Bridge

Gof classification: object structural
Other classification: Interface

... towards the Bridge Pattern

A fresh look at encapsulation

- Remember: encapsulation is more than hiding data
- Multiple levels of encapsulation
  - encapsulation of data, methods, subclasses, algorithms, objects...
- Inheritance as a concept vs. inheritance for reuse
- Design patterns use inheritance for classifying and hiding
- ”Find what varies and encapsulate it”
  - hide classes with inheritance
  - hide objects with e.g. adapters
- many design patterns use encapsulation to define layers between objects – enabling the designer to change things on different sides of the layers without affecting the other

A fresh look at encapsulation

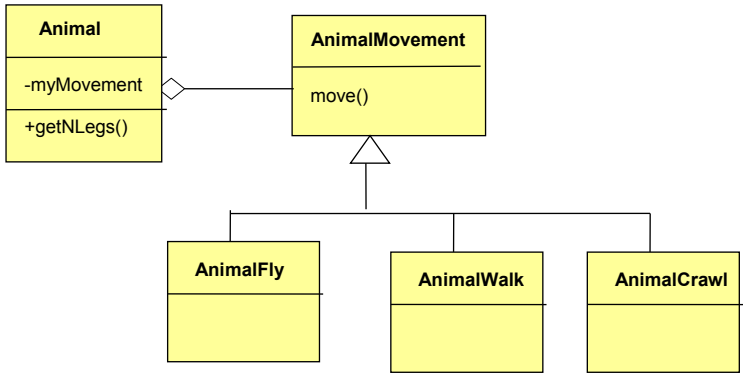
- Example: modeling characteristics of animals
  - each type of animal can have different number of legs (integer)
  - each type of animal can have different type of movement fly, walk or crawl
  - an animal must be able to return the number of legs when asked
  - an animal must be able to calculate how long it would take to move a distance given the type of terrain
- What kind of design would you create?

A fresh look at encapsulation

- Typical way of handling variation in number of legs is to have a data member for storing it and methods for setting and getting it
- Typically we take a different approach for handling variation in movement type
  - use a member flag to indicate the type of movement type and choosing different code in the movement method accordingly
    - problem: tight coupling, messy code if flag starts implying other differences
  - having different types of Animals derived from base Animal class
    - problem: need to manage subtypes of animals, cannot have animals that have more than one type of movement
    - subtyping based on one property of animals, what about classifying them as mammals, reptiles and birds?

A fresh look at encapsulation

- Third possibility for modeling variation in movement: encapsulate movement (behavior) into a class and have animal class contain an object that has the appropriate behavior
- May look like overkill at first, but it is analogous to having a member containing the number of legs, nothing else.
  - Number of legs, type of movement – what is the difference?

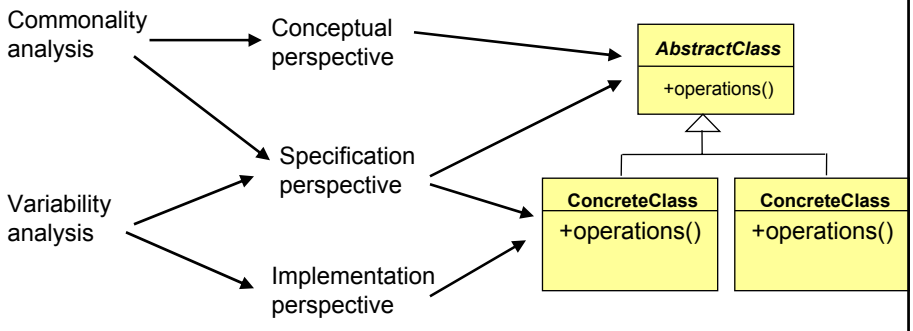


Commonalities and variabilities

- Commonality/variability analysis tells us how to find variations in the problem domain and identify what is common across the domain
- Commonality
  - Coplien: ”Commonality analysis is the search for common elements that helps us to understand how family members are the same.” – *Multi-Paradigm Design for C++*
  - thus process of finding how things are common defines a family in which these elements belong and a context where things vary
  - seeks the structure that is unlikely to change over time
- Variability
  - Variability analysis reveals how things vary *within* a context of commonality
    - variability only makes sense within a given commonality
  - captures structure that is likely to change
- From an architectural perspective, commonality analysis gives the architecture its longevity; variability analysis drives its fitness for use and flexibility

Commonalities and variabilities – a paradigm for finding objects?

