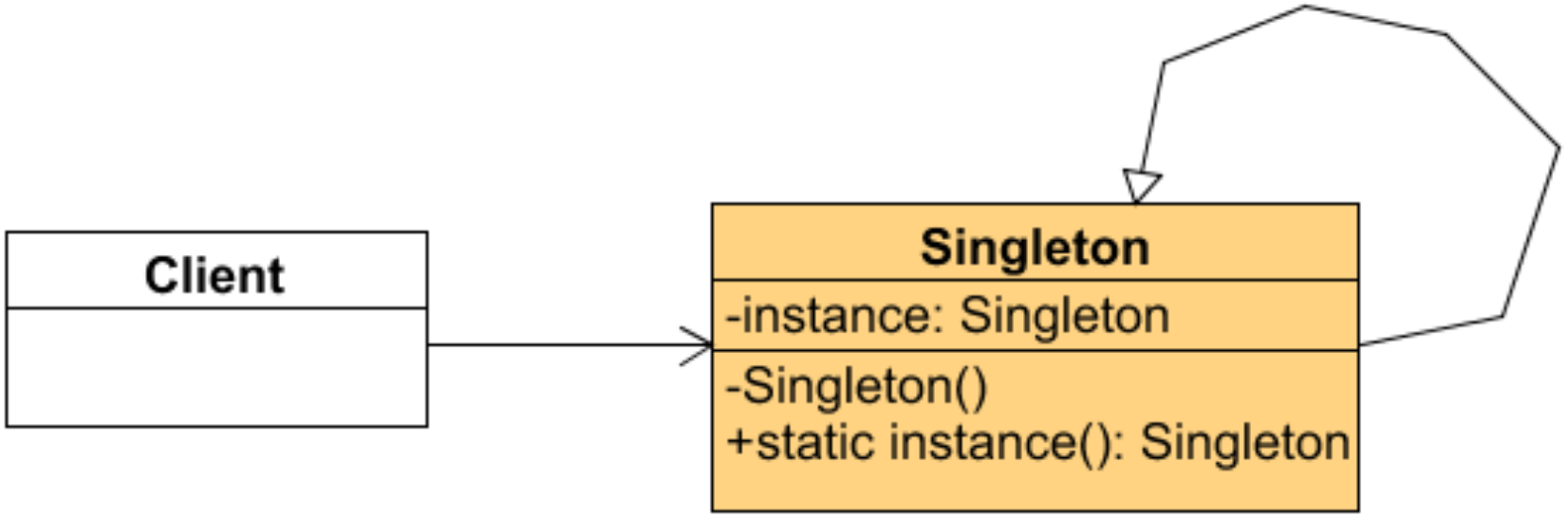
Motivation

- Intent
  - Ensure a class has only one instance, and provide a global point of access to it.
  - Encapsulated "just-in-time initialization" or "initialization on first use".
- Problem
  - Application needs one, and only one, instance of an object. Additionally, lazy initialization and global access are necessary.

Solution

- Define the constructor as private (or protected))
  - private Singleton(String name)
- Define a private static reference to the single class object
  - static private Singleton instance
- Define a acessor method to that instance
  - static public Singleton getInstance ()
  - Customers can access only the singleton object through this method

Structure


Client
Singleton
-instance: Singleton
-Singleton()
+static instance(): Singleton

Example

```
class Singleton {

    private String name;

    static private Singleton instance = new Singleton("Ermita");

    private Singleton(String name) {
        this.name = name;
    }

    static public Singleton getInstance() {
        return instance;
    }
    @Override
    public String toString() {
        return name;
    }
}
```

Example – lazy initialization

```
class LazySingleton {

    private String name;

    static private LazySingleton instance=null;

    private LazySingleton(String name) {
        this.name = name;
    }
    static public synchronized LazySingleton getInstance() {
        if (instance == null) {
            instance = new LazySingleton("Ermita");
        }
        return instance;
    }
    @Override
    public String toString() {
        return name;
    }
}
```

Check list

- Define a private static attribute in the "single instance" class.
- Define a public static accessor function in the class.
- Do "lazy initialization" (creation on first use) in the accessor function.
- Define all constructors to be protected or private.
- Clients may only use the accessor function to manipulate the Singleton.

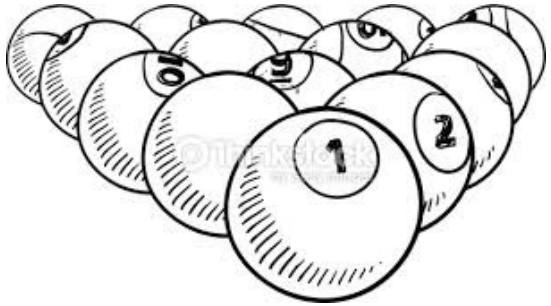
Object Pool

Class

- Factory Method

Object

- Abstract Factory
- Builder
- Singleton
- Object Pool
- Prototype



Motivation

- Intent
  - Object pooling can offer a significant performance boost; it is most effective in situations where:
    - the cost of initializing a class instance is high,
    - the rate of instantiation of a class is high, and
    - the number of instantiations in use at any one time is low.
- Problem
  - Object are used to manage the object caching. A client with access to a Object pool can avoid creating a new Object by simply asking the pool for one that has already been instantiated instead.
  - It is desirable to keep all Reusable objects that are not currently in use in the same object pool so that they can be managed by one coherent policy.

Solution

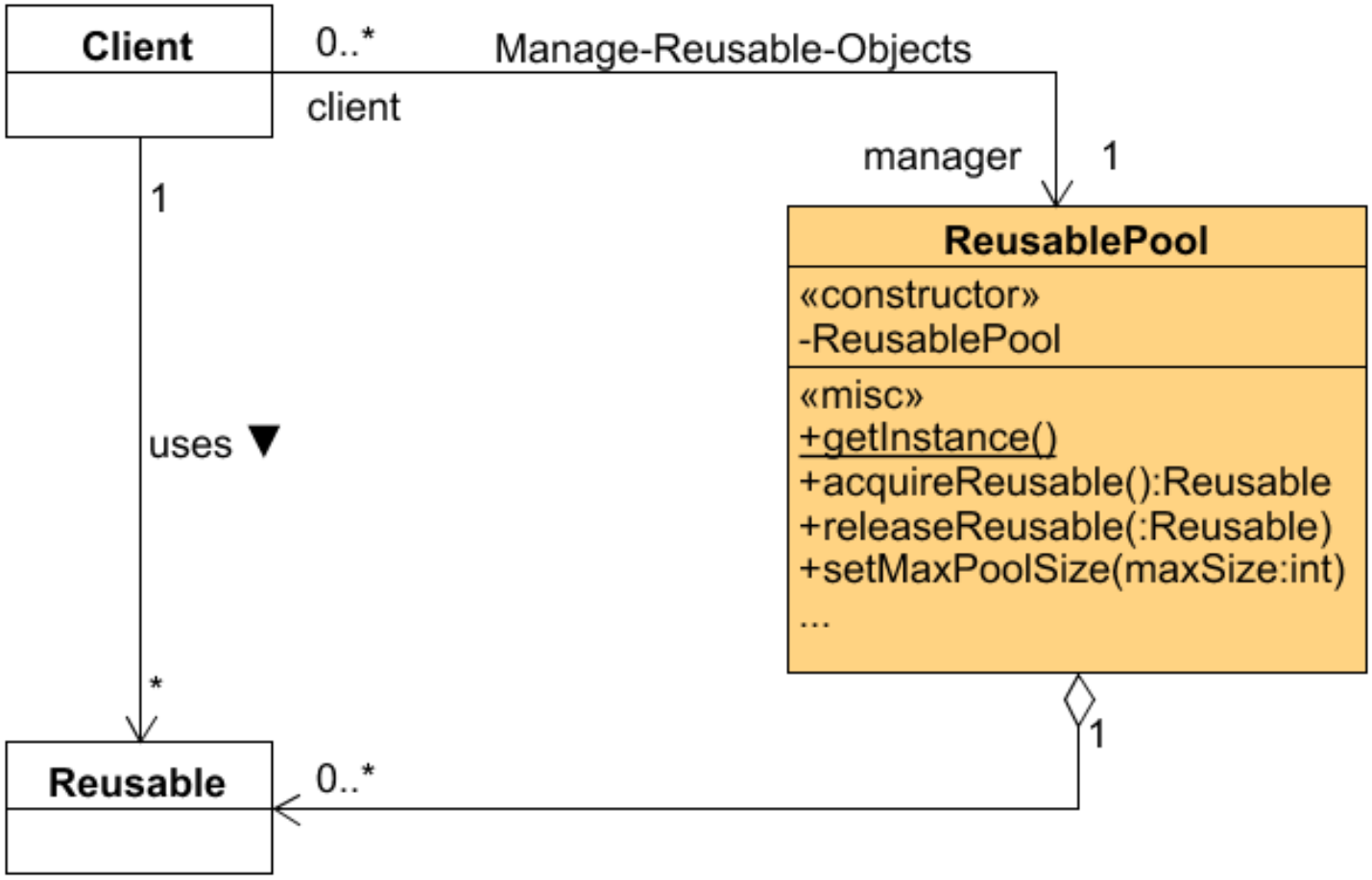
(1) redShoes = Shelf.acquireShoes();

(2) client.wear(redShoes);

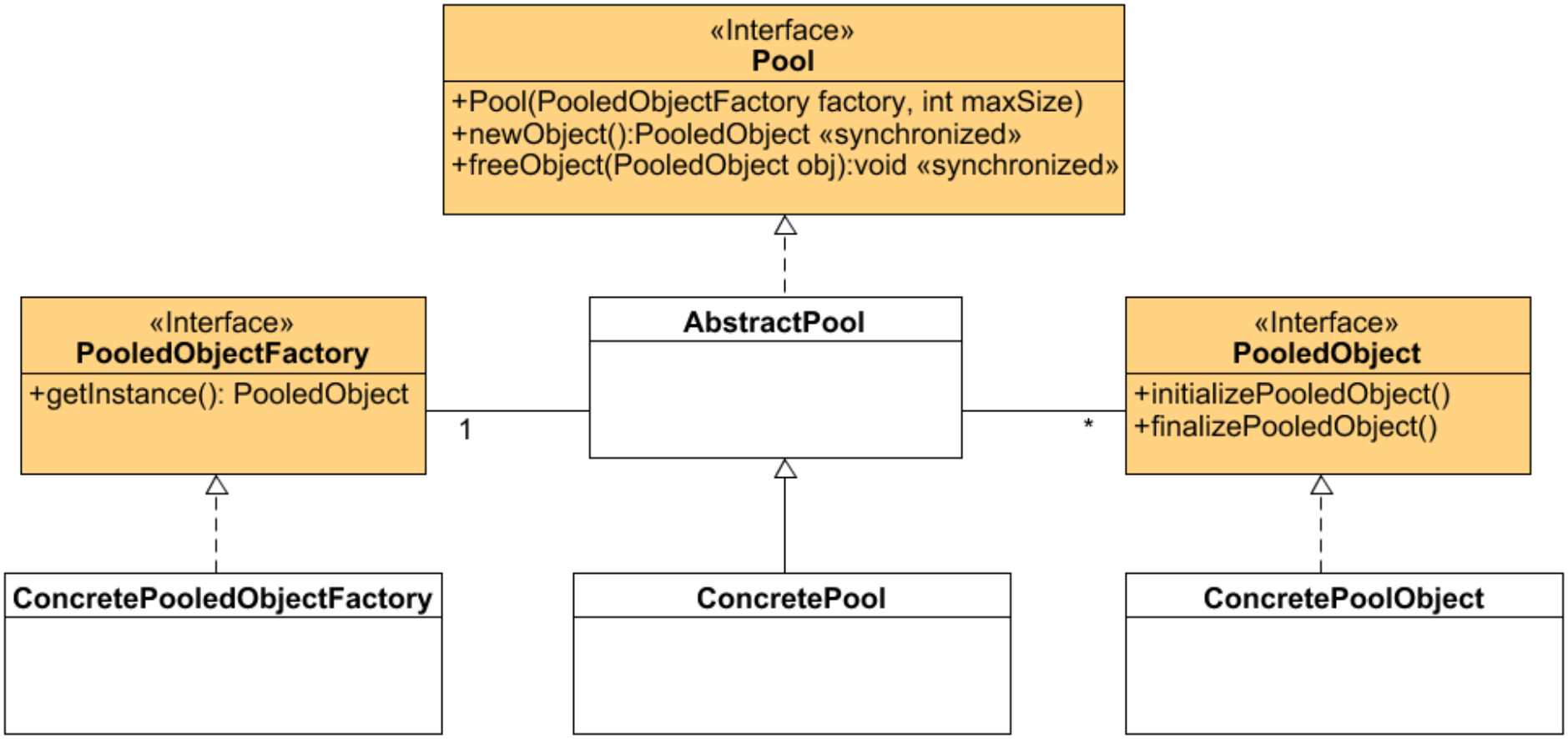
(4) Shelf.releaseShoes(redShoes);

(3) client.play();

Structure

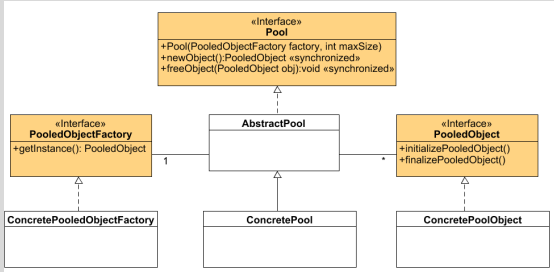

Client
0..*
Manage-Reusable-Objects
client
manager
1
1
ReusablePool
«constructor»
-ReusablePool
«misc»
+getInstance()
+acquireReusable():Reusable
+releaseReusable(:Reusable)
+setMaxPoolSize(maxSize:int)
...
uses
*
1
Reusable
0..*

A more complete Structure



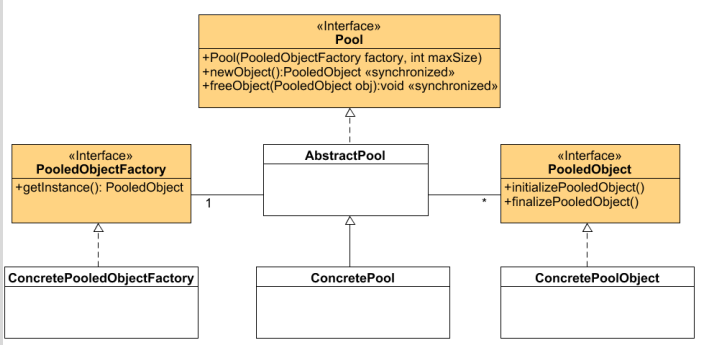
Example - *PooledObject*

```
/**
 * Interface that has to be implemented by an object that can be
 * stored in an object pool through the Pool class.
 * http://www.devahead.com
 */
public interface PooledObject
{
    /**
     * Initialization method. Called when an object is retrieved
     * from the object pool or has just been created.
     */
    public void initializePooledObject();

    /**
     * Finalization method. Called when an object is stored in
     * the object pool to mark it as free.
     */
    public void finalizePooledObject();
}
```



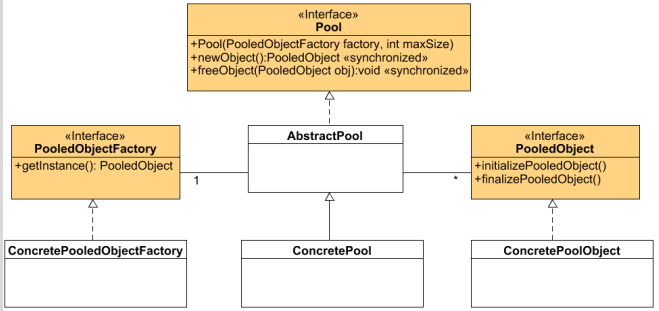
Example - *PooledObjectFactory*

```
/**
 * Interface that has to be implemented by every class that allows
 * the creation of objects for an object pool through the
 * Pool class.
*/
public interface PooledObjectFactory
{
        /**
         * Creates a new object for the object pool.
         *
         * @return new object instance for the object pool
         */
        public PooledObject getInstance();
}
```



Example - *AbstractPool*

```
public class AbstractPool implements Pool
{
    protected final int MAX_FREE_OBJECT_INDEX;

    protected PooledObjectFactory factory;
    protected PooledObject[] freeObjects;
    protected int freeObjectIndex = -1;

    /**
     * @param factory the object pool factory instance
     * @param maxSize the maximum number of instances stored in the pool
     */
    public AbstractPool(PooledObjectFactory factory, int maxSize)
    {
        this.factory = factory;
        this.freeObjects = new PooledObject[maxSize];
        MAX_FREE_OBJECT_INDEX = maxSize - 1;
    }
```



Example – *AbstractPool.newObject*

```
/**
 * Creates a new object or returns a free object from the pool.
 * @return a PooledObject instance already initialized
 */
public synchronized PooledObject newObject() {
    PooledObject obj = null;

    if (freeObjectIndex == -1) {
        // There are no free objects so I just
        // create a new object that is not in the pool.
        obj = factory.getInstance();
    } else {
        // Get an object from the pool
        obj = freeObjects[freeObjectIndex];
        freeObjectIndex--;
    }
    obj.initializePooledObject();
    return obj;
}
```

Example - *AbstractPool.freeObject*

```
    /**
     * Stores an object instance in the pool to make it available for a subsequent
     * call to newObject() (the object is considered free).
     * @param obj the object to store in the pool and that will be finalized
     */
    public synchronized void freeObject(PooledObject obj)
    {
        if (obj != null)   {
            // Finalize the object
            obj.finalizePooledObject();
            // put an object in the pool only if there is still room for it
            if (freeObjectIndex < MAX_FREE_OBJECT_INDEX) {
                freeObjectIndex++;
                // Put the object in the pool
                freeObjects[freeObjectIndex] = obj;
            }
        }
    }
}
```

Check list

- Create the *Pool* class with a collection of *PooledObjects*
- Create *acquire* and *release* methods in *Pool* class

Important remarks

- The creation and destruction of short lived objects (i.e. memory allocation and GC) is more efficient in modern JVMs.
- Object Pool must only be used for special objects whose creation is relatively costly, like DB / network connections, threads etc.

