# Software Design Patterns

CS134

Chris Pollett

Sep. 13, 2004

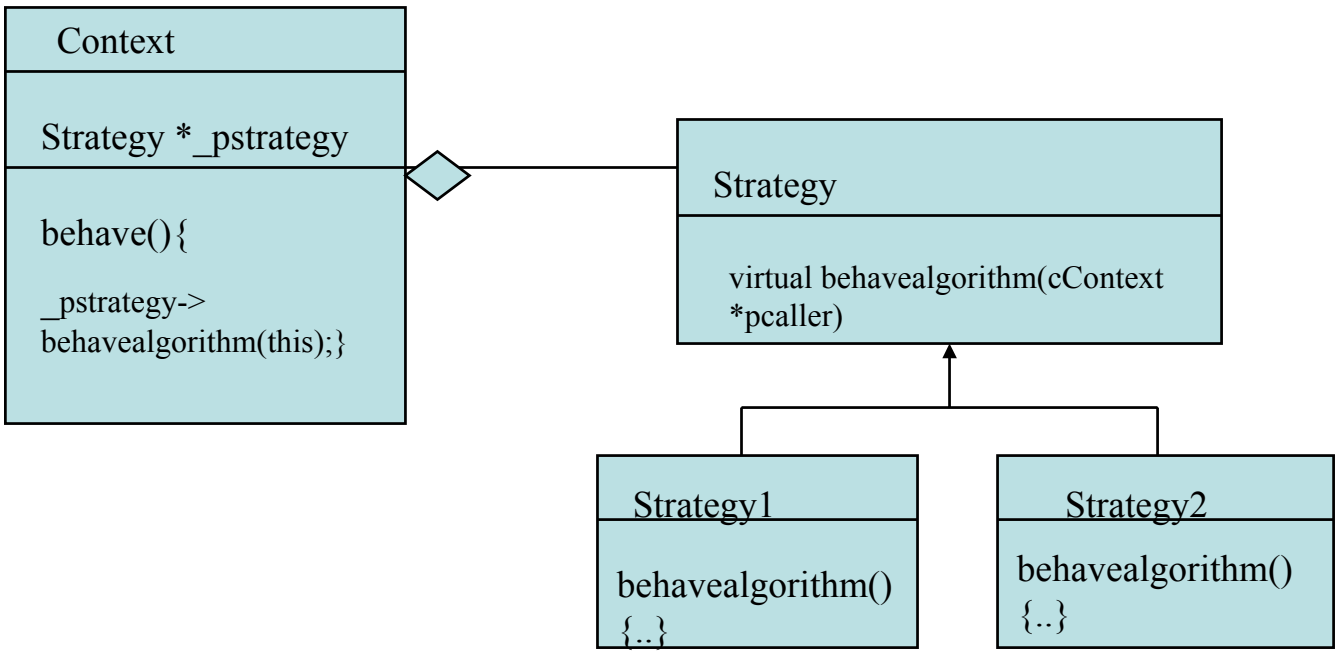
# Introduction

We will discuss the following patterns today:

- Strategy
- Template Method
- Command
- Composite
- Singleton
- Bridge
- Document-View

# Strategy

Is related to delegation mentioned before:



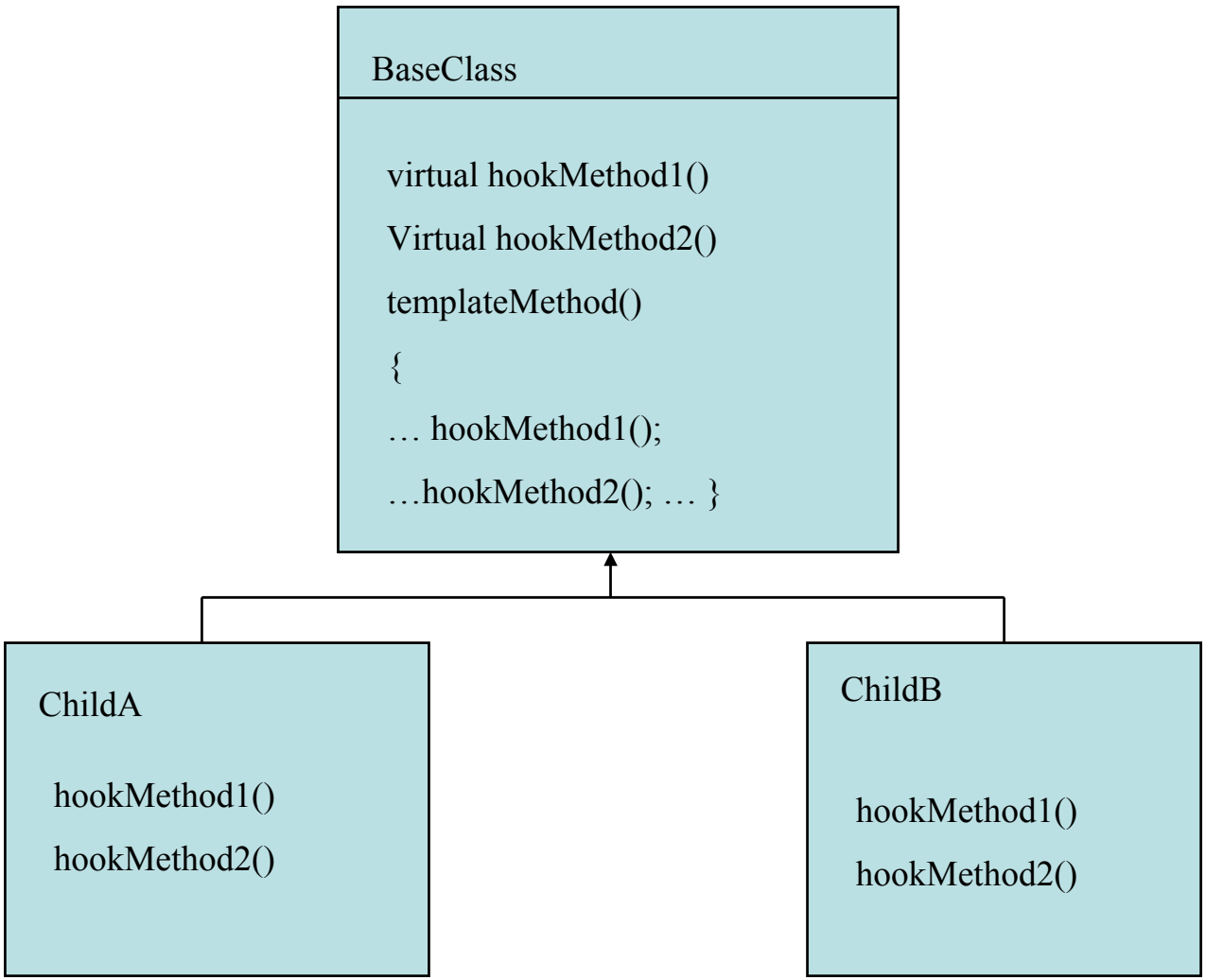

In Pop cCritter::feellistener() calls a cListener::listen(cCritter *pCritter)

# Template Method

Problem:

- Have a bunch of different child classes with a common base class.
- Want each child to execute a fixed sequence of methods in the same order.
- Want to be able to vary what the individual calls do.