- This suggests that you should do the OO analysis from commonality/variability viewpoint, instead of *is-a* based analysis, looking at nouns and verbs



Summary

Mapping with Abstract Classes	**Comments**
Abstract class → the central binding concept	An abstract class represents the core concept that binds together all of the derivatives of the class.
Commonality → which abstract classes to use	The commonalities define the abstract classes I need to use
Variations → derivation of an abstract class	The variations identified *within* a commonality become derivations of the abstract class
Specification → interface for abstract class	The interface for these classes corresponds to the specification level
When defining....	**You must ask yourself...**
An abstract class	What *interface* is needed to handle all of the *responsibilities* of this class?
Derived classes	Given this particular implementation (this variation), how can I implement it with the given specification?

The Bridge Pattern

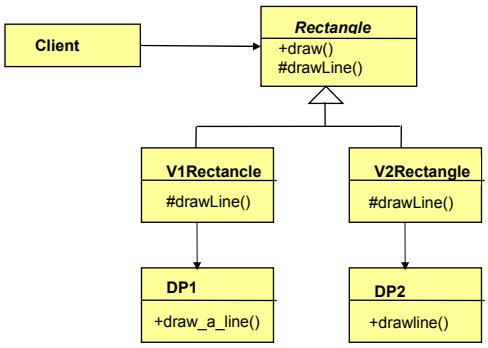
Motivating Example

- The task is to write a program that will draw rectangles with either of the two drawing programs, DP1 or DP2. When instantiating a rectangle it is known which drawing program to use.
- Rectangles are presented as two pairs of points. The differences are presented in the following table.

	DP1	DP2
to draw a line	draw_a_line(x1,y1,x2,y2)	drawline(x1,x2,y1,y2)
to draw a circle	draw_a_circle(x,y,r)	drawcircle(x,y,r)

- The client needs to be unaware of the type of the drawing program
- First idea for design: since the type of drawing program is told at the time of instantiating rectangles, we could have two types of rectangles, one that uses DP1 and other that uses DP2

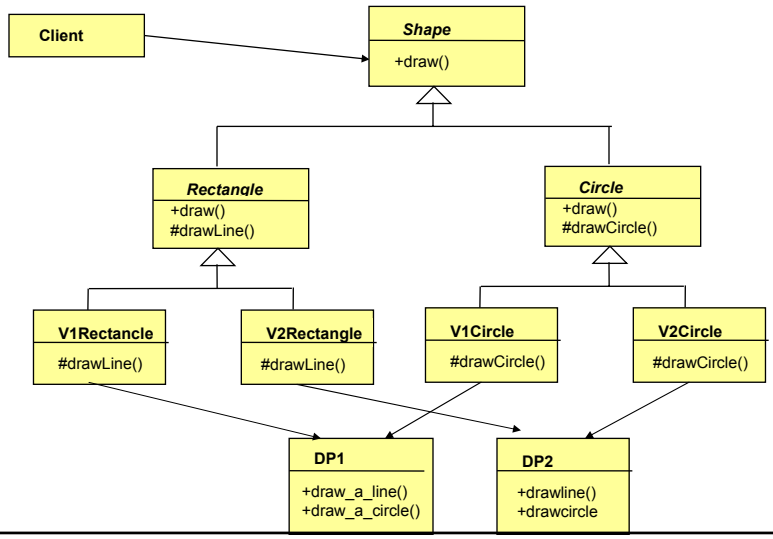
Motivating Example



- A straightforward design using inheritance, solves the problem
- … but requirements change (no surprise !!)
  - Also Circles must be supported