Prototype

Class

- Factory Method

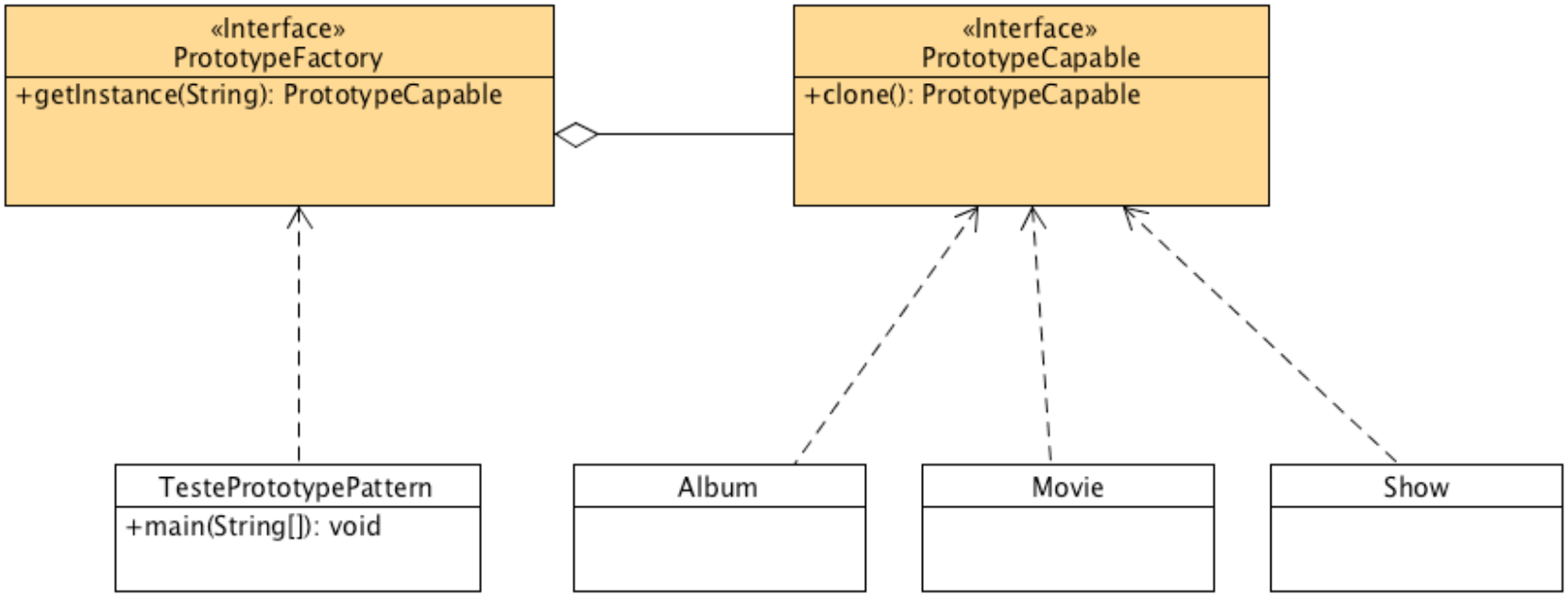
Object

- Abstract Factory
- Builder
- Singleton
- Object Pool
- Prototype

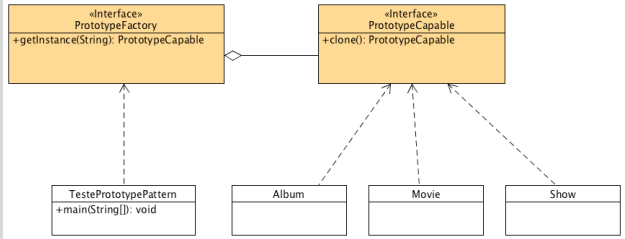
Motivation

- Intent
  - Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
  - Co-opt one instance of a class for use as a breeder of all future instances.
  - The new operator considered harmful.
- Problem
  - Application "hard wires" the class of object to create, in each "new" expression.

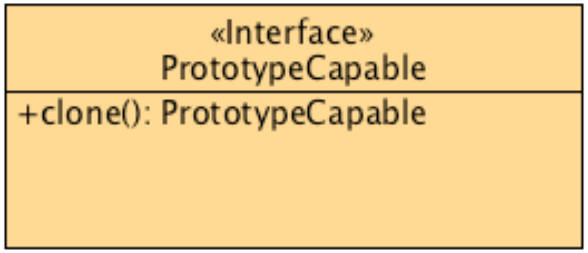
Structure


«Interface»
PrototypeFactory
+getInstance(String): PrototypeCapable
«Interface»
PrototypeCapable
+clone(): PrototypeCapable
TestePrototypePattern
+main(String[]): void
Album
Movie
Show

Example – *the contract*

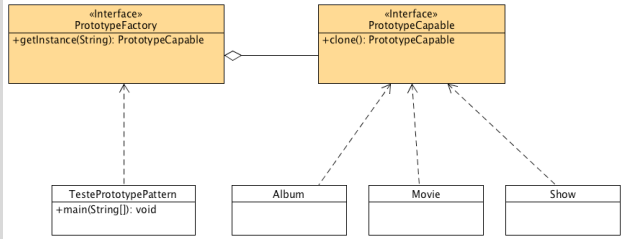



```
public interface PrototypeCapable extends Cloneable
{
    public PrototypeCapable clone() throws CloneNotSupportedException;
}
```



Example – *the model*

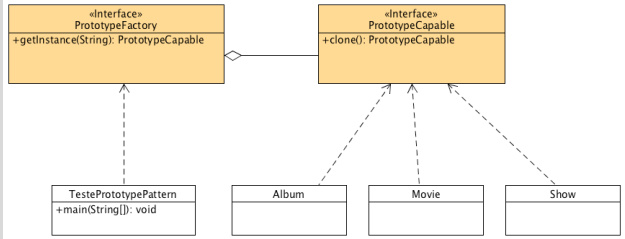
```
public class Album implements PrototypeCapable
{
    private String name = null;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    @Override
    public Album clone() throws CloneNotSupportedException {
        System.out.println("Cloning Album object..");
        return (Album) super.clone();
    }
    @Override
    public String toString() {
        return "Album";
    }
}
```



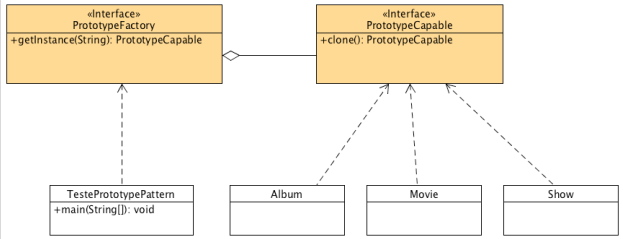
the same for Movie, Show, ..

Example – *the factory*

```
public class PrototypeFactory  {
    public static enum ModelType   {
        MOVIE, ALBUM, SHOW;
    }

    private static Map<ModelType, PrototypeCapable> prototypes =
        new HashMap<>();
    static {
        prototypes.put(ModelType.MOVIE, new Movie());
        prototypes.put(ModelType.ALBUM, new Album());
        prototypes.put(ModelType.SHOW, new Show());
    }

    public static PrototypeCapable getInstance(ModelType s)
                  throws CloneNotSupportedException {
        return (prototypes.get(s)).clone();
    }
}
```



Example – *the client*




```
public class TestPrototypePattern {
    public static void main(String[] args)  {
        try {

            PrototypeCapable proto;
            proto = PrototypeFactory.getInstance(ModelType.MOVIE);
            System.out.println(proto);

            proto = PrototypeFactory.getInstance(ModelType.ALBUM);
            System.out.println(albumPrototype);

            proto = PrototypeFactory.getInstance(ModelType.SHOW);
            System.out.println(proto);
        }
        catch (CloneNotSupportedException e)
            e.printStackTrace();
        }
    }
}
```

```
Cloning Movie object..
Movie
Cloning Album object..
Album
Cloning Show object..
Show
```

Check list

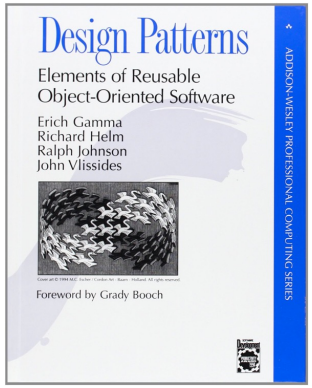
- Add a clone() method to the existing "product" hierarchy.
- Design a "registry" that maintains a cache of prototypical objects. The registry could be encapsulated in a new Factory class, or in the base class of the "product" hierarchy.
- Design a factory method that: may (or may not) accept arguments, finds the correct prototype object, calls clone() on that object, and returns the result.
- The client replaces all references to the new operator with calls to the factory method.

Creational patterns – Summary

- Abstract Factory
  - Creates an instance of several families of classes
- Builder
  - Separates object construction from its representation
- Factory Method
  - Creates an instance of several derived classes
- Singleton
  - A class of which only a single instance can exist
- Object Pool
  - Avoid expensive acquisition and release of resources by recycling objects that are no longer in use
- Prototype
  - A fully initialized instance to be copied or cloned

Resources

- Design Patterns – Elements of Reusable Object-Oriented Software; Gamma, et. al.



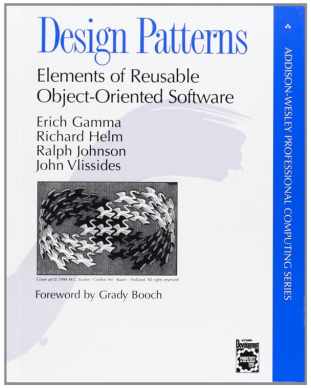
- *Design Patterns Explained Simply (sourcemaking.com)*

Design Patterns – Structural

UA.DETI.PDS
José Luis Oliveira

Resources

- Design Patterns – Elements of Reusable Object-Oriented Software; Gamma, et. al.



- *Design Patterns Explained Simply (sourcemaking.com)*

Structural patterns

- They simplify software design by identifying a simple way to build relationships between entities.

Structural design patterns

Class

- Adapter

Object

- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

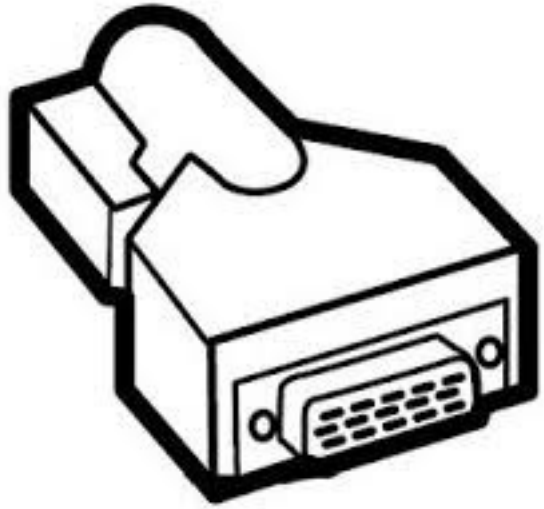
Class Adapter

Class

- Adapter

Object

- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy



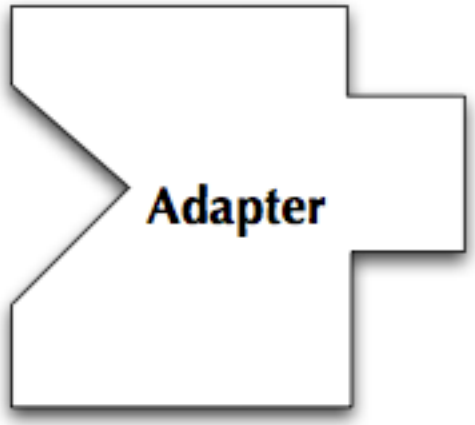
Software Adapters (I)

- Pre-condition
  - You are maintaining an existing system that makes use of a third-party class library from vendor A
- Stimulus
  - Vendor A goes belly up and corporate policy does not allow you to make use of an unsupported class library
- Response
  - Vendor B provides a similar class library but its interface is completely different from the interface provided by vendor A
- Assumptions
  - You don't want to change your code, and you can't change vendor B's code
- Solution?:
  - Write new code that adapts vendor B's interface to the interface expected by your original code

Software Adapters (II)

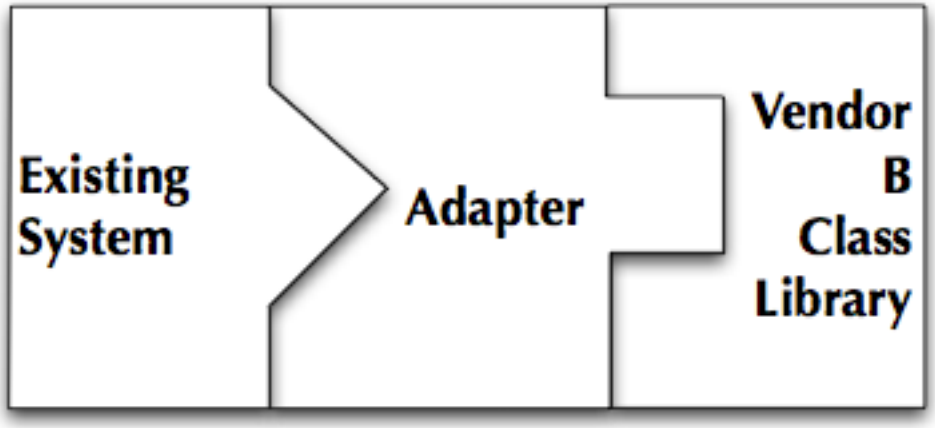
Interface Mismatch
Need Adapter

Create Adapter



And then...

Software Adapters (III)



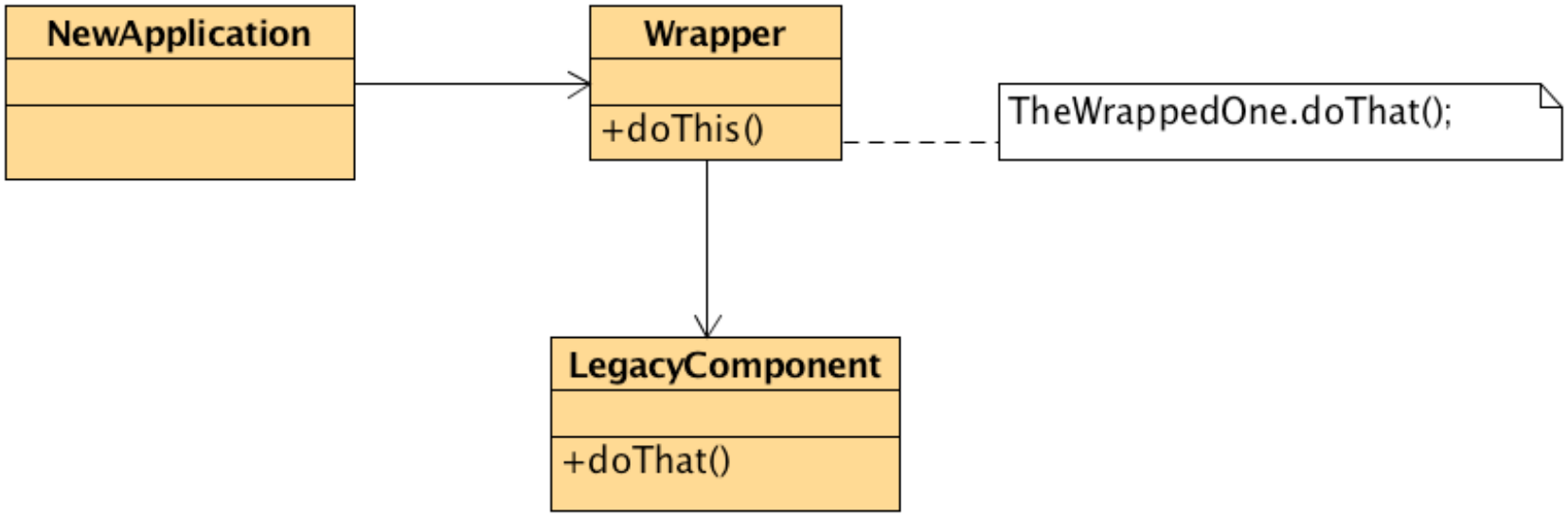
- …plug it in
- Benefit: Existing system and new vendor library do not change - new code is isolated within the adapter