# Template Method Solution



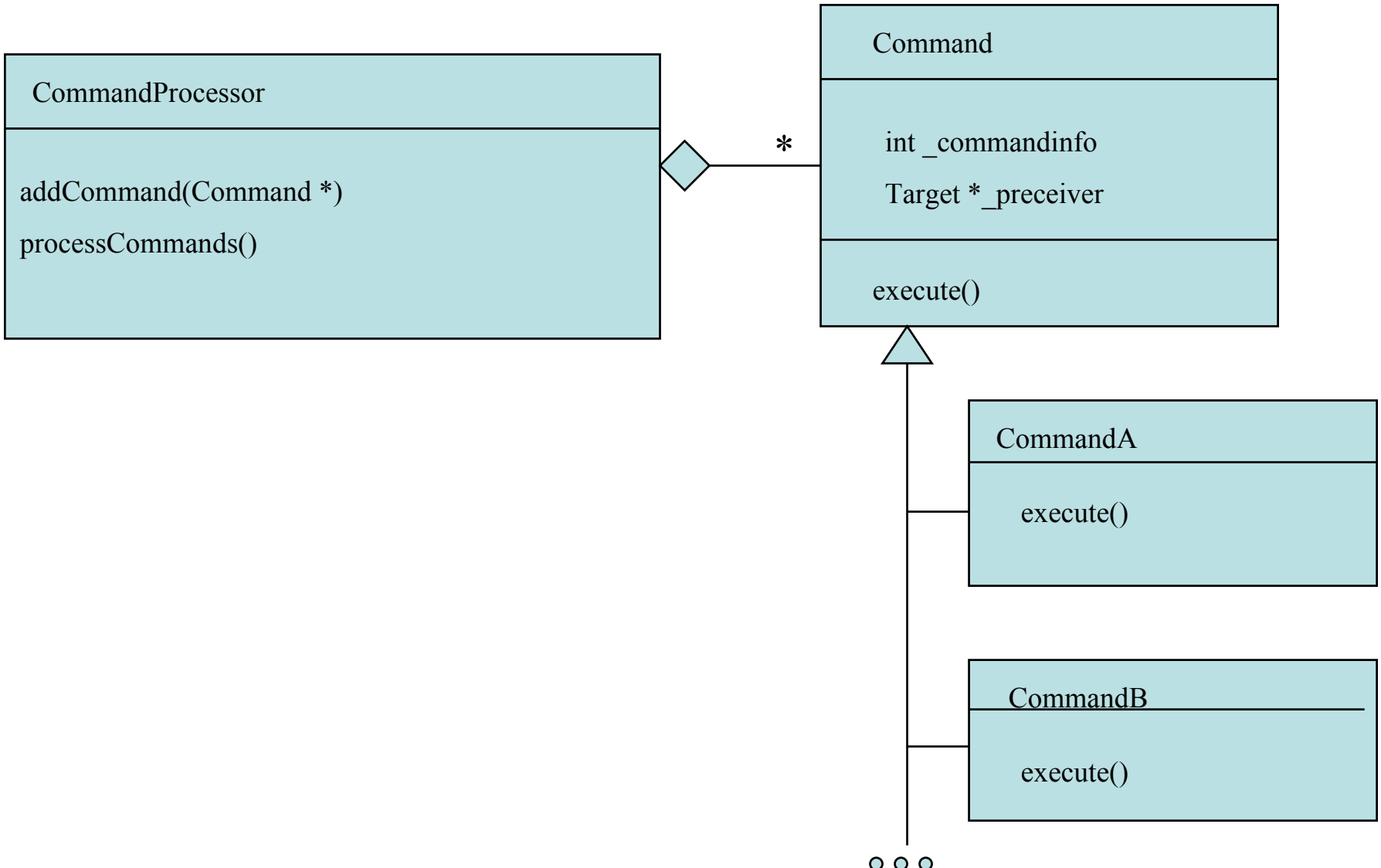
# Template Method in Pop

- cGame::step is a template that has a sequence of calls to cGame methods that update the critters in a certain order. You might override some of these methods.
- In particular, you could rewrite cGame::adjustGameParameters and cGame::gameOverMessage.
- cSprite::draw calls other stuff then cSprite::imageDraw which you can override.

# Command

- You want an object to do something, but you don't know exactly when the object will carry out the request or how it will do it.
- The solution is to create a command object that represents the command to be executed together with info about target object that is supposed to affected.
- Use the Command Object in conjunction with a command processor object which holds a collection of commands.

