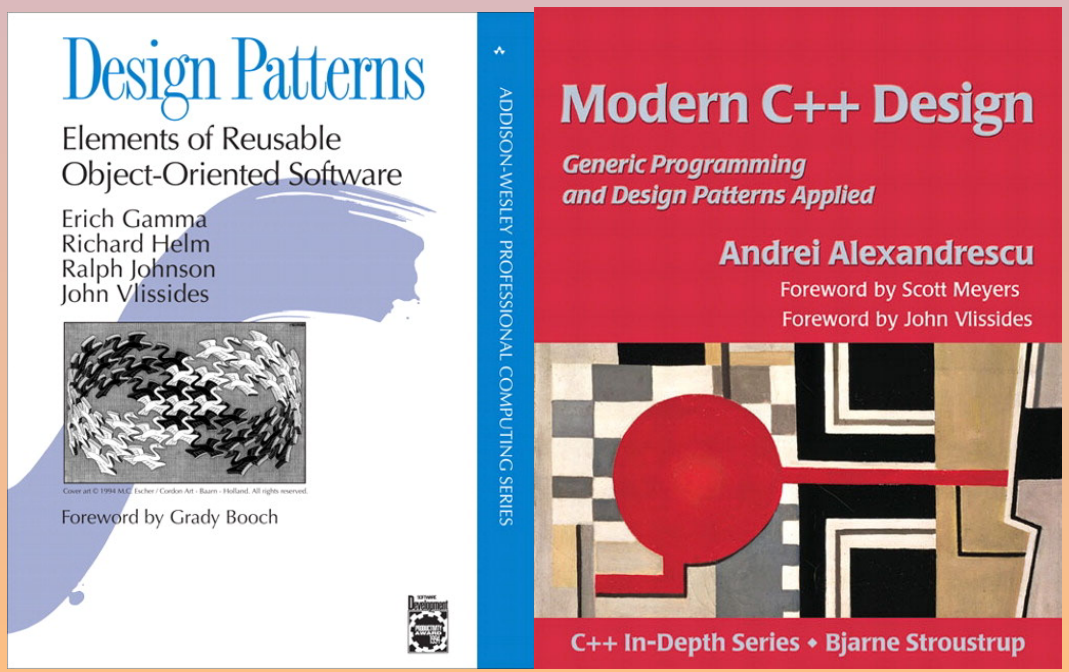
Design Patterns

Akim Demaille akim@lrde.epita.fr

May, 26th 2015

(2016-11-16 09:53:54 +0100 121bea3)



Design Patterns

Design Patterns

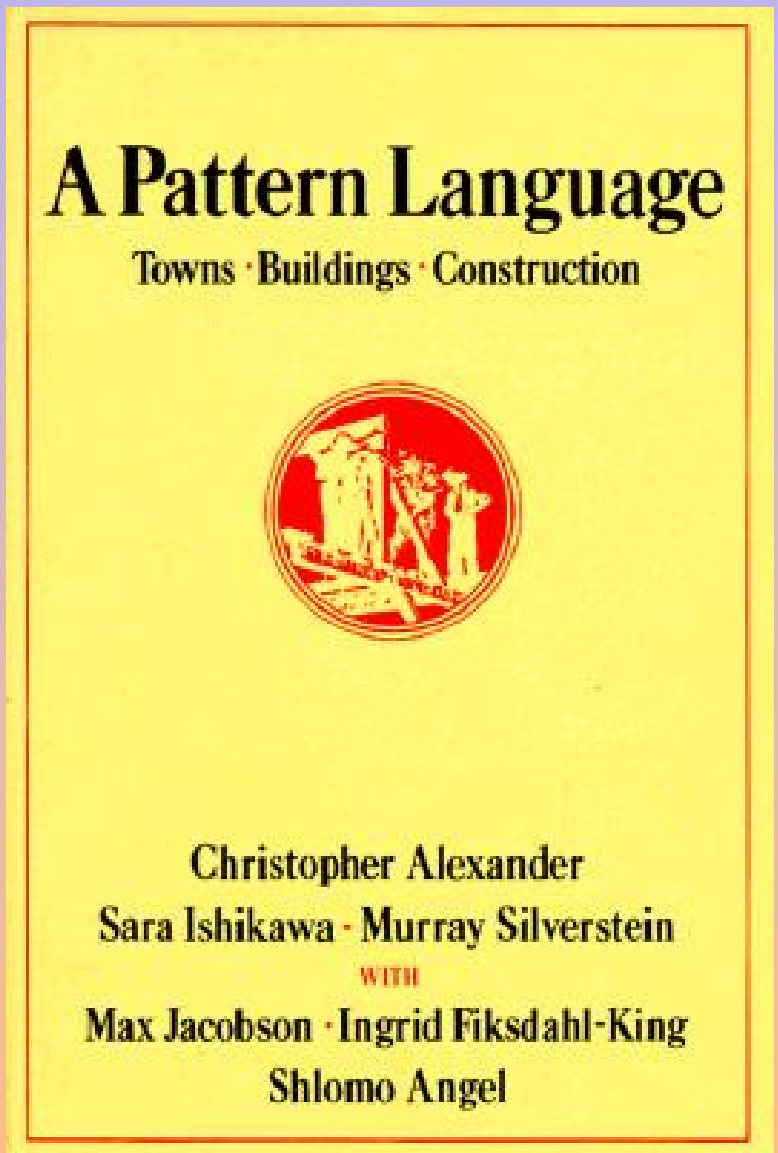
Design Patterns

> *Each 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 Patterns

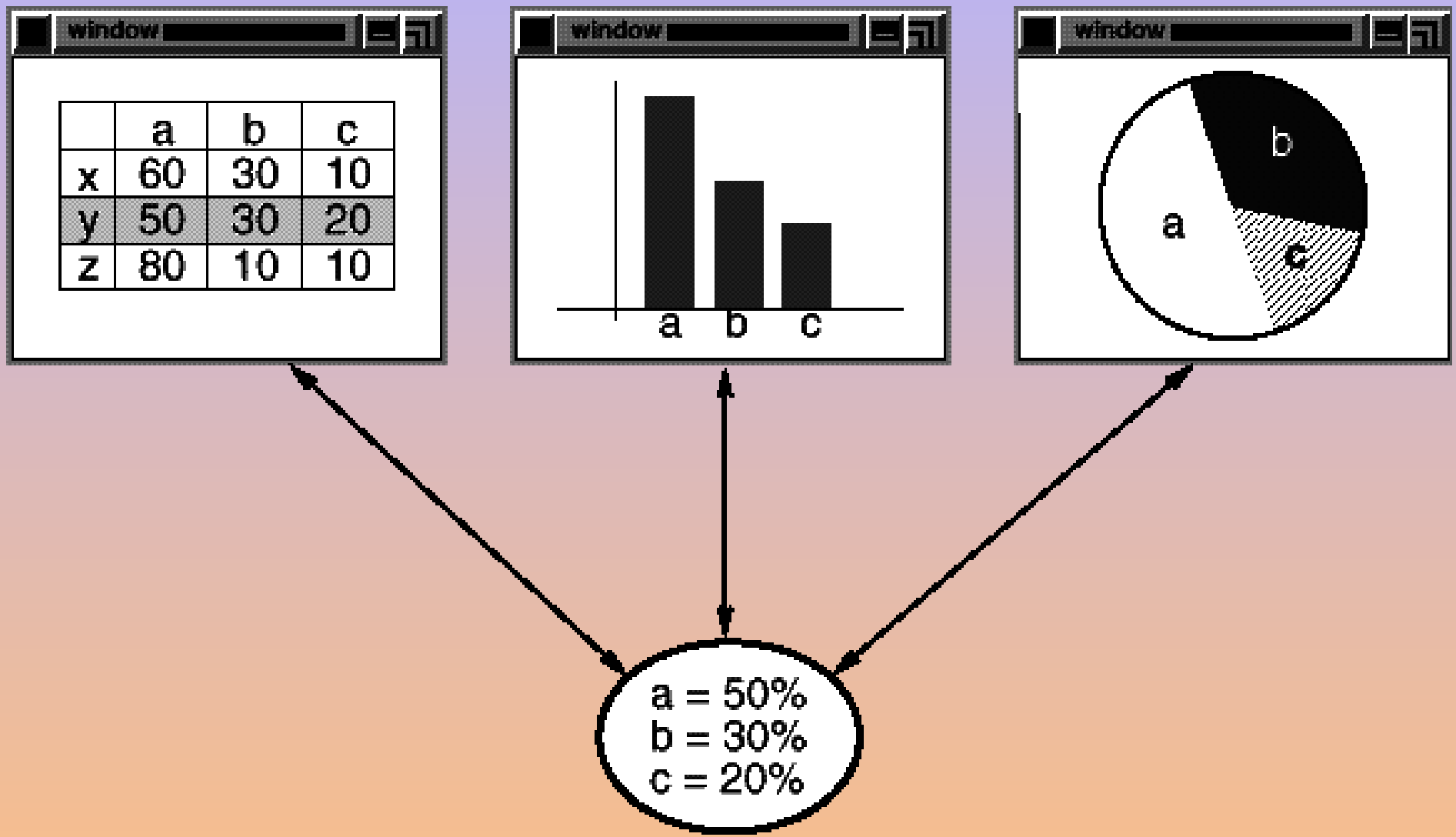
> *Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.*
>
> — [Alexander et al., 1977]

Design Patterns



> *Each 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.*
>
> — [Alexander et al., 1977]

Model/View/Controller in SmallTalk 80



MVC increases flexibility and reuse by decoupling [Gamma et al., 1995]

Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Definition

Definition

Design patterns represent solutions to problems
that arise when developing software
within a particular context.

Patterns facilitate:

- modularity
- reusability
- efficient talk about software design!

[WDesign_Patterns]

Design Pattern

- a recurring *and* generic construct
- with an intention, a motivation, a context
- which is named (as its components)
- and described (pros, cons, examples)

- to provide more reuse opportunities
- coarser grained than classes

Design Patterns

Four essential elements [Gamma et al., 1995]:

pattern name a handle, in a word or two

problem when to apply the pattern

solution the elements that make up the design

consequences results and trade-offs of applying the pattern

- handle, in a word or two
  to describe a design problem, its solutions, and consequences
- increases our design vocabulary
- lets design at a higher level of abstraction
- finding good names was/is hard

- when to apply the pattern
  problem and context
- might describe specific design problems
  e.g., how to represent algorithms as objects
- might describe class or object structures that are symptomatic of an inflexible design
- possibly a list of conditions

- the elements that make up the design
- their relationships, responsibilities, and collaborations
- no concrete design or implementation
  a pattern is like a template
- an abstract description of a design problem
  and how a general arrangement of classes and objects solves it

- results and trade-offs of applying the pattern
- critical for evaluating design alternatives, costs and benefits
- its impact on a system's flexibility, extensibility, or portability
- possibly language and implementation issues

Describing Design Patterns

A consistent format, making DP easier to learn, compare, and use.

- Pattern Name
- Classification
- Intent
- Also Known As
- Motivation
- Applicability
- Structure
- Participants
- Collaborations
- Consequences
- Implementation
- Sample Code
- Known Uses
- Related Patterns

Describing Design Patterns

A consistent format, making DP easier to learn, compare, and use.

- Pattern Name
- Classification
- Intent
- Also Known As
- Motivation
- Applicability
- Structure
- Participants
- Collaborations
- Consequences
- Implementation
- Sample Code
- Known Uses
- Related Patterns

Describing Design Patterns

A consistent format, making DP easier to learn, compare, and use.

- Pattern Name
- Classification
- Intent
- Also Known As
- Motivation
- Applicability
- Structure
- Participants
- Collaborations
- Consequences
- Implementation
- Sample Code
- Known Uses
- Related Patterns

Describing Design Patterns

Pattern Name	The essence of the pattern. Part of the design vocabulary.
Classification	Its purpose.
Intent	What does it do? What rationale and intent? What particular design issue or problem does it address?
Also Known As	Possibly other well-known names.
Motivation	A scenario that illustrates a design problem and how the pattern structure solves the problem.
Applicability	Where/when can it be applied?

Describing Design Patterns

Structure	A graphical representation of the classes (in UML).
Participants	The classes and/or objects. Their responsibilities.
Collaborations	How they collaborate.
Consequences	How does the pattern support its objectives? Trade-offs?

Describing Design Patterns

Implementation	Pitfalls, hints, techniques, language-specific issues.
Sample Code	Illustrating code fragments.
Known Uses	Examples *found* in real systems.
Related Patterns	What design patterns are closely related to this one? Important differences?

The Original Design Patterns

Design Patterns Space

- **Creational patterns**
  initialize and configure classes and objects
- **Structural patterns**
  decouple interface and implementation of classes and objects
- **Behavioral patterns**
  dynamic interactions among societies of classes and objects
- Concurrency
  deal with multithreaded programming paradigm

see also `http://hillside.net/`

Design Patterns Space

- **Creational patterns**
  initialize and configure classes and objects
- **Structural patterns**
  decouple interface and implementation of classes and objects
- **Behavioral patterns**
  dynamic interactions among societies of classes and objects
- **Concurrency**
  deal with multithreaded programming paradigm

see also `http://hillside.net/`

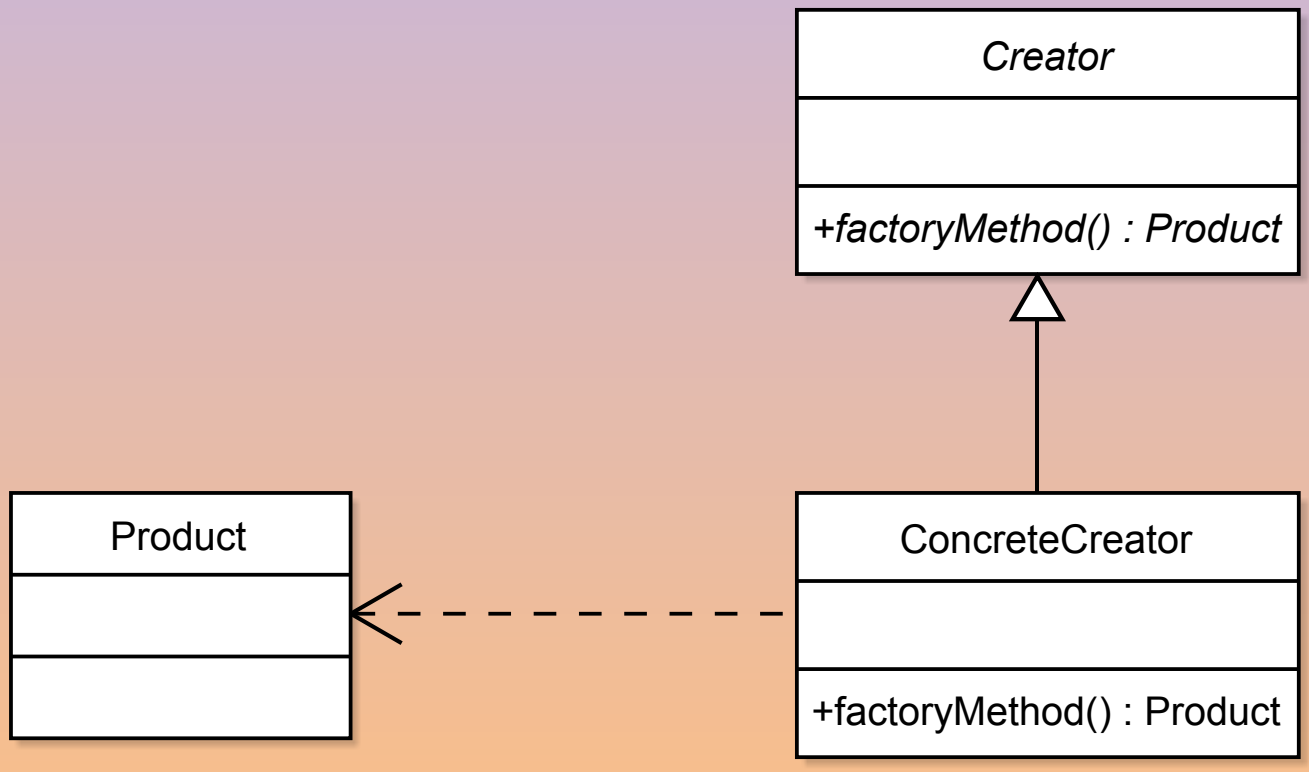
A Classification of Design Patterns

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

Creational Patterns

Factory Method

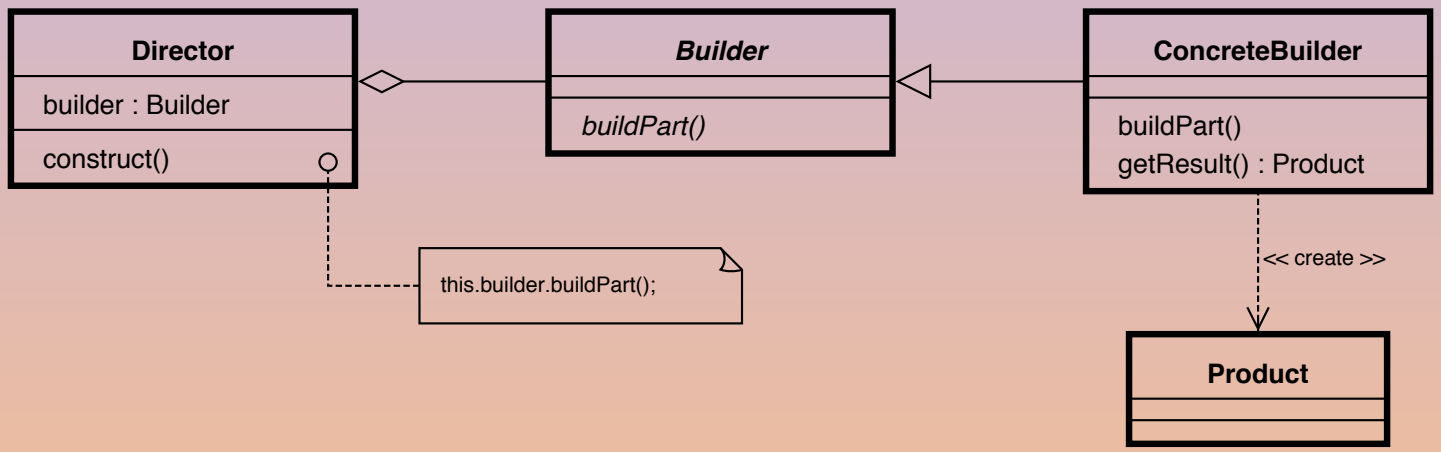
Define an interface for creating an object,
but let subclasses decide which class to instantiate.
E.g., named constructors, FlyWeight, maximal sharing



[WFactory_method_pattern]

Builder

Separate the construction of a complex object from its representation.
E.g., Vcsn 2's Automaton Editor.