Motivation

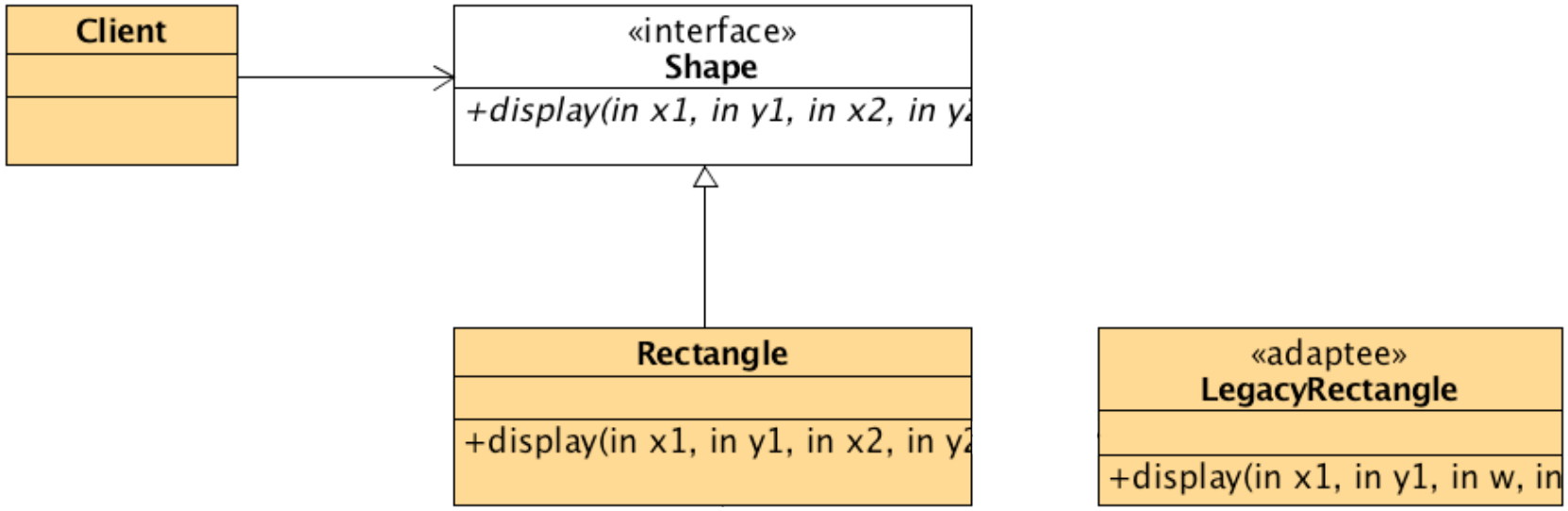
- **Intent**
  - Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
  - Wrap an existing class with a new interface.

- **Problem**
  - An "off the shelf" component offers compelling functionality that you would like to reuse, but its "view of the world" is not compatible with the philosophy and architecture of the system currently being developed.

Solution


NewApplication
Wrapper
+doThis()
TheWrappedOne.doThat();
LegacyComponent
+doThat()

Problem – Solution?



Example – the problem

```
interface Shape {
  void draw(int x1, int y1, int x2, int y2);
}

class Rectangle implements Shape {
    public void draw(int x1, int y1, int x2, int y2) {
        System.out.println("rectangle from (" + x1 + ',' + y1 + ") to (" +
x2
                + ',' + y2 + ')');
    }
}

class LegacyRectangle {
    public void draw(int x, int y, int w, int h) {
        System.out.println("old format rectangle at (" + x + ',' + y
                + ") with width " + w + " and height " + h);
    }
}
```

Example – the problem

```
public class NoAdapterDemo {
    public static void main(String[] args) {
        Object[] shapes = { new Rectangle(), new LegacyRectangle() };
        // A begin and end point from a graphical editor
        int x1 = 10, y1 = 20;
        int x2 = 30, y2 = 60;
        for (int i = 0; i < shapes.length; ++i)
          if (shapes[i].getClass().getSimpleName().equals("Rectangle"))
              ((Rectangle) shapes[i]).draw(x1, y1, x2, y2);
          else if (shapes[i].getClass().getSimpleName()
                .equals("LegacyRectangle"))
              ((LegacyRectangle) shapes[i]).draw(Math.min(x1, x2),
                  Math.min(y1, y2), Math.abs(x2 - x1), Math.abs(y2 - y1));
    }
}
```

rectangle from (10,20) to (30,60)
old format rectangle at (10,20) with width 20 and height 40

Example – the Adapter solution

```
class OldRectangle implements Shape {
    private LegacyRectangle adaptee = new LegacyRectangle();

    public void draw(int x1, int y1, int x2, int y2) {
        adaptee.draw(Math.min(x1, x2), Math.min(y1, y2), Math.abs(x2 - x1),
                Math.abs(y2 - y1));
    }
}

public class AdapterDemo2 {
    public static void main(String[] args) {
        Shape[] shapes = { new Rectangle(), new OldRectangle() };
        // A begin and end point from a graphical editor
        int x1 = 10, y1 = 20;
        int x2 = 30, y2 = 60;
        for (int i = 0; i < shapes.length; ++i)
           shapes[i].draw(x1, y1, x2, y2);
    }
}
```

Another example

```
interface Rectangle {
  void scale(int factor); //grow or shrink by factor
  void setWidth();
  float getWidth();
  float area(); …
}

class Client {
  void clientMethod(Rectangle r) {
    // ...
    r.scale(2);
  }
}

class NonScalableRectangle {
  void setWidth(); …
  // no scale method!
}
```

How to use this rectangle in Client?

Another example: via subclassing

- Class adapter adapts via subclassing

```
class ScalableRectangle1
            extends NonScalableRectangle
            implements Rectangle {

   void scale(int factor) {
     setWidth(factor*getWidth());
     setHeight(factor*getHeight());
   }

}
```

Another example: via delegation

- Object adapter adapts via delegation:
  - it forwards work to delegate

```
class ScalableRectangle2 implements Rectangle {
   NonScalableRectangle r; // delegate
   ScalableRectangle2(NonScalableRectangle r) {
     this.r = r;
   }

   void scale(int factor) {
     setWidth(factor * r.getWidth());
     setHeight(factor * r.getHeight());
   }

   float getWidth() { return r.getWidth(); }
   // ...
}
```

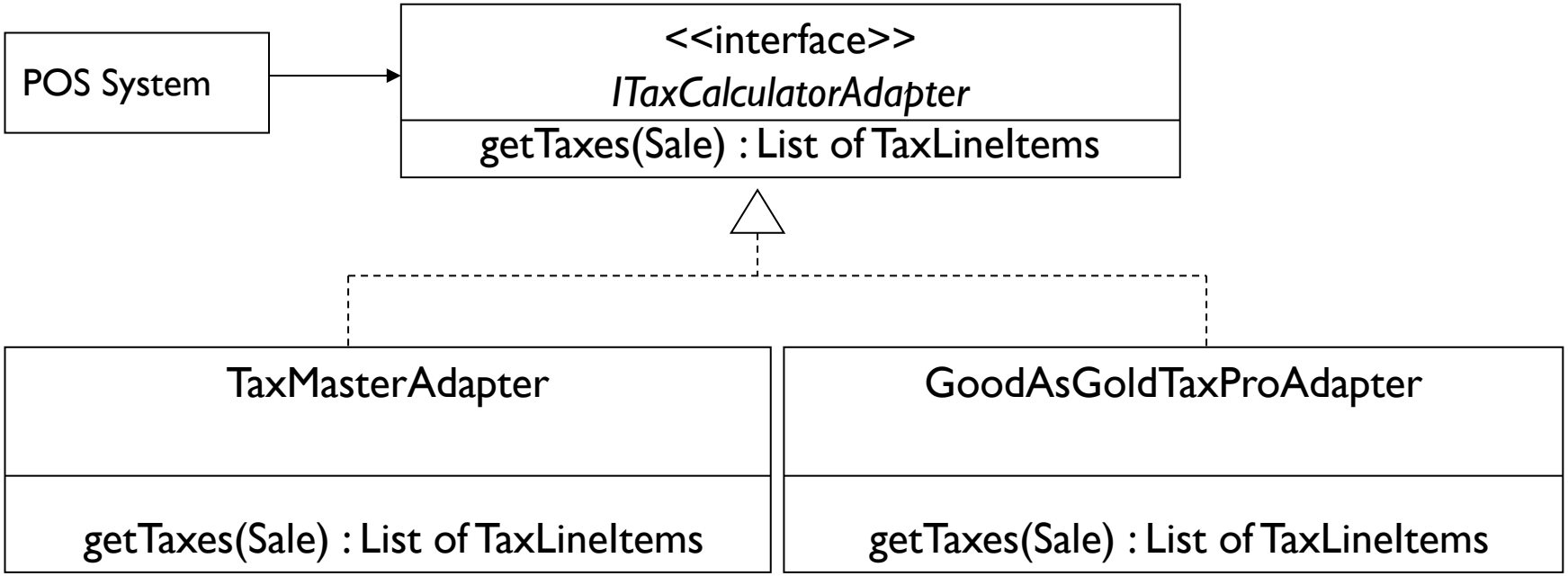
Subclassing versus delegation

- Subclassing
  - Automatically gives access to all methods in the superclass
  - More efficient
- Delegation
  - Permits removal of methods
  - Wrappers can be added and removed dynamically
  - Multiple objects can be composed
  - Bottom line: more flexible

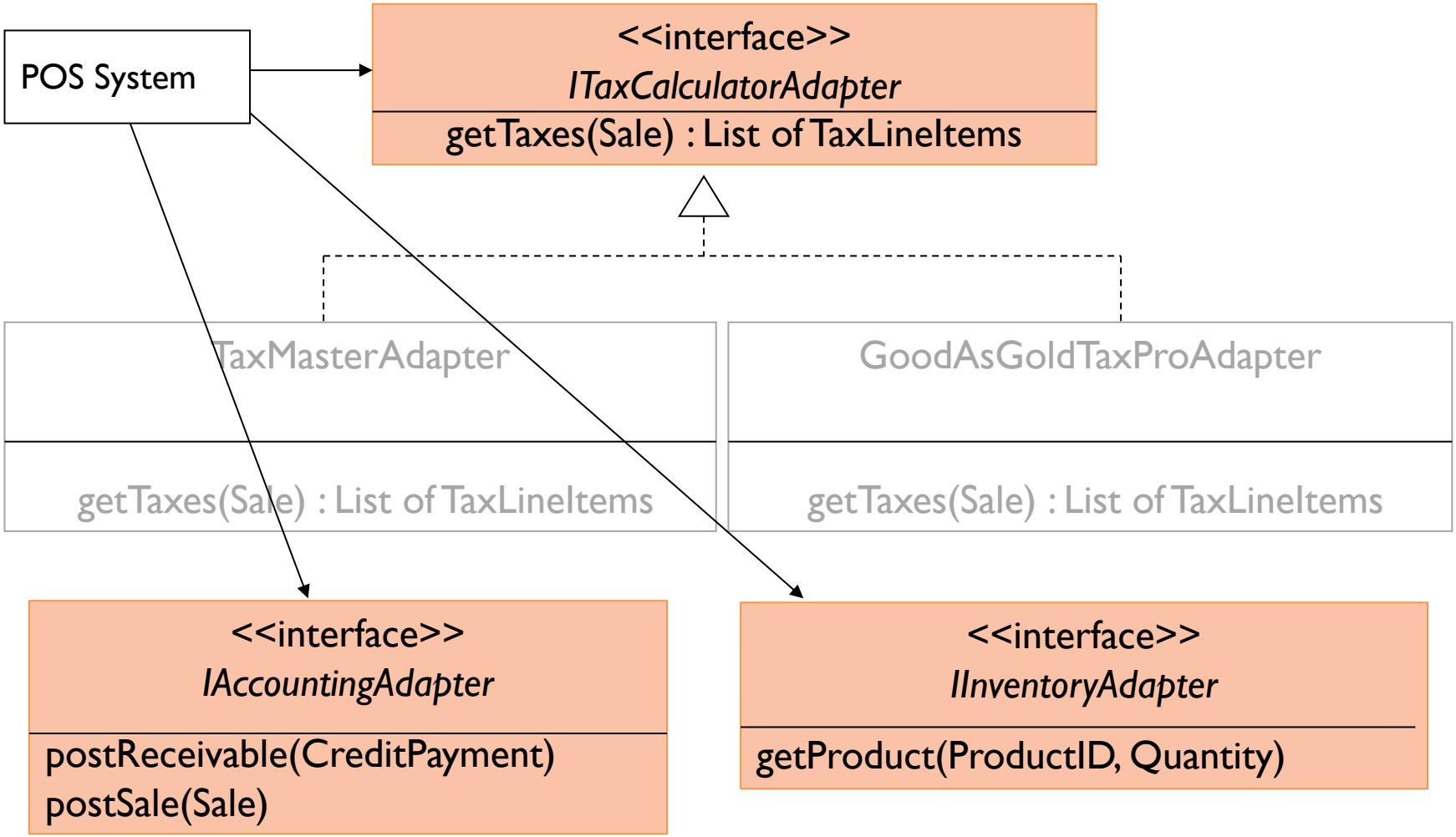
Exercise

- A Point-of-Sale system needs to support services from different third-party vendors:
  - Tax calculator service from different vendors
  - Credit authorization service from different vendors
  - Inventory systems from different vendors
  - Accounting systems from different vendors
- Each vendor service has its own API, which can't be changed
- What design pattern solves this problem?

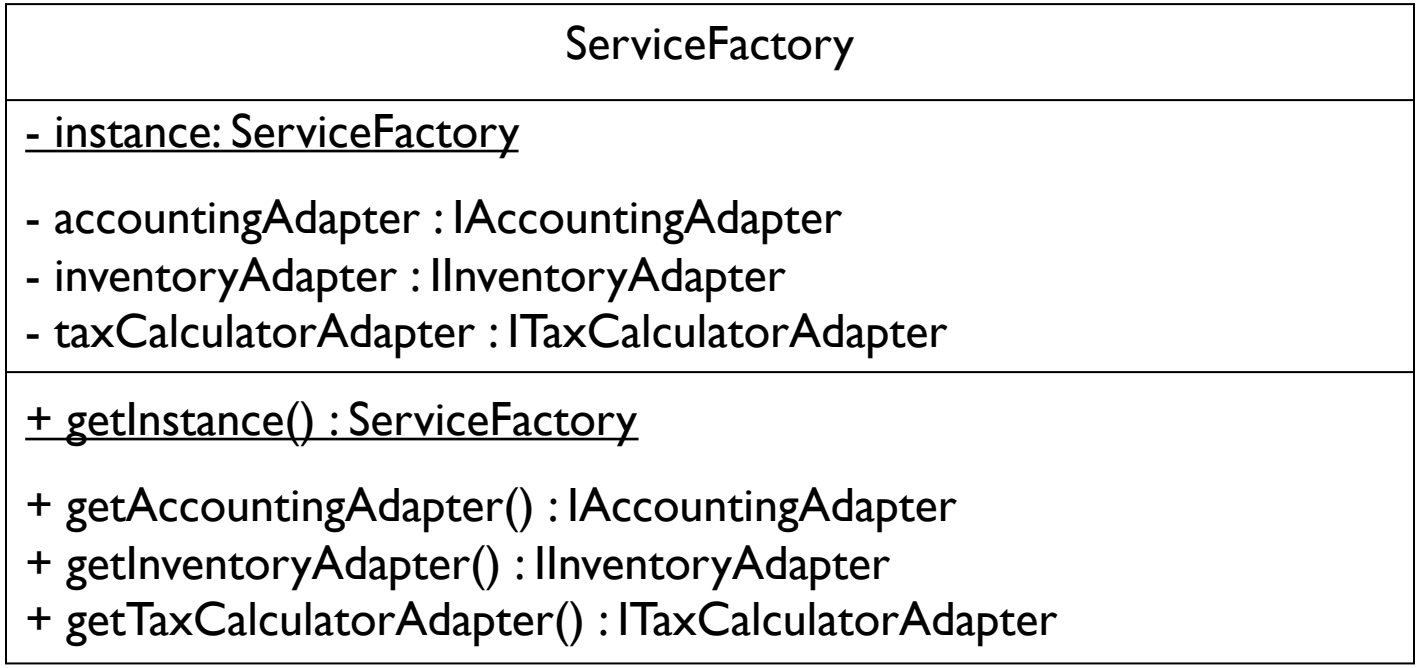
The Solution: Object Adapter



Extending the problem/solution



The Solution



- Single instance of ServiceFactory ensures single instance of adapter objects.
  - underline means static. *instance* and *getInstance* are static.

Check list

- Decide if "platform independence" and creation services are the current source of pain.
- Map out a matrix of "platforms" versus "products".
- Define a factory interface that consists of a factory method per product.
- Define a factory derived class for each platform that encapsulates all references to the new operator.
- The client should retire all references to new, and use the factory methods to create the product objects.