# Command UML


CommandProcessor
addCommand(Command *)
processCommands()
*
Command
int _commandinfo
Target *_preceiver
execute()
CommandA
execute()
CommandB
execute()

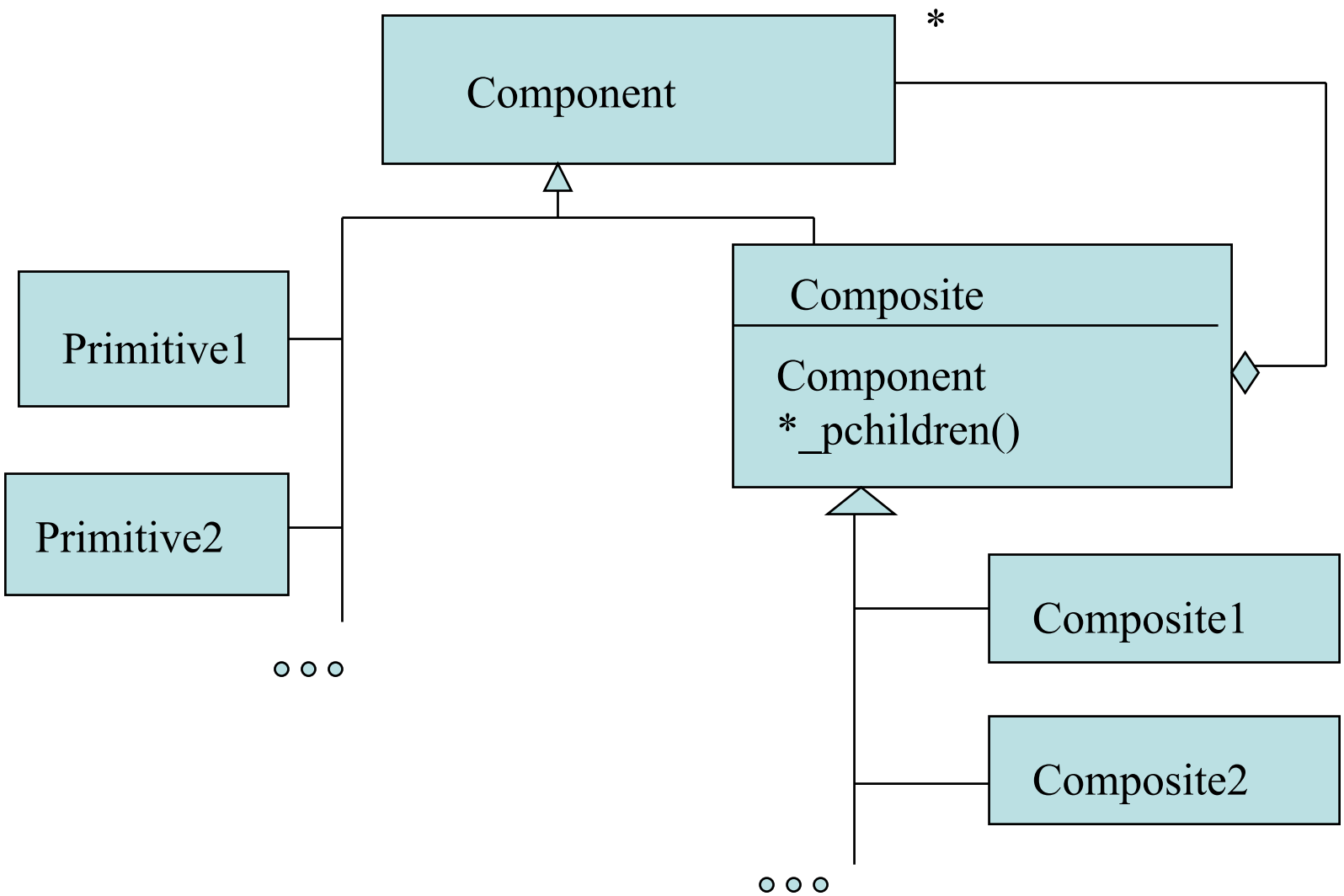
# Command in Pop

- In Windows have a Command pattern facsimile. Objects are stored in message structures with message IDs. Unfortunately, execute is passed off to a big switch depending on ID type.
- Similarly, in Pop there is a cServiceRequest struct which passes the task of executing commands off to a big switch in cBiota
- cGame::step calls cBiota::processServiceRequests to handle all the outstanding requests.

