Introduction to JHotDraw Framework (SB5-MAI)

Jan Corfixen Sørensen

University of Southern Denmark

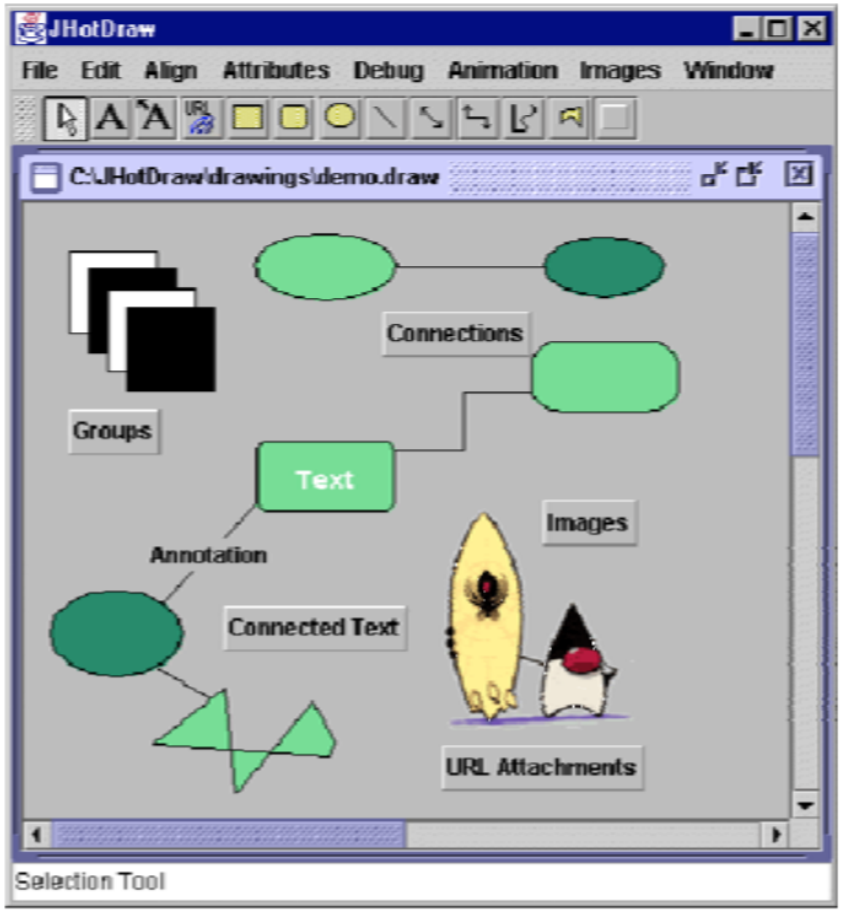
August 30, 2017

Outline

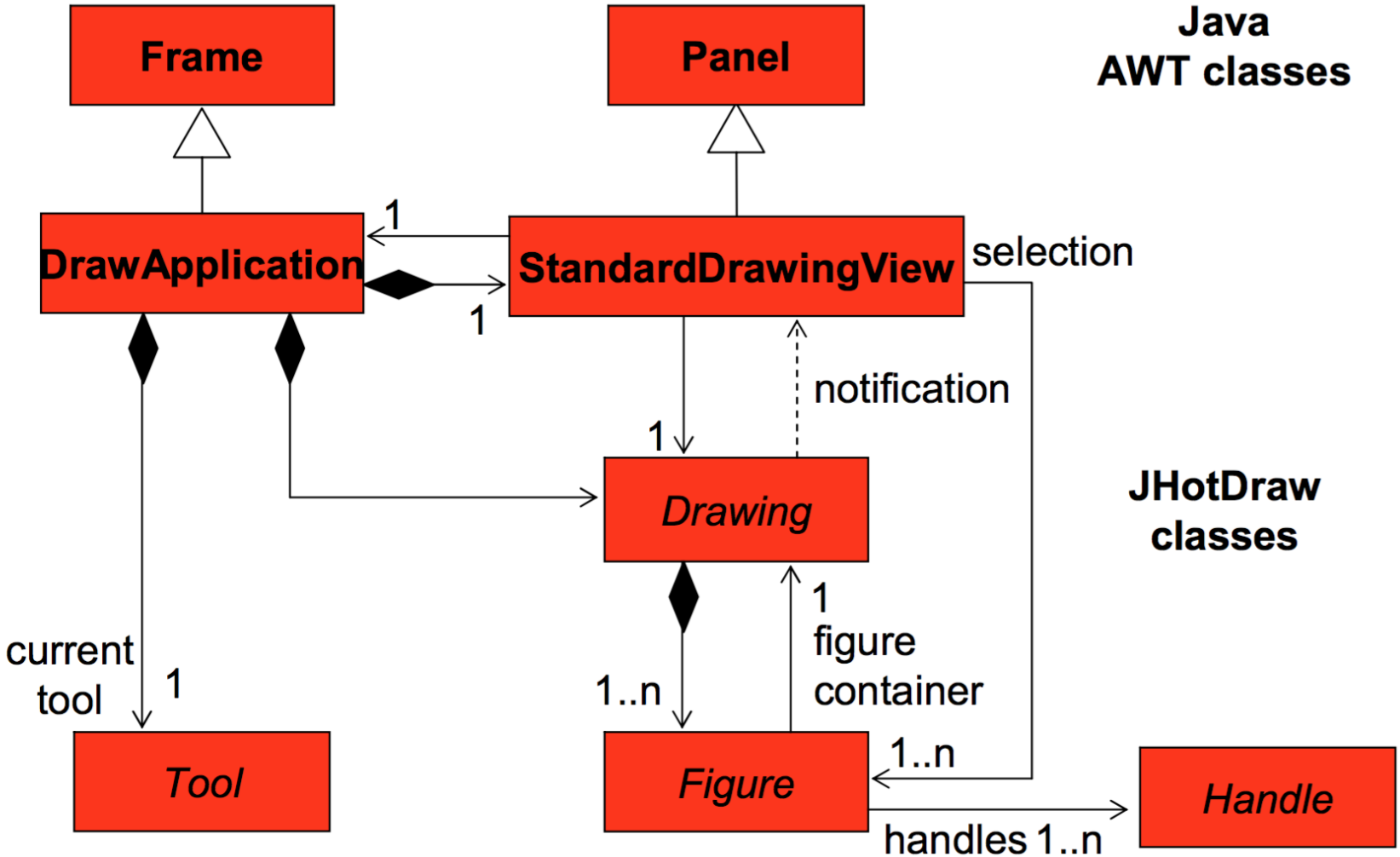
JHotDraw

The JHotDraw Framework

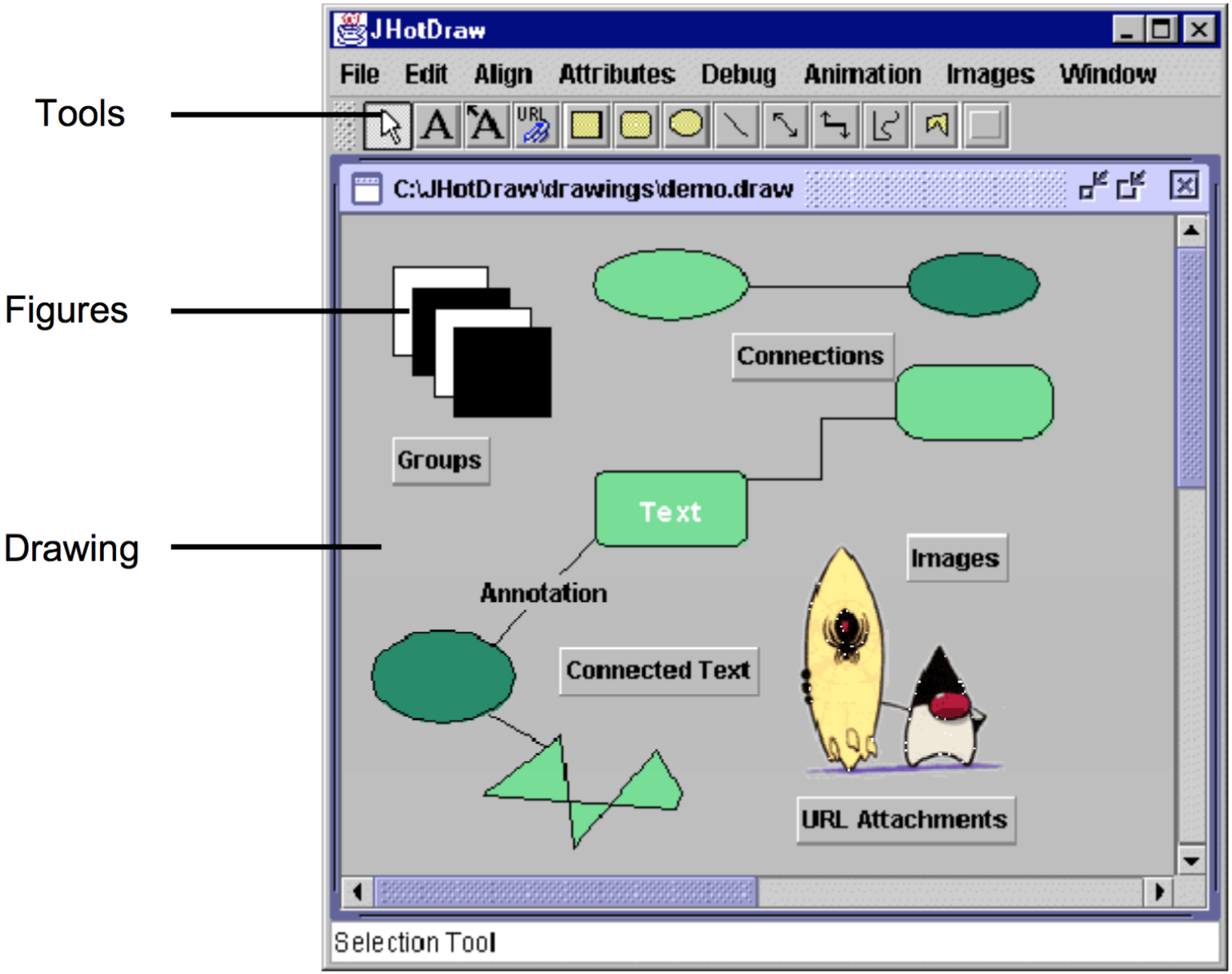
- The JHotDraw framework is targeted at applications for drawing technical and structured graphics – such as network layouts and Gantt diagrams.
- Originally developed in Smalltalk by Kent Beck and Ward Cunningham.
- HotDraw was one of the first software development projects explicitly designed for reuse and labelled a framework.