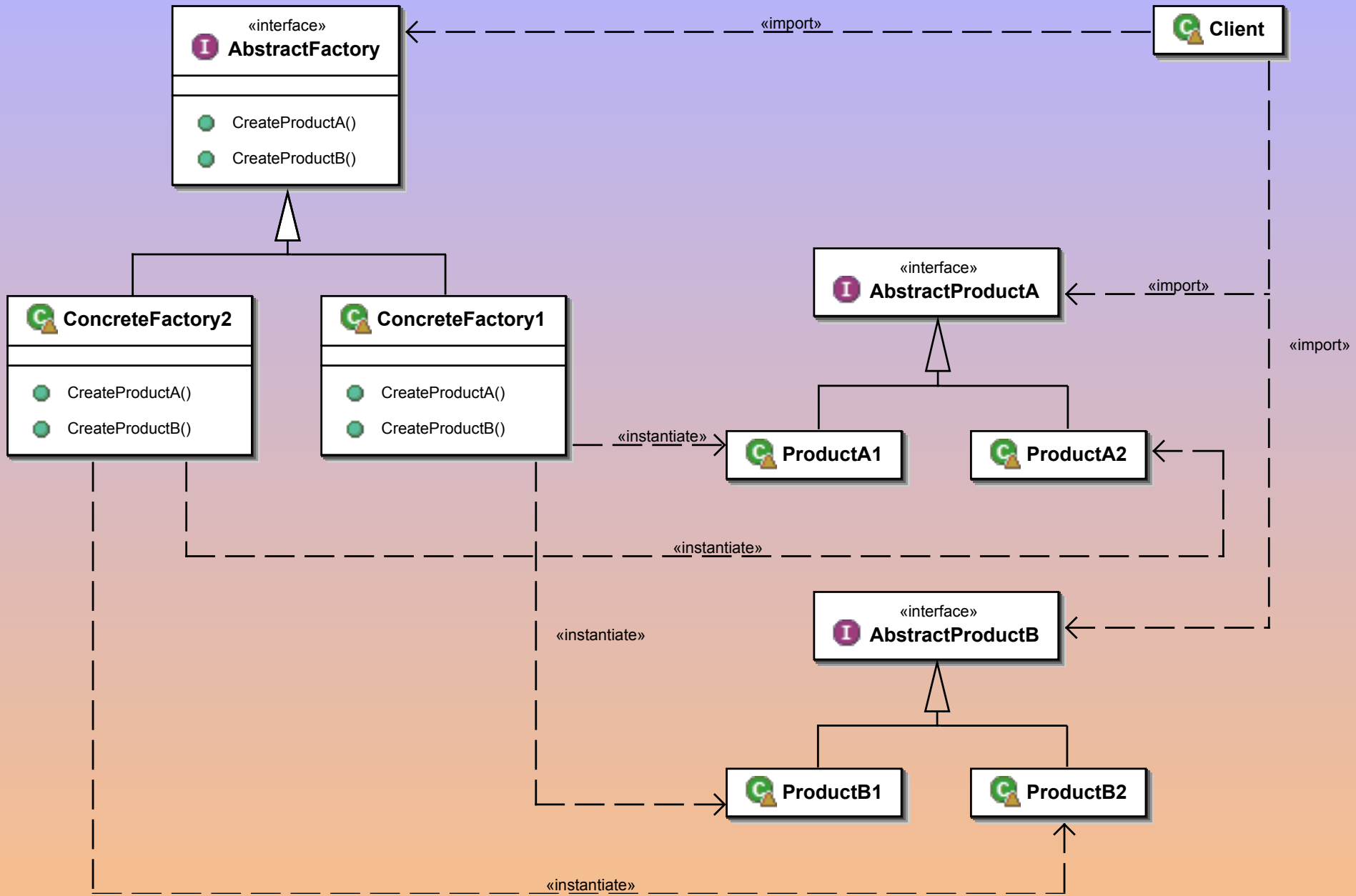


[WBuilder_pattern]

Abstract Factory

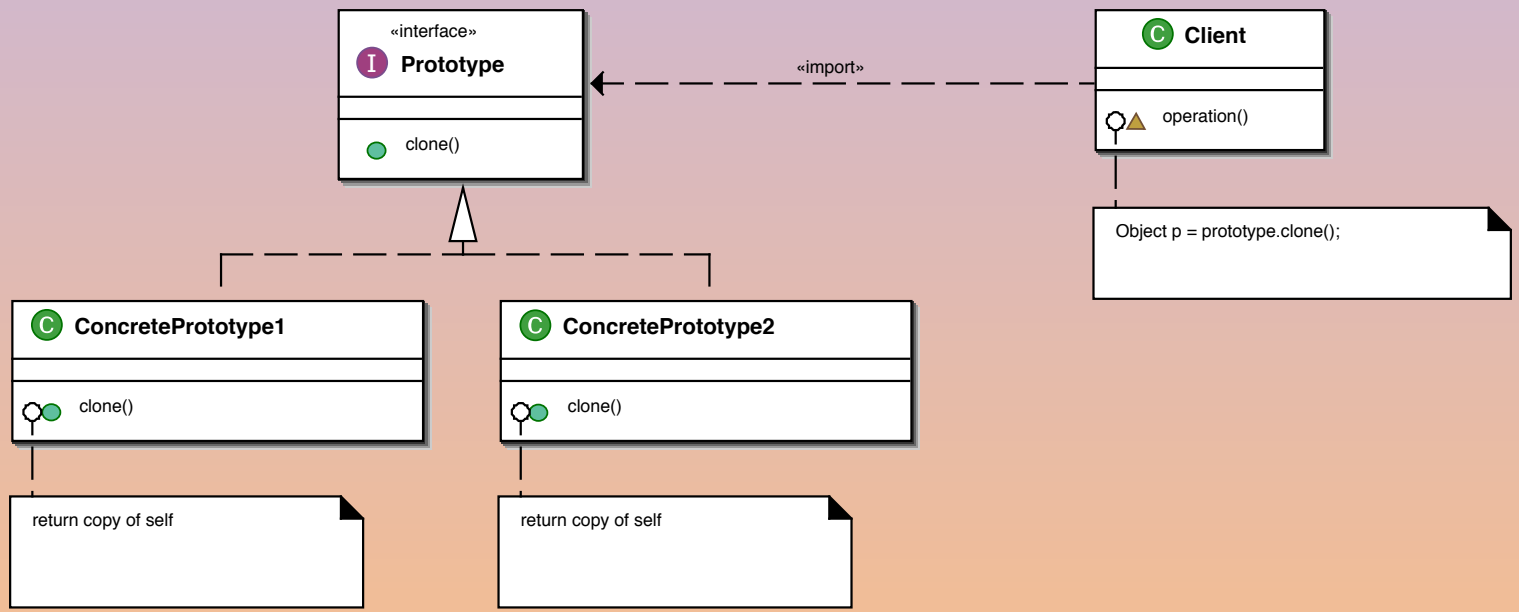
Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
E.g., multiple GUI toolkits.

[W`Abstract_factory_pattern`]


«interface»
AbstractFactory
CreateProductA()
CreateProductB()
«import»
Client
«interface»
AbstractProductA
«import»
«import»
ConcreteFactory2
CreateProductA()
CreateProductB()
ConcreteFactory1
CreateProductA()
CreateProductB()
«instantiate»
ProductA1
ProductA2
«instantiate»
«instantiate»
«interface»
AbstractProductB
ProductB1
ProductB2
«instantiate»

Prototype

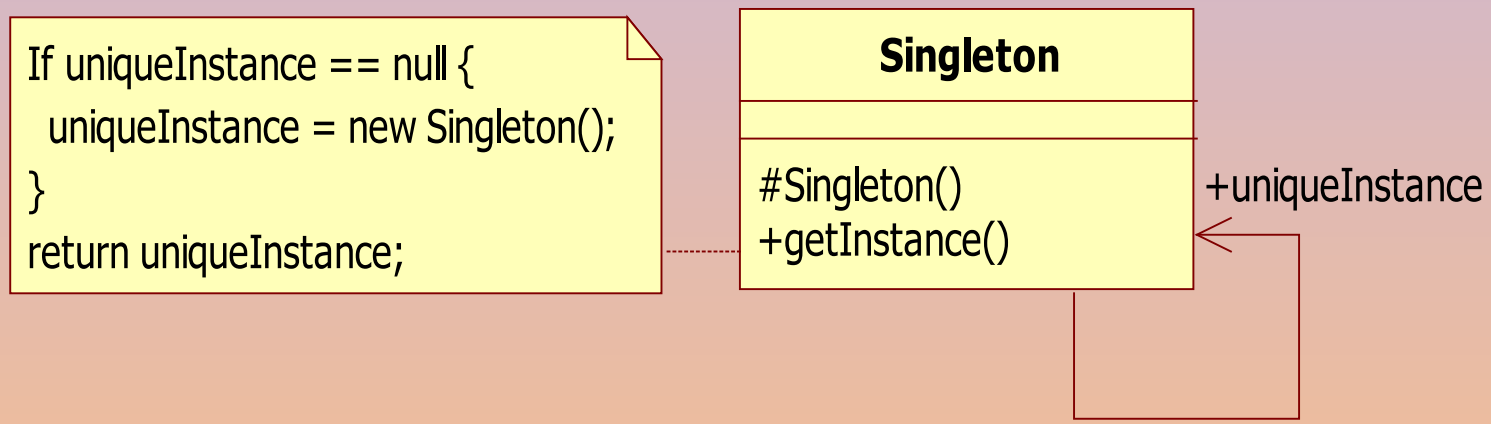
Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
E.g., set of user defined shapes, JavaScript, urbiscript.



[WPrototype_pattern]

Singleton

Ensure a class only has one instance,
and provide a global point of access to it.
E.g., plugin register, logger.

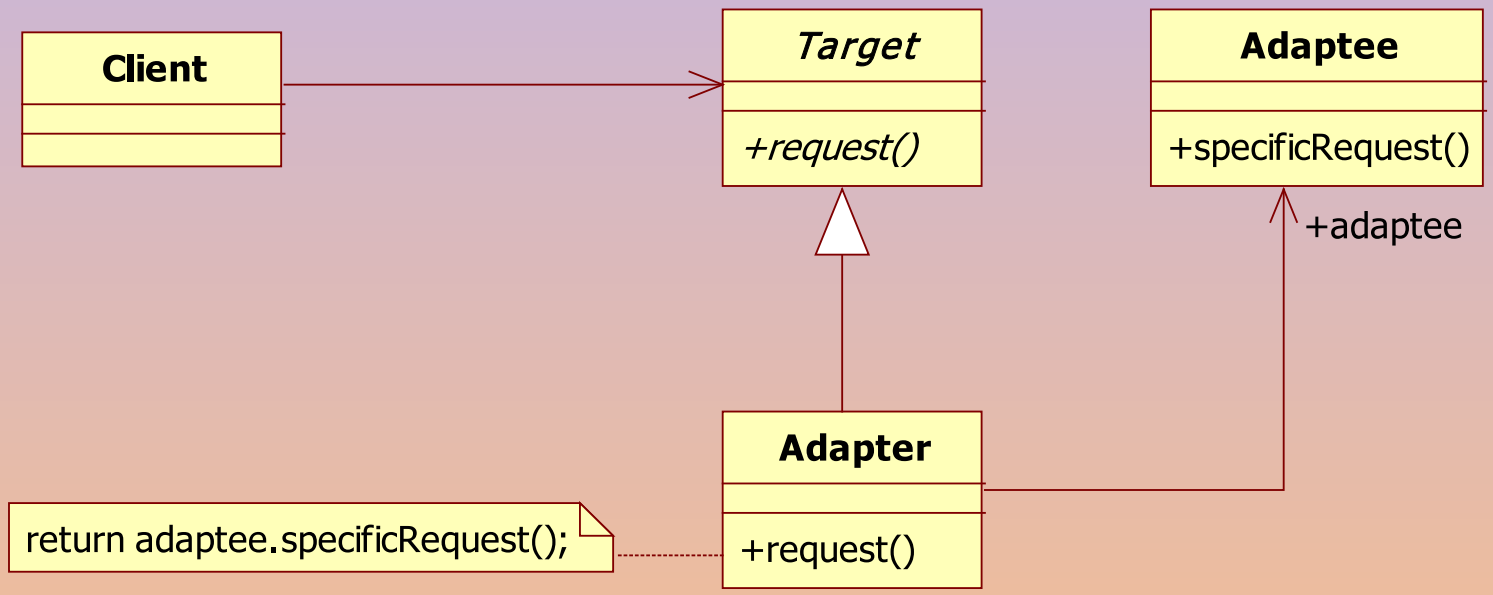


[WSingleton_pattern]

Structural Patterns

Adapter

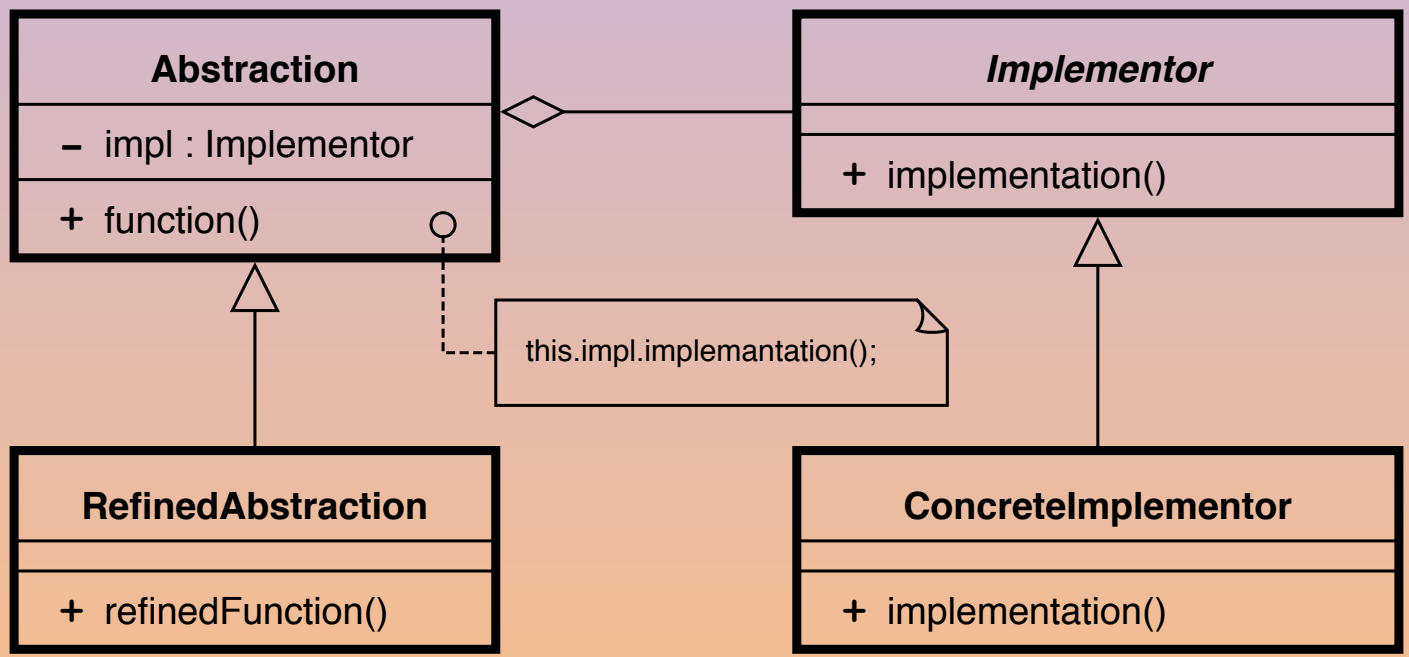
Convert the interface of a class into another interface clients expect.



