Design Patterns
Fundamentals

Software Object-Oriented Design with UML

Contents

- Introduction
  - Rationale
  - Fundamental concepts
- Structural design patterns
  - Adapter
  - Composite
- Behavioral design patterns
  - Observer
  - Strategy
- Creational design patterns
  - Factory
  - Singleton

Introduction

Rationale

Fundamental Concepts

Introduction – *Assumptions*

- ***A1***: The projects we work on are distributed vastly over time and space (teams)
- ***A2***: We spend much more time/effort for *maintenance* than *initial design & implementation*

Introduction – *Goals*

- ***G1***: Avoid modifying existing source code altogether
- ***G2***: Make our designs extendable in the ***right directions***, with the right tools and to the right amount (avoid over-engineering!)

Introduction – *Rules of Thumb*

- ***R1***: Program to *Interfaces*, not *Implementations*
- ***R2***: Depend on *Abstractions*, not *Concrete classes*
- ***R3***: Design your classes so that they are *Open for Extension, but Closed for Modification* (The Open-Closed Principle)
- ***R4***: Strive for *high cohesion* **inside** a function/class/module/component – "Do one thing and do it well"
- ***R5***: Strive for *loose coupling* **between** interacting functions/objects/modules/components
- ***R6***: Encapsulate what varies
- ***R7***: Principle of least knowledge (Law of Demeter)

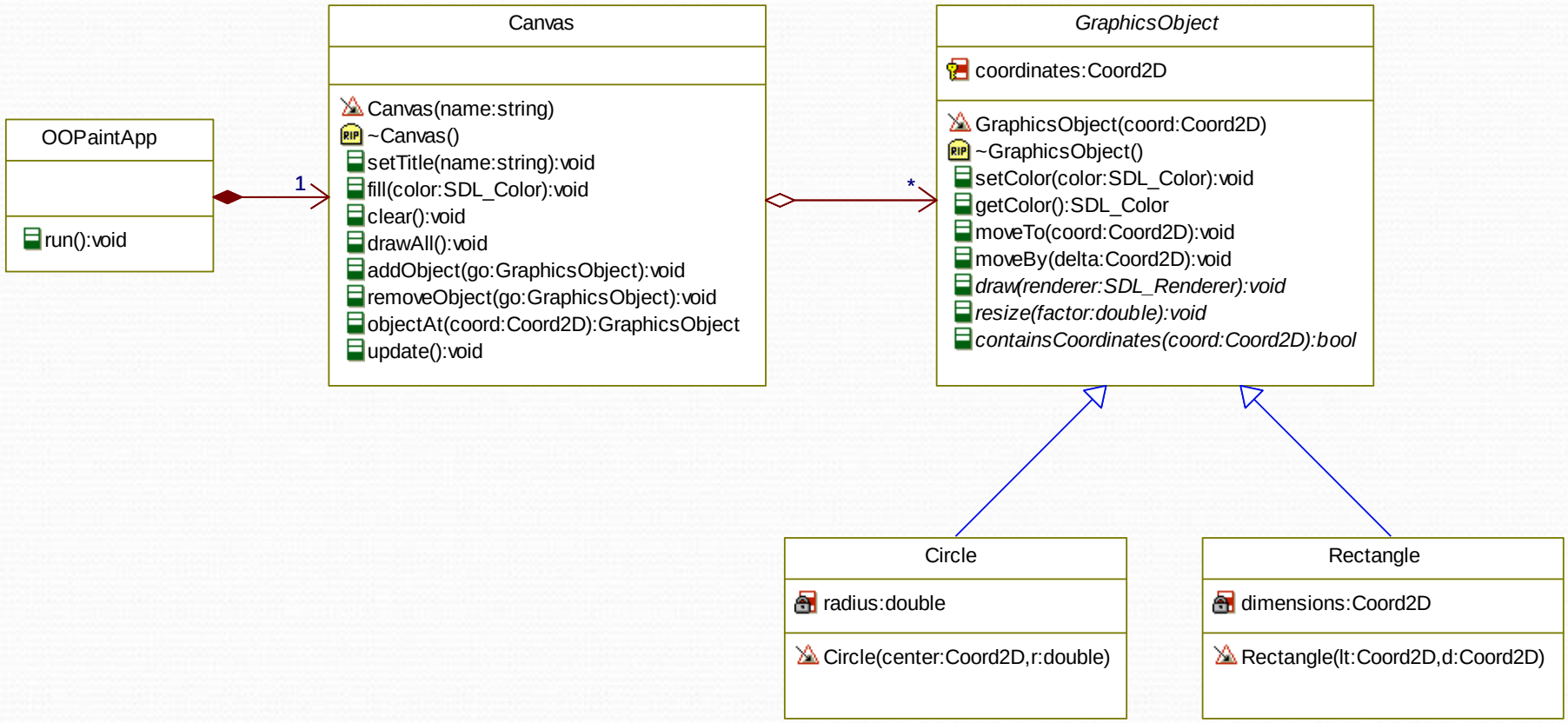
Preparation – Polymorphism (revisited)

- What is Polymorphism?
- Objects of different Type (Class) provide a single Interface
- Important implication: Using Polymorphism, we are able to send messages to Objects without knowing (specifying) their concrete Class (Type)
  - We only use/depend on the common Interface they all implement
- Therefore we don't depend on many concrete types
  - Instead, we only depend on a single (supposedly more stable) Interface
- Which Rules are fulfilled using Polymorphism?
- What changes do we become immune to thanks to this technique?