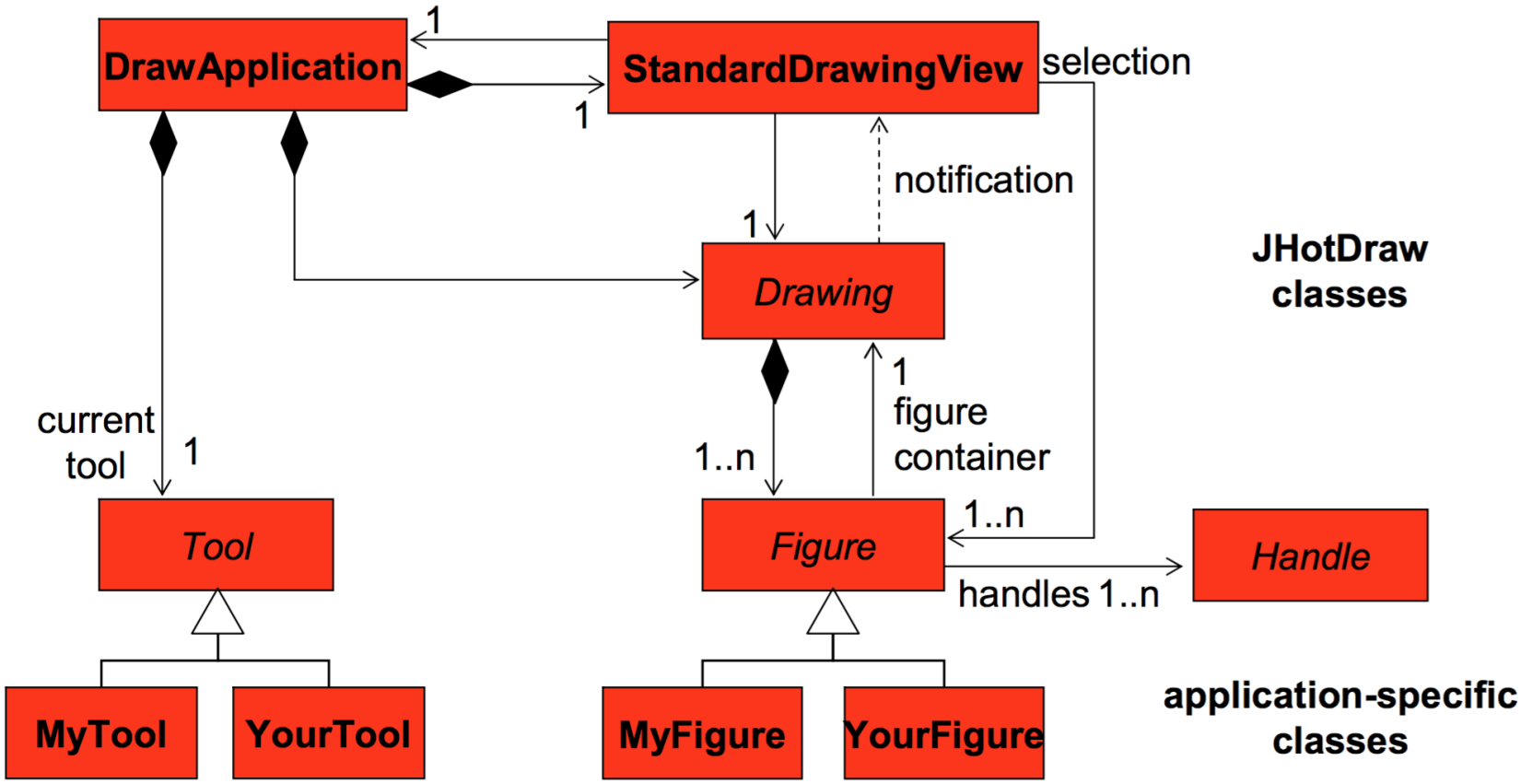
Understanding JHotDraw (1)



Understanding JHotDraw (2)



Using the Framework



GoF Design Patterns

Principles

- Emphasis on flexibility and reuse through decoupling of classes.

- The underlying principles:
  - program to an interface, not to an implementation.
  - favor composition over class inheritance.
  - find what varies and encapsulate it.

General Categories

- 23 patterns are divided into three separate categories:
  - **Creational patterns:** Deal with initializing and configuring classes and objects.
  - **Structural patterns:** Deal with decoupling interface and implementation of classes and objects.
  - **Behavioral patterns:** Deal with dynamic interactions among societies of classes and objects.

Purpose and Scope

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

Creational Patterns

- **Class**
  - **Factory Method:** Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory method lets a class defer instantiation to subclasses.
- **Object**
  - **Abstract Factory:** Provide an interface for creating families of related or dependent objects without specifying their concrete class.
  - **Builder:** Separate the construction of a complex object from its representation so that the same construction process can create different representations.
  - **Prototype:** Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
  - **Singleton:** Ensure a class only has one instance, and provide a global point of access to it.

Structural Patterns

Class/Object

- **Adapter:** Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Object

- **Bridge:** Decouple an abstraction from its implementation so that the two can vary independently.
- **Composite:** Compose objects into tree structures to represent whole- part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- **Decorator:** Attach additional responsibilities to an object dynamically.
- **Facade:** Provide a unified interface to a set of interfaces in a subsystem.
- **Flyweight:** Use sharing to support large numbers of fine-grained objects efficiently.
- **Proxy:** Provide a surrogate or placeholder for another object to control access to it.

Behavioral Patterns

Class

- **Interpreter:** Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
- **Template Method:** Define the skeleton of an algorithm in an operation, deferring some steps to subclasses; lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Object

- **Chain of Responsibility:** Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
- **Command:** Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
- **Iterator:** Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- **Mediator:** Define an object that encapsulates how a set of objects interact.

JHotDraw Design Patterns

Design Patterns in JHotDraw

JHotDraw makes extensive use of Design Patterns [GHJV95]:

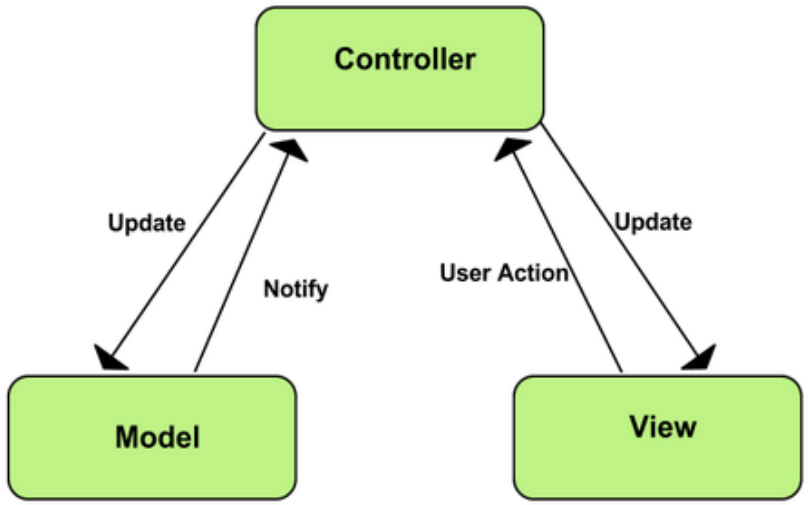
- Model-View-Controller
- Composite
- Strategy
- State
- Template Method
- Decorator
- Factory Method
- Prototype

General observation: Extensibility of frameworks calls for extensive use of design patterns.