Motivating Example

- The design can be easily extended…
  - But the solution suffers (among other problems) from combinatorial explosion of classes



The Motivating Example

- Symptoms
  - there is redundancy
  - there is low cohesion
  - things are tightly coupled
  - would you want to maintain the code?
- But what is the *underlying problem* with the previous solution?
- The *abstraction* (kinds of shapes) and the *implementation* (drawing programs) are tightly coupled
  - each type of shape must know what type of drawing program it is using
  - two variabilities inside different commonalities are coupled via inheritance
- What we need is a way to separate variation in abstraction from variations in implementation so that the number of classes only grow linearly

→ this is what Bridge pattern does

The Motivating Example

- Let's apply commonality/variability thinking:
- The common concepts, commonalities, that will become abstract classes
  - Shapes, drawing programs

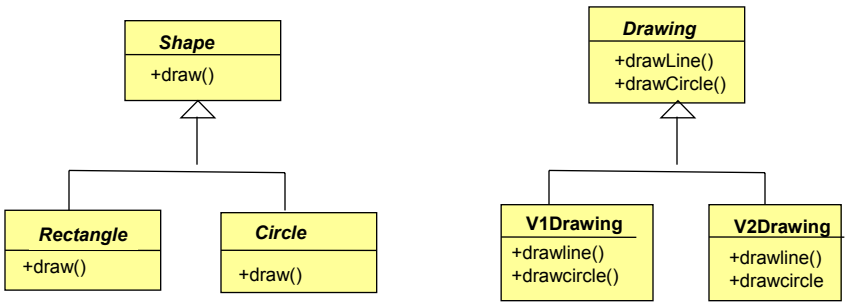
- at this point, shape only encapsulates the *concept* of shape that is responsible for knowing how to draw themselves
- Drawing programs in turn are responsible for drawing lines and circles

- Next step is to look for variability within each commonality
  - for Shape, there are rectangles and circles
  - for drawing programs, there are V1Drawing(based on DP1) and V2Drawing (based on DP2)
    - at this point we use concepts, not the concrete drawing programs

The Motivating Example

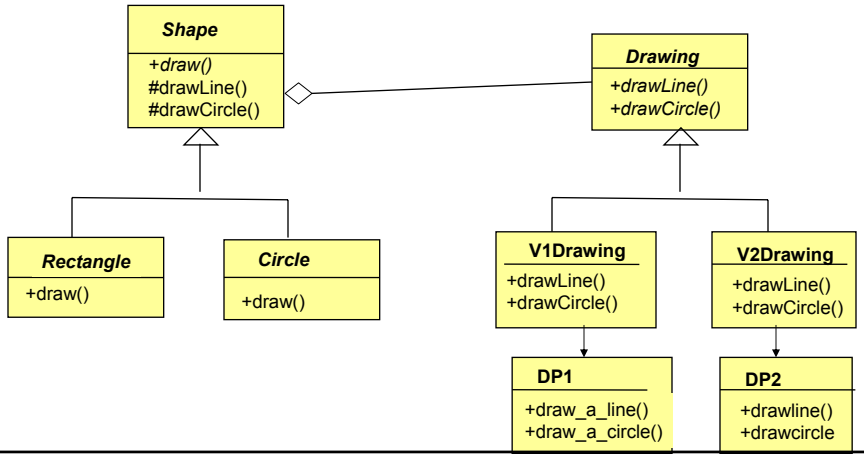
- By representing the variability with derived classes we get ...

- Remember: favor composition over inheritance
- Encapsulate behavior (implementation) with composition
- ... so we must only decide which uses the other



The Motivating Example

- Now we have separated Shape abstraction from Drawing implementation in a way that the two can vary independently
  - (the two protected methods in *Shape* are to obey the ”Once and only once” – rule)
- note that there is an adapter integrated with the bridge. It is not part of the bridge pattern.