Structural design patterns

Class

- Adapter

Object

- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

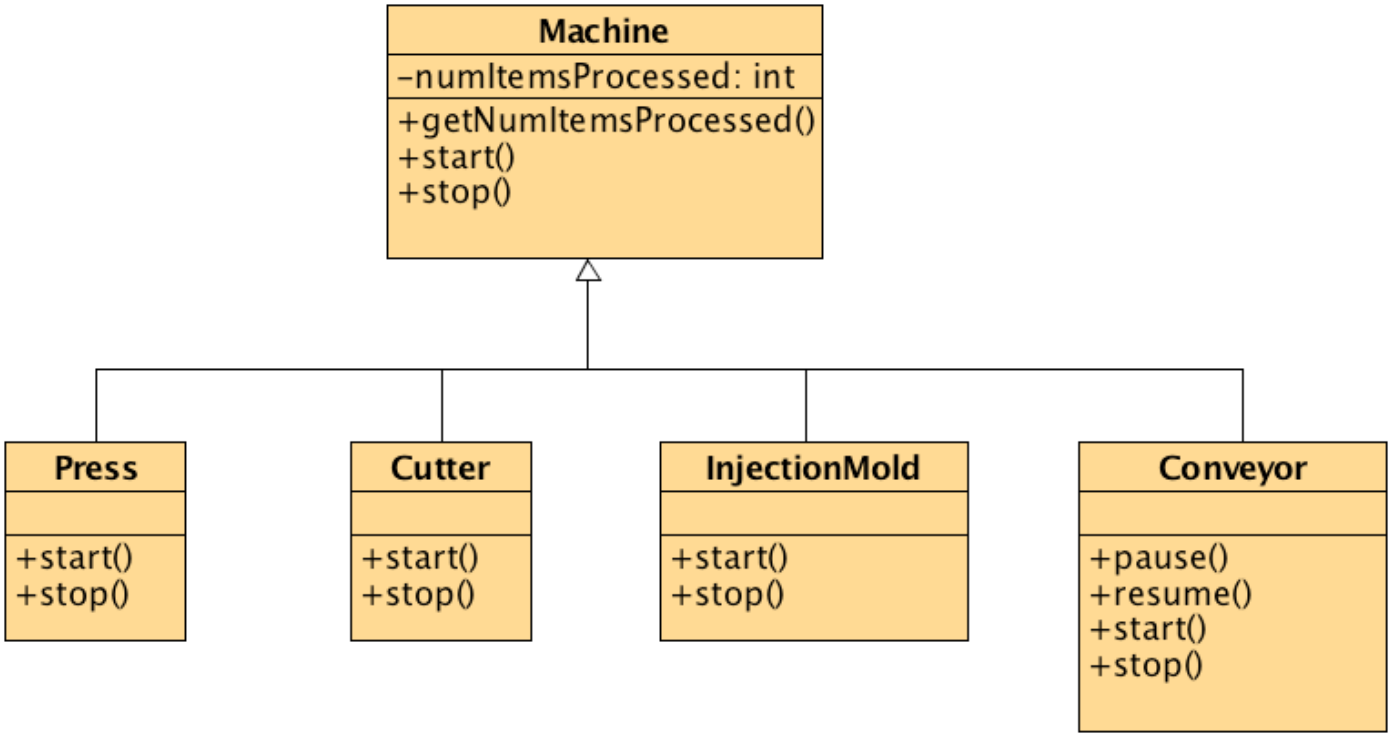
Motivation

- **Intent**
  - Decouple an abstraction from its implementation so that the two can vary independently.
  - Publish interface in an inheritance hierarchy, and bury implementation in its own inheritance hierarchy.

- **Problem**
  - "Hardening of the software arteries" has occurred by using subclassing of an abstract base class to provide alternative implementations. This locks in compile-time binding between interface and implementation. The abstraction and implementation cannot be independently extended or composed.

Bridge

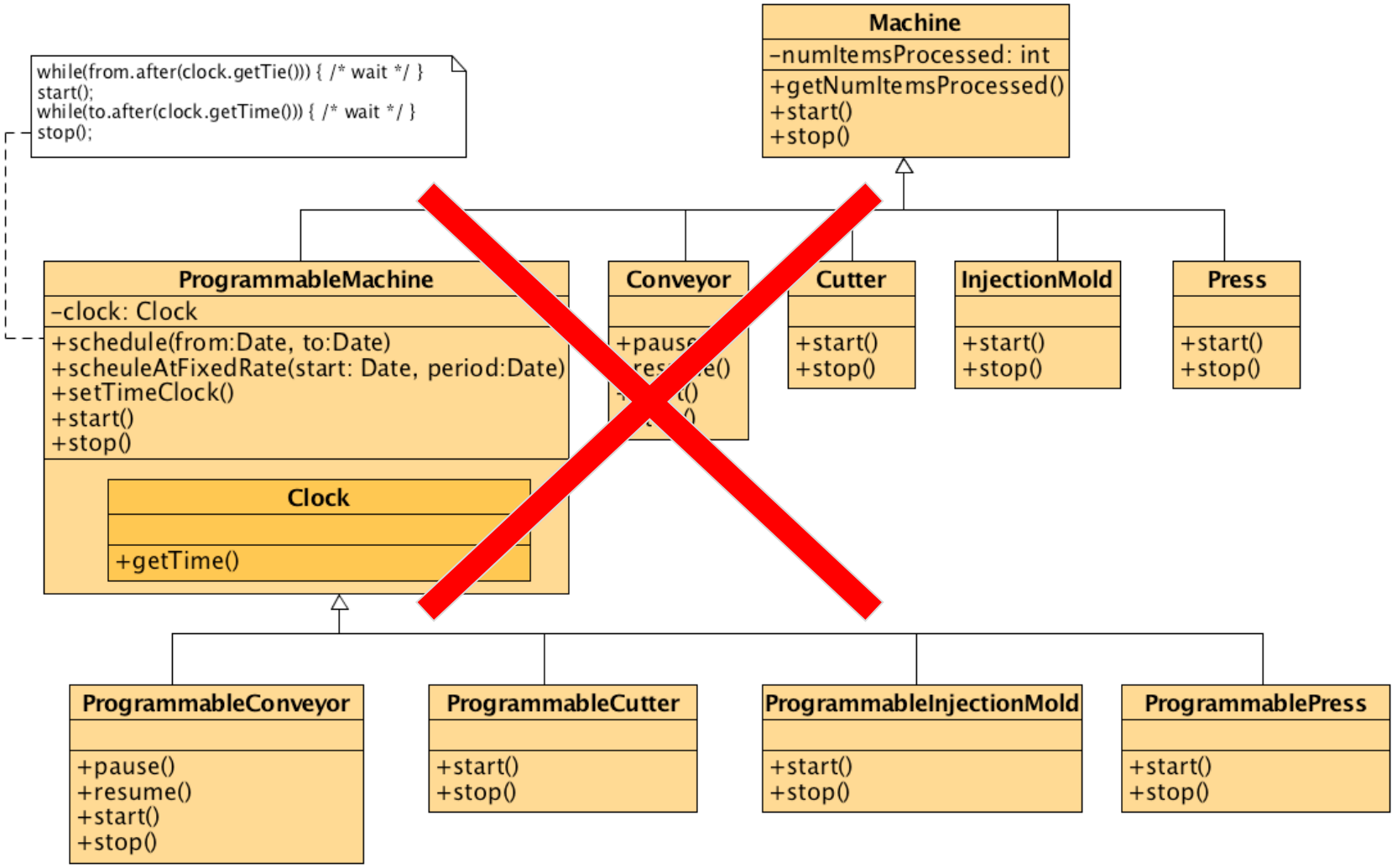
- An abstraction, *Machine*, has one of several possible implementations, which may override or extend *start()* and *stop()*



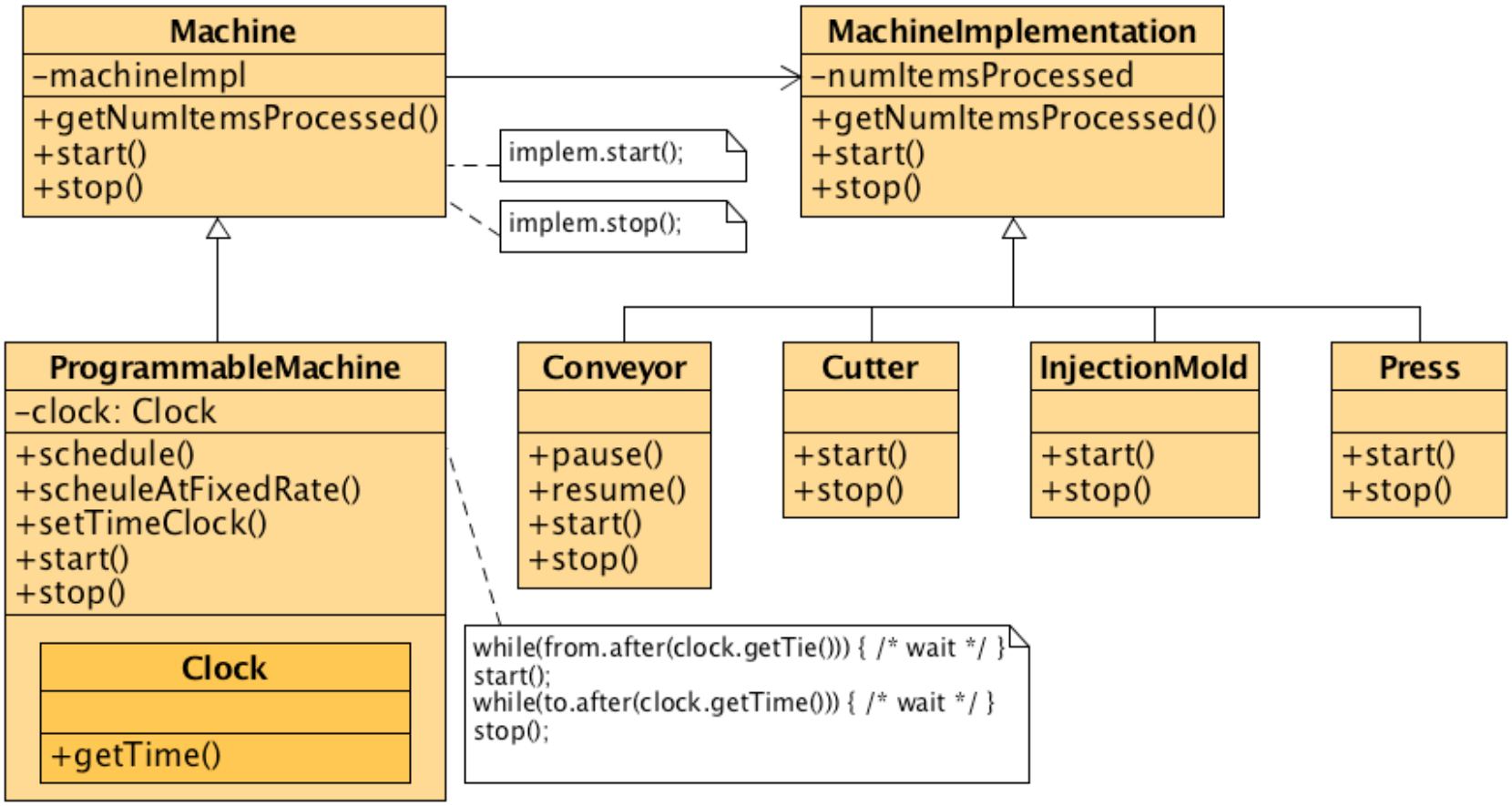
Bridge – the problem

- Later on, machines are bought which can be programmed to start and stop at given times, and even to do it periodically
  - We are forced to add many new classes: every combination of {non-programmable, programmable} × {press, cutter, injection molding, conveyor belt} because start/stop are different and may have or not schedule() capability
- Inheritance binds an implementation to the abstraction permanently
  - makes it difficult to modify, extend, and reuse abstractions and implementations independently

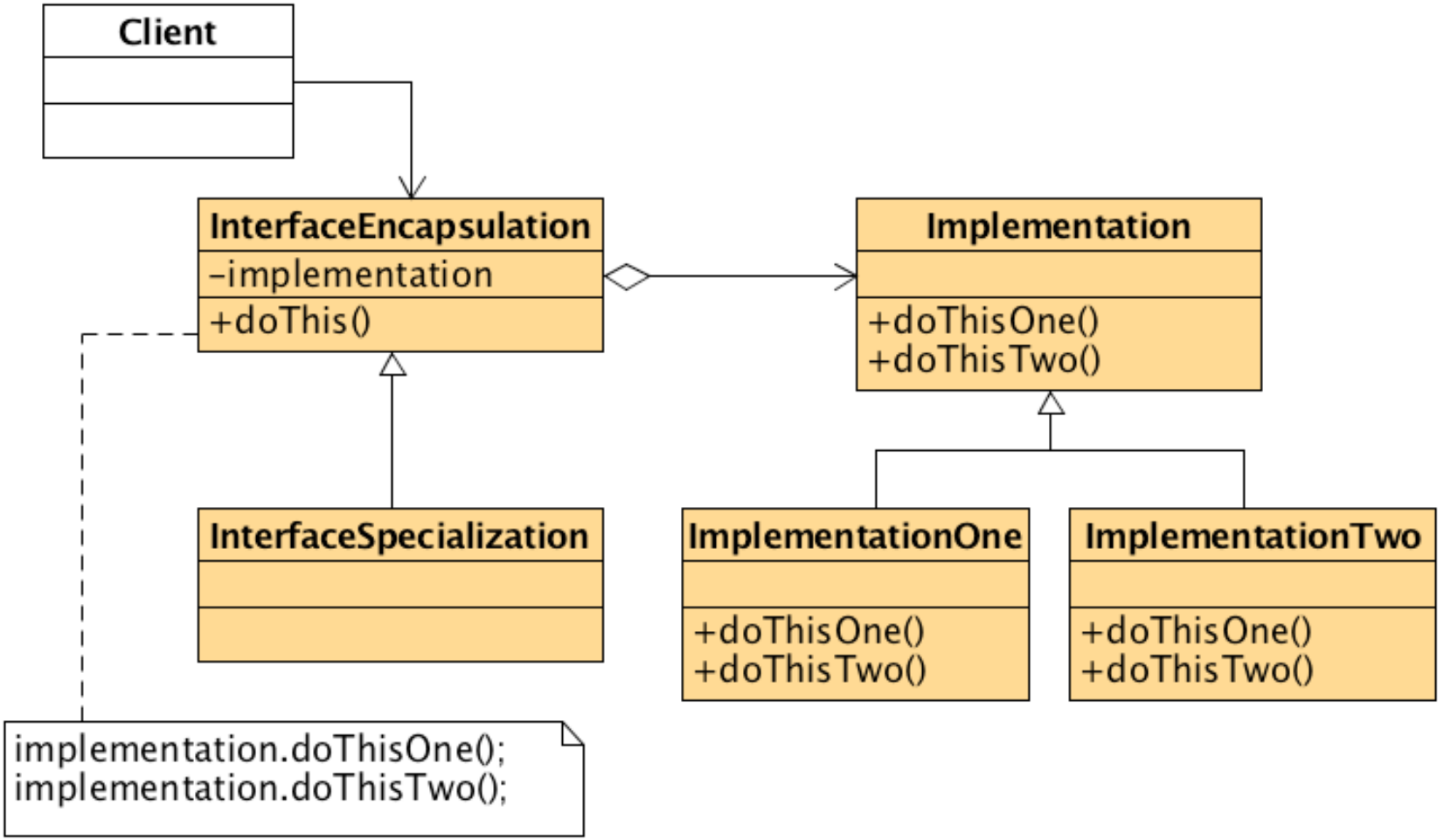
Bridge – the solution?