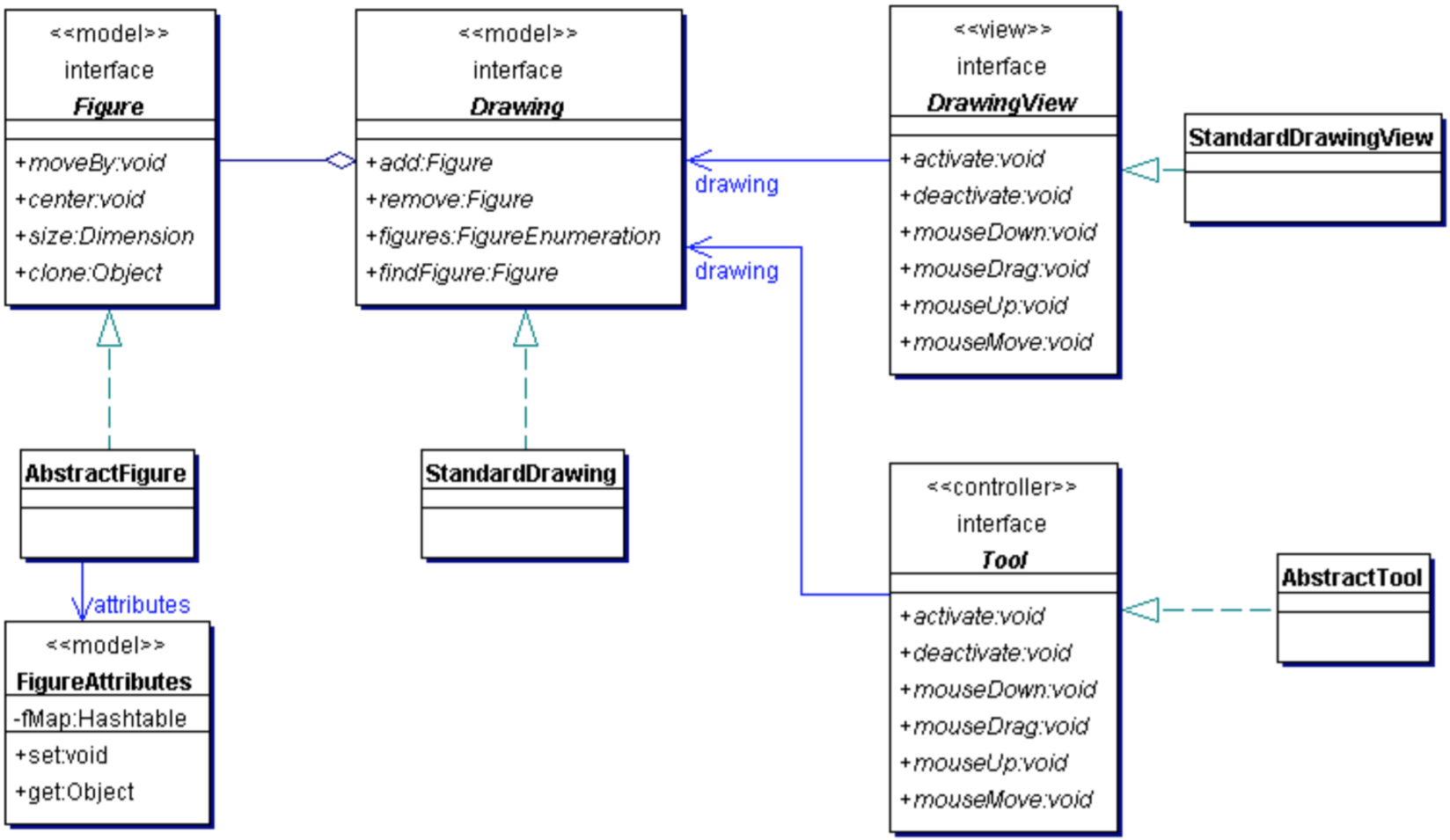
Model View Controller (1)

Model View Controller Pattern

- Model
  - Figures (and their attributes: FillColor, Position)
  - Drawing (figure container)
- View
  - DrawingView (clipping view of a window)
  - DrawingWindow
- Controller
  - Tools to manipulate the model



Model View Controller (2)

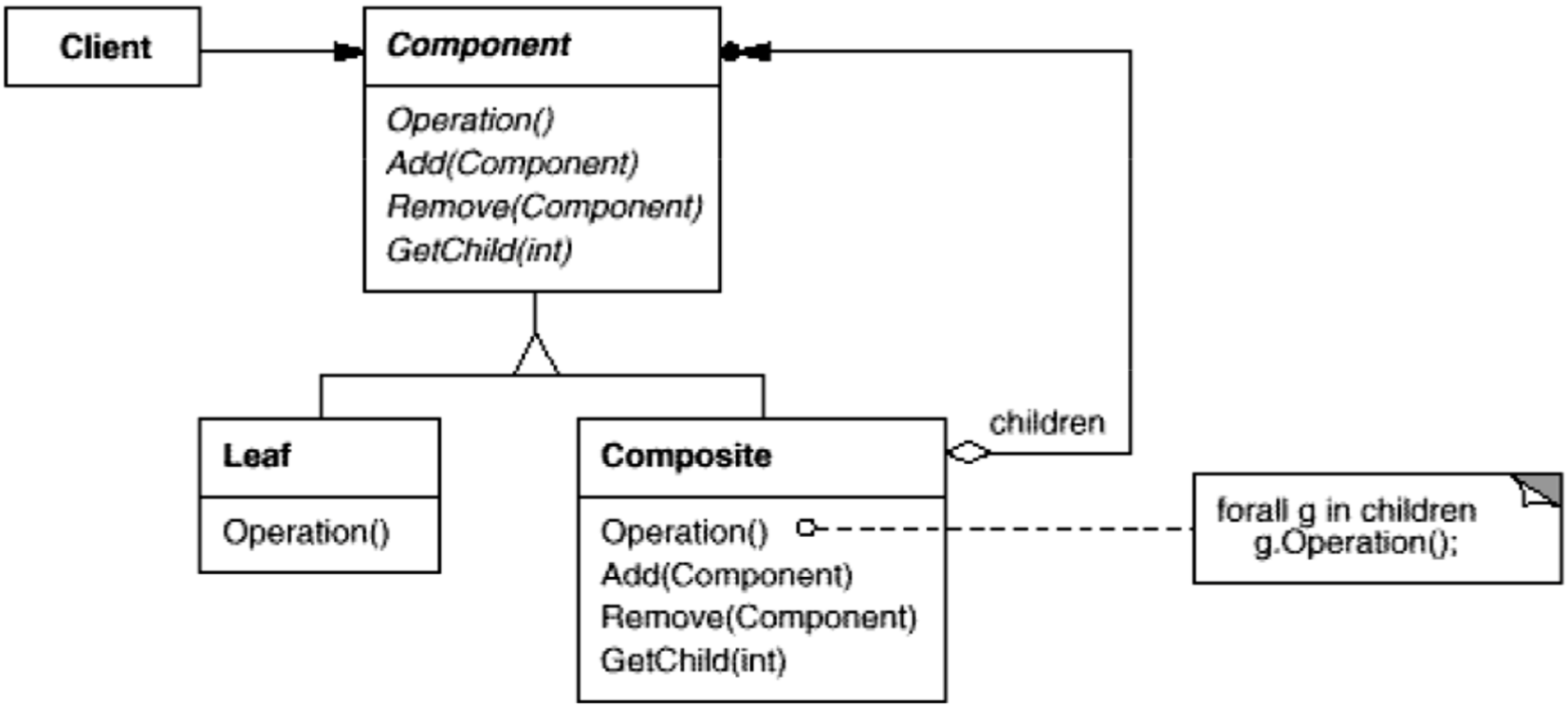


Composite

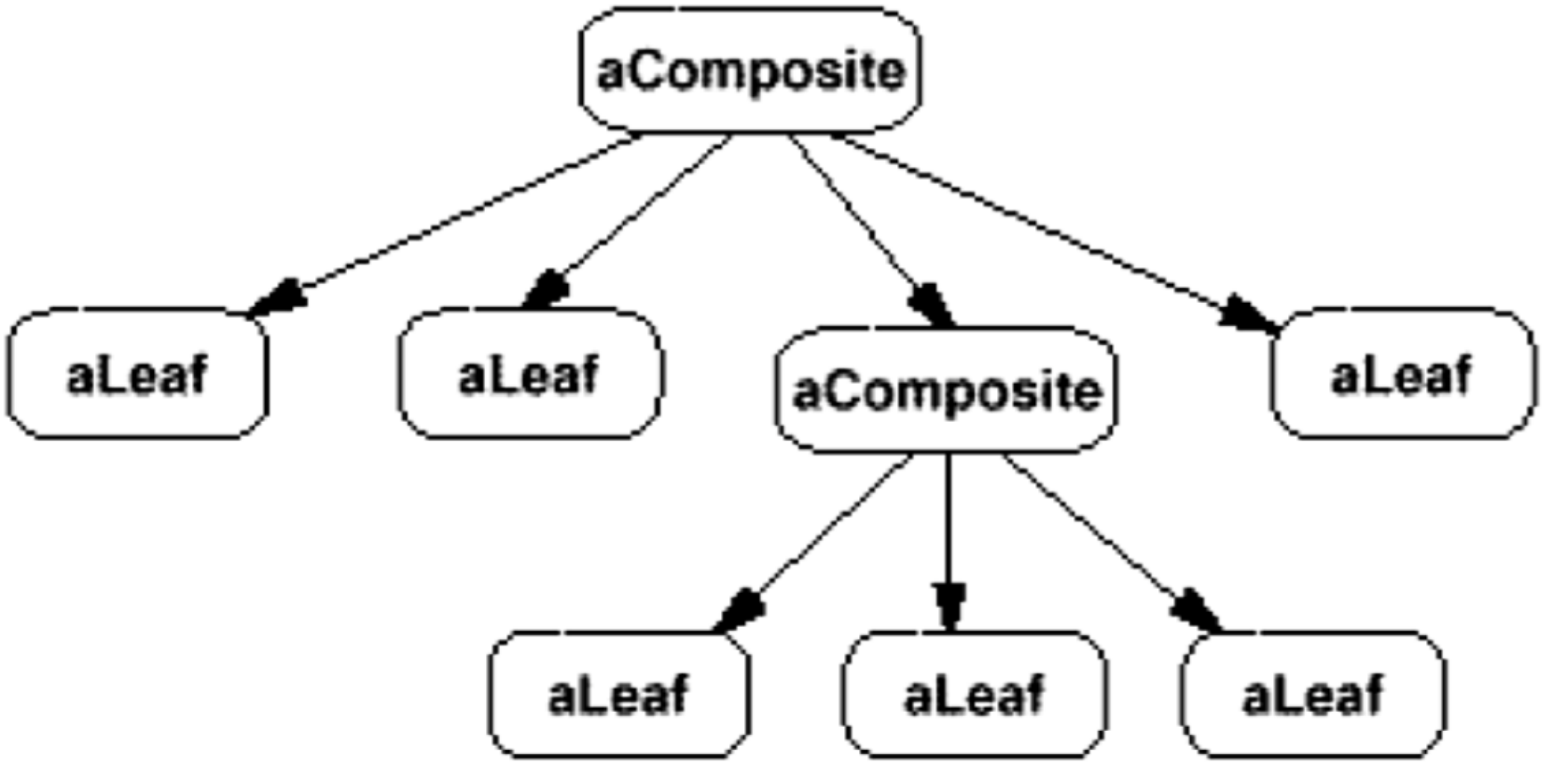
- **Intent:** Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