Preparation – Exercise

- Create a basic Object Oriented drawing program (OOPaint) that has one window (called the Canvas). The User shall be able to draw simple graphic objects on the Canvas – i.e. rectangles and circles
- The application shall use SDL2 as a graphical library. No widgets/menus, just a window with a plain graphical context for 2D drawing
- New shapes are created via keyboard press (i.e. ‘c’ for a Circle, ‘r’ for a Rectangle). User shall be able to move them with the mouse and put them down on the Canvas on left mouse button click
- User shall be able to scale the shapes using the mouse scroller. The resize action applies to the shape User moves currently
- User shall be able to delete shape(s) using right mouse button click
- Implement the UML model shown on the class diagram

Preparation – Exercise (Class Diagram)

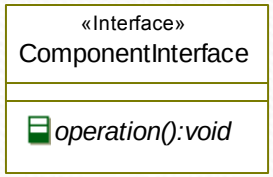


Strategy

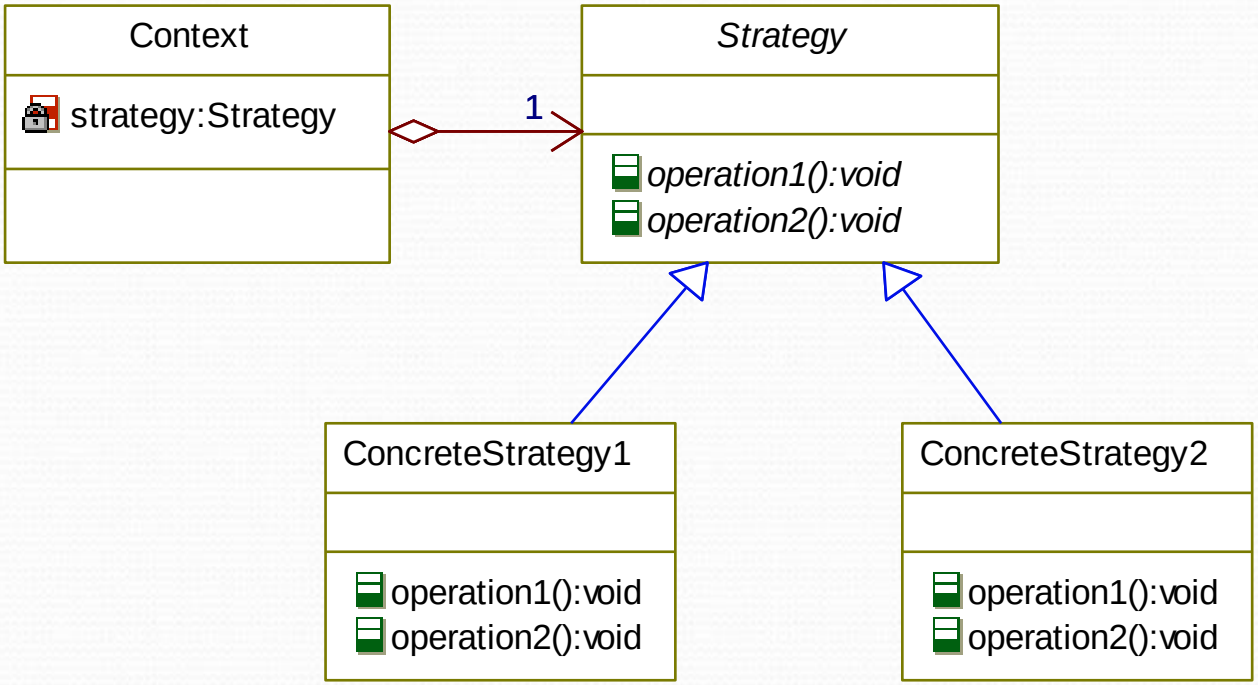
Change Object Behavior

Strategy

- Examples of changing object's behavior in runtime
  - Car changing type of fuel (from petrol to gas), brake type, etc.
  - *Another perspective*: Select among family of algorithms (i.e. sorting)
- Clients call the same method but get new behavior
  - Context may have a *setBehavior()* method. It can be assigned in a *Constructor* too.
  - Essence: Context delegates its behavior to Strategies. It only assigns strategies, and never implements them on its own.
  - This drastically differs from *subclassing*
  - Let's sketch the idea…



Strategy


Context
strategy:Strategy
1
Strategy
operation1():void
operation2():void
ConcreteStrategy1
operation1():void
operation2():void
ConcreteStrategy2
operation1():void
operation2():void

Strategy – Exercise

- Context: OOPaint program from before
- The Customer gets some complaints our program is somewhat difficult to use because objects are hard to situate evenly throughout the canvas. They wonder how come such experienced software team has not provided for *"Snap to grid"* functionality. The feature should be active if the User moves a Shape (or a Group) with Shift button pressed.
- Provide UML model (class diagram) and implementation for this functionality. Use Strategy to vary types of shapes' motion in runtime.