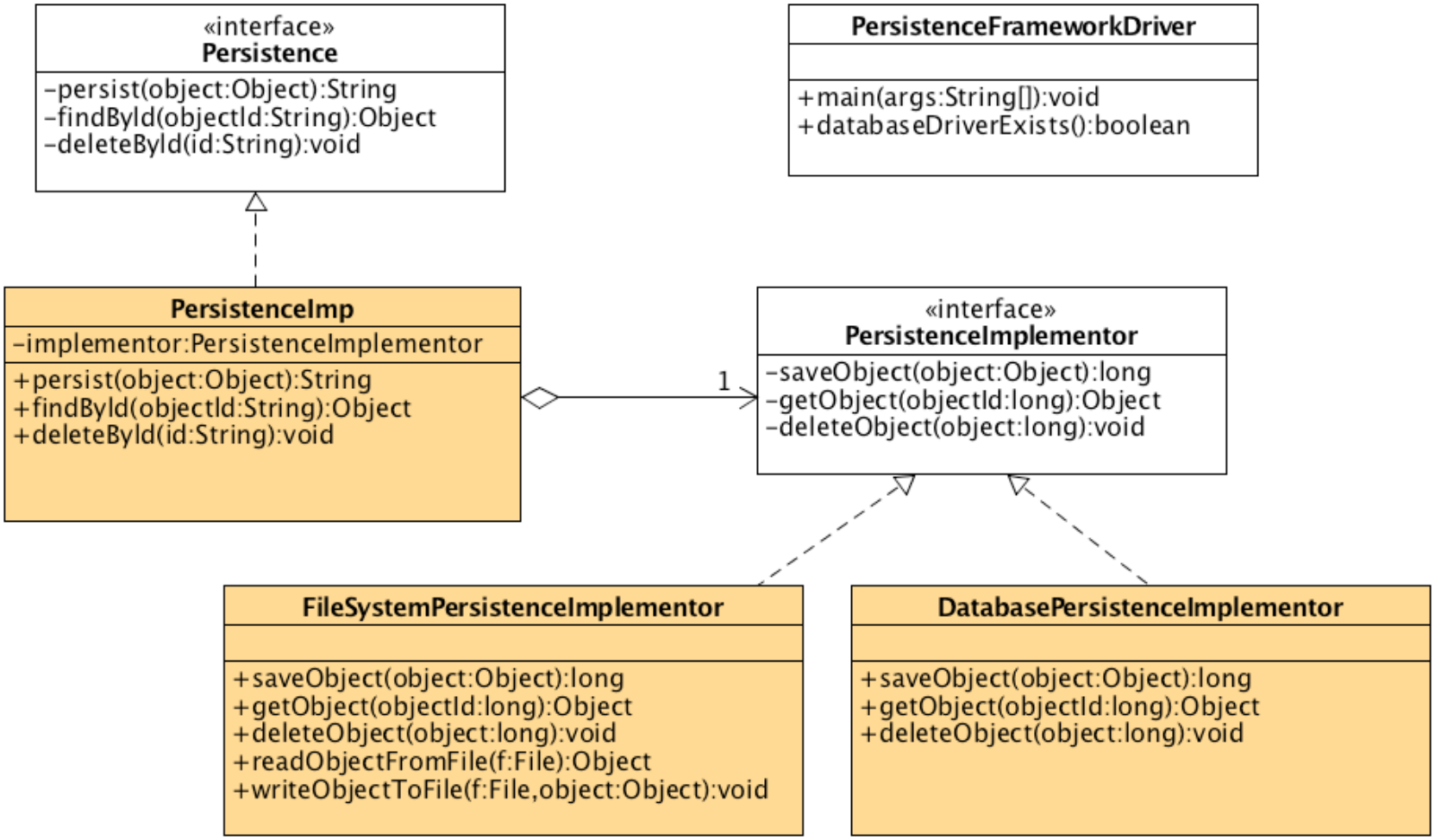
Bridge – the solution



Structure


Client
InterfaceEncapsulation
-implementation
+doThis()
Implementation
+doThisOne()
+doThisTwo()
InterfaceSpecialization
ImplementationOne
+doThisOne()
+doThisTwo()
ImplementationTwo
+doThisOne()
+doThisTwo()
implementation.doThisOne();
implementation.doThisTwo();

Example (1)


«interface»
Persistence
-persist(object:Object):String
-findById(objectId:String):Object
-deleteById(id:String):void
PersistenceFrameworkDriver
+main(args:String[]):void
+databaseDriverExists():boolean
PersistenceImp
-implementor:PersistenceImplementor
+persist(object:Object):String
+findById(objectId:String):Object
+deleteById(id:String):void
1
«interface»
PersistenceImplementor
-saveObject(object:Object):long
-getObject(objectId:long):Object
-deleteObject(object:long):void
FileSystemPersistenceImplementor
+saveObject(object:Object):long
+getObject(objectId:long):Object
+deleteObject(object:long):void
+readObjectFromFile(f:File):Object
+writeObjectToFile(f:File,object:Object):void
DatabasePersistenceImplementor
+saveObject(object:Object):long
+getObject(objectId:long):Object
+deleteObject(object:long):void

Example (1) – Client

```
public class PersistenceFrameworkDriver {
    public static void main(String[] args) {

        PersistenceImplementor implementor = null;
        if(databaseDriverExists()) {
            implementor = new DabatasePersistenceImplementor();
        } else {
            implementor = new FileSystemPersistenceImplementor();
        }
        Persistence persistenceAPI = new PersistenceImp(implementor);
        Object o = persistenceAPI.findById("12343755");
        // do changes to the object ... then persist
        persistenceAPI.persist(o);
        // can also change implementor
        persistenceAPI = new PersistenceImp(
                new DabatasePersistenceImplementor());
        persistenceAPI.deleteById("2323");
    }
}
```

Example (2) – Client

```
public class BridgeDemo {

    public static void main(String[] args) {

        Vehicle vehicle = new BigBus(new SmallEngine());
        vehicle.drive();
        vehicle.setEngine(new BigEngine());
        vehicle.drive();

        vehicle = new SmallCar(new SmallEngine());
        vehicle.drive();
        vehicle.setEngine(new BigEngine());
        vehicle.drive();

    }

}
```

Vehicle and Engine can evolve independently!
How to model this?

Check list

- Decide if **two orthogonal dimensions** exist in the domain (e.g. abstraction/platform, or domain/infrastructure, or front-end/back-end, or interface/implementation).
- Design the **separation of concerns**: what does the client want, and what do the platforms provide.
- Design a platform-oriented interface that is minimal, necessary, and sufficient.
- Define a derived class of that interface for each platform.
- Create the **abstraction base class** that "has a" platform object and delegates the platform-oriented functionality to it.
- Define **specializations** of the abstraction class if desired.

Structural design patterns

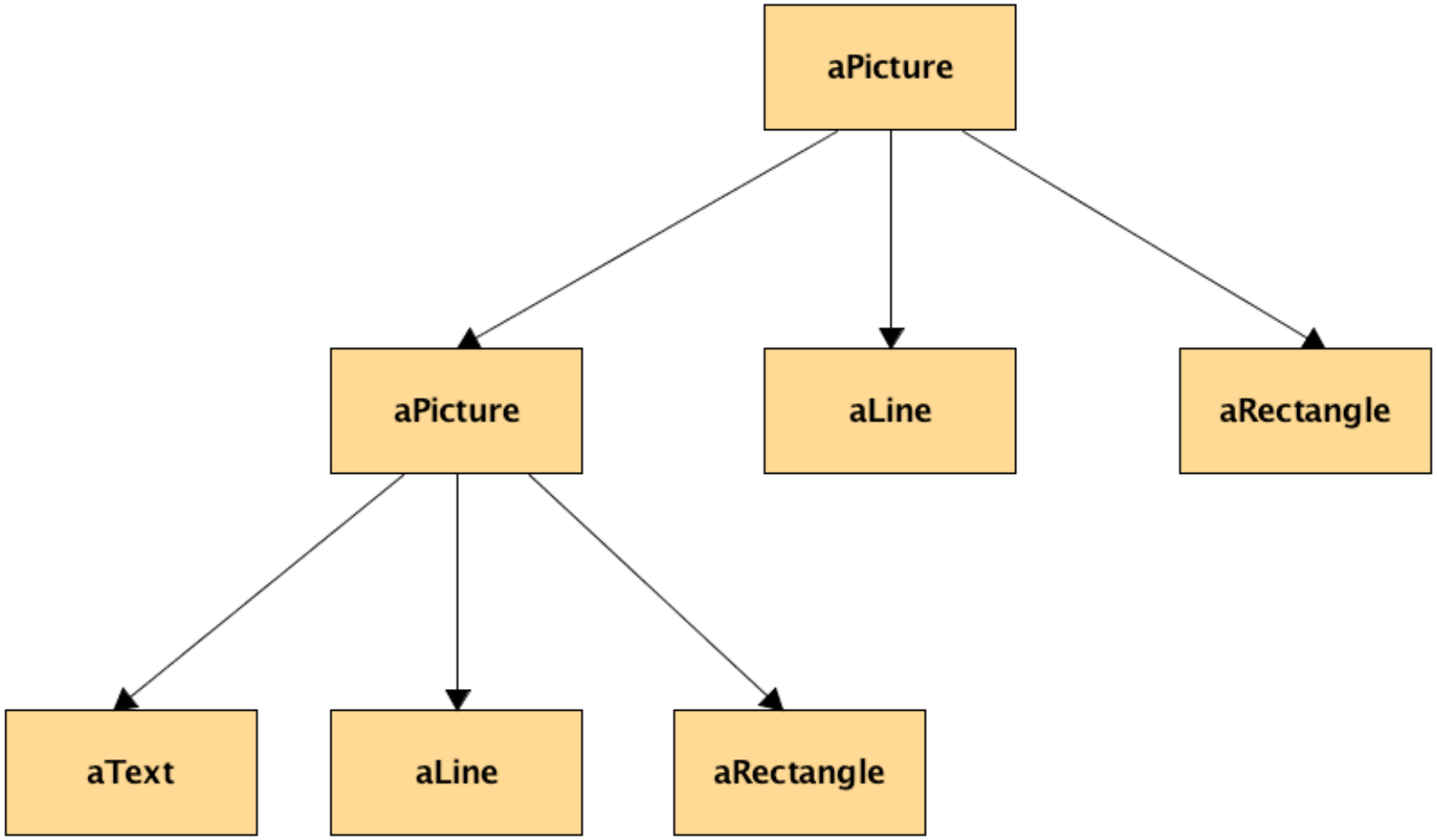
Class

- Adapter

Object

- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

Motivation

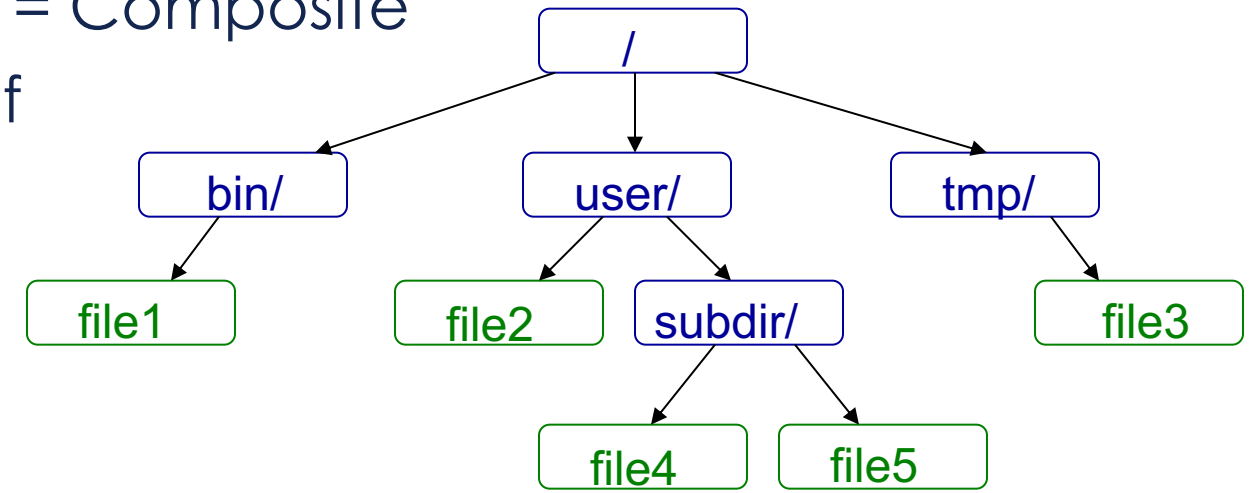

aPicture
aPicture
aLine
aRectangle
aText
aLine
aRectangle

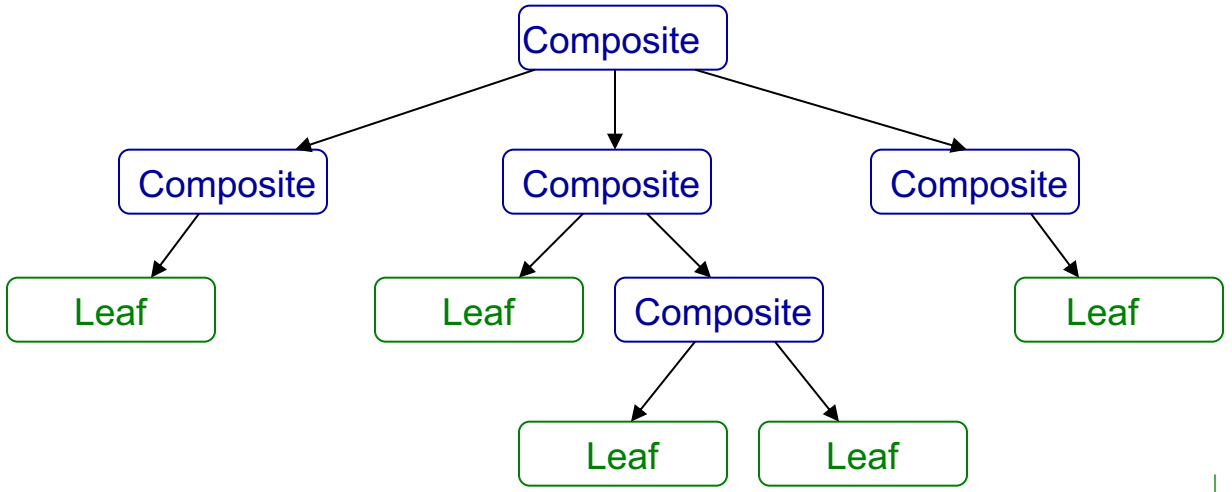
Motivation

- **Intent**
  - Compose objects into tree structures to represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
  - Recursive composition
  - "Directories contain entries, each of which could be a directory."
  - 1-to-many "has a" up the "is a" hierarchy
- **Problem**
  - Application needs to manipulate a hierarchical collection of "primitive" and "composite" objects. Processing of a primitive object is handled one way, and processing of a composite object is handled differently. Having to query the "type" of each object before attempting to process it is not desirable.