- **Use when:**
  - you want to represent whole-part- hierarchies of objects.
  - you want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.

Composite Structure



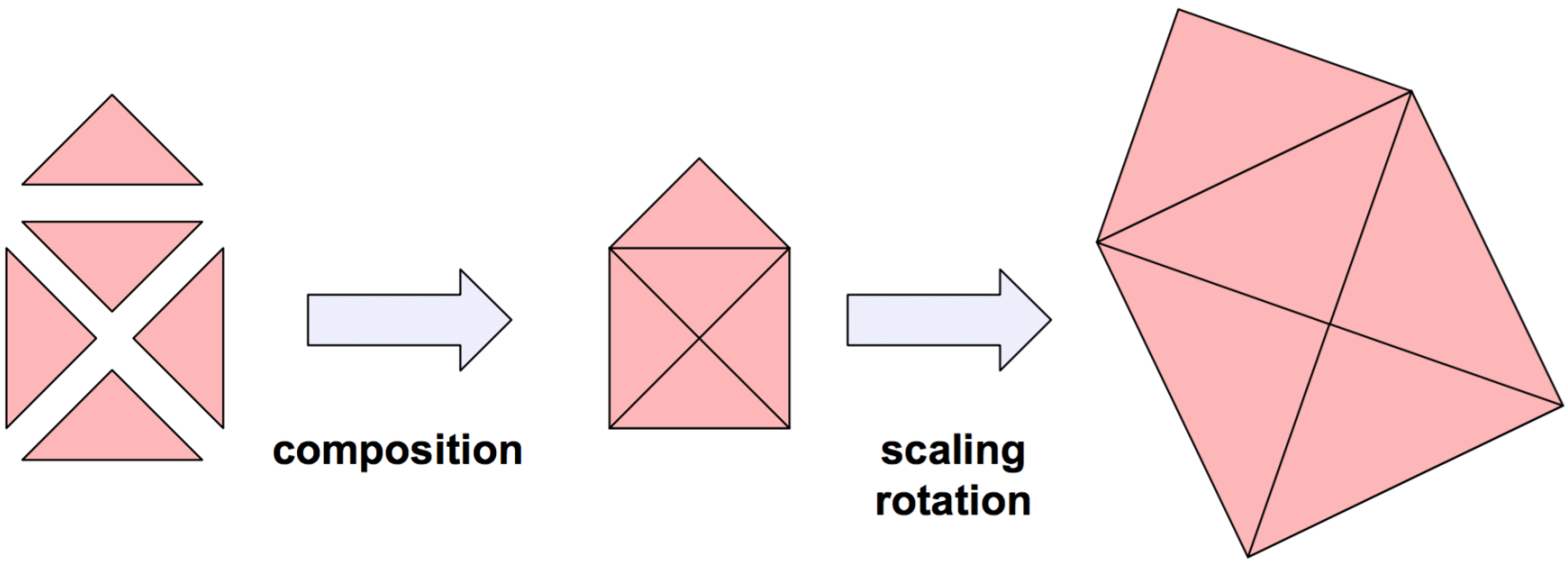
Composite Object Structure



JHotDraw Composite (1)

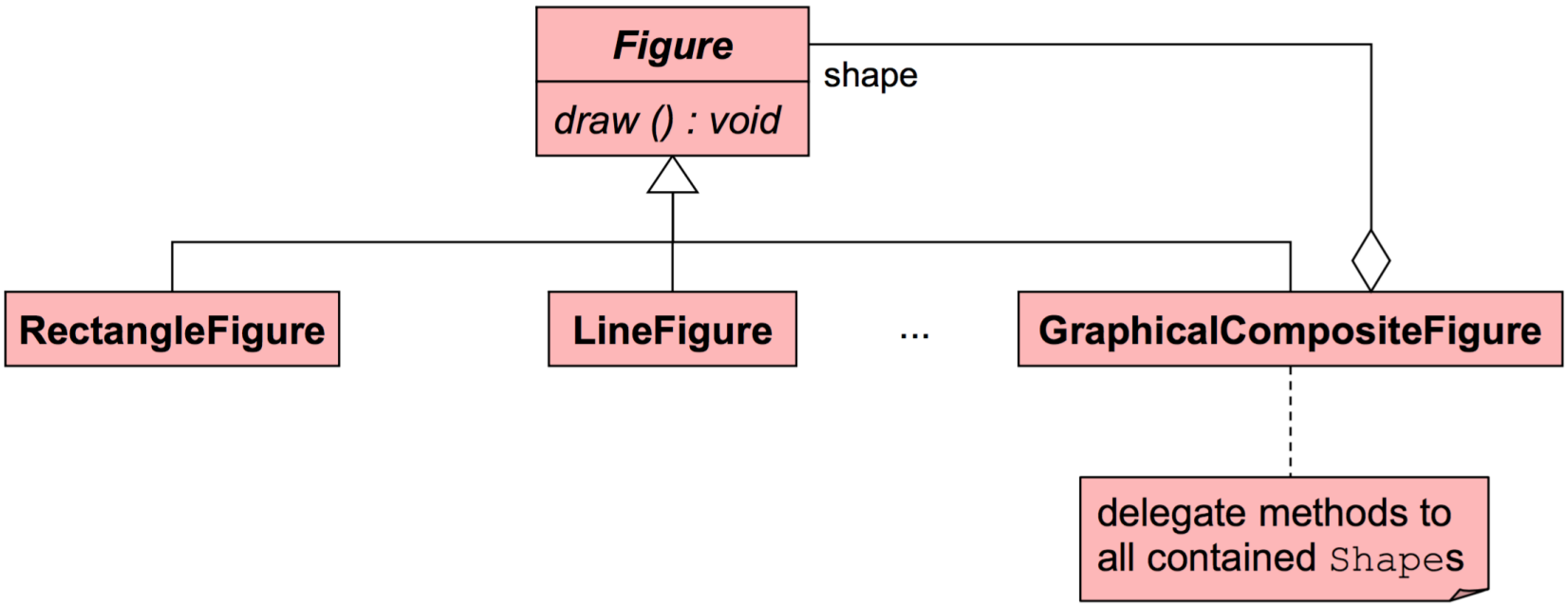
A figure is composed of several figures. Composite figure as well as child figures are treated uniformly. Operations performed on either of the figures will exert a common behavior on all figures (scale, rotation, move).

Example: A composite figure consisting of 5 subordinate figures.



JHotDraw Composite (2)

- Allow nested structures of arbitrary depth.
- Uniformly handle all objects forming a composite.



Composite Consequences

Pros:

- Wherever client code expects a primitive object, it can also take a composite object.
- Makes the client simple. Clients can treat composite structures and individual objects uniformly, and this simplifies their code.
- Makes it easier to add new kinds of components. Clients don't have to be changed for new Component classes.

Cons:

- Can make your design overly general. It makes it harder to restrict the components of a composite.
  - If you want a composite to have only certain components, you can't rely on the type system to enforce those constraints for you. You'll have to use run-time checks instead.