# Composite

- The problem is you have a set of Primitive Objects which you also group into Composite objects and you want to be able to treat the primitive and composite objects the same.
- The solution is  to have a base class Component which has both the primitive and composite objects inheriting from it. Composite has a member which holds any number of Components. Component might have a method do something which you can override in the child classes.

# Composite UML



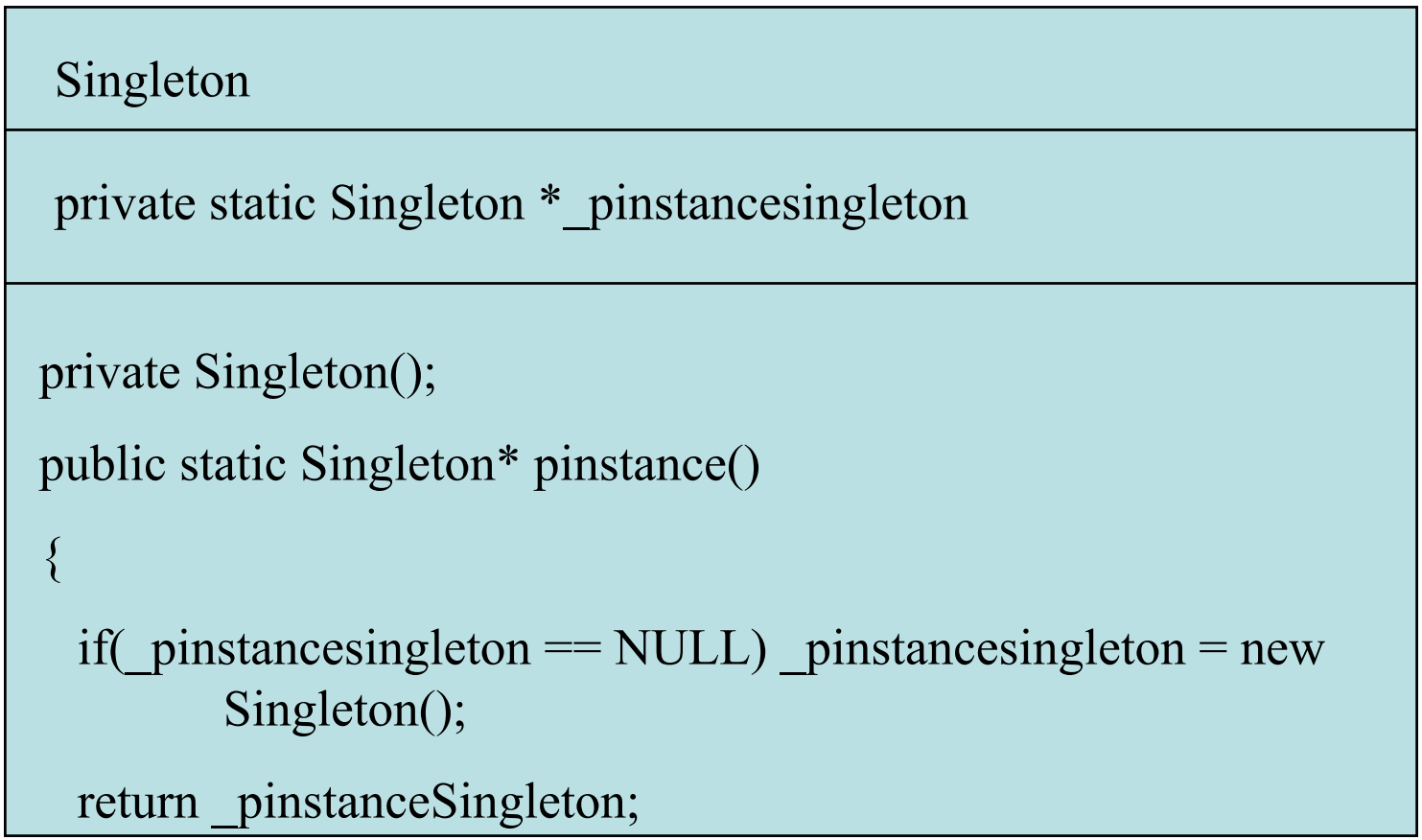
# Composite in Pop

- Composite can be thought as giving a tree whose leaves are Primitive Objects. doSomething() for the whole thing percolates down to the doSomething on the leaves.
- cSpriteComposite uses this pattern to build complex sprite out of simple cSprite's and its subclasses like cSpriteBubble, cSpriteIcon, etc.