[WAdapter_pattern]

Bridge

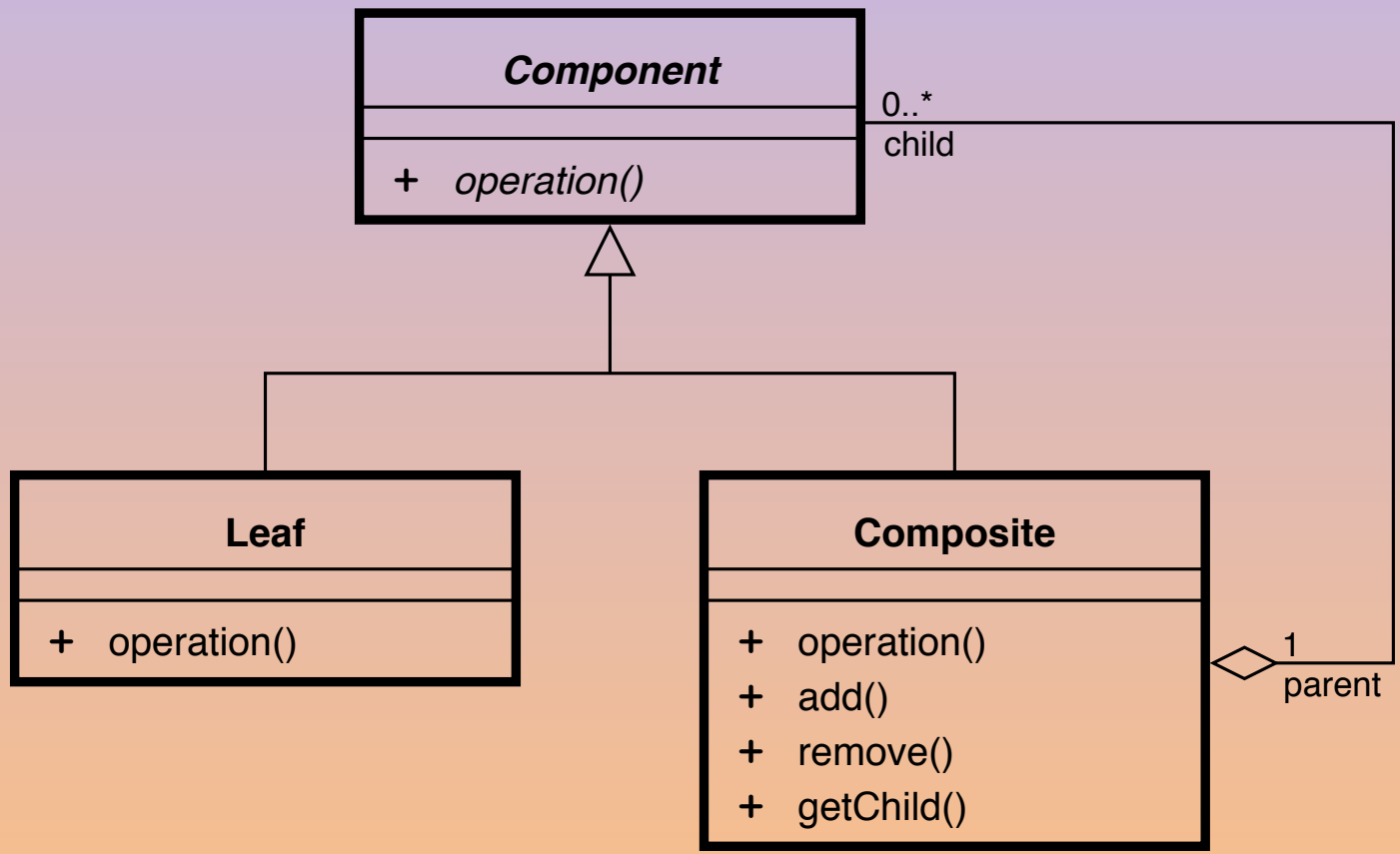
Decouple an abstraction from its implementation
so that the two can vary independently.
E.g., widget hierarchies, Vcsn dyn::RatExps.



[WBridge_pattern]

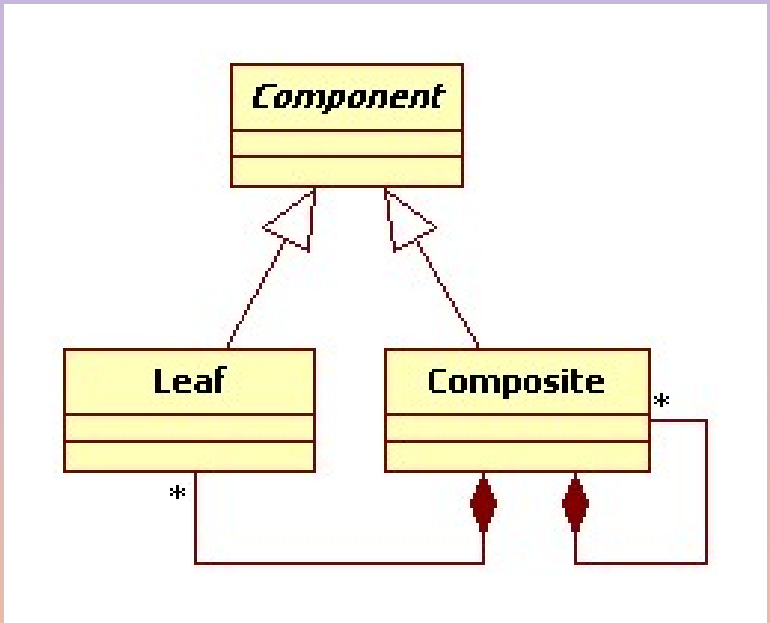
Composite

Compose objects into tree structures to represent part-whole hierarchies.
E.g., AST, composite widgets.



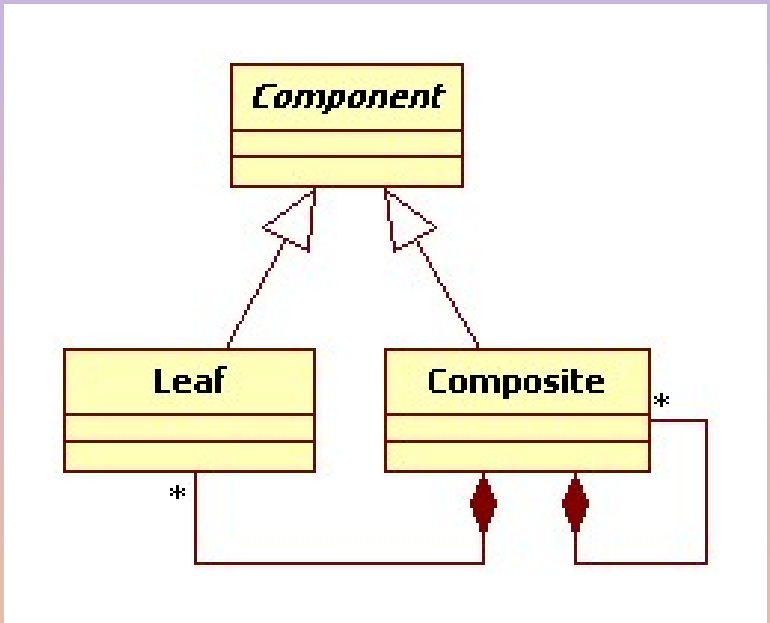
[Wikipedia Composite_pattern]

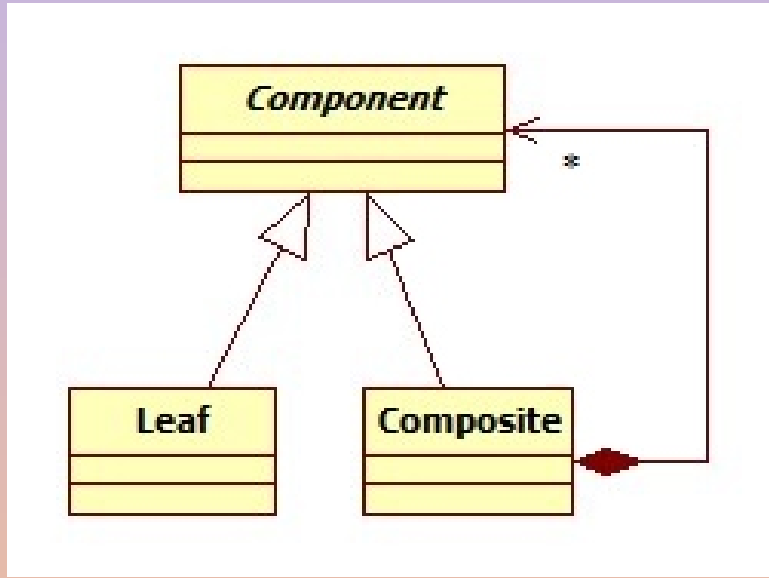
Misaligned/Spoiled Patterns



[Bouhours, 2012]

Misaligned/Spoiled Patterns

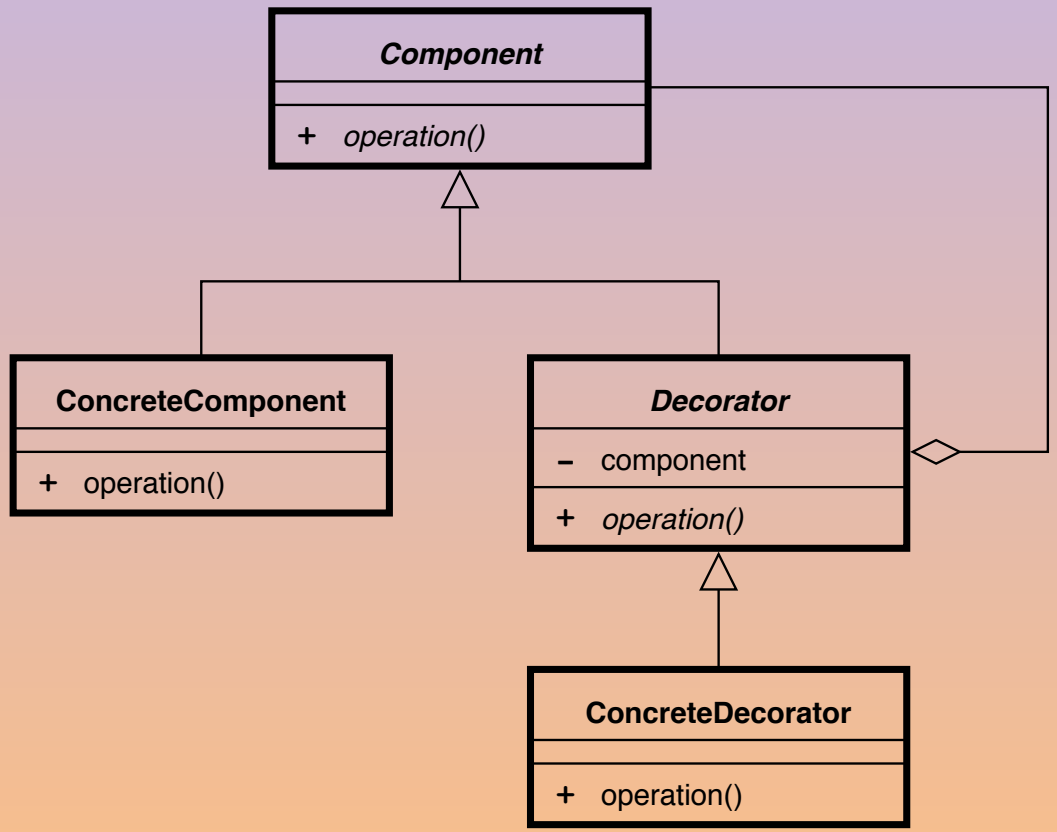




[Bouhours, 2012]

Decorator

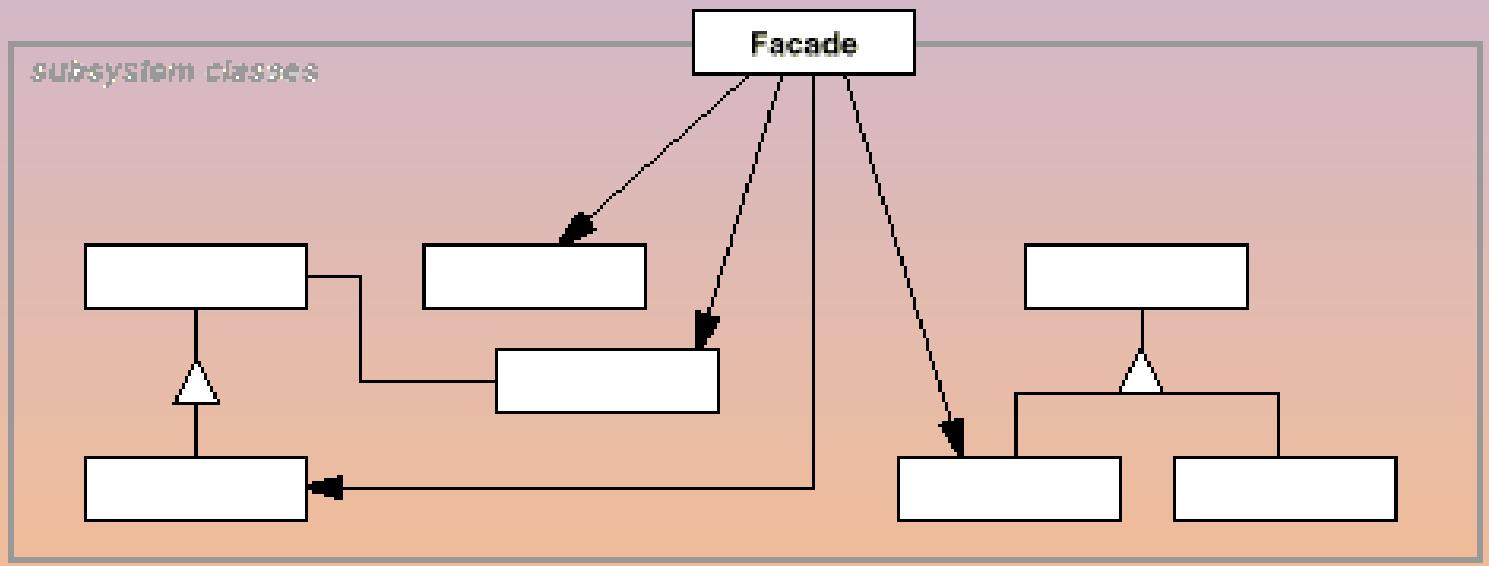
Attach additional responsibilities to an object dynamically.
E.g., widget decorators.



[WDecorator_pattern]

Facade

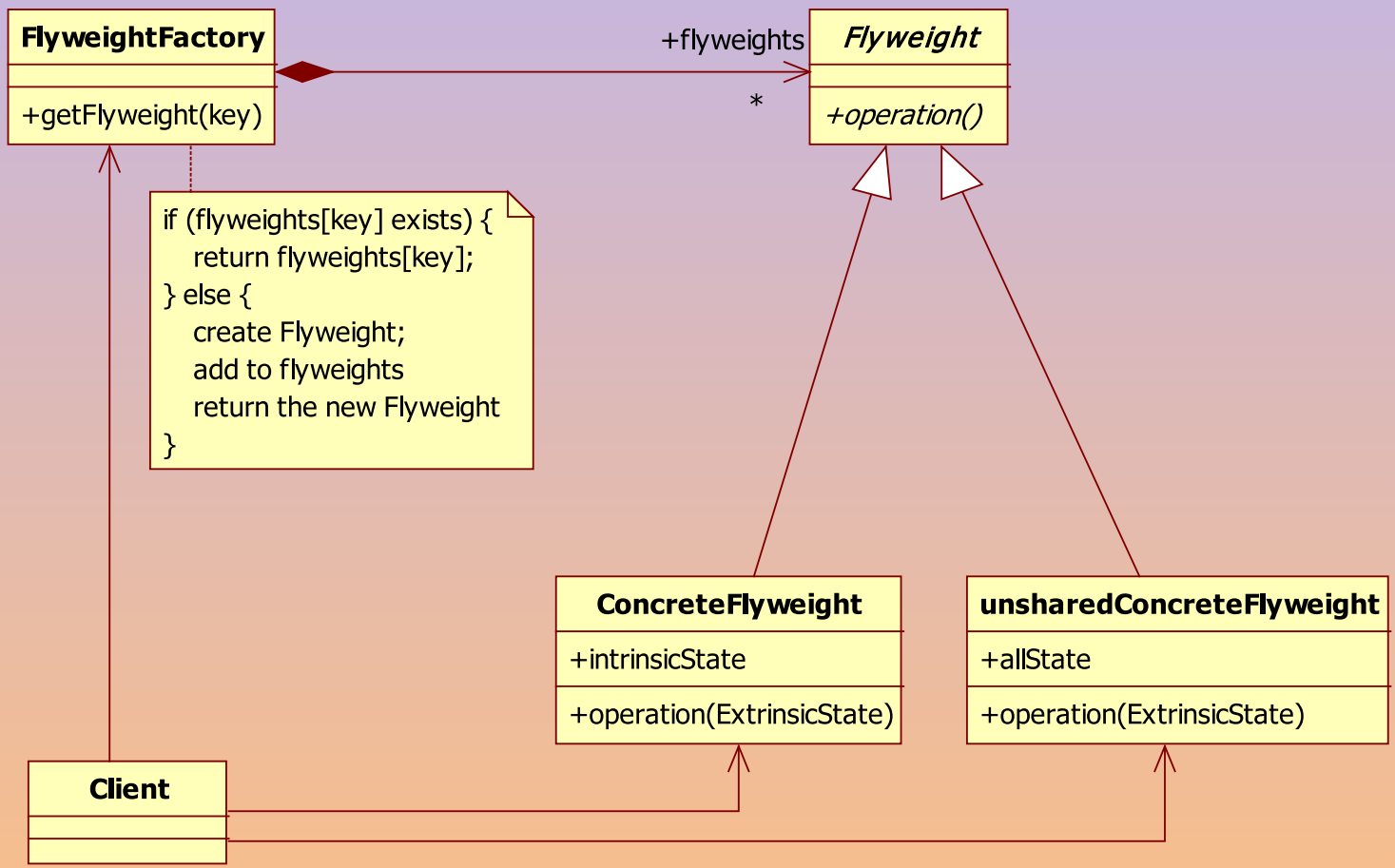
Provide a unified interface to a set of interfaces in a subsystem.
Define a higher-level interface that makes the subsystem easier to use.
E.g., urbiscript.