Strategy

Intent:

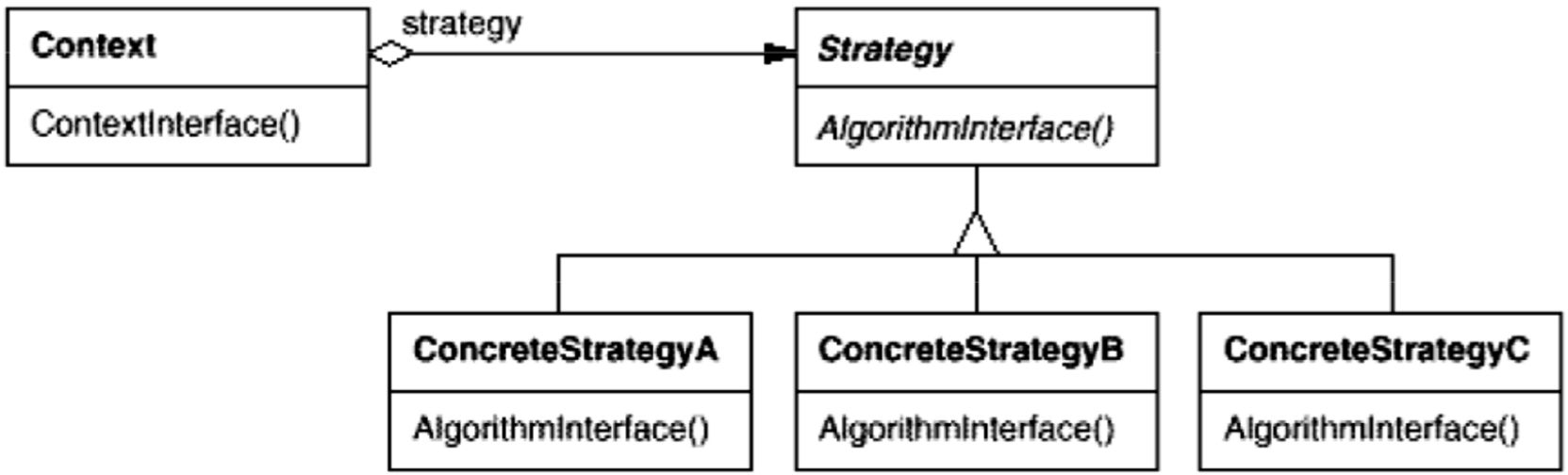
- Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Strategy Applicability

Use when:

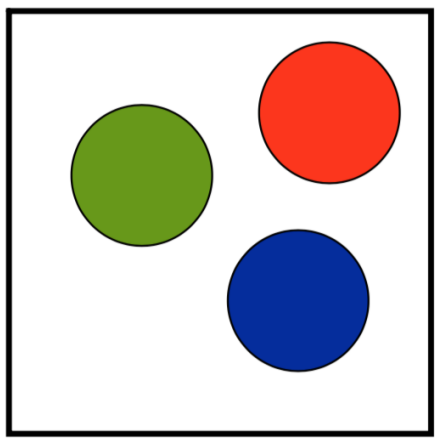
- Many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.
- You need different variants of an algorithm. For example, you might define algorithms reflecting different space/time trade-offs.
- An algorithm uses data that clients shouldn't know about. Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures.
- A class defines many behaviors, and these appear as multiple conditional statements in its operations.

Strategy Structure

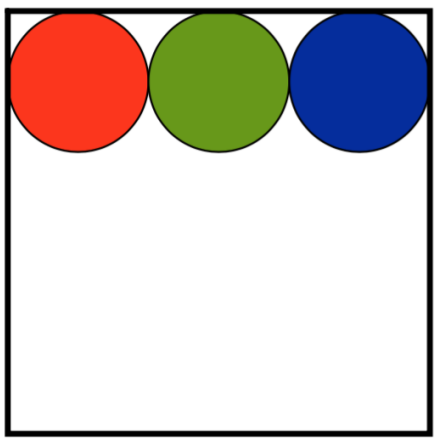


JHotDraw Strategy

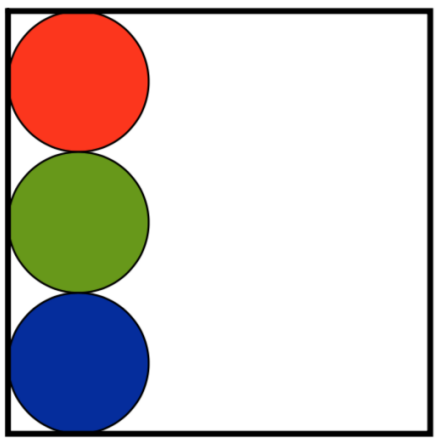
- It has advantages to separate layout algorithms from objects to be laid out (see also Java AWT, Swing). Here, the Strategy Pattern is used.
- Every layouter (Java Swing / AWT: layout manager) can be attached to a composite figure rendering it.



A composite figure with no layout manager



A composite figure with an “align to top” layout manager



A composite figure with an “align to left” layout manager

Strategy Consequences

Pros:

- Families of related algorithms.
- An alternative to subclassing.
- Strategies eliminate conditional statements.
- A choice of implementations. Strategies can provide different implementations of the same behavior. The client can choose among strategies with different time and space trade-offs.

Cons:

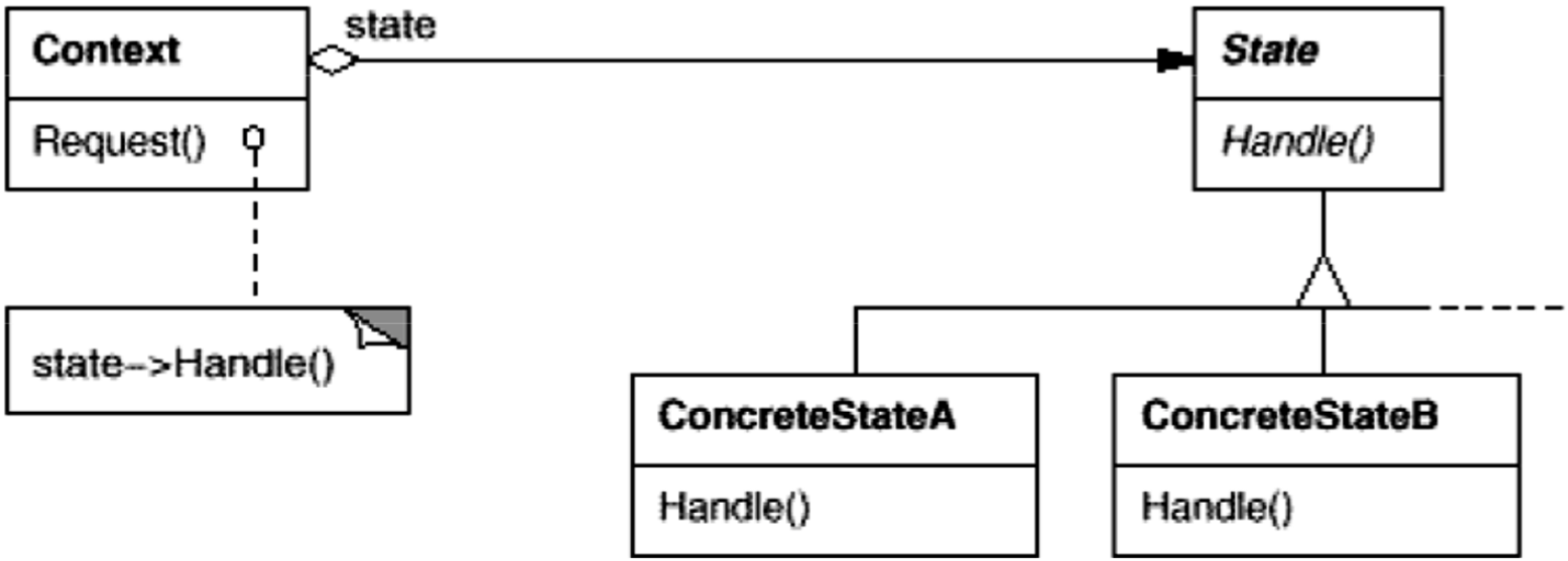
- Clients must be aware of different Strategies.
- Communication overhead between Strategy and Context.
- Increased number of objects.

State

- **Intent:**
  - Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

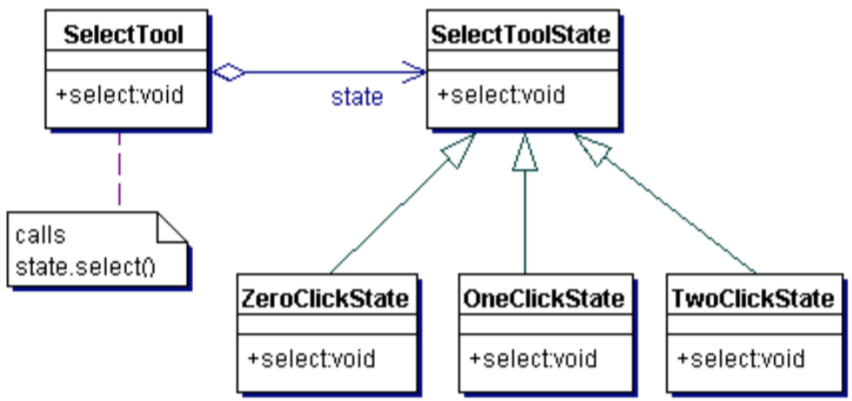
- **Use when:**
  - An object's behavior depends on its state, and it must change its behavior at run-time depending on that state.
  - Operations have large, multipart conditional statements that depend on the object's state.