Composite

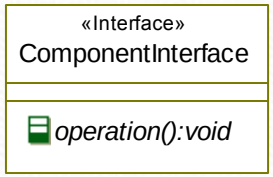
Represent a Composition of Objects as one Object

Composite

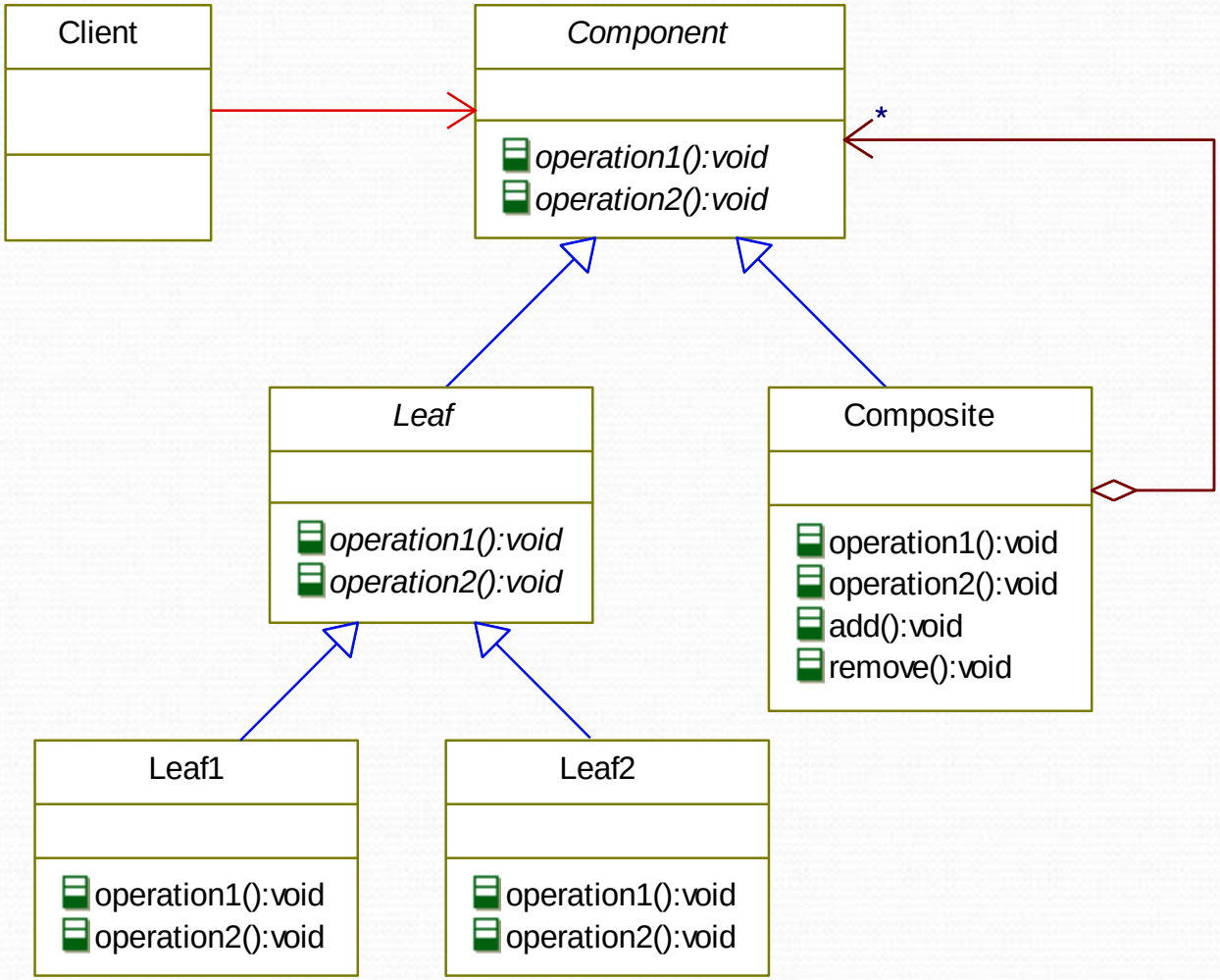
- Problem domain
  - Sometimes we need to work with tree-like structures
  - There are "**leaf**" ("child") nodes which represent simple (non-composite) items and "**branch**" ("parent") nodes which represent compositions (groups of items).
- Examples of nested Objects Compositions
  - **Directories** containing Files & nested **Directories**, etc.
  - Document with **Sections** with Content & nested **Sections**
  - A GUI **Frame** containing Buttons, Edit controls and also nested **Frames** which in turn can contain controls etc.
  - Often code that deals with such tree-structured data is full of conditional statements to test whether an item is a "leaf" or "branch" in order to process it further