# Singleton

- The problem is that one wants only one instance of a class to be possible.
- The solution is to give the class a single static Singleton *_pinstancesingleton member which is initially NULL. The class has an accessor function public static accessor which either returns the only instance or constructs one instance.

# Singleton UML



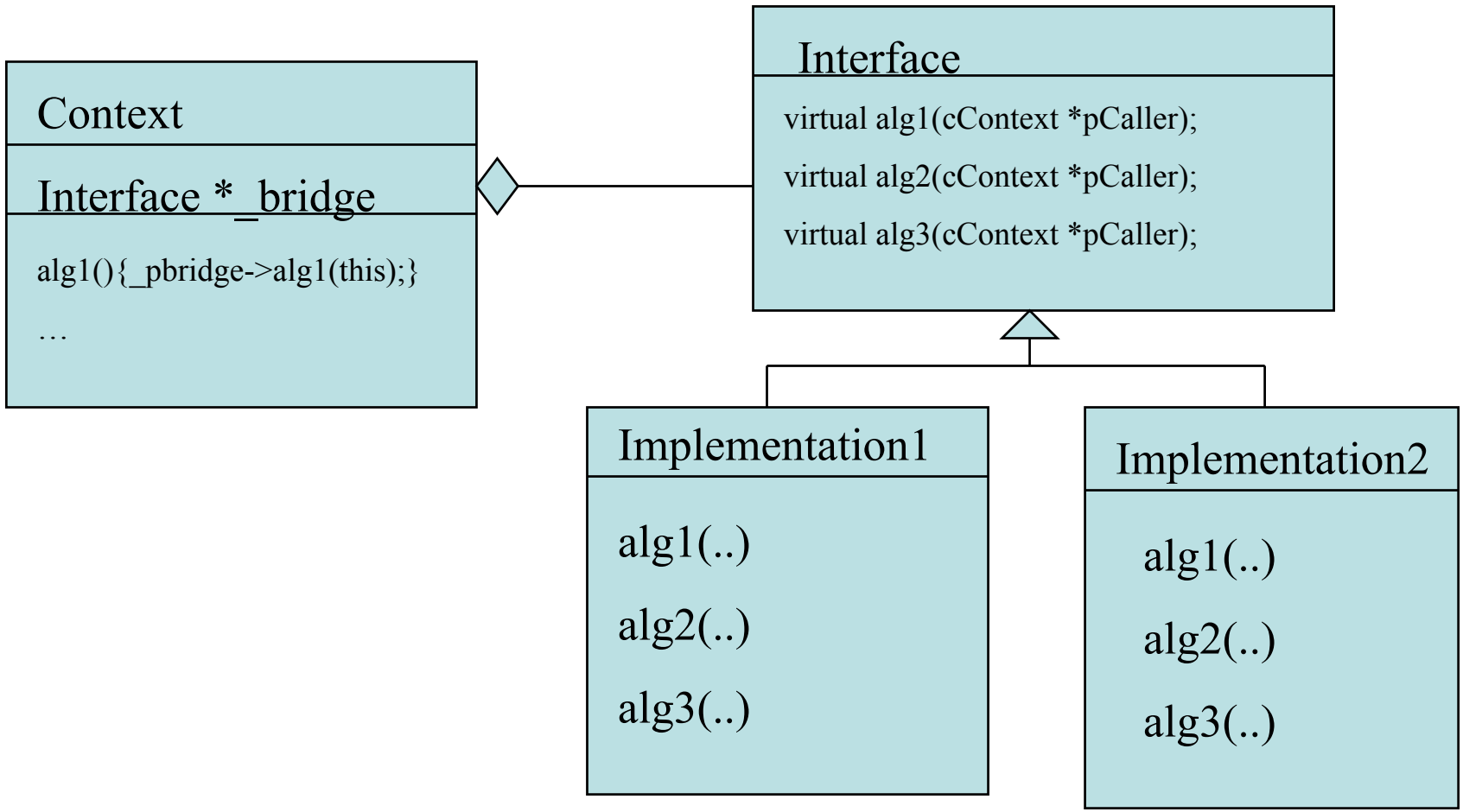
# Singleton in Pop

- Used in cRandomizer. This class has a large number of useful functions for returning random integers, reals, vectors, colors and so on. The method that returns random values can't be static because the internal state of cRandomizer is changed with each call.
- Should have a deleteSingleton() function to get rid of stuff when done. Otherwise can get memory leak.

# Bridge

- The problem is that one has a set of methods that have been implemented in multiples ways. Don’t want to mess up the rest of your code with always spelling out which way.
- The solutions is to make a common interface from which both implementation inherit and then use the interface functions for the rest of your code.