State Structure



JHotDraw State

- **Goal:** Externalize the state of a tool. A tool can subdivide its state. This is necessary to provide means to make a tool operate in different modes (states).
- **Example:** A drawing tool's operation may differ whether the tool has already been used before and is obvious for toggling tools:
  - zoom tool: toggle between "zoom in" and "zoom out"
  - selection tool: toggle between "select border" and "select text"



E.g., text field:

•set the cursor with 1 click

•select a word with 2 clicks

•select the whole line with 3 clicks

State Consequences

Pros:

- It localizes state-specific behavior and partitions behavior for different states. New states and transitions can be added easily by defining new subclasses.
- It makes state transitions explicit.
- State objects can be shared.

Template Method

Intents:

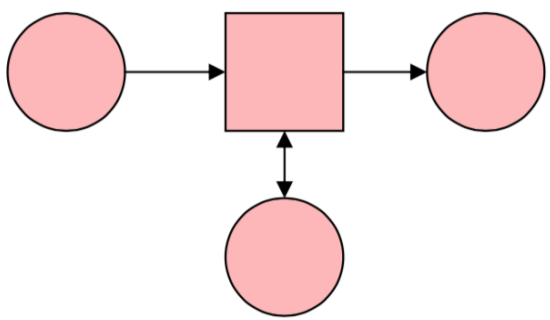
- Define the skeleton of an algorithm in an operation, deferring some steps to subclasses; lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure

JHotDraw Template Method

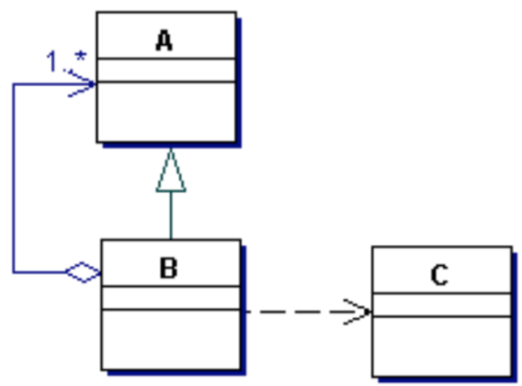
Goal: Define the skeleton of an algorithm in the framework, deferring application-specific steps or additions to the application-specific classes. From the framework's point of view, connecting figures via a line is always the same operation. From the application's point of view, it is quite a difference depending on the semantics of the connection. To join these two views, the Template Method pattern is used.

The framework's algorithm for connecting figures (its invariant parts) are placed in one class (LineConnection), exposing the variant parts of the algorithm in empty methods that subclasses can overwrite.

Example:



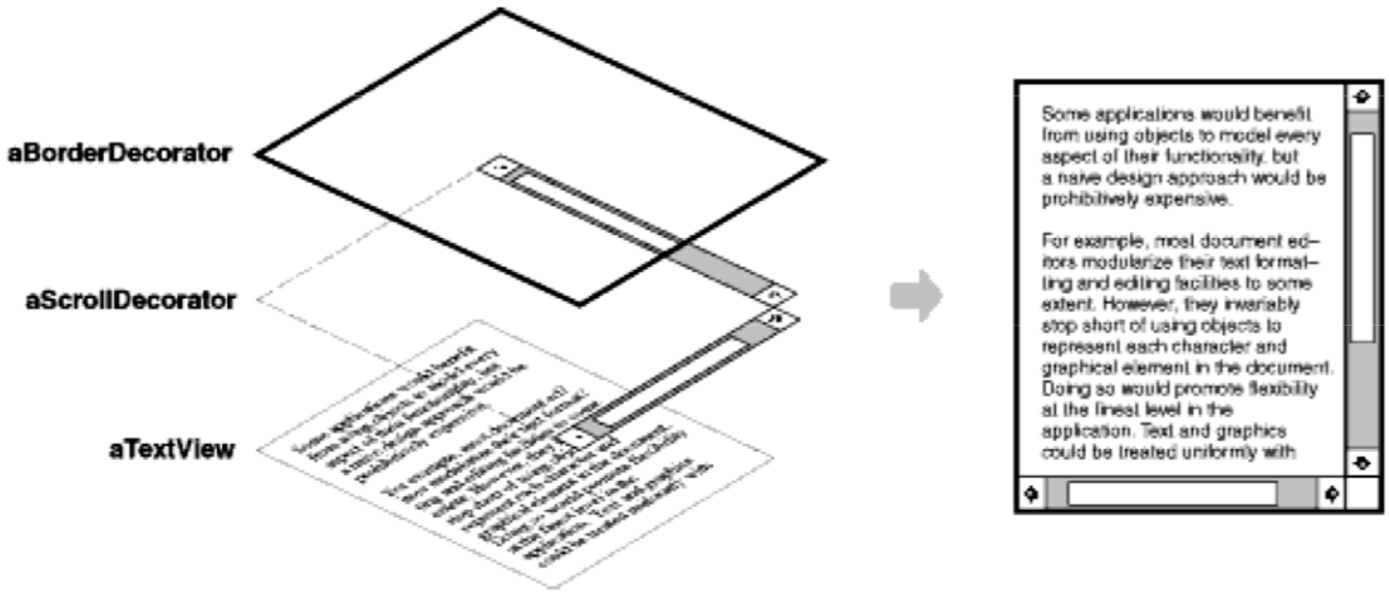
Petri net connections

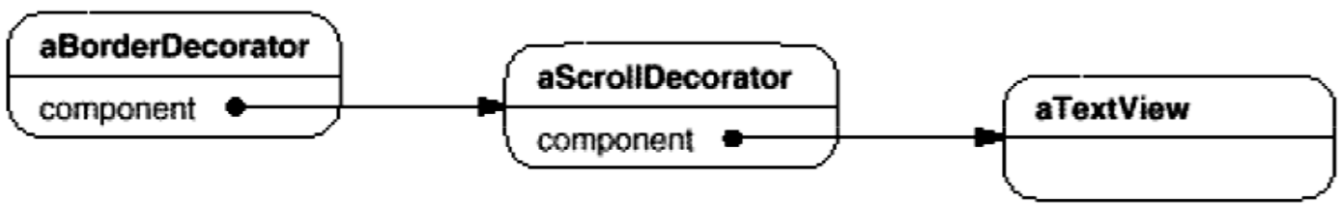


Class diagram connections

Decorator

- **Intent:** Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

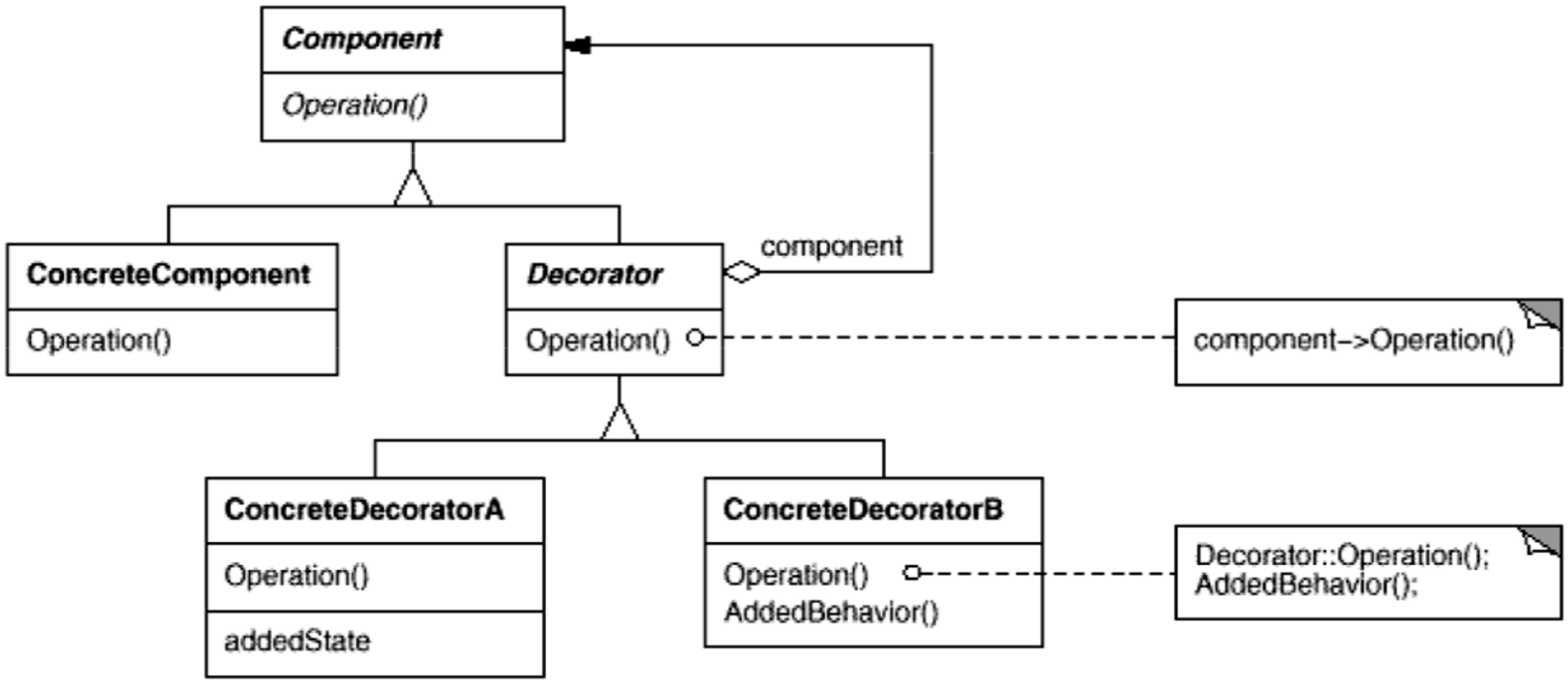




Decorator Use

- To add responsibilities to individual objects dynamically and transparently, that is, without affecting other objects.
- For responsibilities that can be withdrawn.
- When extension by subclassing is impractical. Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses.