Composite

- However, in reality, most of the operations we need can be applied to both *leafs* and *compositions (groups)*
- Examples: move, change color, delete (for a shape / group of shapes)
- Therefore, it would be better if we could find a way to treat a group of items as a single item and avoid checks
- In software terms, we need the Composition (as a whole) to expose the same Interface as its Objects
- This way, we can perform the same operations on the Composition as on its individual items
- Let's sketch the idea…



Composite


Client
Component
operation1():void
operation2():void
*
Leaf
operation1():void
operation2():void
Composite
operation1():void
operation2():void
add():void
remove():void
Leaf1
operation1():void
operation2():void
Leaf2
operation1():void
operation2():void

Composite – Roles & Responsibilities

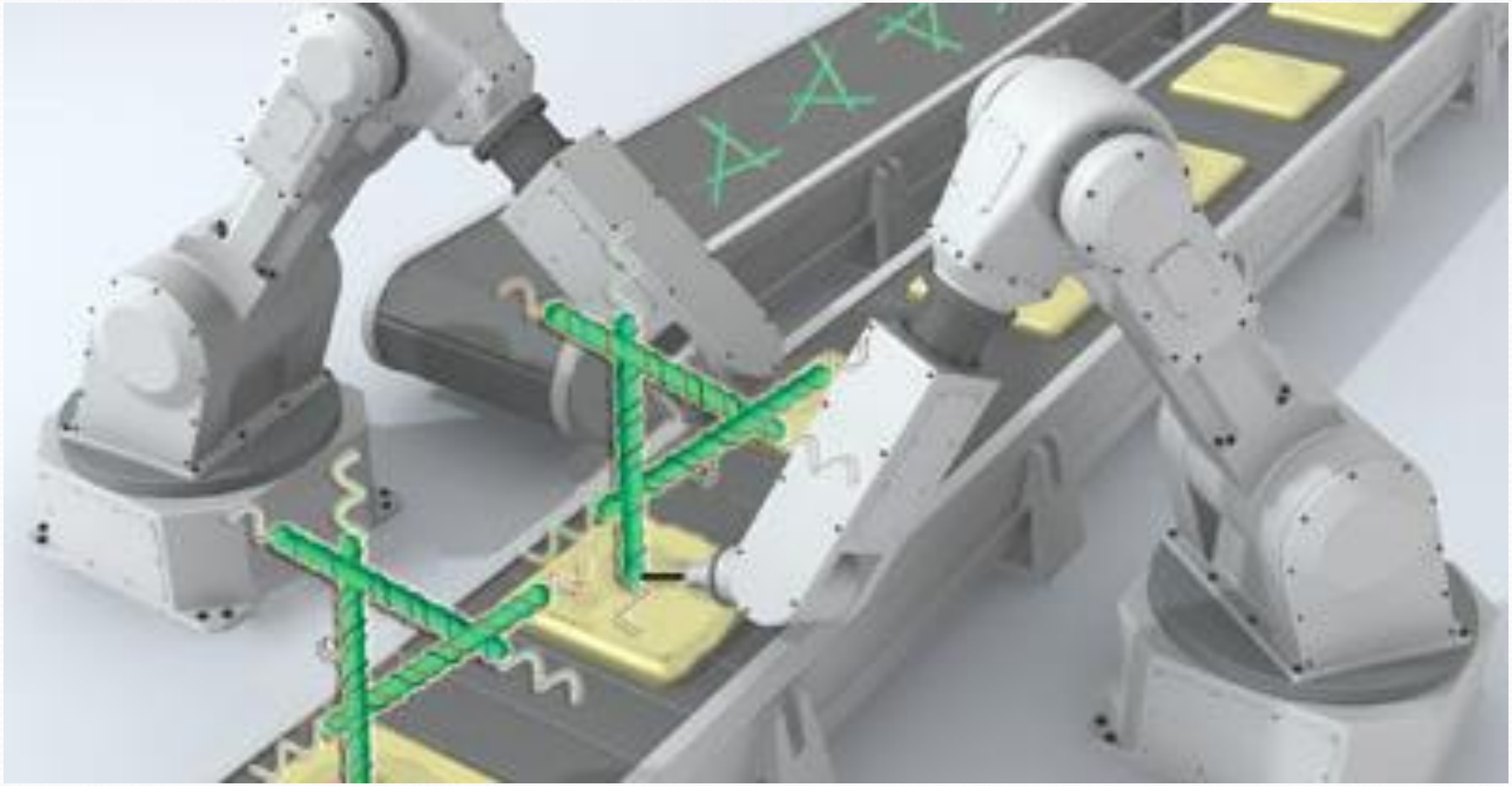
- *Component*
  - Abstraction for both Leaf and Composite components
- *Leaf*
  - Abstraction for Leaf components only (non-composite)
- Leaf1, Leaf2
  - Concrete Leaf classes
- Composite
  - Represents composite components (that group/compose other components – children)
  - Has methods dedicated for manipulation of children (add, remove, find, etc.)
  - Usually implements Component methods by delegating their behavior to its children