[WFacade_pattern]

Flyweight

Use sharing to support large numbers of fine-grained objects.
E.g., internalized strings (`tc`'s `symbol`), maximal sharing.



[WFlyweight_pattern]

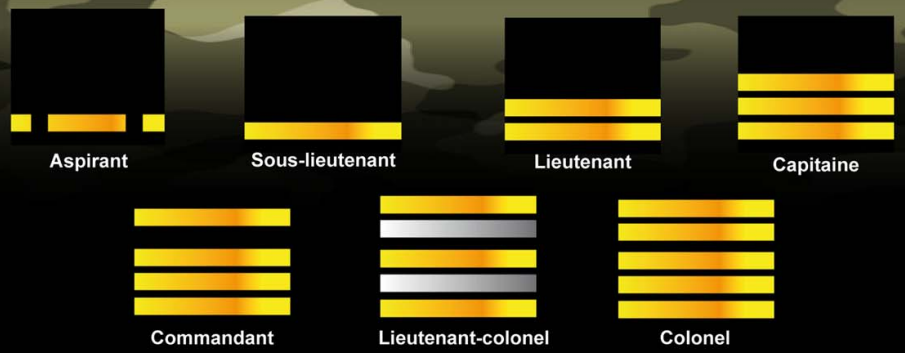
Behavioral Patterns

LES GRADES DE L'ARMEE DE TERRE

MILITAIRES DU RANG

SOUS-OFFICIERS

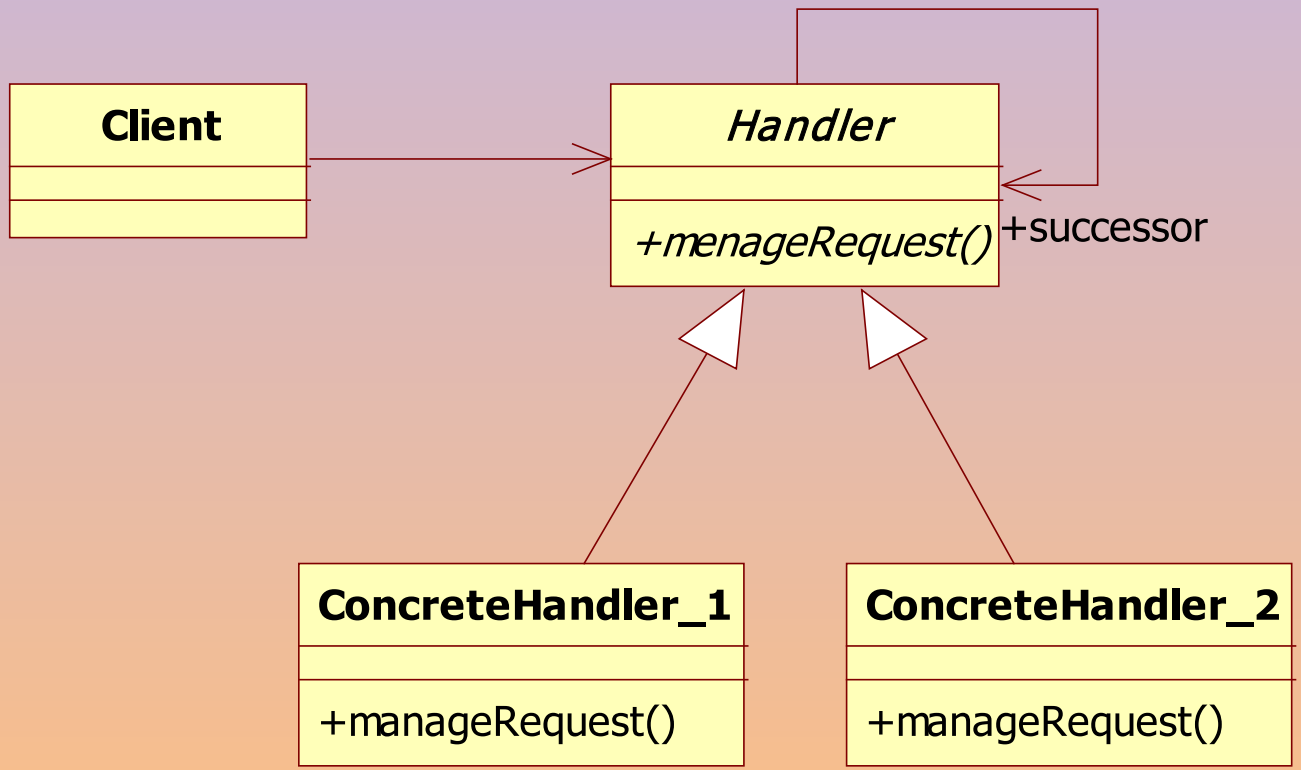
OFFICIERS



OFFICIERS GENERAUX

Chain of Responsibility

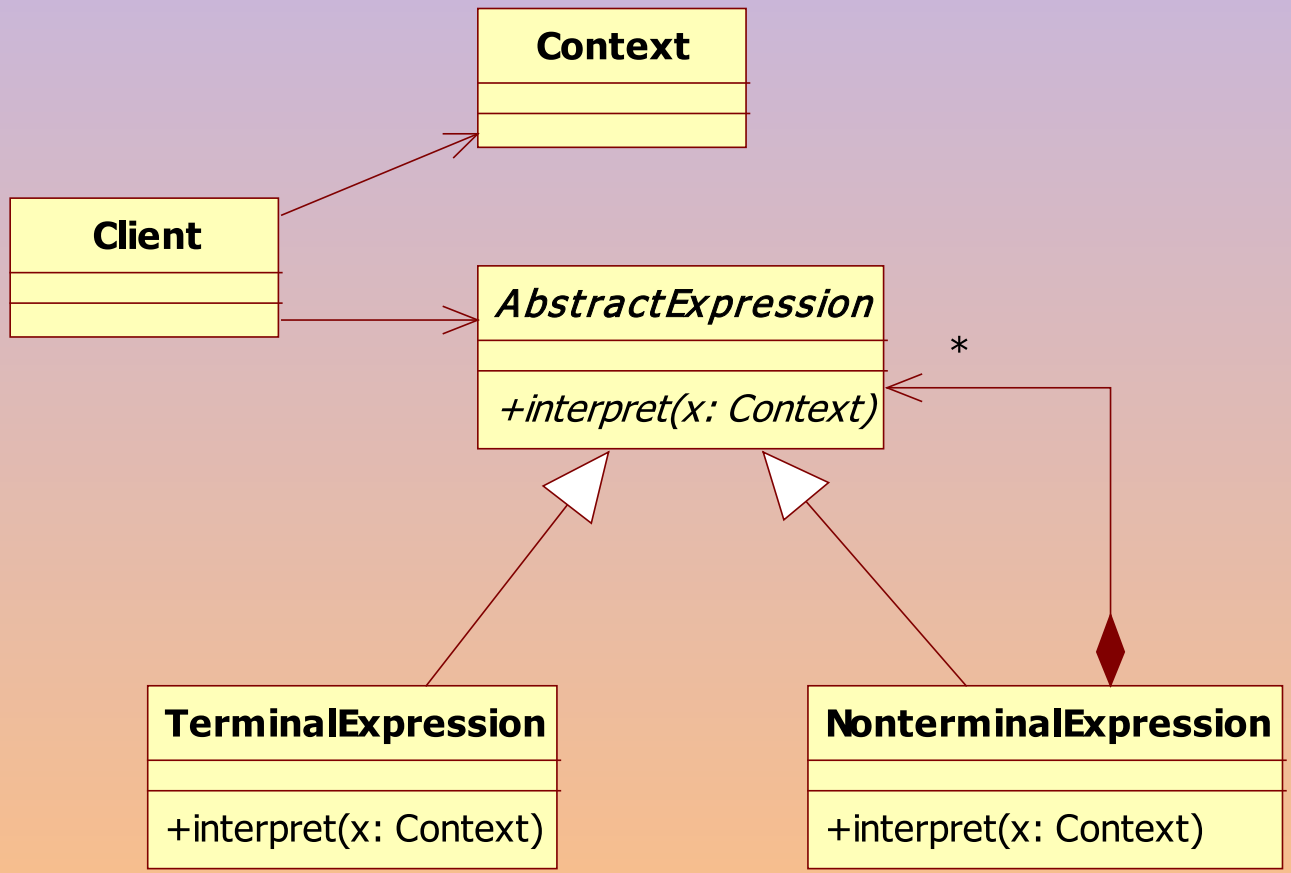
Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request.
E.g., GUIs, `catch`.



[WChain-of-responsibility_pattern]

Interpreter

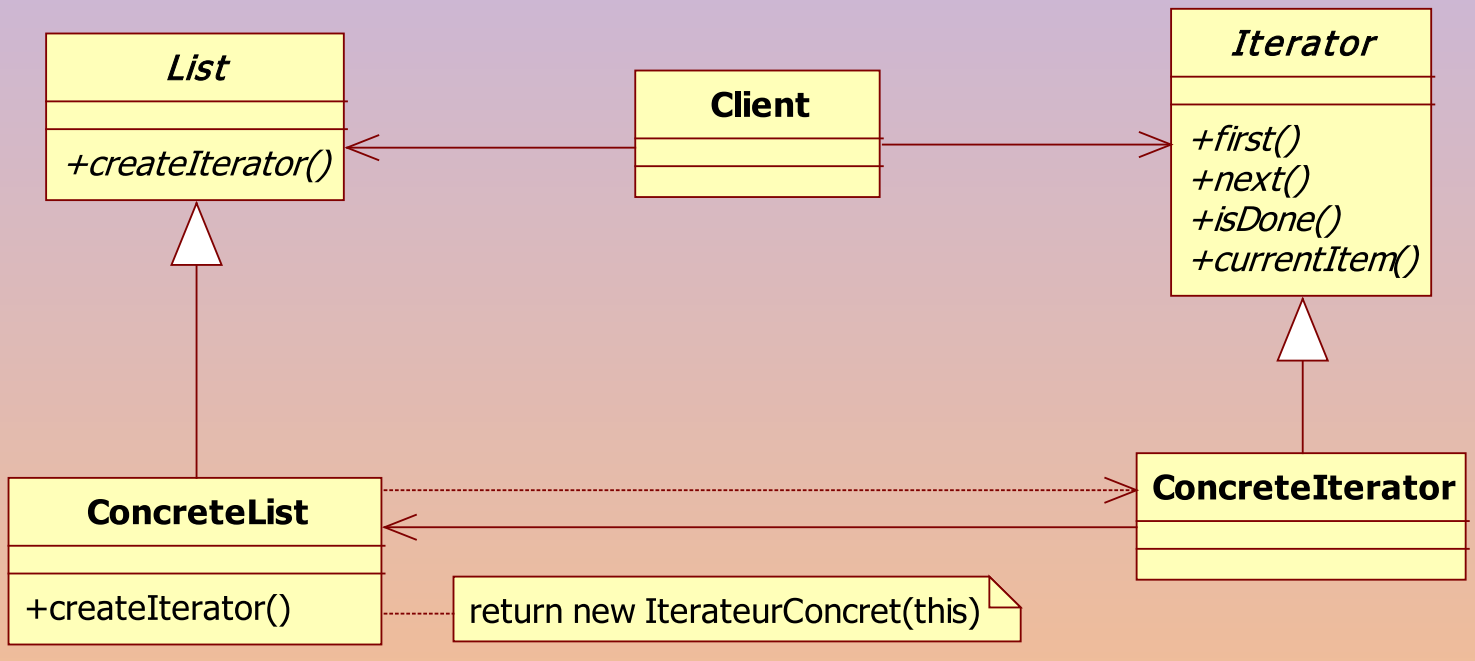
Given a language, define a representation for its grammar along with an interpreter.



[WInterpreter_pattern]

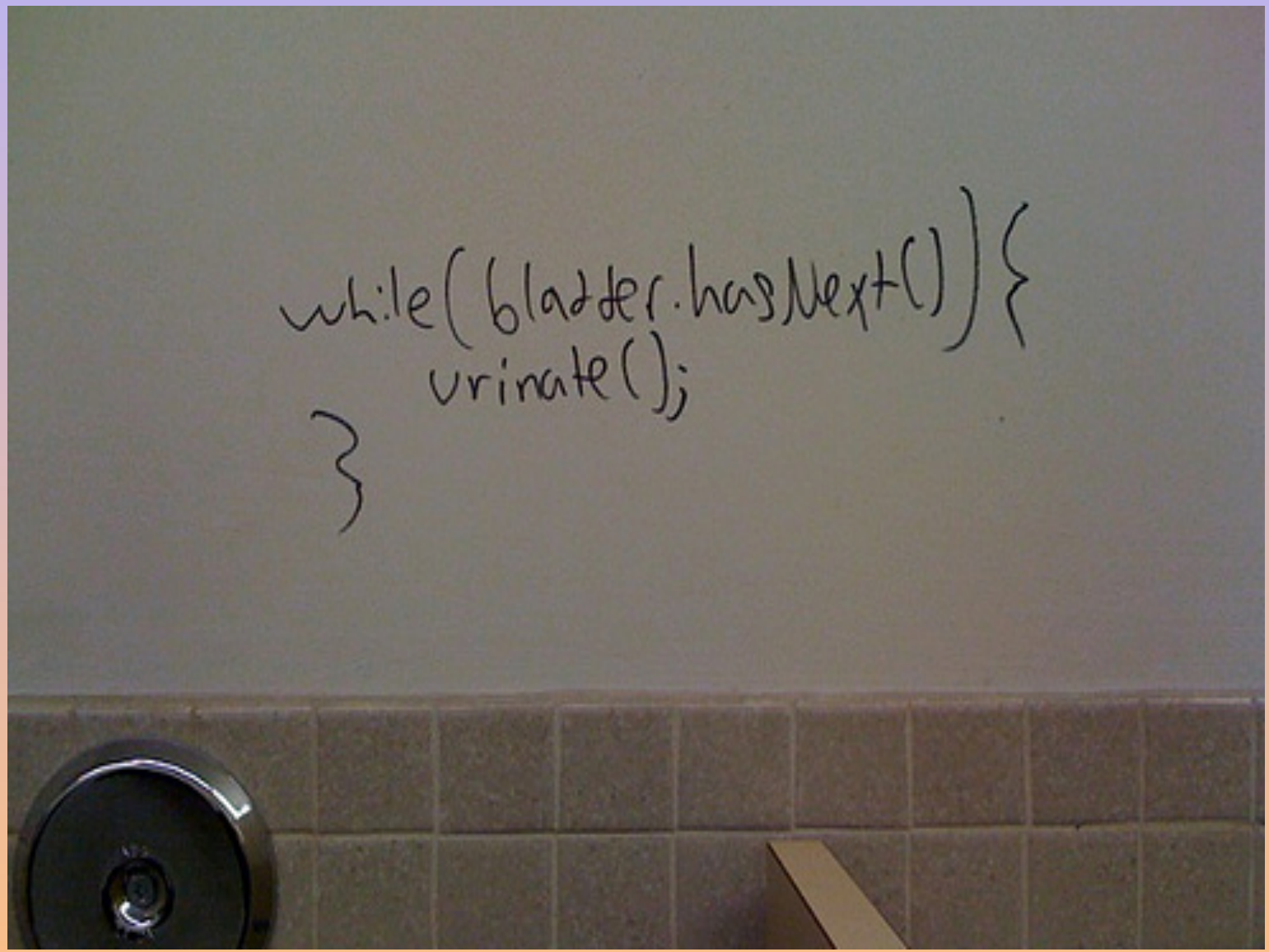
Iterator

Decouple algorithms from “containers”.