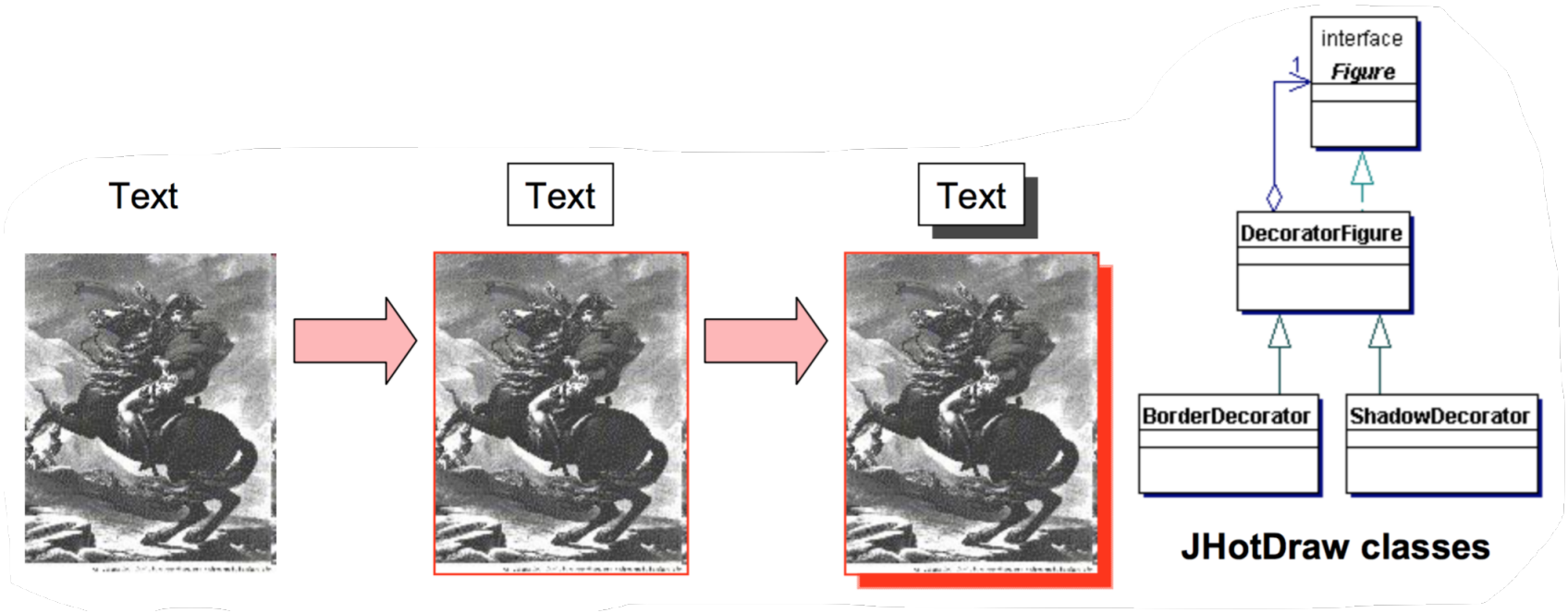
Decorator Structure



JHotDraw Decorator

Goal: Add specific visualization to generic visualization.

Example: Add borders and shadows to a figure.



Note: A problem of the decorator pattern is that the type of the decorated object is difficult to obtain if:

- it is not made public by access methods like getDecoratedType() (the Java instanceof operator does not work) and
- no separate list of all figures (including the decorated) exists.

Decorator Consequences

Pros:

- More flexibility than static inheritance.
- Avoids feature-laden classes high up in the hierarchy.

Cons:

- A decorator and its component aren't identical.
- Lots of little objects.

Factory Method

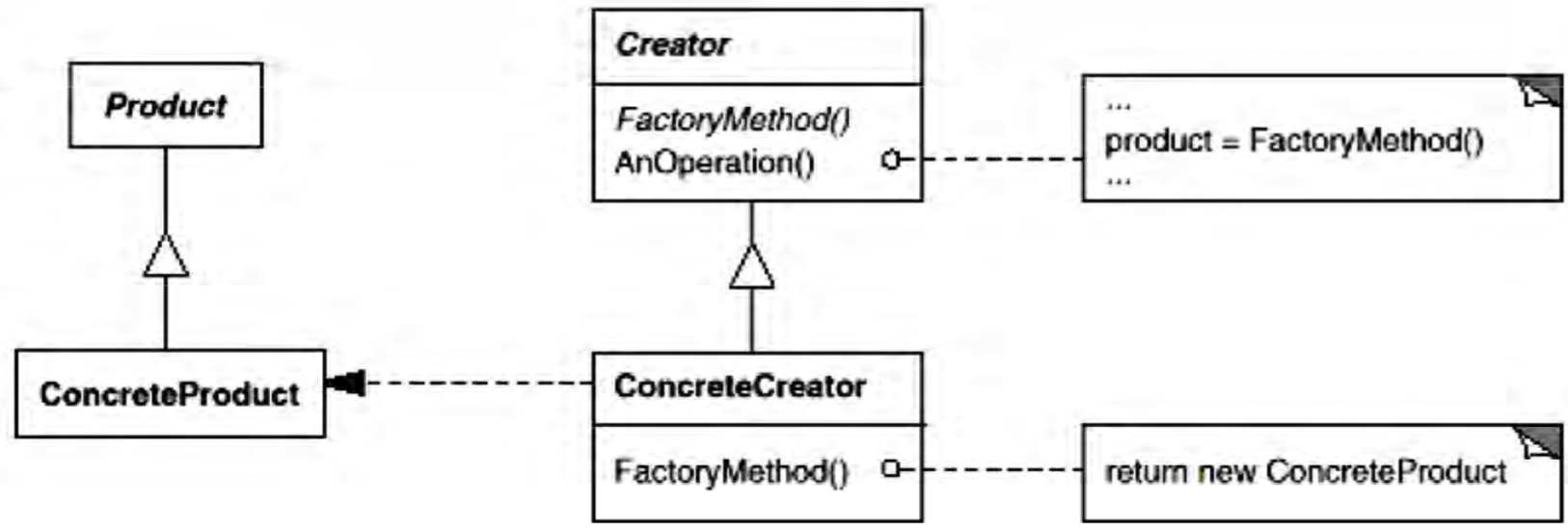
Intent:

- Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Use when:

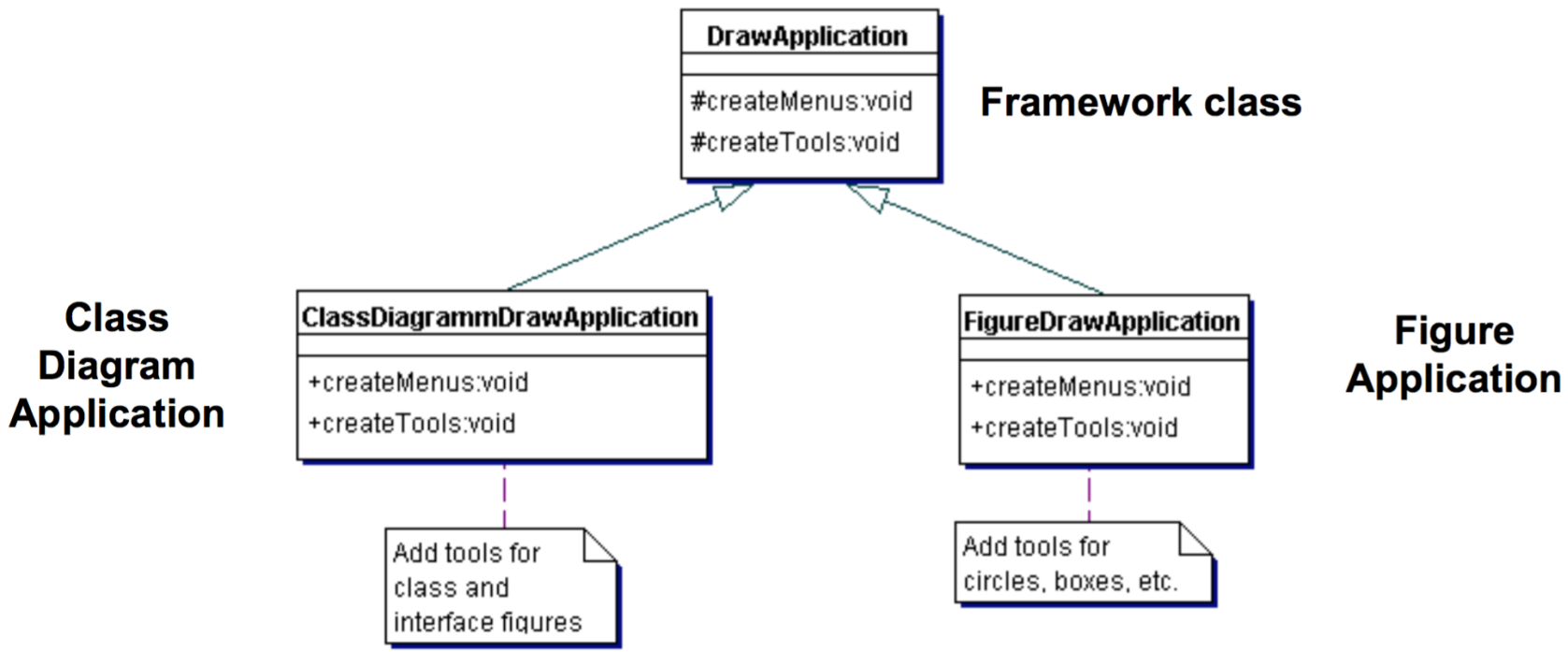
- a class can't anticipate the class of objects it must create.
- a class wants its subclasses to specify the objects it creates.
- classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

Factory Method Structure



JHotDraw Factory Method

Goal: Abstract from the concrete classes to be instantiated, which helps creation of customized components. This is used in JHotDraw to keep menus and tool structure flexible. Factory methods in the `DrawApplication` class are called `createMenus()` and `createTools()`. The customized application class (`MyDrawApplication`) inherits from `DrawApplication` and overwrites the `createMenus()` and `createTools()` methods.