Composite – Exercise

- Context: OOPaint program from before
- Assume single graphics sheet (for now)
- User works with simple geometric Shapes (Point, Rectangle, Circle, … *maybe more?*)
- Once added to the sheet, Shapes can be Rotated, Scaled, Deleted, Moved… *(add 1-2 more what you deem necessary)*
- The Customer wants *Grouping functionality*. The User should be able to group Shapes together and perform the same basic operations on Groups as apply to all Shapes
- Provide UML model (class diagram) & implement for this functionality

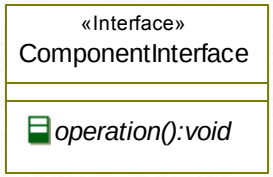
Factory

Create Objects

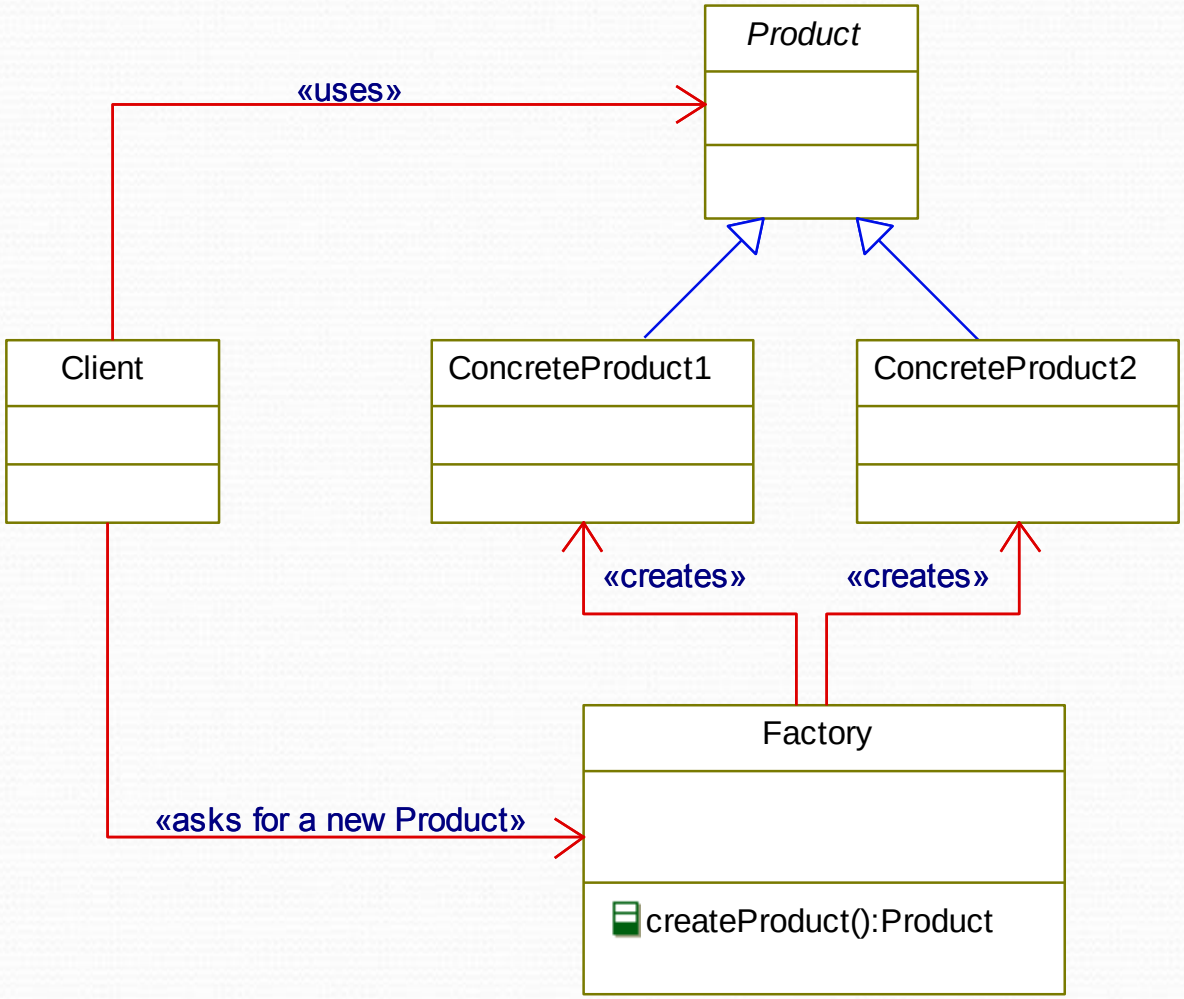


Factory

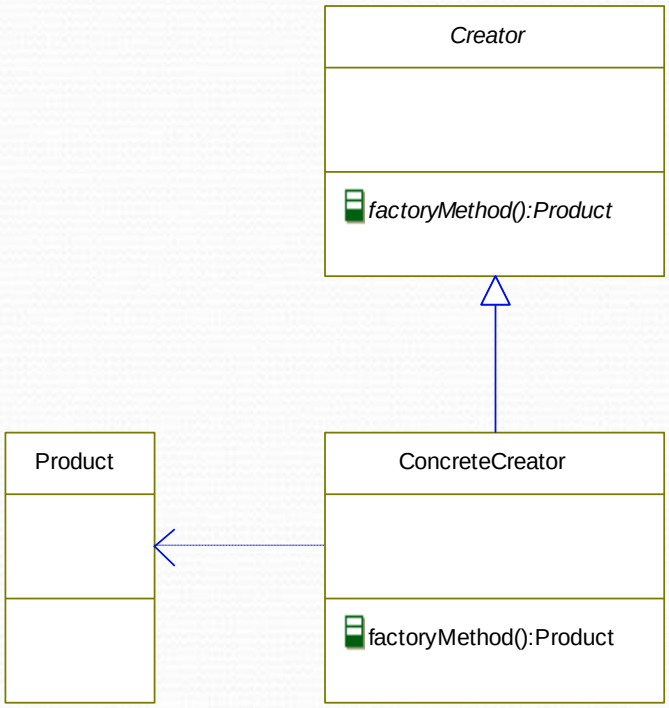
- Examples for object creation?
  - Umm… each and every application?
  - Everyone knows how to create objects, right?
  - Right… we can do it so well, we construct objects just about *everywhere*
  - *… which is a real disaster*
- Encapsulate class instantiation!
  - We strive to extend without modification (create new classes)
  - So we don't want to have class instantiation all over our app!
  - Let's sketch the idea…