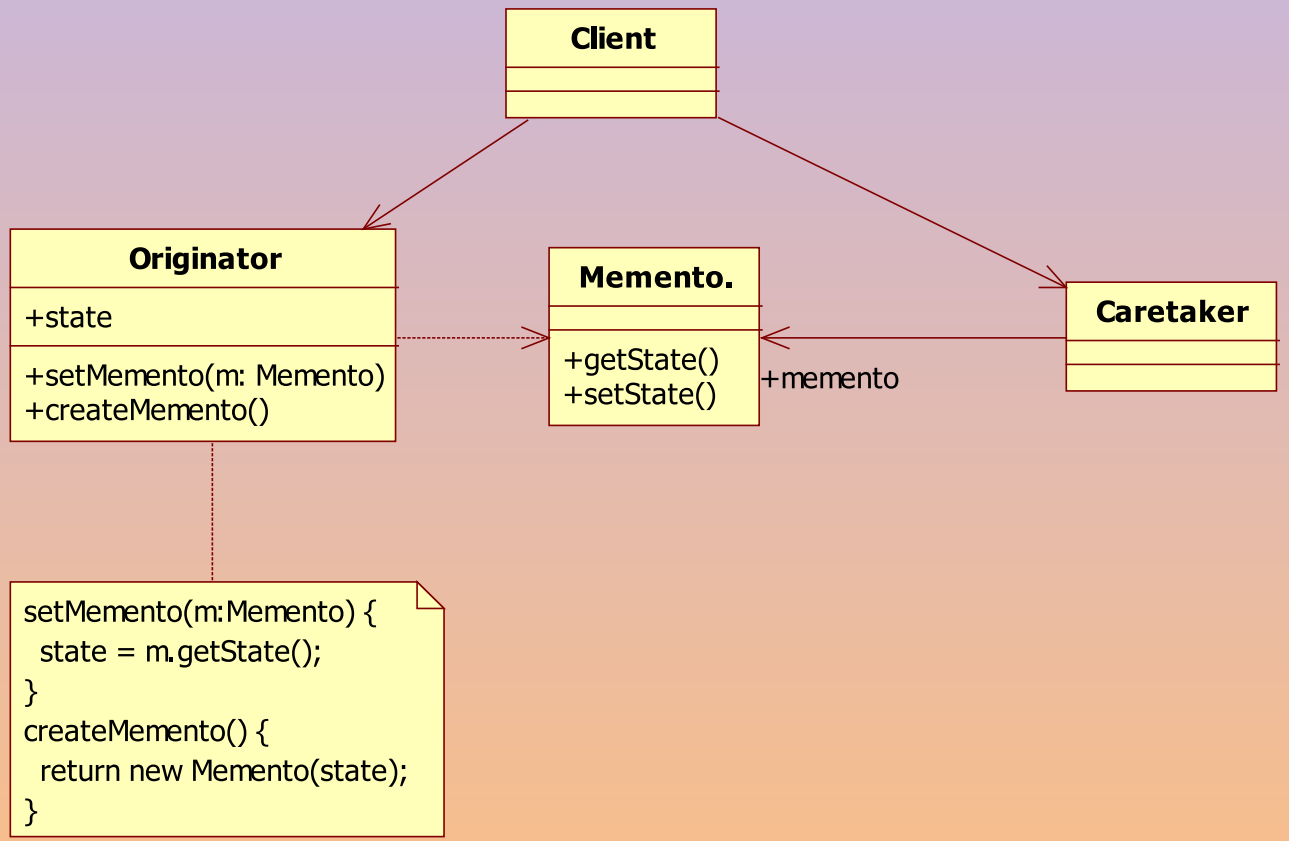
[WIterator_pattern]

Iterator



Memento

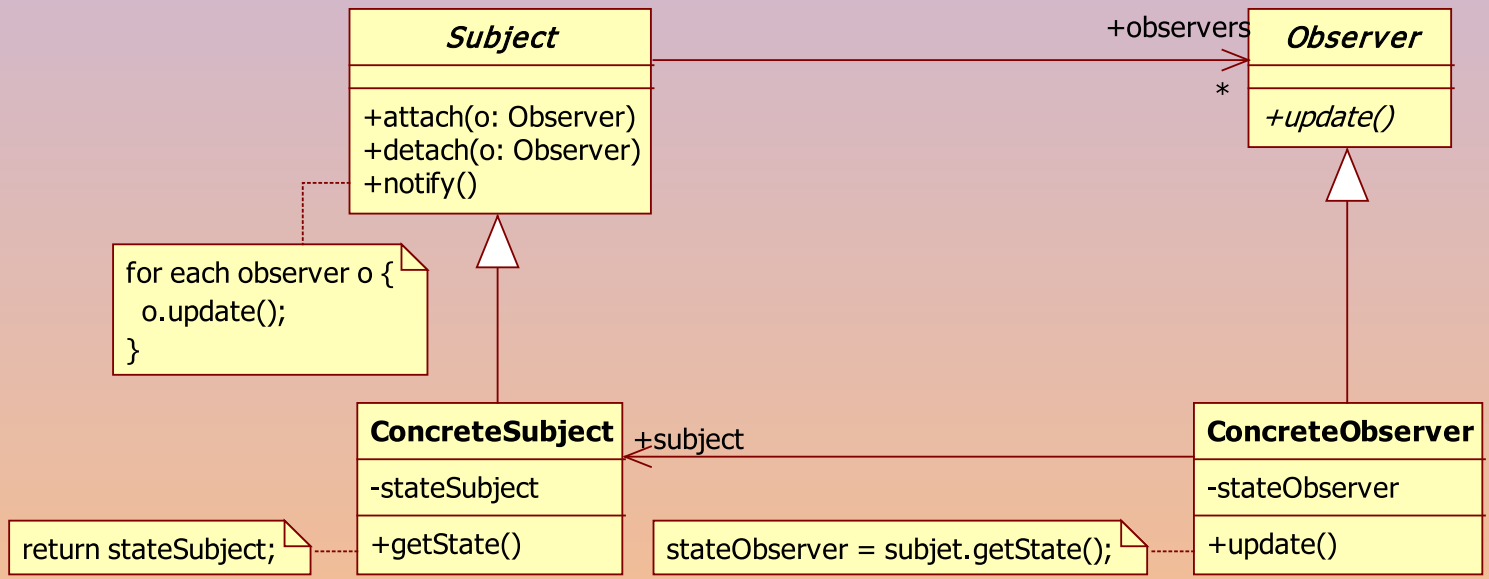
Capture and externalize an object's internal state
so that the object can be restored to this state later.
E.g., Undo/redo stacks, seeds from PRNGs.



[WMemento_pattern]

Observer

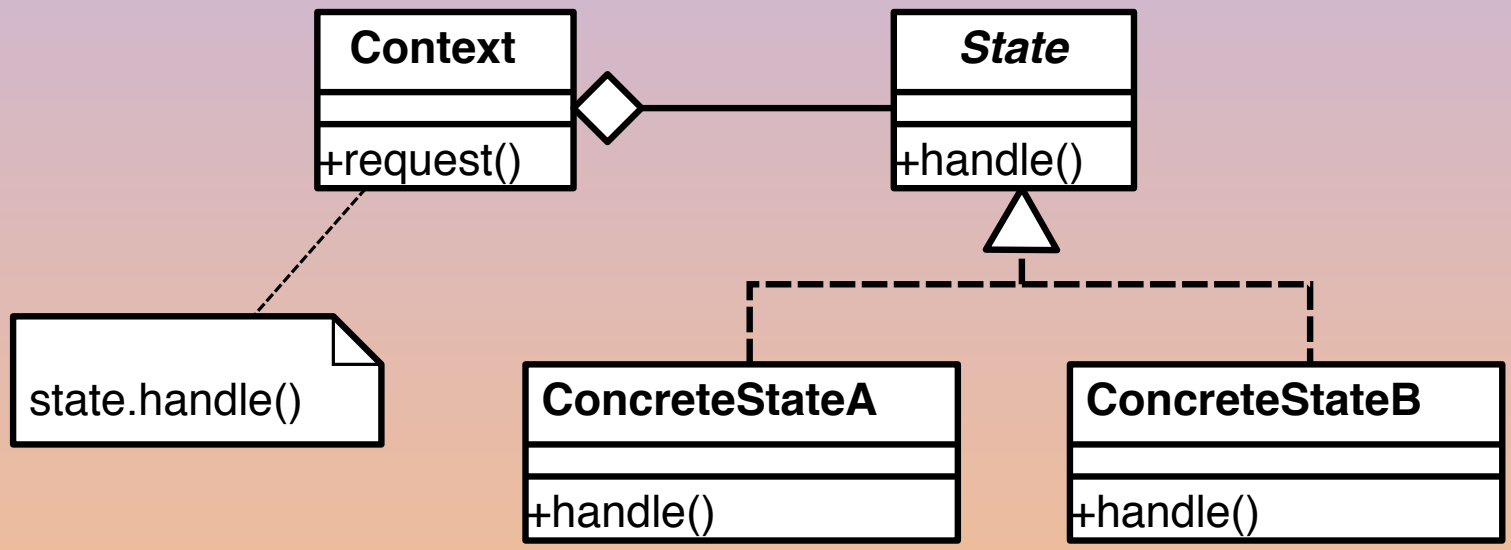
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.



[WObserver_pattern]

State

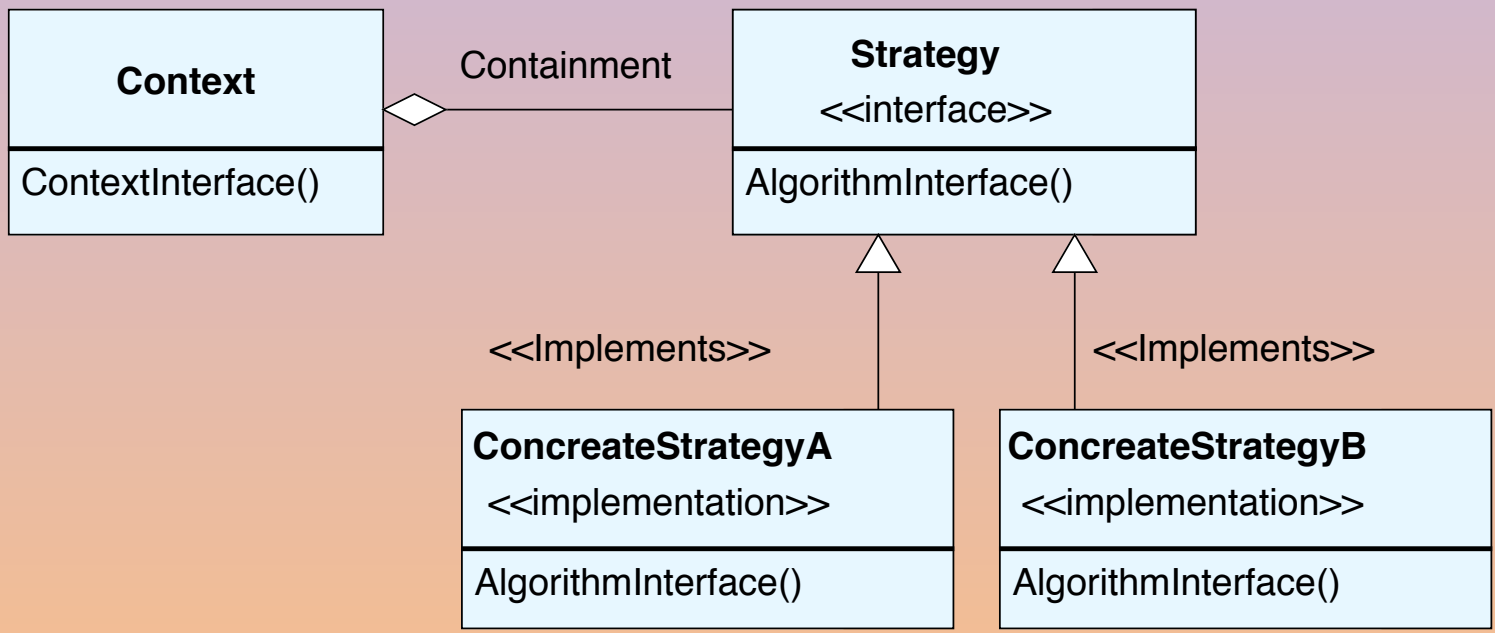
Allow an object to alter its behavior when its internal state changes.
The object will appear to change its class.



[WState_pattern]

Strategy

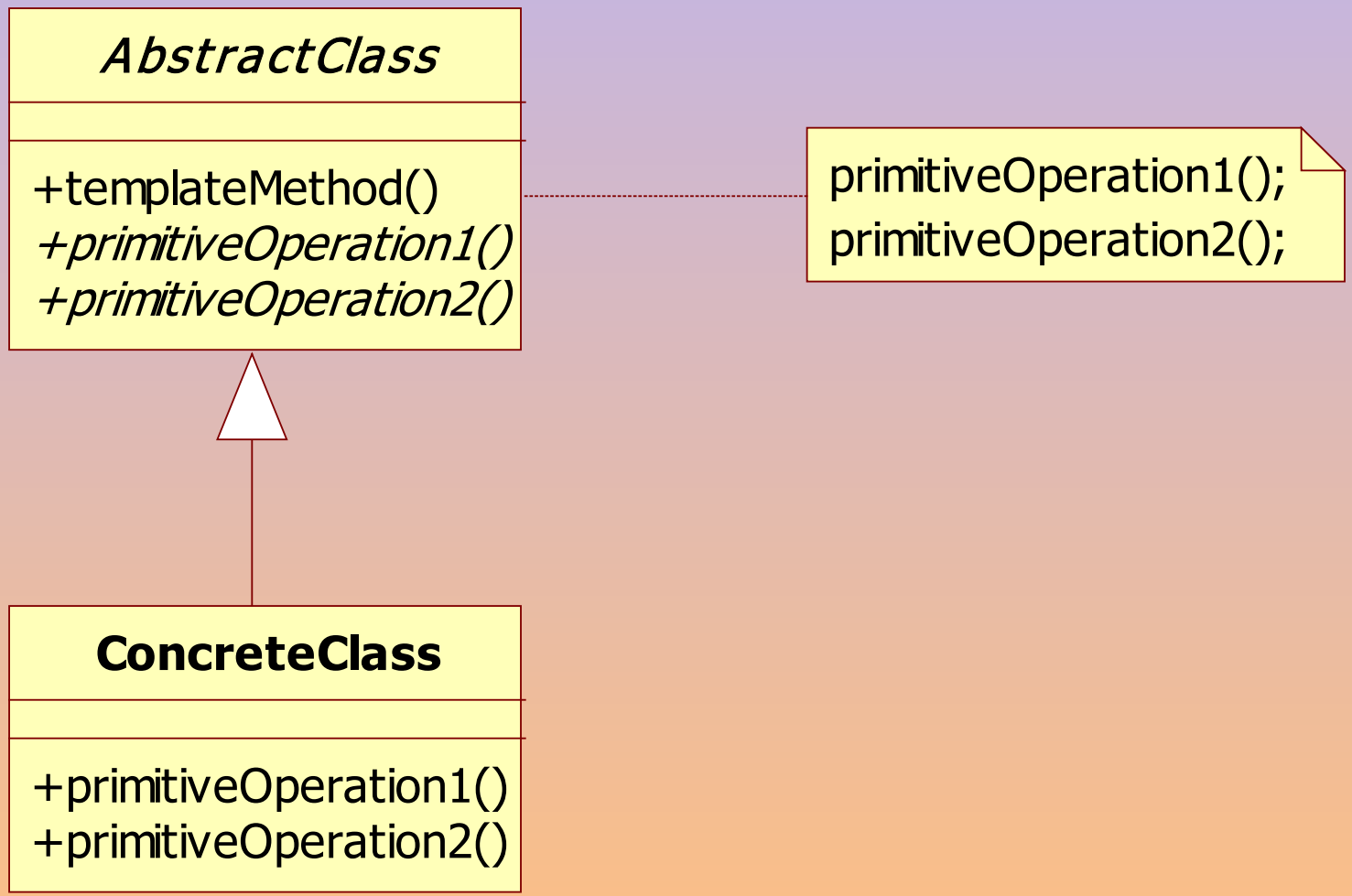
Define a family of algorithms, encapsulate each one, and make them interchangeable.
Alike Bridge, but Strategy is meant for behavior, and Bridge for structure.



[WStrategy_pattern]

Template Method

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.