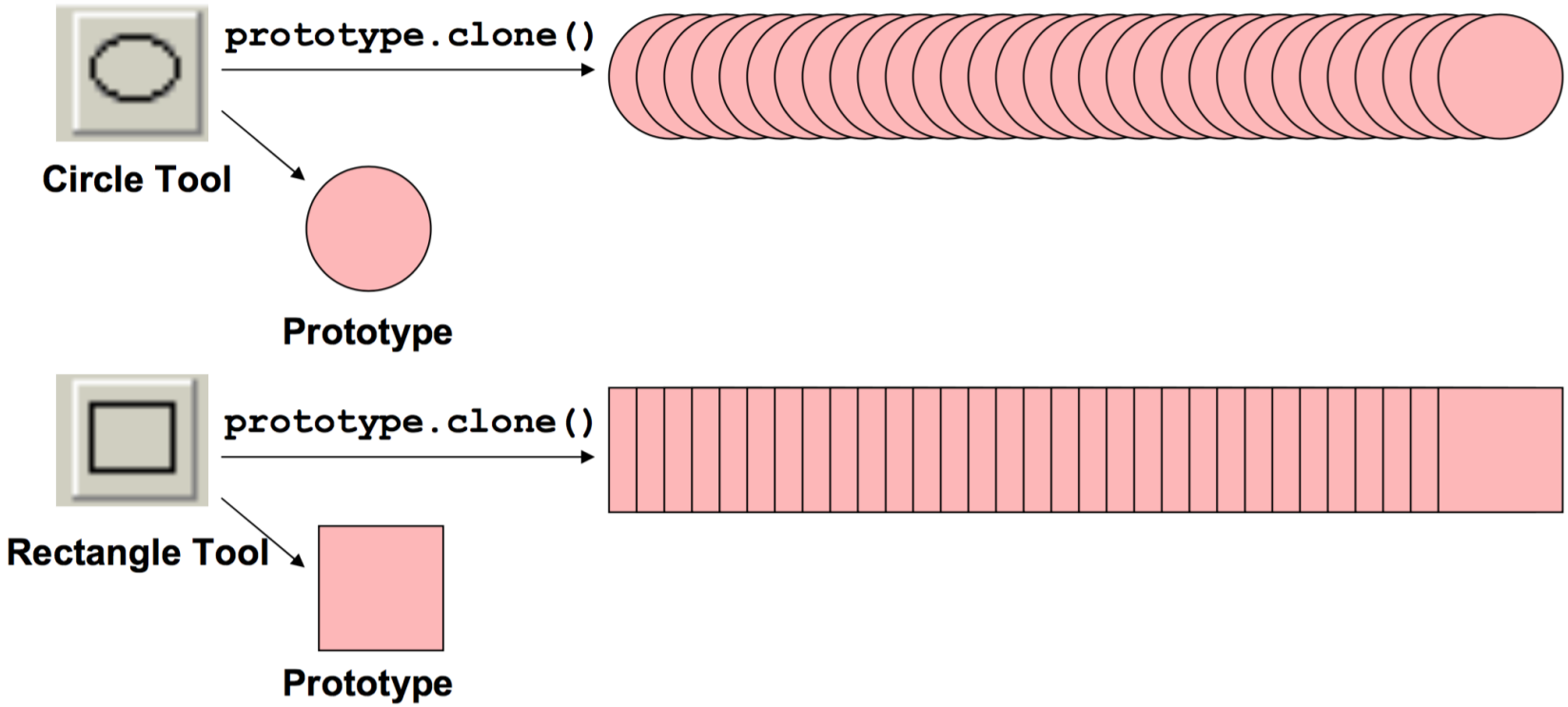
Factory Method Consequences

- It provides hooks for the subclasses.
- It connects parallel class hierarchies.

Prototype

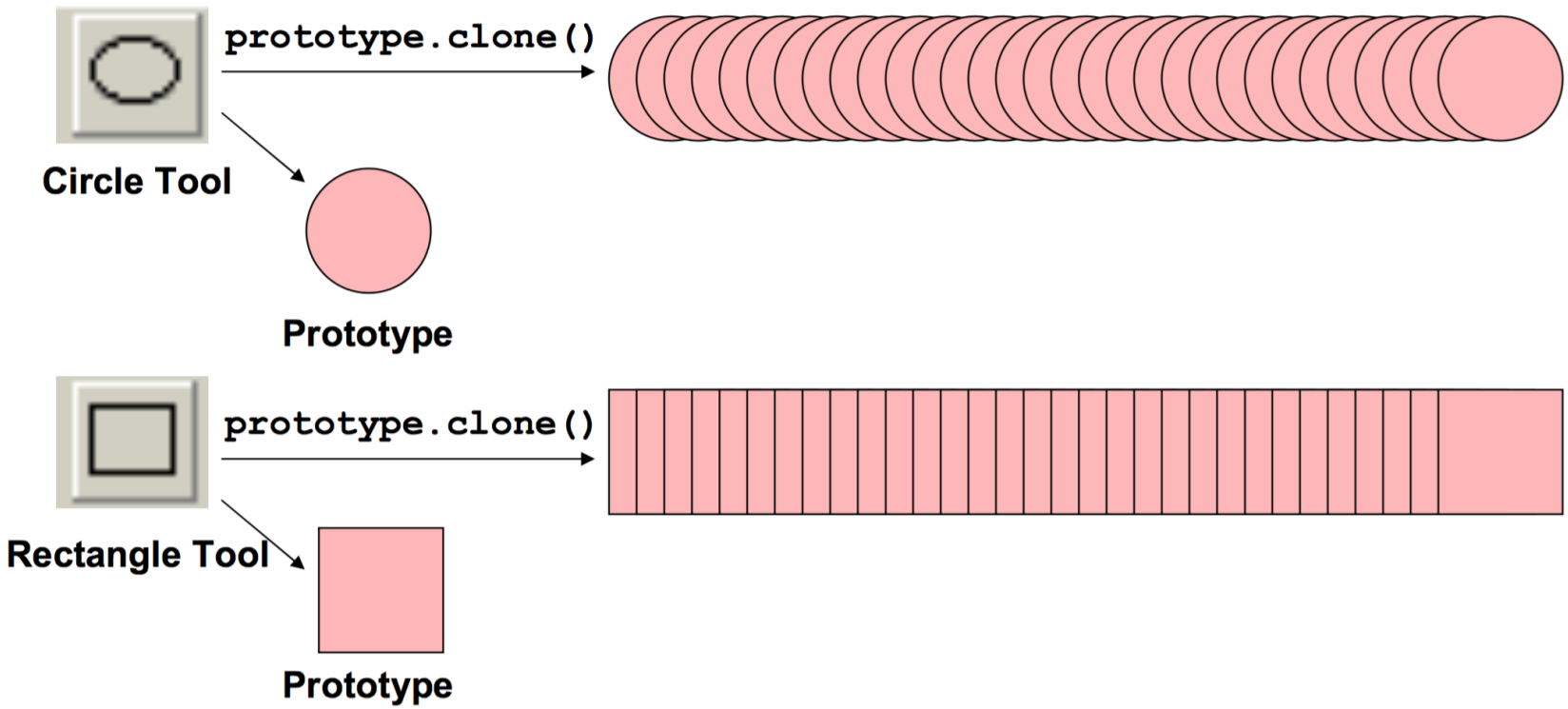
- **Intent:** Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
- **Use when:**
  - the classes to instantiate are specified at run-time, for example, by dynamic loading.
  - building a class hierarchy of factories that parallels the class hierarchy of products should be avoided.
  - instances of a class can have one of only a few different combinations of state.
    - It may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually.

Prototype Structure



JHotDraw Prototype

Each tool is initialized with an instance (a prototype) of the figure it is meant to create. When creating a new figure, the tool clones the prototype.



Prototype Consequences

- It hides the concrete product classes from the clients, thereby reducing the number of names clients know about.
- It lets a client work with application-specific classes without modification.
- It lets you add and remove products at run-time.
- It lets you specify new objects by varying values.

Other Design Patterns