Simple Factory



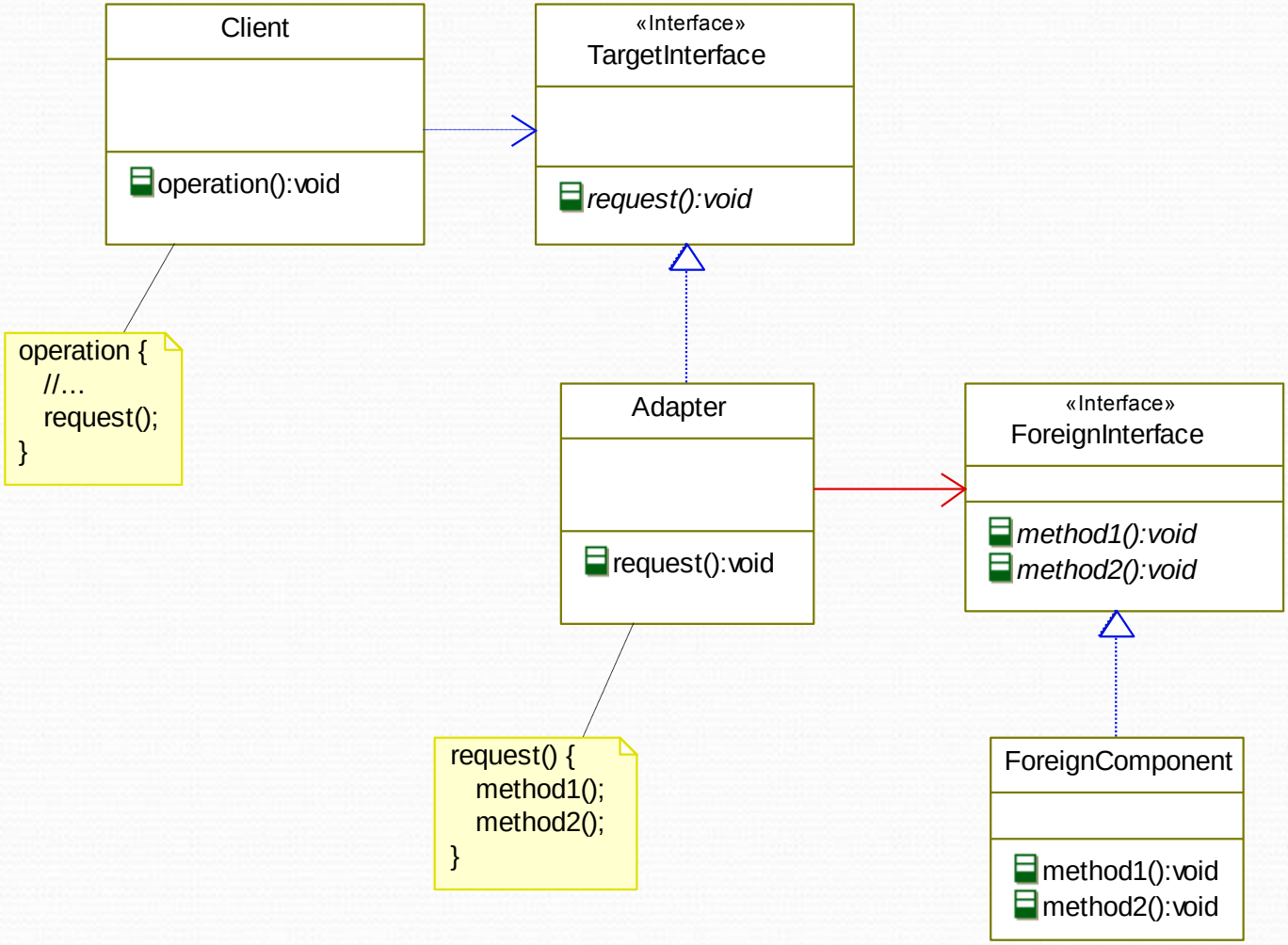
Factory method



Adapter

"Things Are Not What They Appear"

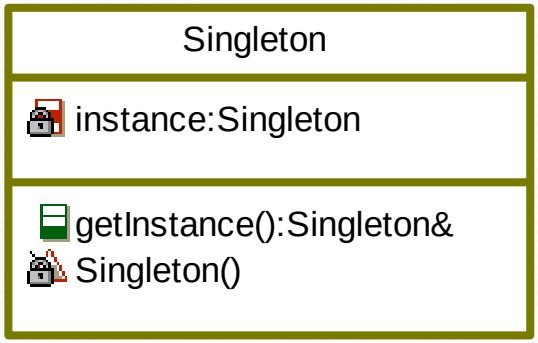
Adapter


Client
operation():void
«Interface»
TargetInterface
request():void
operation {
//…
request();
}
Adapter
request():void
«Interface»
ForeignInterface
method1():void
method2():void
request() {
method1();
method2();
}
ForeignComponent
method1():void
method2():void

Singleton

The One And Only

Singleton


Singleton
instance:Singleton
getInstance():Singleton&
Singleton()

Singleton

- Advantages
  - Cannot create inadvertently more than one instance of a Class when it does not make sense (i.e. when that Class models exactly 1 real-world entity)
  - Makes use of *lazy initialization* where resources for the Singleton are only allocated the first time they are requested (compare to a global variable which allocates resource from program start)
  - Simplifies API by avoiding parameters solely dedicated to pass a single object around *(Beware! See Disadvantages page)*

Singleton

- Disadvantages
  - Singletons represent a global state. Global state is transitive (it's "contagious") – i.e. it spreads wildly throughout the whole application
  - Singletons tend to spoil object-oriented design and style of implementation. Undermines *encapsulation* and *loose coupling*
  - Classes that use Singletons **hide** dependencies from the Singletons they use and from each other (their API is misleading)
  - As a result, the complexity of the system can easily increase out of control
  - There are multiple types of complexity Singletons bring (see the next page)

Singleton

- Disadvantages (contd.)
  - Implementation complexity: Singletons easily can increase complexity by forcing you to take more things into consideration simultaneously while writing each line of code
  - As a consequence, the same code executed several times on the same input might lead to different results without obvious reasons (due to global state) – i.e. create a new object and call one of its methods with the same input: you expect the same output but it's different
  - Unit test complexity: UT results are no longer reliable! Sometimes they pass, sometimes they fail because of (hidden) dependency on some global state
  - Debugging complexity: it's harder to reproduce and debug a problem when it depends on some global state