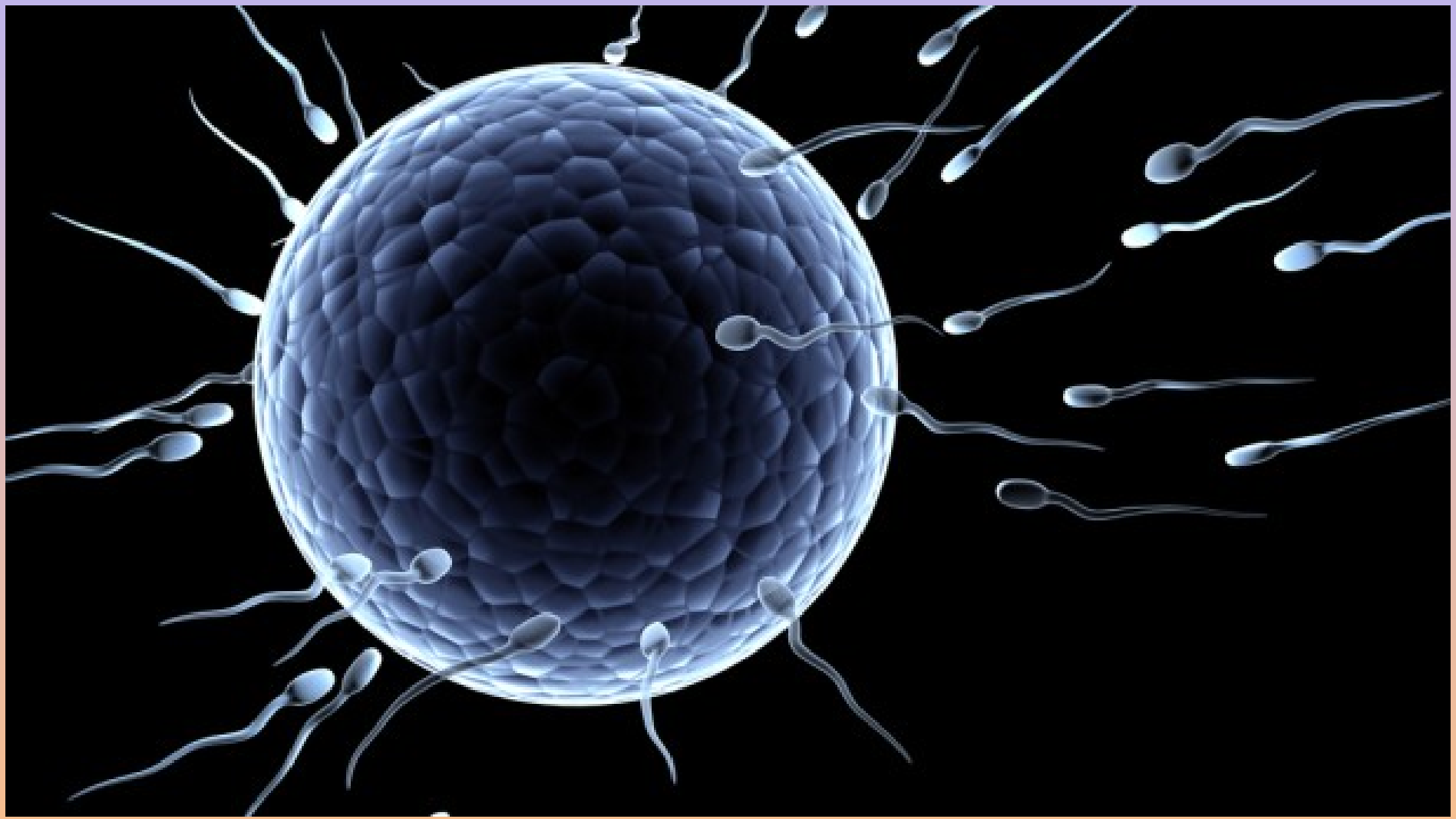
[W Template Method pattern]

Creational Patterns

Singleton

Singleton

Singleton



By The Book

Singleton (Object Creational)

Intent
- Ensure a class only has one instance
- provide a global point of access to it

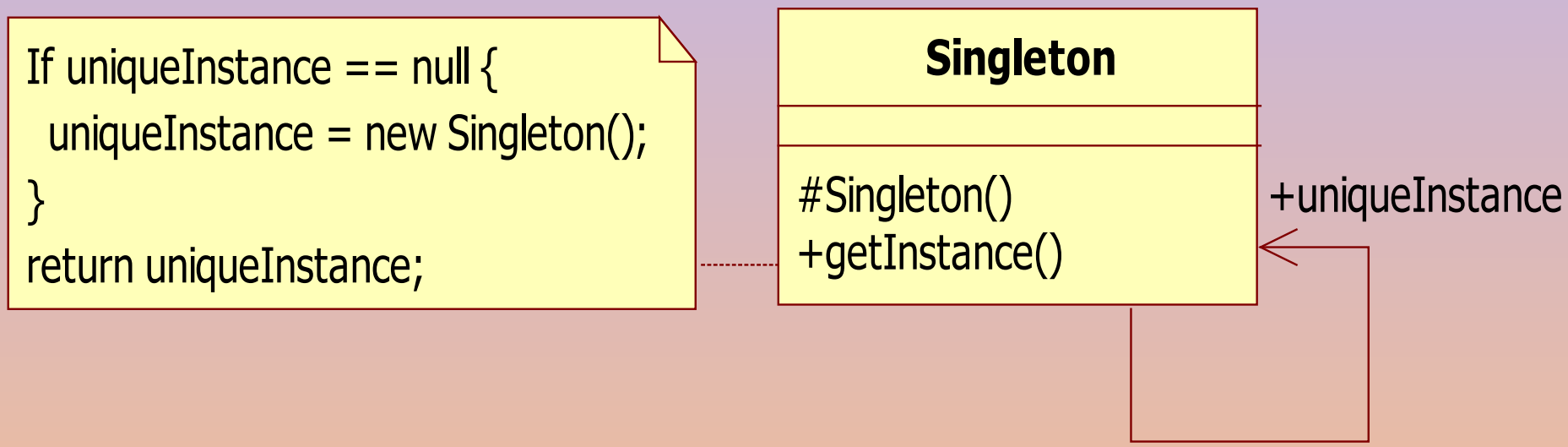
AKA

Motivation One file system, one window manager, etc.

Applicability
- there must be exactly one instance of a class
- accessible to clients from a well-known access point
- the sole instance should be extensible by subclassing
- clients should be able to use an extended instance (without modifying their code)

Singleton (Object Creational)

Structure



Singleton (Object Creational)

Consequences

- strictly controlled access to sole instance
- reduced name space (fewer globals)
- permits refinement of operations and representation (subclassing)
- permits a variable number of instances
- more flexible than static members (cannot be virtual either)
- guaranteed initialization

Implementation

- reclaiming the memory

Implementation in C++

Static Namespace-level Variables

'display.so'

```
static Display display;
```

'keyboard.so'

```
static Keyboard keyboard;
```

'log.so'

```
static Log log;
```

Static Namespace-level Variables

'display.so'

```
static Display display;
```

'keyboard.so'

```
static Keyboard keyboard;
```

'log.so'

```
static Log log;
```

- no guaranteed order across compilers
- no portable order for a given toolchain
- depending on the 'Makefile' is insane

Static Namespace-level Variables

'display.so'

```
static Display display;
```

'keyboard.so'

```
static Keyboard keyboard;
```

'log.so'

```
static Log log;
```

- no guaranteed order across compilers
- no portable order for a given toolchain
- depending on the 'Makefile' is insane

- guaranteed initialization
- guaranteed initialization order within the file
- guaranteed cleanup
- guaranteed cleanup order within the file

Not a Singleton

```
class MyOnlyPrinter
{
public:
  static void AddPrintJob(PrintJob& newJob) { ... }
private:
  // All data is static
  static std::queue<PrintJob> printQueue_;
  static PrinterPort printingPort_;
  static Font defaultFont_;
};
```

```
PrintJob somePrintJob("MyDocument.txt");
MyOnlyPrinter::AddPrintJob(somePrintJob);
```

Not a Singleton

```
class MyOnlyPrinter
{
public:
  static void AddPrintJob(PrintJob& newJob) { ... }
private:
  // All data is static
  static std::queue<PrintJob> printQueue_;
  static PrinterPort printingPort_;
  static Font defaultFont_;
};
```

```
PrintJob somePrintJob("MyDocument.txt");
MyOnlyPrinter::AddPrintJob(somePrintJob);
```

- no `virtual` function
- unclear initialization
- unclear cleanup

Singleton: static member

```
class Singleton
{
public:
  static Singleton* instance();
protected:                                  // or private.
  Singleton();                              // Forbid foreign instantiation.
private:
  Singleton(const Singleton&) = delete; // Forbid duplication.
  static Singleton* instance_;
};
```

```
Singleton* Singleton::instance_ = nullptr;

Singleton* Singleton::instance()
{
  if (!instance_)
    instance_ = new Singleton;
  return instance_;
}
```

Singleton: static member by reference

```
class Singleton
{
public:
  static Singleton& instance();          // Claim ownership.
protected:                               // or private.
  Singleton();                           // Forbid foreign instantiation.
private:
  Singleton(const Singleton&) = delete; // Forbid duplication.
  Singleton& operator=(const Singleton&) = delete; // Forbid nonsense.
  ~Singleton();                          // Enforce ownership.

  static Singleton* instance_;
};
```

Singleton Leaks!

- one `new`, no `delete` $\Rightarrow$ trouble!
- really?
- the OS is a fine garbage collector!
- but the OS does not recover everything
- especially OS-wide resources
- a leak is. . . a leak

Singleton Leaks!

- one `new`, no `delete` $\Rightarrow$ trouble!
- really?
- the OS is a fine garbage collector!
- but the OS does not recover everything
- especially OS-wide resources
- a leak is. . . a leak

Singleton Leaks!

- one `new`, no `delete` $\Rightarrow$ trouble!
- really?
- the OS is a fine garbage collector!
- but the OS does not recover everything
- especially OS-wide resources
- a leak is. . . a leak

Singleton Leaks!

- one `new`, no `delete` $\Rightarrow$ trouble!
- really?
- the OS is a fine garbage collector!
- but the OS does not recover everything
- especially OS-wide resources
- a leak is. . . a leak

Singleton Leaks!

- one `new`, no `delete` $\Rightarrow$ trouble!
- really?
- the OS is a fine garbage collector!
- but the OS does not recover everything
- especially OS-wide resources
- a leak is. . . a leak

Singleton Leaks!

- one `new`, no `delete` $\Rightarrow$ trouble!
- really?
- the OS is a fine garbage collector!
- but the OS does not recover everything
- especially OS-wide resources
- a leak is. . . a leak

Local Static Variable [Meyers, 1996, Item 26]

```
Singleton& Singleton::instance()
{
  static Singleton instance;
  return instance;
}
```

Local Static Variable [Alexandrescu, 2001, 6.4]

```
Singleton& Singleton::instance()
{
  // Functions generated by the compiler.
  extern void __ConstructSingleton(void* memory);
  extern void __DestroySingleton();
  // Variables generated by the compiler.
  static bool __initialized = false;
  // Buffer that holds the singleton (assumed properly aligned).
  static char __buffer[sizeof(Singleton)];
  if (!__initialized)
  {
    // Invokes Singleton::Singleton in the __buffer memory.
    __ConstructSingleton(__buffer);
    // Register destruction.
    atexit(__DestroySingleton);
    __initialized = true;
  }
  return *reinterpret_cast<Singleton*>(__buffer);
}
```

Issues with Meyers's Implementation of Singletons

- guaranteed order of initializations
- dependency resolution
- guaranteed order of cleanups (`atexit` works lifo)
- not robust to failing initializations
- maybe lifo is not right for you
- see [Alexandrescu, 2001, 6.6–6.8] for a top notch implementation

Issues with Meyers's Implementation of Singletons

- guaranteed order of initializations
- dependency resolution
- guaranteed order of cleanups (`atexit` works lifo)
- not robust to failing initializations
- maybe lifo is not right for you
- see [Alexandrescu, 2001, 6.6–6.8] for a top notch implementation

Issues with Meyers's Implementation of Singletons

- guaranteed order of initializations
- dependency resolution
- guaranteed order of cleanups (`atexit` works lifo)
- not robust to failing initializations
- maybe lifo is not right for you
- see [Alexandrescu, 2001, 6.6–6.8] for a top notch implementation

Issues with Meyers's Implementation of Singletons

- guaranteed order of initializations
- dependency resolution
- guaranteed order of cleanups (`atexit` works lifo)
- not robust to failing initializations
- maybe lifo is not right for you
- see [Alexandrescu, 2001, 6.6–6.8] for a top notch implementation

Issues with Meyers's Implementation of Singletons

- guaranteed order of initializations
- dependency resolution
- guaranteed order of cleanups (`atexit` works lifo)
- not robust to failing initializations
- maybe lifo is not right for you
- see [Alexandrescu, 2001, 6.6–6.8] for a top notch implementation

Issues with Meyers's Implementation of Singletons

- guaranteed order of initializations
- dependency resolution
- guaranteed order of cleanups (`atexit` works lifo)
- not robust to failing initializations
- maybe lifo is not right for you
- see [Alexandrescu, 2001, 6.6–6.8] for a top notch implementation

Thread Safety? Multicore Safety?

```
Singleton& Singleton::instance()
{
  static Singleton instance;
  return instance;
}
```

- guaranteed to be atomic by GCC
- VC++ (circa 2010) fails

> *What you see here is not a compiler bug.*
> *This behavior is required by the C++ standard.*
>
> —C++ scoped static initialization is not thread-safe, on purpose!
>
> blogs.msdn.com/b/oldnewthing/archive/2004/03/08/85901.aspx

(100% pure BS).

Thread Safety? Multicore Safety?

```
Singleton& Singleton::instance()
{
  static Singleton instance;
  return instance;
}
```

- guaranteed to be atomic by GCC
- VC++ (circa 2010) fails

> *What you see here is not a compiler bug.*
> *This behavior is required by the C++ standard.*
>
> — C++ scoped static initialization is not thread-safe, on purpose!
>
> blogs.msdn.com/b/oldnewthing/archive/2004/03/08/85901.aspx

(100% pure BS).

Thread Safety? Multicore Safety?

```
Singleton& Singleton::instance()
{
  static Singleton instance;
  return instance;
}
```

- guaranteed to be atomic by GCC
- VC++ (circa 2010) fails

> *What you see here is not a compiler bug.*
> ***This behavior is required by the C++ standard.***
>
> — C++ scoped static initialization is not thread-safe, on purpose!
>
> blogs.msdn.com/b/oldnewthing/archive/2004/03/08/85901.aspx

(100% pure BS).

Thread Safety

```
1  Singleton& Singleton::instance()
2  {
3    if (!instance_)
4      instance_ = new Singleton;
5    return *instance_;
6  }
```

- thread A executes 3 and is interrupted
- thread B runs 3, 4 and 5
- when resumed, thread A runs 4 and 5
- two instances!

Thread Safety

```
1  Singleton& Singleton::instance()
2  {
3    if (!instance_)
4      instance_ = new Singleton;
5    return *instance_;
6  }
```

- thread A executes 3 and is interrupted
- thread B runs 3, 4 and 5
- when resumed, thread A runs 4 and 5
- two instances!

Thread Safety

```
1  Singleton& Singleton::instance()
2  {
3    if (!instance_)
4      instance_ = new Singleton;
5    return *instance_;
6  }
```

- thread A executes 3 and is interrupted
- thread B runs 3, 4 and 5
- when resumed, thread A runs 4 and 5
- two instances!

Thread Safety

```
1  Singleton& Singleton::instance()
2  {
3    if (!instance_)
4      instance_ = new Singleton;
5    return *instance_;
6  }
```

- thread A executes 3 and is interrupted
- thread B runs 3, 4 and 5
- when resumed, thread A runs 4 and 5
- two instances!

Thread Safety

```
1  Singleton& Singleton::instance()
2  {
3    if (!instance_)
4      instance_ = new Singleton;
5    return *instance_;
6  }
```

- thread A executes 3 and is interrupted
- thread B runs 3, 4 and 5
- when resumed, thread A runs 4 and 5
- two instances!

Once Routines in Eiffel

```
make_window: WINDOW
        do
            create Result.make (...)
        end
```

```
console: WINDOW
            -- Shared console window
        once
            create Result.make (...)
        end
```

```
shared_object: SOME_TYPE
            -- An object that can be shared among threads
            -- without being reinitialized.
        once ("PROCESS")
            create Result.make (...)
        end
```

Once Routines in C++ 11

```
#include <mutex>

class Singleton {
public:
  static Singleton& instance()
  {
    std::call_once(once_flag_, []() { instance_ = new Singleton; } );
    return *instance_;
  }

private:
  Singleton() = default;
  Singleton(const Singleton&) = delete;
  Singleton& operator=(const Singleton&) = delete;
  static Singleton* instance_;
  static std::once_flag once_flag_;
};

Singleton* Singleton::instance_ = nullptr;
std::once_flag Singleton::once_flag_;
```

Thread Safety: Lock it!

```
1  Singleton& Singleton::instance()
2  {
3    std::lock_guard<std::mutex> lock(mutex_);
4    if (!instance_)
5      instance_ = new Singleton;
6    return *instance_;
7  }
```

- yeah!
- (but how is `mutex_` defined?)
- can be expensive!

Thread Safety: Lock it!

```
1  Singleton& Singleton::instance()
2  {
3    std::lock_guard<std::mutex> lock(mutex_);
4    if (!instance_)
5      instance_ = new Singleton;
6    return *instance_;
7  }
```

- yeah!
- (but how is `mutex_` defined?)
- can be expensive!

Thread Safety: Lock it!

```
1  Singleton& Singleton::instance()
2  {
3    std::lock_guard<std::mutex> lock(mutex_);
4    if (!instance_)
5      instance_ = new Singleton;
6    return *instance_;
7  }
```

- yeah!
- (but how is `mutex_` defined?)
- can be expensive!

Thread Safety: Lock it!

```
1  Singleton& Singleton::instance()
2  {
3    std::lock_guard<std::mutex> lock(mutex_);
4    if (!instance_)
5      instance_ = new Singleton;
6    return *instance_;
7  }
```

- yeah!
- (but how is `mutex_` defined?)
- can be expensive!

Double Check Locking Pattern

[Schmidt, 1996]

```
1  Singleton& Singleton::instance()
2  {
3    if (!instance_)
4    {
5      std::lock_guard<std::mutex> lock(mutex_);
6      if (!instance_)
7        instance_ = new Singleton;
8    }
9    return *instance_;
10 }
```

- yahoo!
- or?

Double Check Locking Pattern

[Schmidt, 1996]

```
1  Singleton& Singleton::instance()
2  {
3    if (!instance_)
4    {
5      std::lock_guard<std::mutex> lock(mutex_);
6      if (!instance_)
7        instance_ = new Singleton;
8    }
9    return *instance_;
10 }
```

- yahoo!
- or?