Directory / File Example

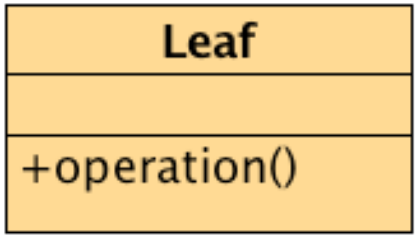
- Directory = Composite
- File = Leaf



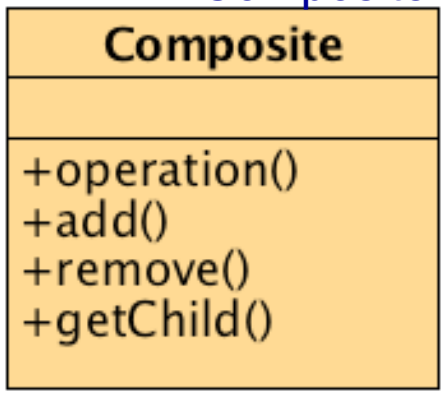


Directory / File Example – Classes

Leaf Class: File

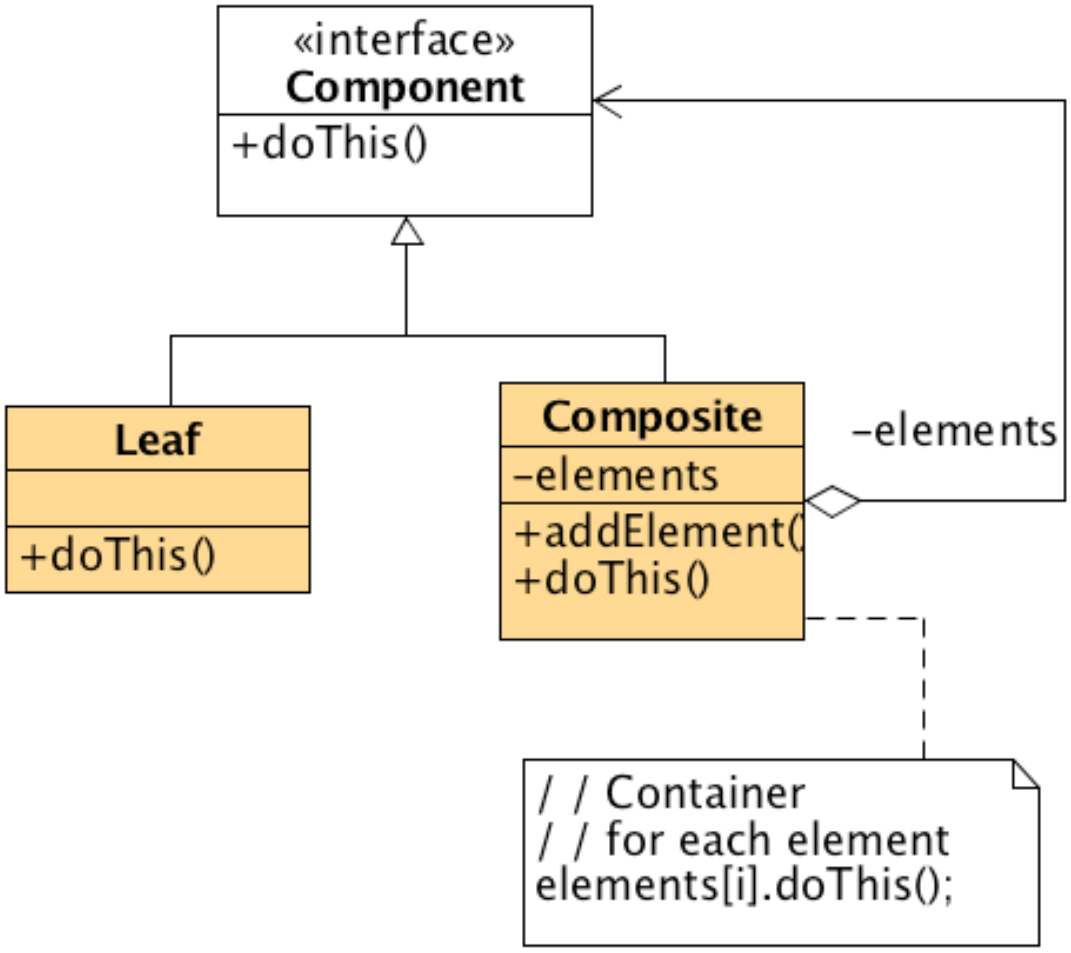


Composite Class: Directory

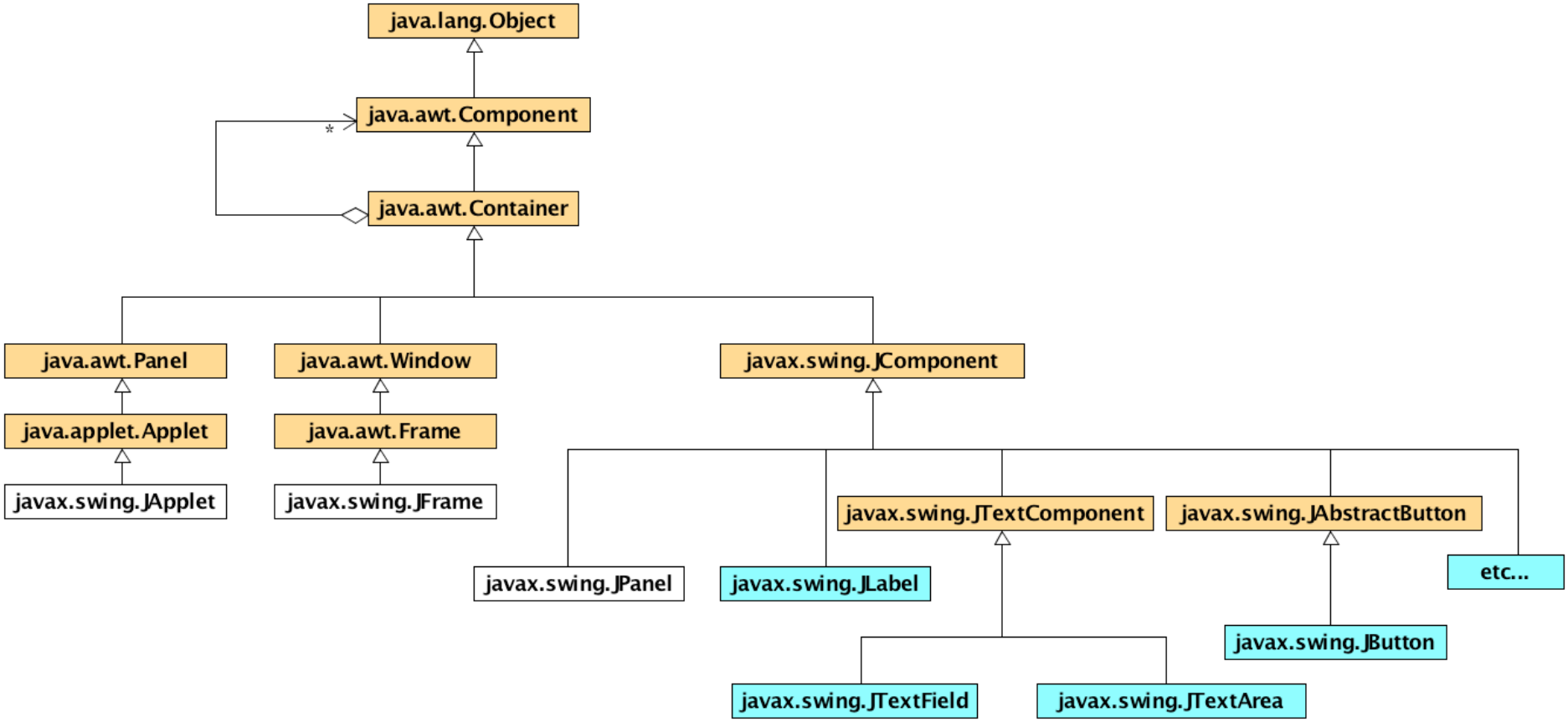


- One class for Files (Leaf nodes)
- One class for Directories (Composite nodes)
  - Collection of Directories and Files
- How do we make sure that Leaf nodes and Composite nodes can be handled uniformly?
  - Derive them from the same abstract base class

Structure


«interface»
Component
+doThis()
Leaf
+doThis()
Composite
-elements
+addElement()
+doThis()
-elements
/ / Container
/ / for each element
elements[i].doThis();

Java Swing – where is the composite?



Example – Entity/Product/Box

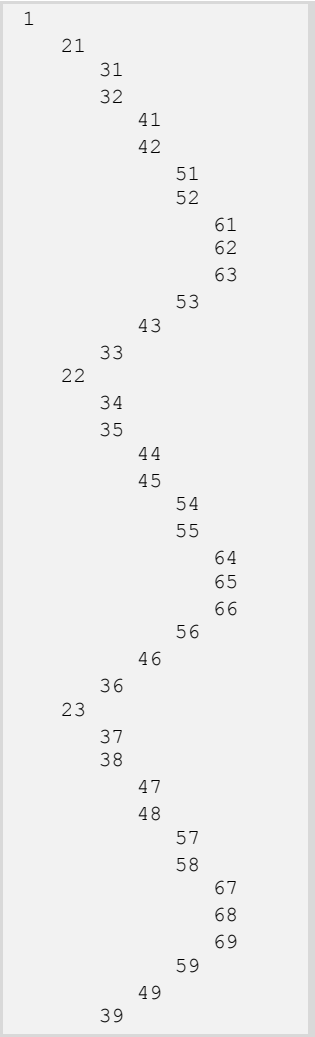
```
abstract class Entity {
    protected static StringBuffer indent = new StringBuffer();
    public abstract void traverse();
}

class Product extends Entity {
    private int value;

    public Product(int val) {
        value = val;
    }

    public void traverse() {
        System.out.println(indent.toString() + value);
    }
}
```

Example – Entity/Product/Box

```
class Box extends Entity {
    private List<Entity> children = new ArrayList<>();
    private int value;

    public Box(int val) {
        value = val;
    }

    public void add(Entity c) {
        children.add(c);
    }

    public void traverse() {
        System.out.println(indent.toString() + value);
        indent.append("   ");
        for (int i = 0; i < children.size(); i++)
            children.get(i).traverse();
        indent.setLength(indent.length() - 3);
    }
}
```

Example – Entity/Product/Box

```
public class CompositeLevels {
    public static void main(String[] args) {
        Box root = initialize();   root.traverse();
    }
    private static Box initialize() {
        Box[] nodes = new Box[7];
        nodes[1] = new Box(1);
        int[] s = { 1, 4, 7 };
        for (int i = 0; i < 3; i++) {
            nodes[2] = new Box(21 + i);
            nodes[1].add(nodes[2]);
            int lev = 3;
            for (int j = 0; j < 4; j++) {
                nodes[lev - 1].add(new Product(lev * 10 + s[i]));
                nodes[lev] = new Box(lev * 10 + s[i] + 1);
                nodes[lev - 1].add(nodes[lev]);
                nodes[lev - 1].add(new Product(lev * 10 + s[i] + 2));
                lev++;
            }
        }
        return nodes[1];
    }
}
```



Check list

- Ensure that your problem is about representing "**whole-part" hierarchical relationships**.
- Consider the heuristic, "Containers that contain containees, each of which could be a container."
- Create a "**lowest common denominator**" interface that makes your containers and containees interchangeable.
- All container and containee classes declare an "is a" relationship to the interface.
- All container classes declare a one-to-many "has a" relationship to the interface.
- Child management methods [e.g. addChild(), removeChild()] should normally be defined in the Composite class.

Structural design patterns

Class

- Adapter

Object

- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

Decorator

- **Intent**
  - Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
  - Client-specified embellishment of a core object by recursively wrapping it.
  - Wrapping a gift, putting it in a box, and wrapping the box.
- **Problem**
  - You want to add behavior or state to individual objects at run-time. Inheritance is not feasible because it is static and applies to an entire class.

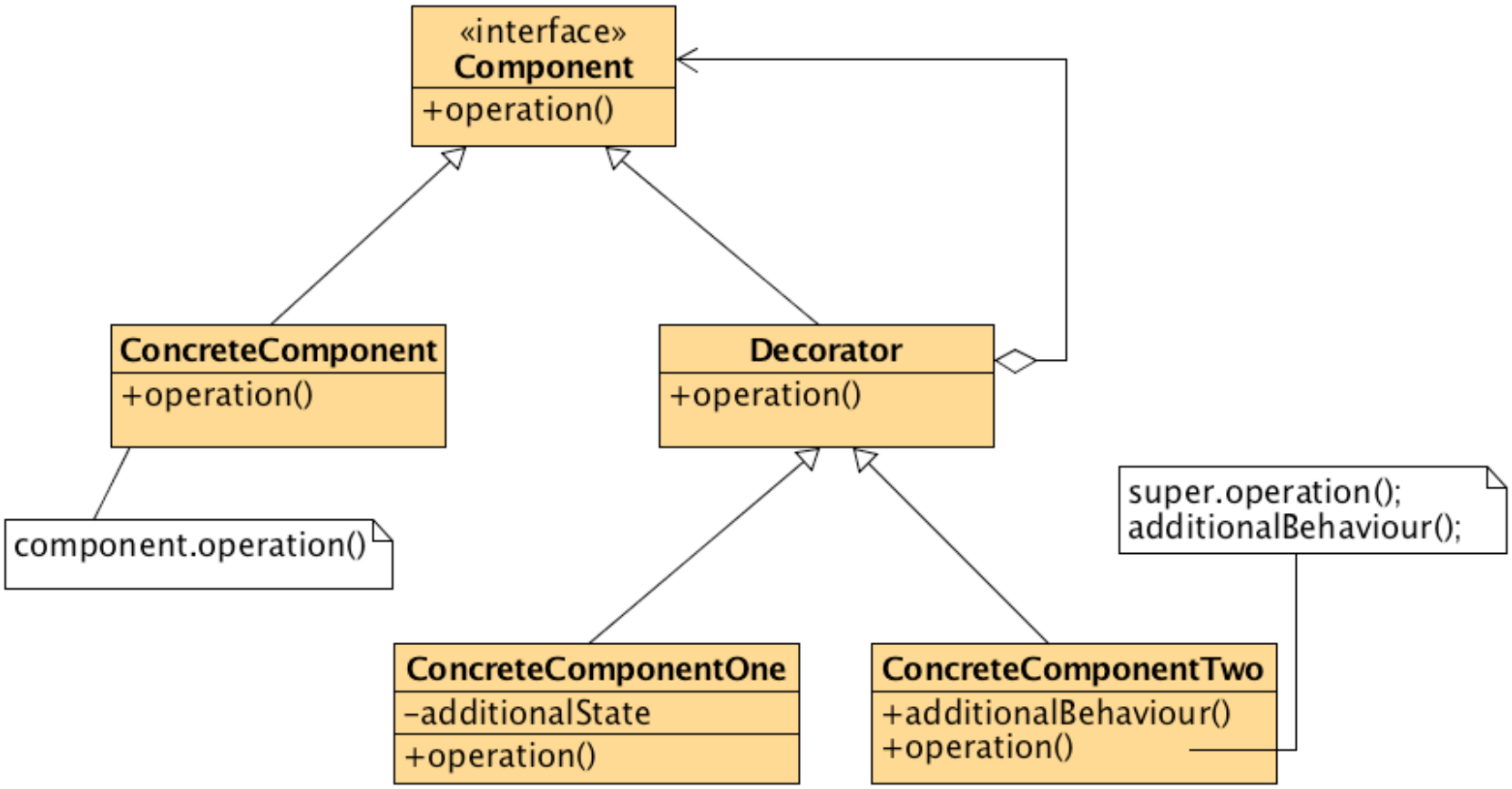
Motivation

- Consider the following entities:
  - Futebolista (joga, passa, remata),
  - Tenista (joga, serve),
  - Jogador (joga)

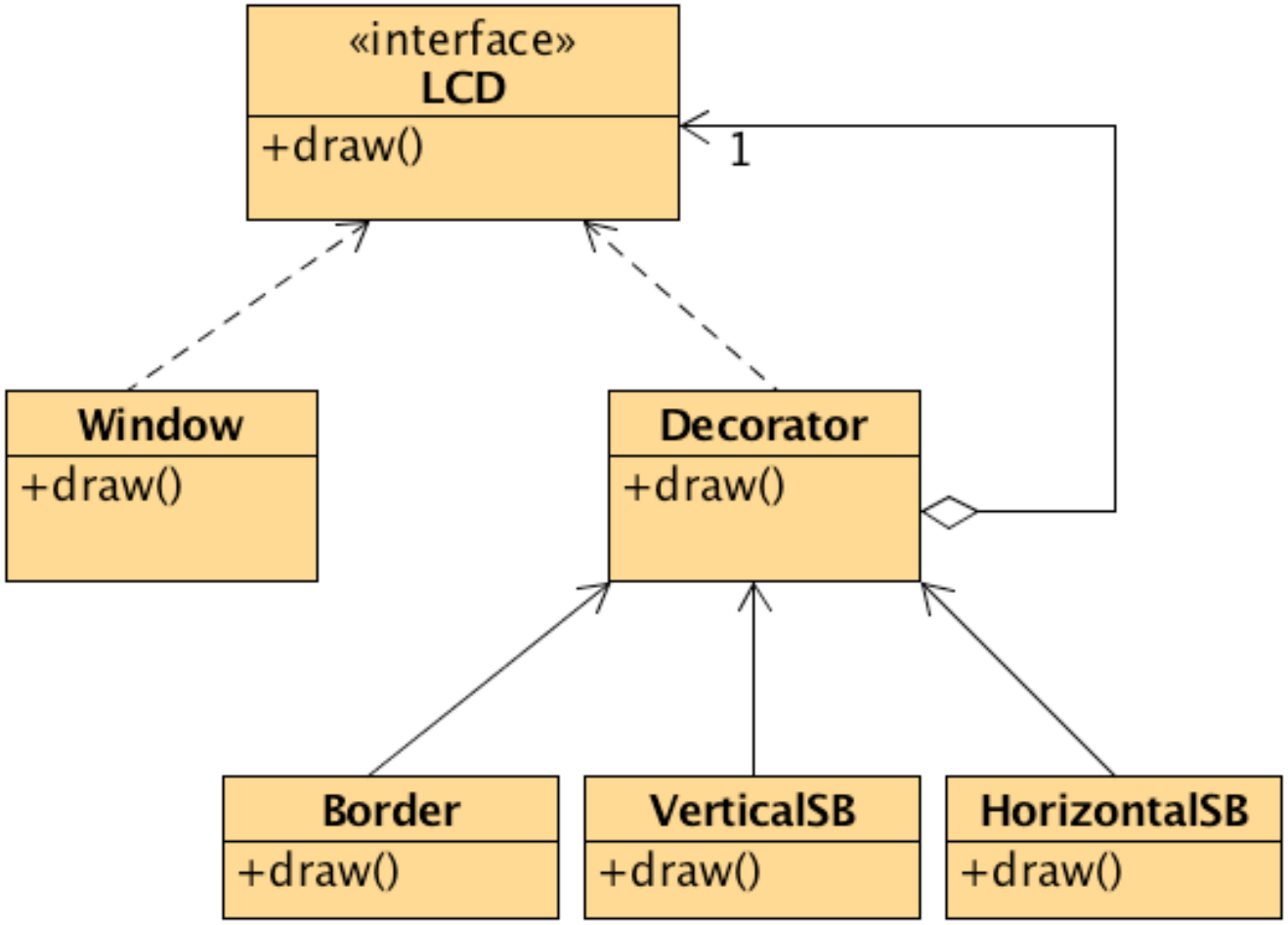
- Let's complicate:
  - O Rui joga Basquete e Futebol
  - A Ana joga Badminton e Basquete
  - O Paulo joga Xadrez, Futebol e Basquete

- .... Solution?