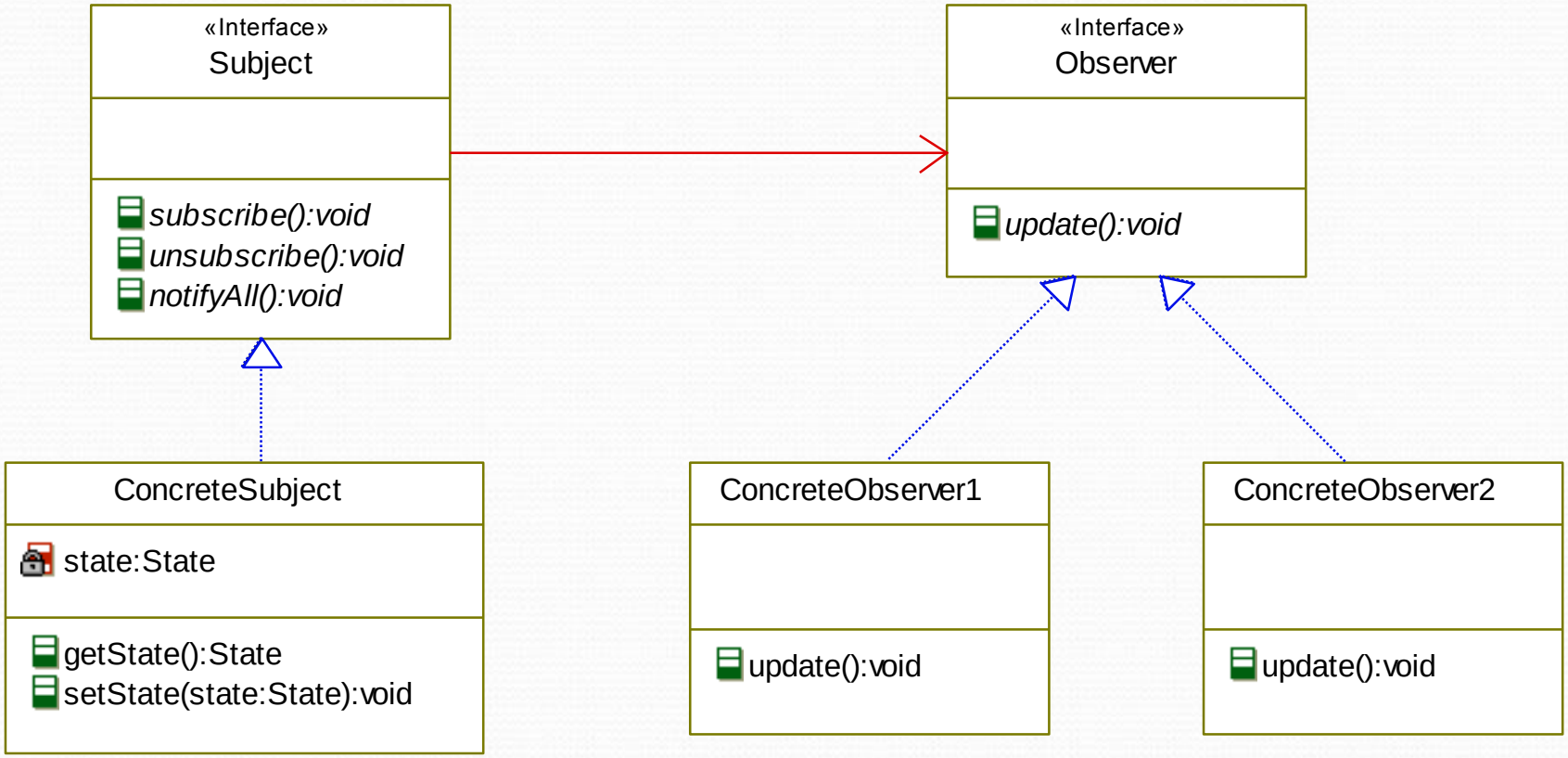
Singleton

- Challenging question to discuss
  - Singletons (being an object that encapsulates a global state) allow for two classes to interact *without knowing about each other*
  - Now, how is this a *bad thing*? Isn't it what *loose coupling?*
  - *(define what loose coupling really means)*

Observer

Subscribe / Notify

Observer


«Interface»
Subject
subscribe():void
unsubscribe():void
notifyAll():void
«Interface»
Observer
update():void
ConcreteSubject
state:State
getState():State
setState(state:State):void
ConcreteObserver1
update():void
ConcreteObserver2
update():void

Observer

- Roles & Responsibilities
  - *Subject Interface* – implemented by all classes that wish to become Subjects (to be Observed via subscribe/notify mechanism)
  - *Observer Interface* – implemented by all classes that wish to be Observers (to be able to subscribe for concrete Subjects and be notified on changes of their states)
  - Concrete Subjects– classes that need to act as *Subjects* (support subscriptions and notify all Observers that subscribe to them). Those classes need to have only 1 thing in common: they implement *Subject Interface*
  - Concrete Observers – classes that need to observe concrete Subjects. They are able to subscribe for notifications and receive such from the corresponding Subject

Observer

- Dangers
  - *Lapsed Listener Problem*

Observer – Exercise

- Context: OOPaint program from before
- Customer wants a special graphic object (a *"marker"*) which if put on the Canvas, allows to move all graphic objects (like a caption) seemingly moving the whole Canvas around. A variant of that: move all graphic objects that are created *after* the marker.
- Provide UML model (class diagram) and implementation for this functionality

References

- *Freeman, Eric, Elisabeth Robson, Bert Bates, Kathy Sierra (2004)* **Head First Design Patterns**
- http://misko.hevery.com/2008/11/21/clean-code-talks-global-state-and-singletons/