Double Check Locking Pattern

[Schmidt, 1996]

```
1  Singleton& Singleton::instance()
2  {
3    if (!instance_)
4    {
5      std::lock_guard<std::mutex> lock(mutex_);
6      if (!instance_)
7        instance_ = new Singleton;
8    }
9    return *instance_;
10 }
```

- yahoo!
- or?

Double Check Locking Pattern

Troubles Ahead

The "Double-Checked Locking is Broken" Declaration

David Bacon (IBM Research), Joshua Bloch (Javasoft), Jeff Bogda, Cliff Click (Hotspot JVM project), Paul Haahr, Doug Lea, Tom May, Jan-Willem Maessen, Jeremy Manson, John D. Mitchell (jGuru), Kelvin Nilsen, Bill Pugh, Emin Gun Sirer

> *Lots of very smart people have spent lots of time looking at this. There is* no way *to make it work without requiring each thread that accesses the helper object to perform synchronization.*
>
> — [Bacon et al.,]

Double Check Locking Pattern

Troubles Ahead

The "Double-Checked Locking is Broken" Declaration

David Bacon (IBM Research), Joshua Bloch (Javasoft), Jeff Bogda, Cliff Click (Hotspot JVM project), Paul Haahr, Doug Lea, Tom May, Jan-Willem Maessen, Jeremy Manson, John D. Mitchell (jGuru), Kelvin Nilsen, Bill Pugh, Emin Gun Sirer

> *Lots of very smart people have spent lots of time looking at this. There is* no way *to make it work without requiring each thread that accesses the helper object to perform synchronization.*
>
> — [Bacon et al.,]

Newsflash: Double-checked locking is still broken

www.javaworld.com/javaqa/2002-04/01-qa-0412-doublelock.html?page=1

- "The 'Double-Checked Locking Is Broken' Declaration," David Bacon, et al.
- "Double-Checked Locking: Clever, but Broken," Brian Goetz (JavaWorld, February 2001)
- "Warning! Threading in a Multiprocessor World," Allen Holub (JavaWorld, February 2001)
- "Can Double-Checked Locking Be Fixed?," Brian Goetz (JavaWorld, May 2001)
- "Can ThreadLocal Solve the Double-Checked Locking Problem?," Brian Goetz (JavaWorld, November 2001)

What about C++?

> *Very experienced multithreaded programmers know that even the Double-Checked Locking pattern, although correct on paper, is not always correct in practice. [...] Thus, sadly, the Double-Checked Locking pattern is known to be defective for [multicore] systems.*
>
> *In conclusion, you should check your compiler documentation before implementing the Double-Checked Locking pattern. (This makes it the Triple-Checked Locking pattern.)*
>
> *Usually the platform offers alternative, nonportable concurrency-solving primitives, such as memory barriers, which ensure ordered access to memory. At least, put a `volatile` qualifier next to `instance_`. A reasonable compiler should generate correct, nonspeculative code around volatile objects.*
>
> — [Alexandrescu, 2001, 6.9]

What about C++?

“Very experienced multithreaded programmers know that even the Double-Checked Locking pattern, although correct on paper, is not always correct in practice. [...] Thus, sadly, the Double-Checked Locking pattern is known to be defective for [multicore] systems.
In conclusion, you should check your compiler documentation before implementing the Double-Checked Locking pattern. (This makes it the Triple-Checked Locking pattern.)
Usually the platform offers alternative, nonportable concurrency-solving primitives, such as memory barriers, which ensure ordered access to memory. At least, put a `volatile` qualifier next to `instance_`. A reasonable compiler should generate correct, nonspeculative code around volatile objects.

— [Alexandrescu, 2001, 6.9]

What about C++?

“Very experienced multithreaded programmers know that even the Double-Checked Locking pattern, although correct on paper, is not always correct in practice. [...] Thus, sadly, the Double-Checked Locking pattern is known to be defective for [multicore] systems.
In conclusion, you should check your compiler documentation before implementing the Double-Checked Locking pattern. (This makes it the Triple-Checked Locking pattern.)
Usually the platform offers alternative, nonportable concurrency-solving primitives, such as memory barriers, which ensure ordered access to memory. At least, put a `volatile` qualifier next to `instance_`. A reasonable compiler should generate correct, nonspeculative code around volatile objects.

— [Alexandrescu, 2001, 6.9]

Double Check Locking Pattern

[Meyers and Alexandrescu, 2004]

```
1  Singleton& Singleton::instance()
2  {
3    if (!instance_)
4    {
5      std::lock_guard<std::mutex> lock(mutex_);
6      if (!instance_)
7      {
8        instance_ =                          // Step 3
9          operator new(sizeof(Singleton)); // Step 1
10       new (instance_) Singleton;          // Step 2
11     }
12   }
13   return *instance_;
14 }
```

- a compiler might invert steps 2 and 3
- thread A runs steps 1 and 3
- thread B enters and see `instance_` nonnull

Double Check Locking Pattern

[Meyers and Alexandrescu, 2004]

```
1  Singleton& Singleton::instance()
2  {
3    if (!instance_)
4    {
5      std::lock_guard<std::mutex> lock(mutex_);
6      if (!instance_)
7      {
8        instance_ =                          // Step 3
9          operator new(sizeof(Singleton)); // Step 1
10       new (instance_) Singleton;          // Step 2
11     }
12   }
13   return *instance_;
14 }
```

- a compiler might invert steps 2 and 3
- thread A runs steps 1 and 3
- thread B enters and see `instance_` nonnull

Double Check Locking Pattern

[Meyers and Alexandrescu, 2004]

```
1  Singleton& Singleton::instance()
2  {
3    if (!instance_)
4    {
5      std::lock_guard<std::mutex> lock(mutex_);
6      if (!instance_)
7      {
8        instance_ =                          // Step 3
9          operator new(sizeof(Singleton)); // Step 1
10       new (instance_) Singleton;          // Step 2
11     }
12   }
13   return *instance_;
14 }
```

- a compiler might invert steps 2 and 3
- thread A runs steps 1 and 3
- thread B enters and see `instance_` nonnull

Double Check Locking Pattern in C++

> *DCLP will work only if steps 1 and 2 are completed before step 3 is performed, but*
>
> *there is no way to express this constraint in C or C++* .
>
> — [Meyers and Alexandrescu, 2004]

Compilers

- reorder statements
- to take advantage of their multiple process units
- to avoid spilling
- to perform common subexpression elimination
- to keep the pipeline full
- to do what ever they like to *optimize*
- provided the observable behavior is preserved

- "observable" relatively to the standard's abstract machine
- and neither C nor C++ *languages* feature threads
- so *escape from C/C++* (`asm`)
- or force a variable to be considered as observable?

Compilers

- reorder statements
- to take advantage of their multiple process units
- to avoid spilling
- to perform common subexpression elimination
- to keep the pipeline full
- to do what ever they like to *optimize*
- provided the observable behavior is preserved

- "observable" relatively to the standard's abstract machine
- and neither C nor C++ *languages* feature threads
- so *escape from C/C++* (`asm`)
- or force a variable to be considered as observable?

Compilers

- reorder statements
- to take advantage of their multiple process units
- to avoid spilling
- to perform common subexpression elimination
- to keep the pipeline full
- to do what ever they like to *optimize*
- provided the observable behavior is preserved

- "observable" relatively to the standard's abstract machine
- and neither C nor C++ *languages* feature threads
- so *escape from C/C++* (`asm`)
- or force a variable to be considered as observable?

Compilers

- reorder statements
- to take advantage of their multiple process units
- to avoid spilling
- to perform common subexpression elimination
- to keep the pipeline full
- to do what ever they like to *optimize*
- provided the observable behavior is preserved

- "observable" relatively to the standard's abstract machine
- and neither C nor C++ *languages* feature threads
- so *escape from C/C++* (`asm`)
- or force a variable to be considered as observable?

Compilers

- reorder statements
- to take advantage of their multiple process units
- to avoid spilling
- to perform common subexpression elimination
- to keep the pipeline full
- to do what ever they like to *optimize*
- provided the observable behavior is preserved

- "observable" relatively to the standard's abstract machine
- and neither C nor C++ *languages* feature threads
- so *escape from C/C++* (`asm`)
- or force a variable to be considered as observable?

Volatile to the Rescue

Volatile to the (Alexand)Rescue

```
class Singleton {
public:
  static volatile Singleton& volatile instance();
  ...
private:
  static volatile Singleton* volatile instance_;
};
```

```
volatile Singleton* volatile Singleton::instance_ = nullptr;
volatile Singleton& volatile Singleton::instance() {
  if (!instance_)
  {
    std::lock_guard<std::mutex> lock(mutex_);
    if (!instance_) {
      volatile Singleton* volatile temp = new volatile Singleton;
      instance_ = temp;
    }
  }
  return *instance_;
}
```

Volatiles to the Rescue

Volatiles to the Rescue

Volatiles Don't Suffice

- the standards does not ensure reordering *across* threads
- `volatile` is enforced once the construction completed

```
volatile Singleton* volatile temp = new volatile Singleton;
```

- can be fought against via more `volatile` *in* the `Singleton`
- and what about multicores?

Use Protection

```
Singleton* Singleton::instance()
{
  Singleton* tmp = instance_;
  MemoryBarrier();     // "acquire": prevent arrival.
  if (!res)
  {
    std::lock_guard<std::mutex> lock(mutex_);
    res = instance_;
    if (!res)
    {
      res = new Singleton;
      MemoryBarrier(); // "release": prevent departure.
      instance_ = tmp;
    }
  }
  return *res;
}
```

Acquire and Release Semantics

http://preshing.com/20120913/acquire-and-release-semantics/

Acquire semantics is a property which can only apply to operations which read from shared memory, whether they are read-modify-write operations or plain loads.
The operation is then considered a *read-acquire*.
Acquire semantics prevent memory reordering of the read-acquire with any read or write operation which follows it in program order.

Release semantics is a property which can only apply to operations which write to shared memory, whether they are read-modify-write operations or plain stores.
The operation is then considered a *write-release*.
Release semantics prevent memory reordering of the write-release with any read or write operation which precedes it in program order.

With TBB (Lazy Initialization)

[Intel, 2011, Lazy Initialization]

```
template <typename T, typename Mutex=tbb::mutex>
class lazy {
public:
  lazy() : value() {}          // Initializes value to nullptr
  ~lazy() { delete value; }
  T& get()
  {
    if (!value)                // Read of value has acquire semantics.
    {
      Mutex::scoped_lock lock(mut);
      if (!value)
        value = new T();       // Write of value has release semantics
    }
    return *value;
  }
private:
  tbb::atomic<T*> value;
  Mutex mut;
};
```

With TBB's CAS (Lazy Initialization)

[Intel, 2011, Lazy Initialization]

```
// if x equals z, then do x=y. In either case, return old value of x.
x.compare_and_swap(y,z)
```

```
template <typename T> class lazy {
public:
  lazy() = default;                    // Initializes value to nullptr.
  ~lazy() { delete value; }
  T& get() {
    if (!value) {
      T* tmp = new T();
      if (value.compare_and_swap(tmp, nullptr) != nullptr)
        // Another thread installed the value, so throw away mine.
        delete tmp;
    }
    return *value;
  }
private:
  tbb::atomic<T*> value;
};
```

Ways Out

- do you really need to optimize the Singleton with DLCP?
- if you do, use thread-local caches
- or use one Singleton per thread
- initialize *before* going multithreaded
- C++11

Ways Out

- do you really need to optimize the Singleton with DLCP?
- if you do, use thread-local caches
- or use one Singleton per thread
- initialize *before* going multithreaded
- C++11

Ways Out

- do you really need to optimize the Singleton with DLCP?
- if you do, use thread-local caches
- or use one Singleton per thread
- initialize *before* going multithreaded
- C++11

Ways Out

- do you really need to optimize the Singleton with DLCP?
- if you do, use thread-local caches
- or use one Singleton per thread
- initialize *before* going multithreaded
- C++11

Ways Out

- do you really need to optimize the Singleton with DLCP?
- if you do, use thread-local caches
- or use one Singleton per thread
- initialize *before* going multithreaded
- C++11

In C++ 11

```
static std::atomic<Singleton*> instance_;

static Singleton& instance()
{
  static thread_local Singleton *instance;
  if (!instance &&
      !(instance = instance_.load(std::memory_order_acquire)))
  {
    std::lock_guard<std::mutex> lock(m_mutex);
    if (!(instance = instance_.load(std::memory_order_relaxed)))
    {
      instance = new Singleton;
      instance_.store(instance, std::memory_order_release);
    }
  }
  return *instance;
}
```

stackoverflow.com/questions/6086912

Safe in C++ 11 [Meyers, 1996, Item 26]

```
Singleton& Singleton::instance()
{
  static Singleton instance;
  return instance;
}
```

But

- do you really need a Singleton?
- this is a glorified global
- that's a smell for bad design
- it hides dependencies (in the code)
- a PITA for test suites
  - tight coupling
  - how can you inject some mock singleton?
  - accumulates state between test runs
- this is not OO! [Yegge, 2004]
- poor memory/resource management
- are you really really sure two will never be needed?
- have a look at Dependency Injection

But

- do you really need a Singleton?
- this is a glorified global
- that's a smell for bad design
- it hides dependencies (in the code)
- a PITA for test suites
  - tight coupling
  - how can you inject some mock singleton?
  - accumulates state between test runs
- this is not OO! [Yegge, 2004]
- poor memory/resource management
- are you really really sure two will never be needed?
- have a look at Dependency Injection

But

- do you really need a Singleton?
- this is a glorified global
- that's a smell for bad design
- it hides dependencies (in the code)
- a PITA for test suites
  - tight coupling
  - how can you inject some mock singleton?
  - accumulates state between test runs
- this is not OO! [Yegge, 2004]
- poor memory/resource management
- are you really really sure two will never be needed?
- have a look at Dependency Injection

But

- do you really need a Singleton?
- this is a glorified global
- that's a smell for bad design
- it hides dependencies (in the code)
- a PITA for test suites
  - tight coupling
  - how can you inject some mock singleton?
  - accumulates state between test runs
- this is not OO! [Yegge, 2004]
- poor memory/resource management
- are you really really sure two will never be needed?
- have a look at Dependency Injection

But

- do you really need a Singleton?
- this is a glorified global
- that's a smell for bad design
- it hides dependencies (in the code)
- a PITA for test suites
  - tight coupling
  - how can you inject some mock singleton?
  - accumulates state between test runs
- this is not OO! [Yegge, 2004]
- poor memory/resource management
- are you really really sure two will never be needed?
- have a look at Dependency Injection

But

- do you really need a Singleton?
- this is a glorified global
- that's a smell for bad design
- it hides dependencies (in the code)
- a PITA for test suites
  - tight coupling
  - how can you inject some mock singleton?
  - accumulates state between test runs
- this is not OO! [Yegge, 2004]
- poor memory/resource management
- are you really really sure two will never be needed?
- have a look at Dependency Injection

But but

- it is really handy
- adequate for things such as logger, service locator
- group “Singletons” into an single one

Factory Method

Factory Method

Factory Method (Class Creational)

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.

AKA Virtual Constructor, Named Constructor

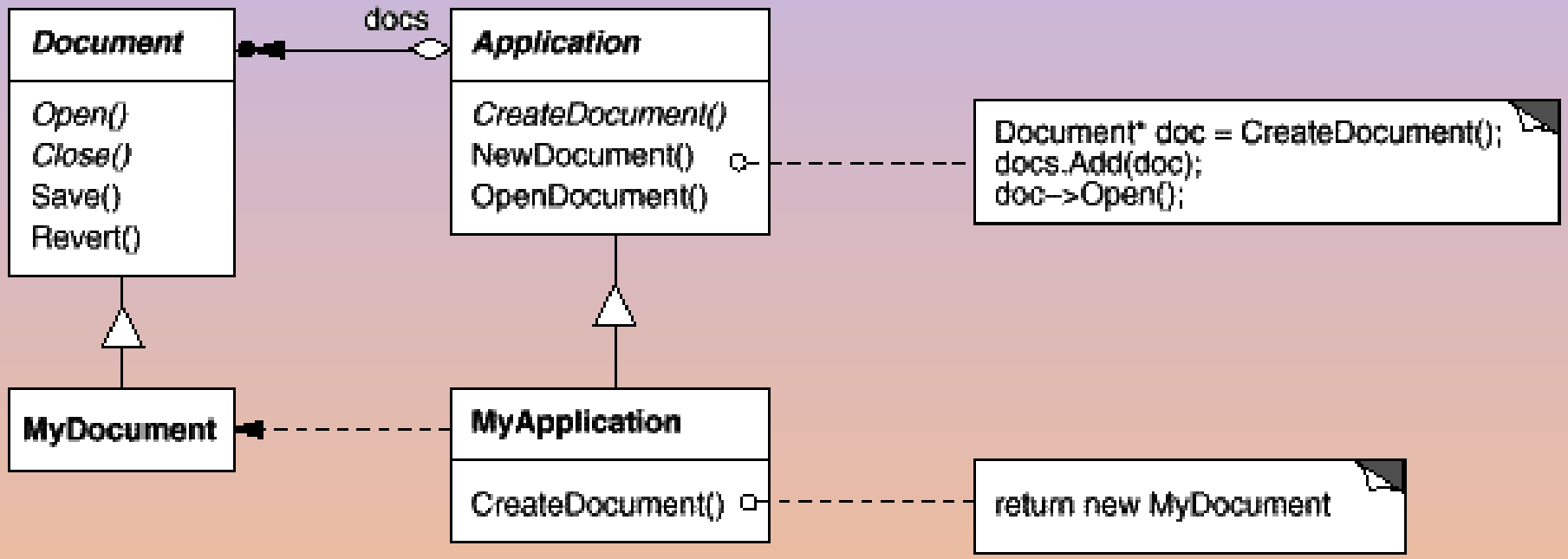
Motivation A framework for applications that can present multiple documents to the user.