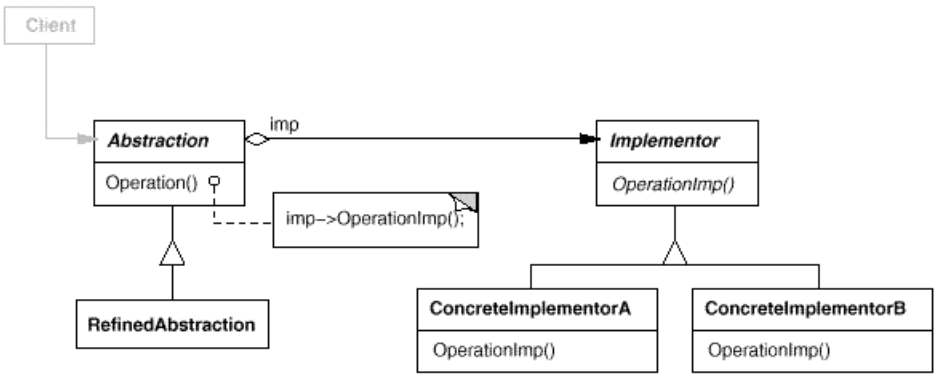
Basic Aspects of Bridge Pattern

- Intent
  - decouple an abstraction from its implementation
  - allow implementation to vary independently from its abstraction
  - abstraction defines and implements the interface
    - all operations in abstraction call methods from its implementation object
- In the Bridge pattern ...
  - ... an abstraction can use different implementations
  - ... an implementation can be used in different abstractions

Applicability

- Avoid permanent binding between an abstraction and its implementation
- Abstractions and their implementations should be *independently extensible* by subclassing
- Hide the implementation of an abstraction completely from clients
  - their code should not have to be recompiled when implementation changes
- Share an implementation among multiple objects
  - and this fact should be hidden from the client

Structure



Participants

- Abstraction
  - defines the abstraction's interface
  - maintains a reference to an object of type Implementor
- Implementor
  - defines the interface for implementation classes
    - does not necessarily correspond to the Abstraction's interface
    - Implementor contains primitive operations,
    - Abstraction defines the higher-level operations based on these primitives
- RefinedAbstraction
  - extends the interface defines by Abstraction
- ConcreteImplementer
  - implements the Implementor interface, defining a concrete impl.

Consequences

- Decoupling interface and implementation
  - implementation **configurable** and **changeable** at run-time
  - reduce compile-time dependencies
    - implementation changes do not require Abstraction to recompile
- Improved extensibility
  - extend by subclassing independently Abstractions and Implementations
- Hiding implementation details from clients
  - shield clients from implementations details
    - e.g. sharing implementor objects together with reference counting

Implementation

- Only one Implementor
  - not necessary to create an abstract implementor class
  - degenerate, but useful due to decoupling
- Which Implementor should I use ?
  - Variant 1: let Abstraction know all concrete implementors and choose
  - Variant 2: choose initially default implementor and change later
  - Variant 3: use an Abstract Factory
    - no coupling between Abstraction and concrete implementor classes

A few notes

- Adapter makes things work after they're designed; Bridge makes them work before they are. [GOF, p219]
- Bridge is designed up-front to let the abstraction and the implementation vary independently. Adapter is retrofitted to make unrelated classes work together. [GOF, 216]
- State, Strategy, Bridge (and to some degree Adapter) have similar solution structures. They all share elements of the "handle/body" idiom [Coplien, *Advanced C++*, p58]. They differ in intent - that is, they solve different problems.
- The Bridge is not perfect: you may get a new shape that cannot be implemented with the implementations already in place, e.g. an ellipse. This requires changes to implementation, however these changes are fairly well localized
- *The Bottom Line: Patterns do not give perfect solutions. However, chances are good that they provide better solutions that you or I might come up with on our own.*