Structure


«interface»
Component
+operation()
ConcreteComponent
+operation()
Decorator
+operation()
component.operation()
super.operation();
additionalBehaviour();
ConcreteComponentOne
-additionalState
+operation()
ConcreteComponentTwo
+additionalBehaviour()
+operation()

Structure – Example



Example

```
interface JogadorInterface {
    void joga();
}

class Jogador implements JogadorInterface {
    private String name;
    Jogador(String n) { name = n; }
    @Override public void joga()
        { System.out.print("\n"+name+" joga "); }
}

abstract class JogDecorator implements JogadorInterface {
    protected JogadorInterface j;
    JogDecorator(JogadorInterface j) { this.j = j; }
    public void joga()       { j.joga(); }
}
```

Example

```
class Futebolista extends JogDecorator {
    Futebolista(JogadorInterface j) { super(j); }
    @Override public void joga()
        { j.joga(); System.out.print("futebol ");  }
    public void remata()  { System.out.println("-- Remata!"); }
}

class Xadrezista extends JogDecorator {
    Xadrezista(JogadorInterface j) { super(j); }
    @Override public void joga()  { j.joga();
        System.out.print("xadrez ");  }
}

class Tenista extends JogDecorator {
    Tenista(JogadorInterface j) { super(j); }
    @Override public void joga()
        { j.joga(); System.out.print("tenis ");  }
    public void serve()  { System.out.println("-- Serve!"); }
}
```

Example

```
public class PlayTest{
    public static void main(String args[]) {
        JogadorInterface j1 = new Jogador("Rui");
        Futebolista f1 = new Futebolista(new Jogador("Luis"));
        Xadrezista x1 = new Xadrezista(new Jogador("Ana"));
        Xadrezista x2 = new Xadrezista(j1);
        Xadrezista x3 = new Xadrezista(f1);
        Tenista t1 = new Tenista(j1);
        Tenista t2 = new Tenista(
                new Xadrezista(
                        new Futebolista(
                                new Jogador("Bruna"))));
        JogadorInterface lista[] = { j1, f1, x1, x2, x3, t1, t2 };
        for (JogadorInterface ji: lista)
            ji.joga();
    }
}
```

```
Rui joga
Luis joga futebol
Ana joga xadrez
Rui joga xadrez
Luis joga futebol xadrez
Rui joga ténis
Bruna joga futebol xadrez ténis
```

Decorator example: Java I/O

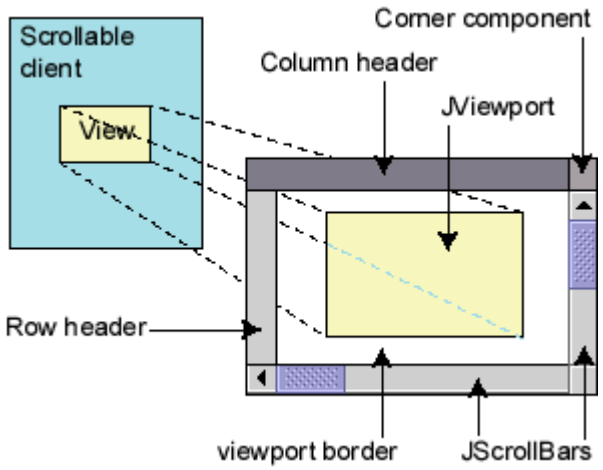
- *InputStream* class has only public *int read()* method to read one letter at a time
- decorators such as *BufferedReader* or *Scanner* add additional functionality to read the stream more easily

```
// InputStreamReader/BufferedReader decorate InputStream
InputStream in = new FileInputStream("hardcode.txt");
InputStreamReader isr = new InputStreamReader(in);
BufferedReader br = new BufferedReader(isr);

// because of decorator streams, we can read an
// entire line from the file in one call
// (InputStream only provides public int read() )
String wholeLine = br.readLine();
```

Decorator example: GUI

- Common GUI components don't have scroll bars
- *JScrollPane* is a container with scroll bars to which we can add any component to make it scrollable

```
// JScrollPane decorates GUI components
JTextArea area = new JTextArea(20, 30);
JScrollPane scrollPane =
  new JScrollPane(area);
contentPane.add(scrollPane);
```



Exercise

Create the required classes to the following main function

```
public class TestDecorator {

   public static void main(String args[]) {
       Icecream icecream =
         new HoneyDecorator(
            new NuttyDecorator(
                  new SimpleIcecream()));
       System.out.println(icecream.makeIcecream());
   }

}
```

```
Base Icecream + cruncy nuts + sweet honey
```

Check list

- Ensure the context is: a single **core** (or non-optional) component, several optional embellishments or wrappers, and an interface that is common to all.
- Create a "Lowest Common Denominator" **interface** that makes all classes interchangeable.
- Create a second level base class (**Decorator**) to support the optional wrapper classes.
- The Core class and Decorator class inherit from the interface.
- The Decorator class declares a composition relationship to the interface, and this data member is initialized in its constructor.
- Define a Decorator derived class for each optional embellishment.

Structural design patterns

Class

- Adapter

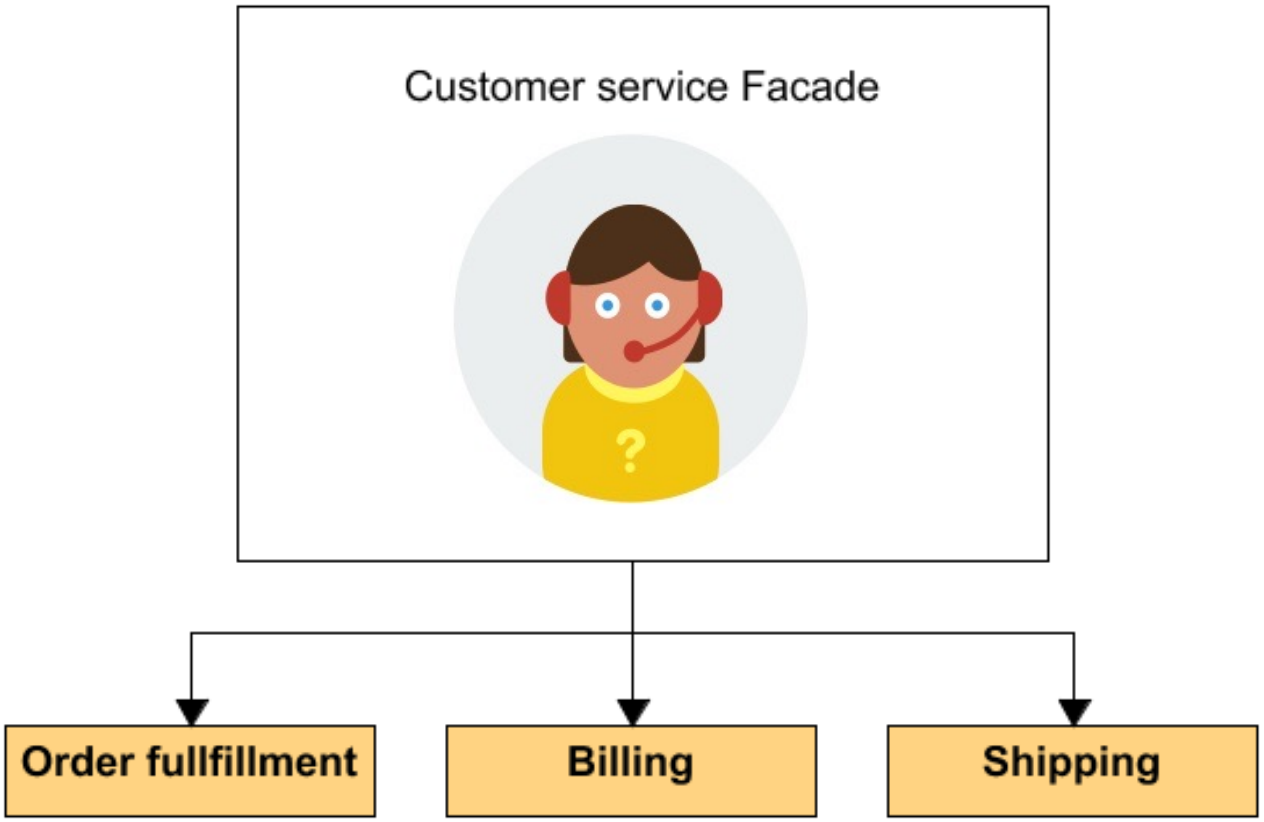
Object

- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

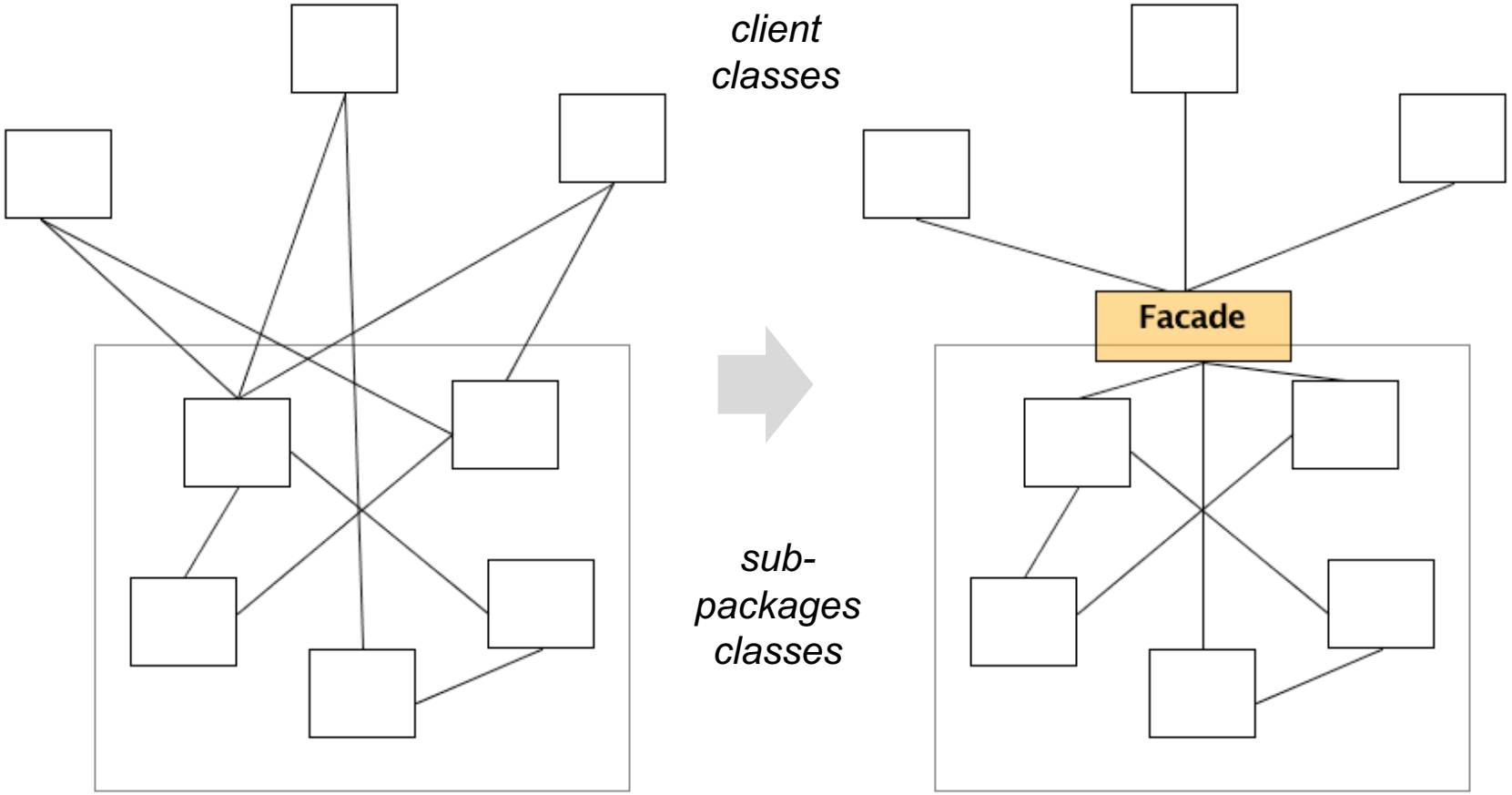
Motivation

- **Problem**
  - A segment of the client community needs a simplified interface to the overall functionality of a complex subsystem.
- **Intent**
  - Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
  - Wrap a complicated subsystem with a simpler interface.

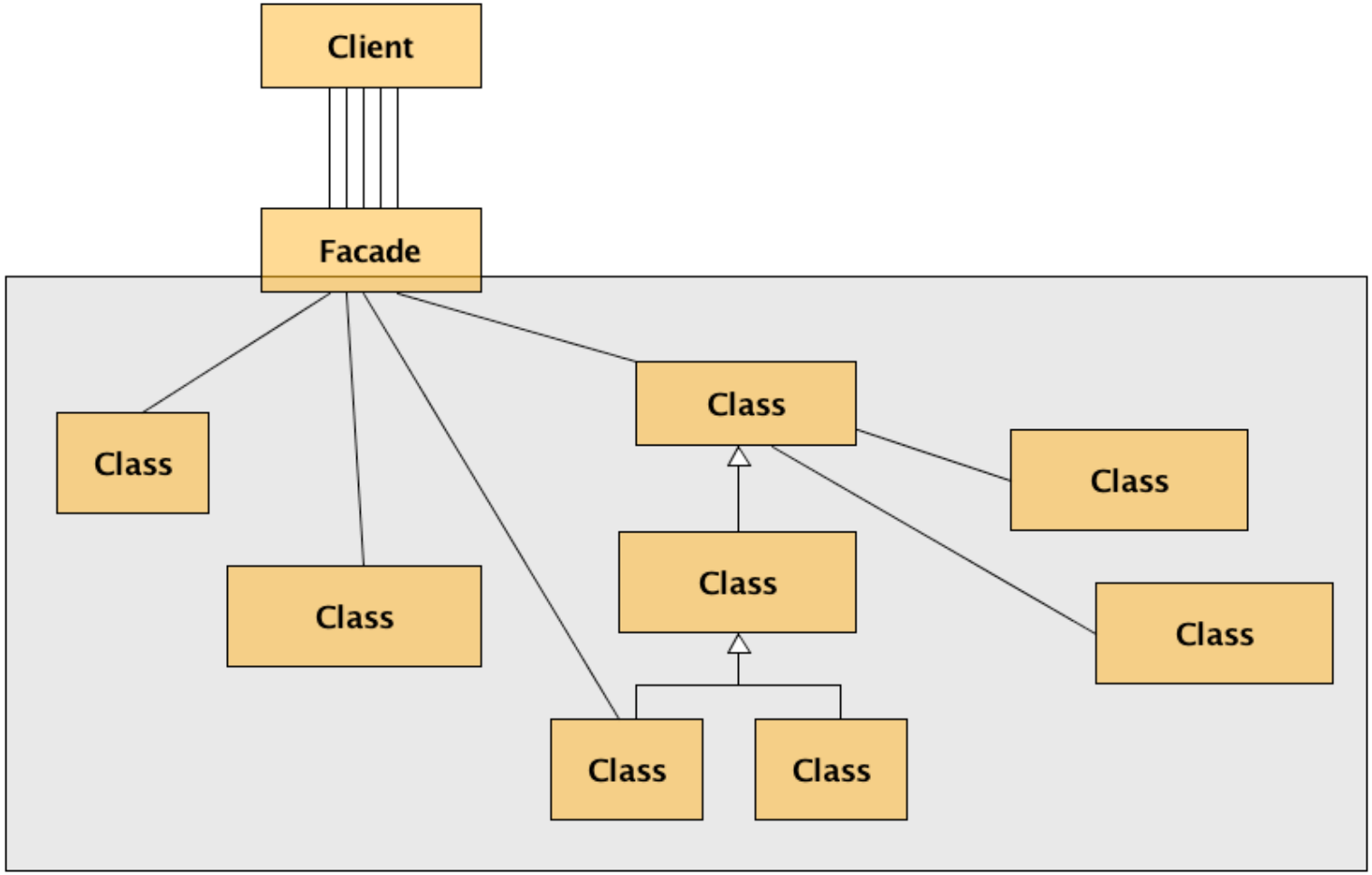
Motivation


Customer service Facade
?
Order fullfillment
Billing
Shipping

Motivation



Structure


Client
Facade
Class
Class
Class
Class
Class
Class
Class
Class

Example

```
// ...
class TravelFacade {
    private HotelBooker hotelBooker;
    private FlightBooker flightBooker;
    private LocalTourBooker tourBooker;
    public void getFlightsAndHotels(City dest, Date from, Data to) {
        List<Flight> flights = flightBooker.getFlightsFor(dest, from, to);
        List<Hotel> hotels = hotelBooker.getHotelsFor(dest, from, to);
        List<Tour> tours = tourBooker.getToursFor(dest, from, to);
        // process and return
    }
}

public class FacadeDemo {
    public static void main(String[] args) {
        TravelFacade facade = new TravelFacade();
        facade.getFlightsAndHotels(destination, from, to);
    }
}
```

Consequences

- Benefits
  - It hides the implementation of the subsystem from clients, making the subsystem easier to use
  - It promotes weak coupling between the subsystem and its clients. This allows you to change the classes that comprise the subsystem without affecting the clients.
  - It reduces compilation dependencies in large software systems
- It does not add any functionality, it just simplifies interfaces
- It does not prevent clients from accessing the underlying classes