# Bridge UML


Context
Interface *_bridge
alg1(){_pbridge->alg1(this);}
…
Interface
virtual alg1(cContext *pCaller);
virtual alg2(cContext *pCaller);
virtual alg3(cContext *pCaller);
Implementation1
alg1(..)
alg2(..)
alg3(..)
Implementation2
alg1(..)
alg2(..)
alg3(..)

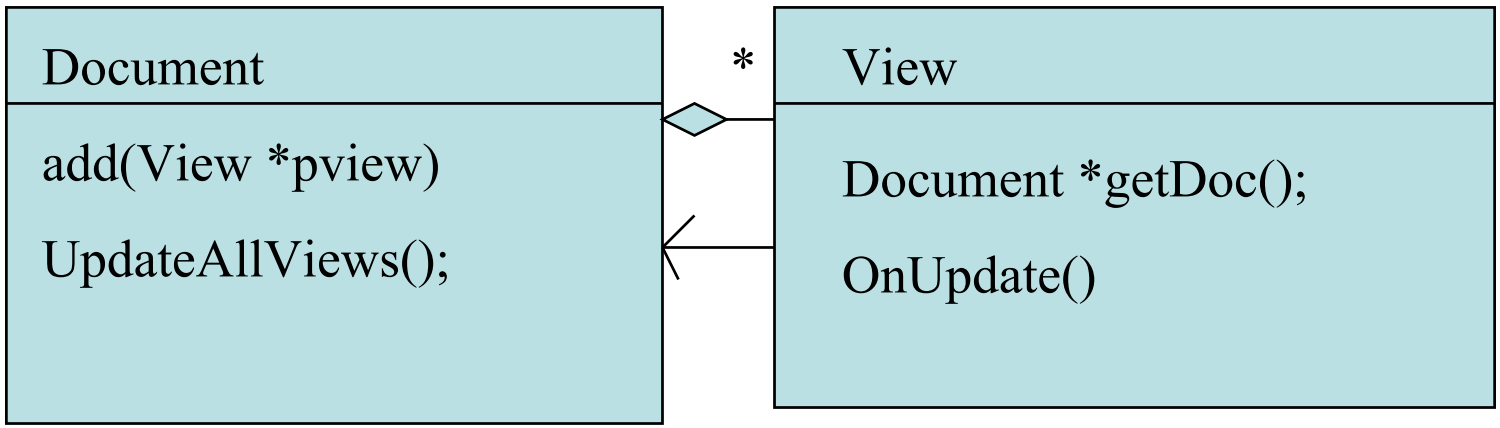
# Bridge in Pop

- CPopView owns a cGraphics interface which uses dynamically either cGraphicsMFC or cGraphicsOpenGL to actually implement it.

# Document-View

- The problem is one has a number of different representations of the same information. How do we keep the various views in-sync with each other.
- The solution is to have two classes Document and View. The Document has a list of Views and has a method UpdateAllViews to notify views of changes in data. A view has a *getDoc() function and an OnUpdate() function called by UpdateAllViews().

# Document-View UML



# Document View in Pop

- CDocument and CView -- latter could control if see things in wireframe.