Structural design patterns

Class

- Adapter

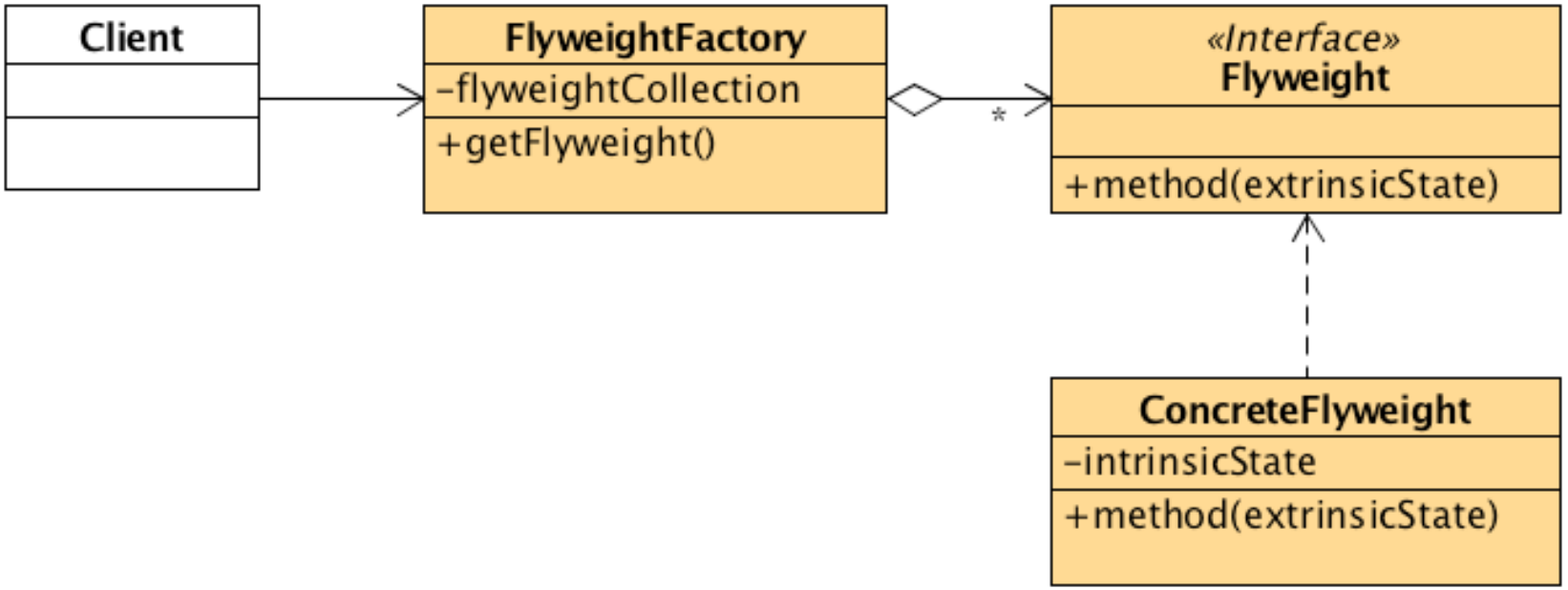
Object

- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

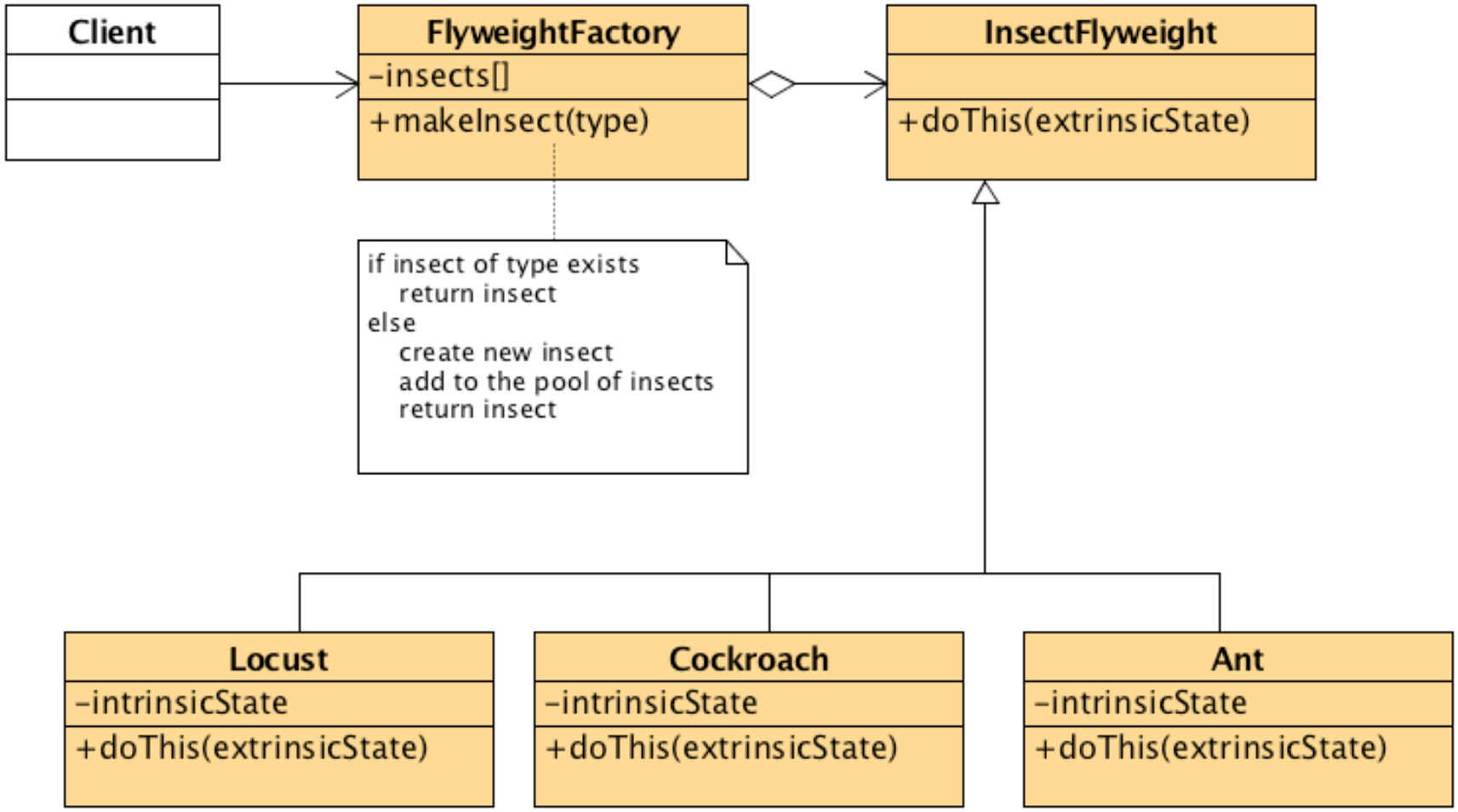
Motivation

- **Intent**
  - Use sharing to support large numbers of fine-grained objects efficiently.
  - The Motif GUI strategy of replacing heavy-weight widgets with light-weight gadgets.
- **Problem**
  - Designing objects down to the lowest levels of system "granularity" provides optimal flexibility, but can be unacceptably expensive in terms of performance and memory usage.

Structure


Client
FlyweightFactory
-flyweightCollection
+getFlyweight()
*
«Interface»
Flyweight
+method(extrinsicState)
ConcreteFlyweight
-intrinsicState
+method(extrinsicState)

Structure - example



Example (java.lang.Integer:valueOf)

```
public final class Integer extends Number implements Comparable<Integer> {

// ...
public static Integer valueOf(int i) {
    final int offset = 128;
    if (i >= -128 && i <= 127) { // must cache
        return IntegerCache.cache[i + offset];
    }
    return new Integer(i);
}

// ...
private static class IntegerCache {
     static final Integer cache[] = new Integer[-(-128) + 127 + 1];
     static {
        for(int i = 0; i < cache.length; i++)
             cache[i] = new Integer(i - 128);
     }
}
```

Example – web browser cache

Browser loads images just once and then reuses them from pool:

Check list

- Ensure that object overhead is an issue needing attention.
- Divide the target class's state into: shareable (intrinsic) state, and non-shareable (extrinsic) state.
- Remove the non-shareable state from the class attributes, and add it the calling argument list of affected methods.
- Create a Factory that can cache and reuse existing class instances.
- The client must use the Factory instead of the new operator to request objects.

Structural design patterns

Class

- Adapter

Object

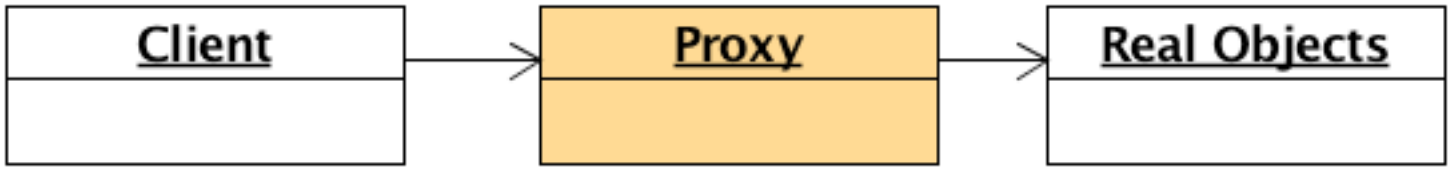
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

Motivation

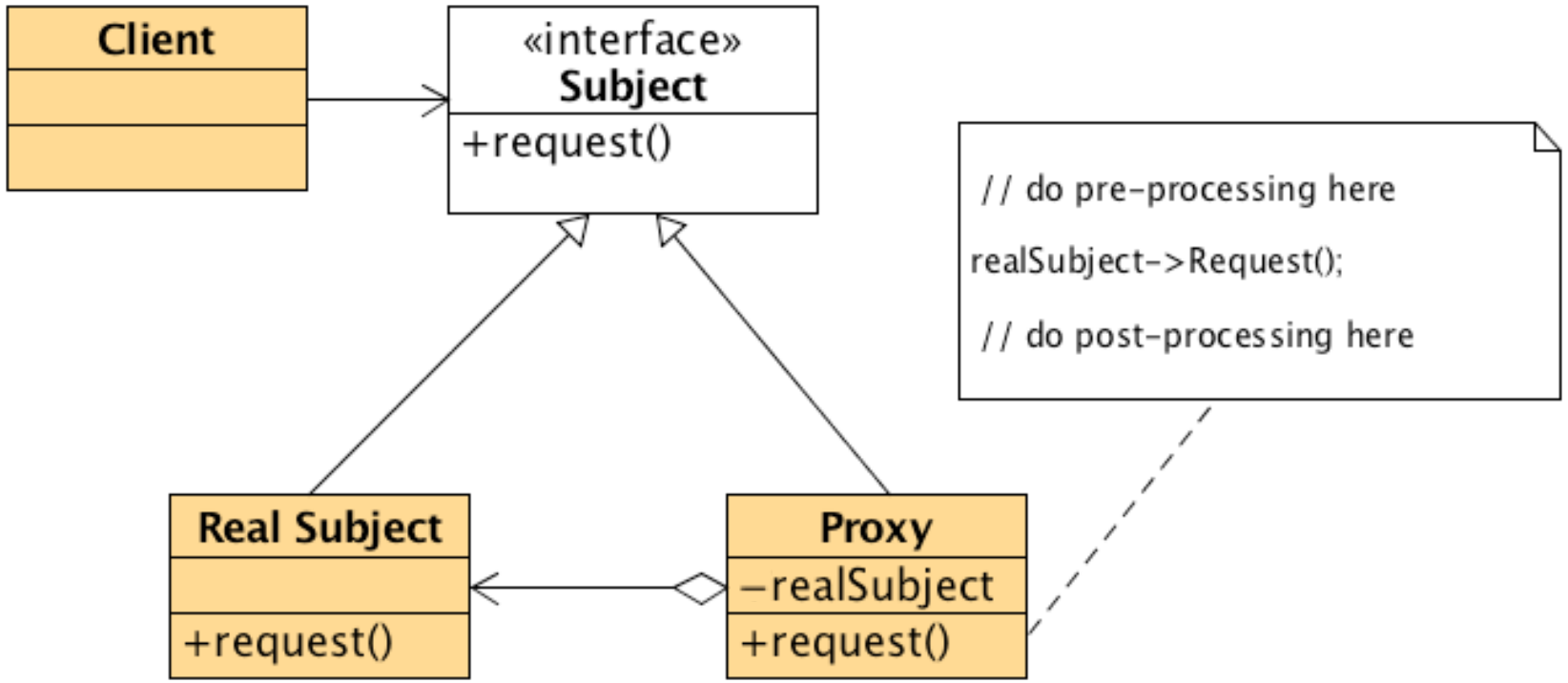
- **Problem**
  - You need to support resource-hungry objects, and you do not want to instantiate such objects unless and until they are actually requested by the client.
- **Intent**
  - Provide a surrogate or placeholder for another object to control access to it.
  - Use an extra level of indirection to support distributed, controlled, or intelligent access.
  - Add a wrapper and delegation to protect the real component from undue complexity.

Solution

- Create a Proxy object that implements the same interface as the real object
  - The Proxy object (usually) contains a reference to the real object
  - Clients are given a reference to the Proxy, not the real object
  - All client operations on the object pass through the Proxy, allowing the Proxy to perform additional processing



Structure


Client
«interface»
Subject
+request()
// do pre-processing here
realSubject->Request();
// do post-processing here
Real Subject
+request()
Proxy
−realSubject
+request()

Consequences

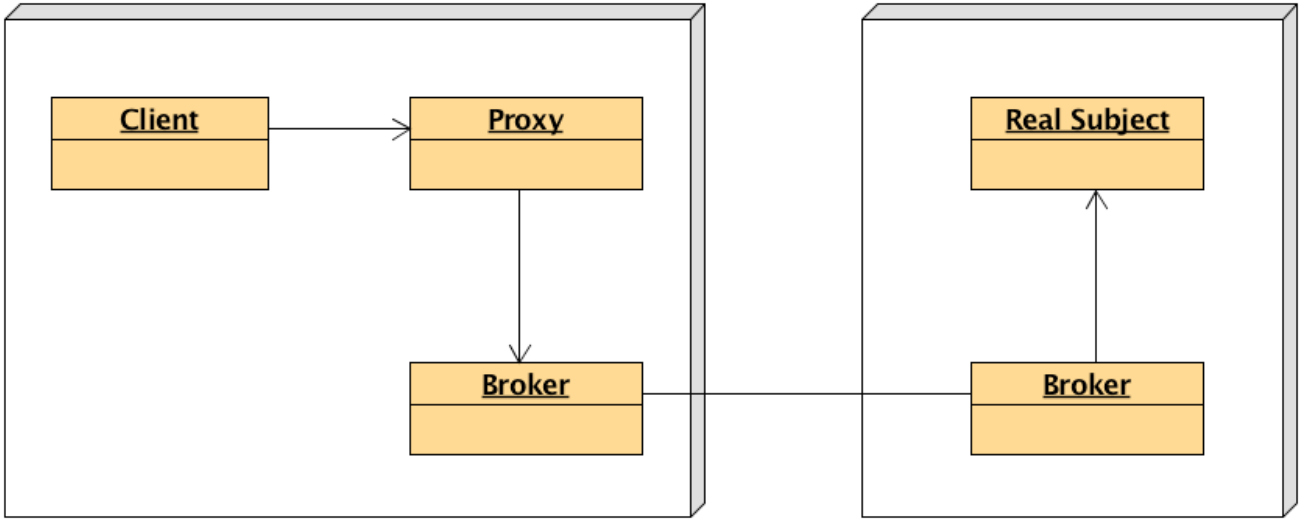
- Provides an additional level of indirection between client and object that may be used to insert arbitrary services
- Proxies are invisible to the client, so introducing proxies does not affect client code

Known Uses: Java Collections

- Read-only Collections
  - Wrap collection object in a proxy that only allows read-only operations to be invoked on the collection
  - All other operations throw exceptions
  - List Collections.unmodifiableList(List list);
    - Returns read-only List proxy
- Synchronized Collections
  - Wrap collection object in a proxy that ensures only one thread at a time is allowed to access the collection
  - Proxy acquires lock before calling a method, and releases lock after the method completes
  - List Collections.synchronizedList(List list);
    - Returns a synchronized List proxy

Known Uses: Distributed Objects

- The Client and Real Subject are in different processes or on different machines, and so a direct method call will not work
- The Proxy's job is to pass the method call across process or machine boundaries, and return the result to the client (with Broker's help)



Known Uses: Secure Objects

- Different clients have different levels of access privileges to an object
- Clients access the object through a proxy
- The proxy either allows or rejects a method call depending on what method is being called and who is calling it (i.e., the client's identity)

Known Uses: Lazy Loading

- Some objects are expensive to instantiate (i.e., consume lots of resources or take a long time to initialize)
- Create a proxy instead, and give the proxy to the client
  - The proxy creates the object on demand when the client first uses it
  - Proxies must store whatever information is needed to create the object on-the-fly (file name, network address, etc.)

Known Uses: Copy-on-Write

- Multiple clients share the same object as long as nobody tries to change it
- When a client attempts to change the object, they get their own private copy of the object
- Read-only clients continue to share the original object, while writers get their own copies
- Allows resource sharing, while making it look like everyone has their own object
- When a write operation occurs, a proxy makes a private copy of the object on-the-fly to insulate other clients from the changes

Check list

- Identify the functionality that is best implemented as a wrapper or surrogate.
- Define an interface that will make the proxy and the original component interchangeable.
- Consider defining a Factory that can encapsulate the decision of whether a proxy or original object is desirable.
- The wrapper class holds a pointer to the real class and implements the interface.

Structural design patterns

Class

- Adapter

Object

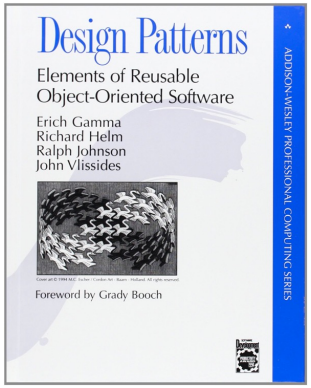
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

Structural patterns – Summary

- Adapter
  - Match interfaces of different classes
- Bridge
  - Separates an object's interface from its implementation
- Composite
  - A tree structure of simple and composite objects
- Decorator
  - Add responsibilities to objects dynamically
- Facade
  - A single class that represents an entire subsystem
- Flyweight
  - A fine-grained instance used for efficient sharing
- Proxy
  - An object representing another object

Resources

- Design Patterns – Elements of Reusable Object-Oriented Software; Gamma, et. al.



- *Design Patterns Explained Simply (sourcemaking.com)*