Applicability

- a class can't anticipate the class of objects it must create
- a class wants its subclasses to specify the objects it creates
- to delegate responsibility from superclass to one of several helper subclasses

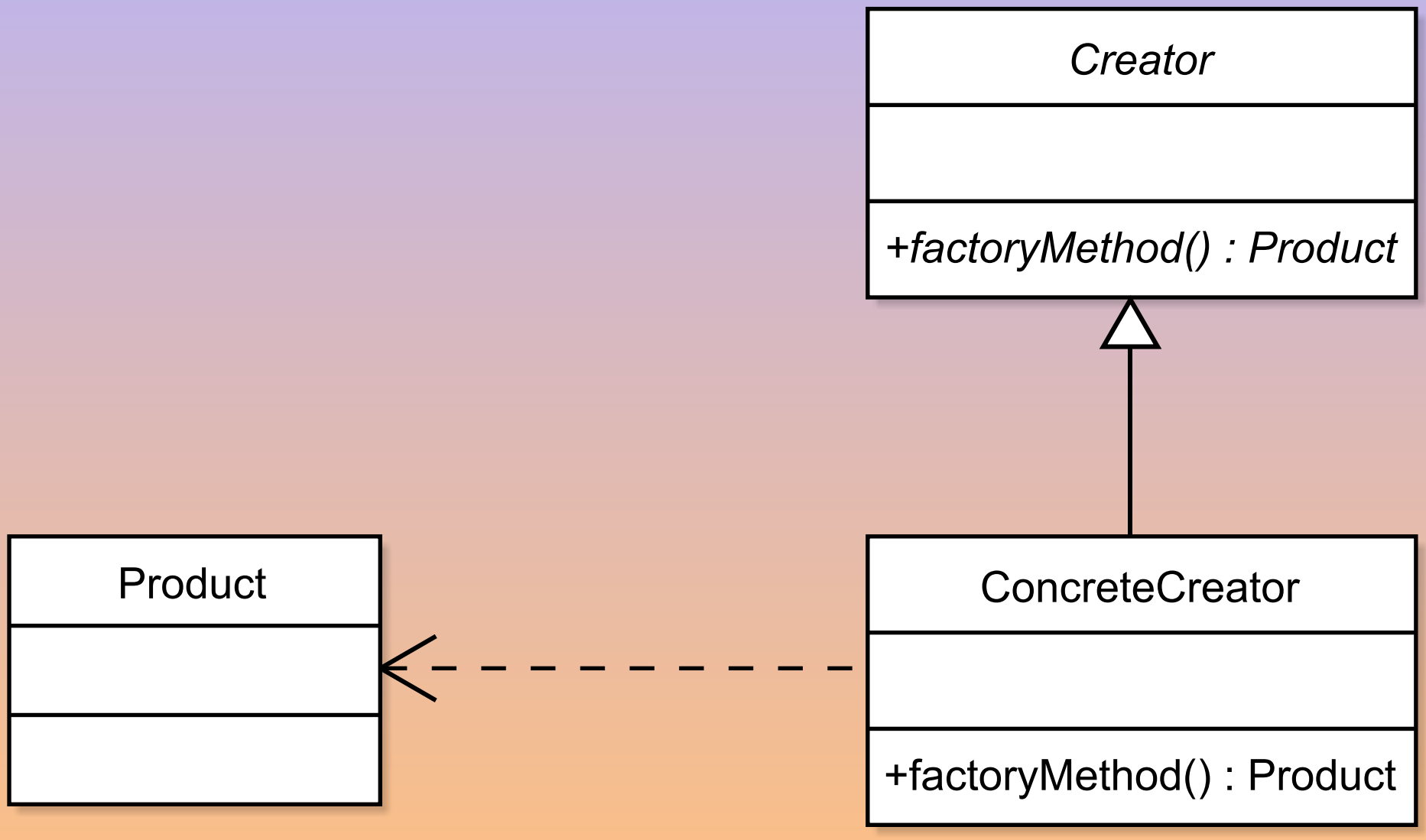
Factory Method (Class Creational)

Motivation



Factory Method (Class Creational)

Structure



Factory Method (Class Creational)

- **Consequences**
  1. eliminates the need to bind application-specific classes into code
  2. (-?) clients might have to subclass `Creator` just to create a particular `ConcreteProduct`
  3. provides hooks for subclasses
  4. connects parallel class hierarchies
- **Implementation**
  1. Abstract creator, or concrete with a default implementation?
  2. Parameterized factory methods
  3. In C++, possibly use templates to avoid subclassing

Factory Method (Class Creational)

Sample Without Design Patterns

```
Maze* MazeGame::CreateMaze()
{
  auto* res = new Maze;
  auto* r1 = new Room{1};  res->AddRoom(r1);
  auto* r2 = new Room{2};  res->AddRoom(r2);
  auto* d = new Door{r1, r2};

  r1->SetSide(North, new Wall);
  r1->SetSide(East,  d);
  r1->SetSide(South, new Wall);
  r1->SetSide(West,  new Wall);

  r2->SetSide(North, new Wall);
  r2->SetSide(East,  new Wall);
  r2->SetSide(South, new Wall);
  r2->SetSide(West,  d);

  return res;
}
```

Factory Method (Class Creational)

Introducing Factory Methods

```
class MazeGame
{
public:
  Maze* CreateMazeGame();

  // Factory methods.
  virtual Door* MakeDoor(Room* r1, Room* r2) const
    { return new Door{r1, r2}; }
  virtual Maze* MakeMaze() const
    { return new Maze; }
  virtual Room* MakeRoom(int n) const
    { return new Room{n}; }
  virtual Wall* MakeWall() const
    { return new Wall; }
};
```

Factory Method (Class Creational)

Using Factory Methods

```
Maze* MazeGame::CreateMaze() {
  auto* res = MakeMaze();
  auto* r1  = MakeRoom(1);  res->AddRoom(r1);
  auto* r2  = MakeRoom(2);  res->AddRoom(r2);
  auto* d   = MakeDoor(r1, r2);

  r1->SetSide(North, MakeWall());
  r1->SetSide(East,  d);
  r1->SetSide(South, MakeWall());
  r1->SetSide(West,  MakeWall());

  r2->SetSide(North, MakeWall());
  r2->SetSide(East,  MakeWall());
  r2->SetSide(South, MakeWall());
  r2->SetSide(West,  d);

  return res;
}
```

Factory Method (Class Creational)

Subclassing MazeGame

```
class EnchantedMazeGame : public MazeGame
{
public:
  EnchantedMazeGame();

  virtual Room* MakeRoom(int n) const
  {
    return new EnchantedRoom{n, CastSpell()};
  }

  virtual Door* MakeDoor(Room* r1, Room* r2) const
  {
    return new DoorNeedingSpell{r1, r2};
  }

protected:
  Spell* CastSpell() const;
};
```

Abstract Factory

Abstract Factory

[WBattersea_Power_Station]

[WQuark,_Strangeness_and_Charm] (Hawkwind, 1977)

Abstract Factory (Object Creational)

Intent Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

AKA Kit

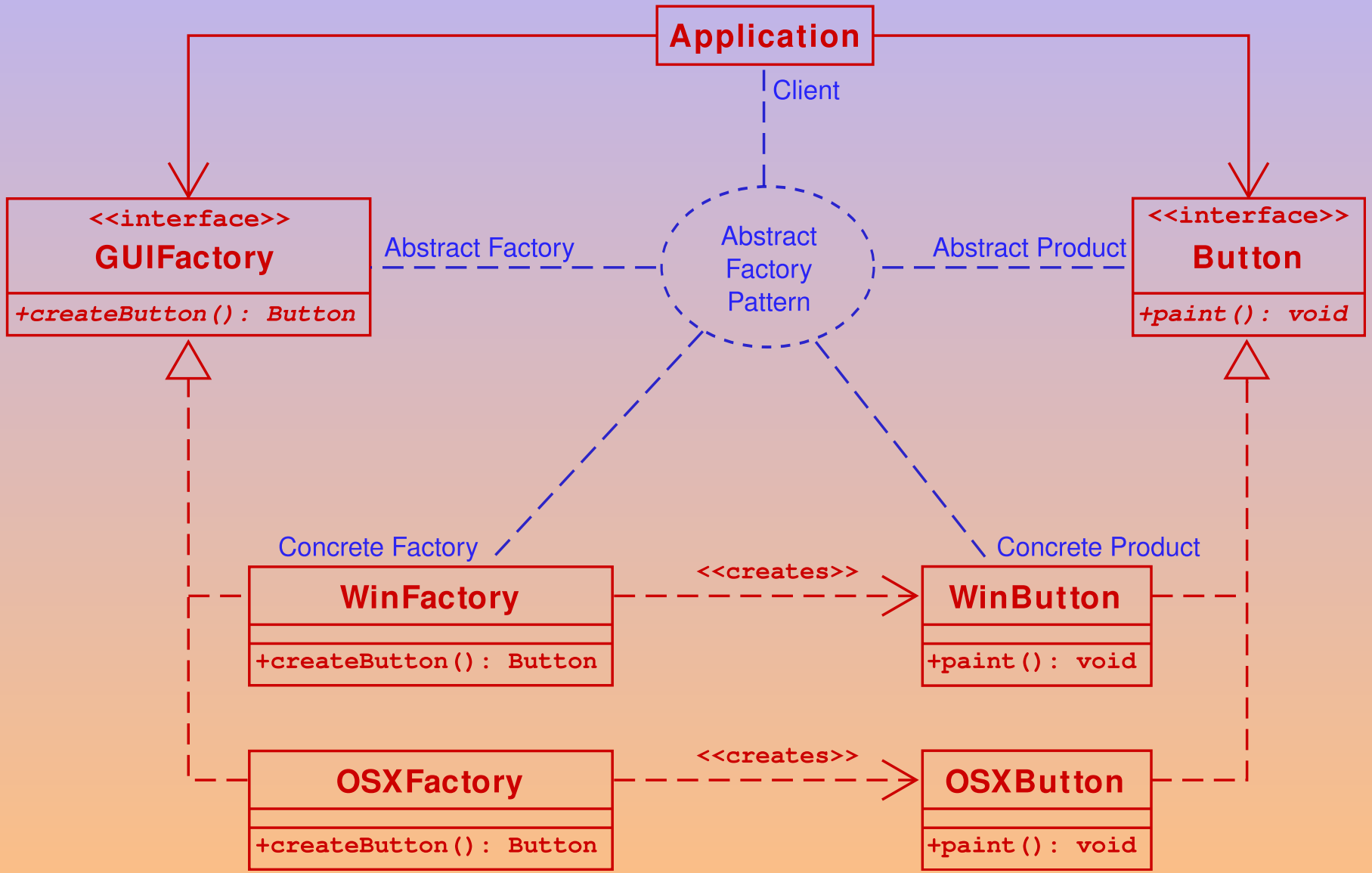
Motivation A user interface toolkit that supports multiple toolkits.

Applicability

- to be independent of the implementation of products
- to be configured with one of multiple product families
- to enforce consistency between members of a family
- to expose only interfaces of a class library of products

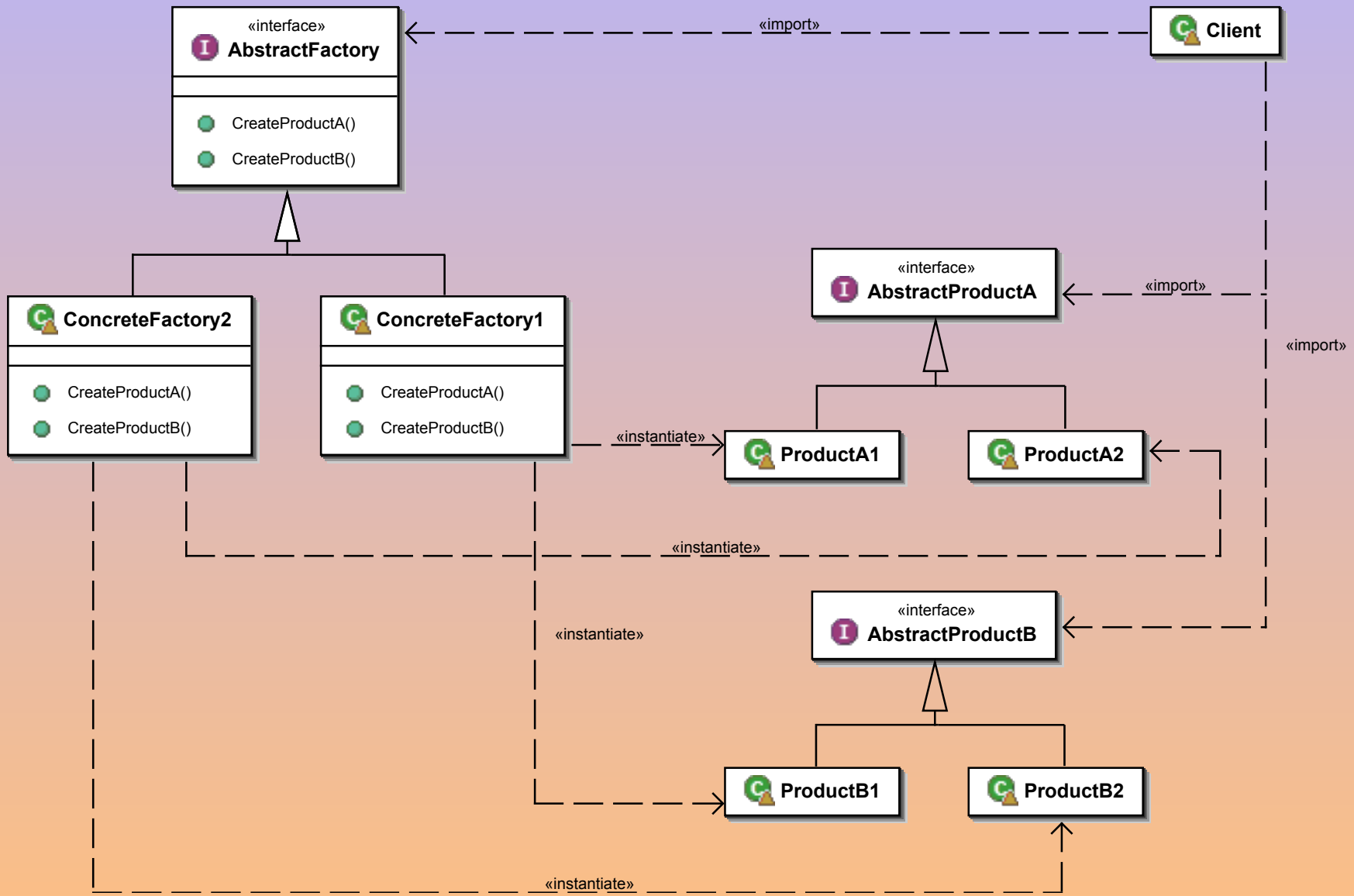
Abstract Factory (Object Creational)

Motivation



Abstract Factory (Object Creational)

Structure



Abstract Factory (Object Creational)

Consequences

1. isolates concrete classes
2. makes exchanging product families easy
3. promotes consistency among products
4. supporting new kinds of products is difficult

Implementation

1. factories as singletons
2. creating the products
   Use a Factory Method, Prototype, etc.

Abstract Factory (Object Creational)

Sample

```
class MazeFactory
{
public:
  MazeFactory();

  virtual Door* MakeDoor(Room* r1, Room* r2) const
    { return new Door{r1, r2}; }
  virtual Maze* MakeMaze() const
    { return new Maze; }
  virtual Room* MakeRoom(int n) const
    { return new Room{n}; }
  virtual Wall* MakeWall() const
    { return new Wall; }
};
```

Abstract Factory (Object Creational)

Sample With Abstract Factory

```
Maze* MazeGame::CreateMaze(MazeFactory& factory) {
  auto* res = factory.MakeMaze();
  auto* r1  = factory.MakeRoom(1);  res->AddRoom(r1);
  auto* r2  = factory.MakeRoom(2);  res->AddRoom(r2);
  auto* d   = factory.MakeDoor(r1, r2);

  r1->SetSide(North, factory.MakeWall());
  r1->SetSide(East,  d);
  r1->SetSide(South, factory.MakeWall());
  r1->SetSide(West,  factory.MakeWall());

  r2->SetSide(North, factory.MakeWall());
  r2->SetSide(East,  factory.MakeWall());
  r2->SetSide(South, factory.MakeWall());
  r2->SetSide(West,  d);

  return res;
}
```

Abstract Factory (Object Creational)

Sample

```
class EnchantedMazeFactory : public MazeFactory
{
public:
  EnchantedMazeFactory();

  virtual Room* MakeRoom(int n) const
    { return new EnchantedRoom{n, CastSpell()}; }

  virtual Door* MakeDoor(Room* r1, Room* r2) const
    { return new DoorNeedingSpell{r1, r2}; }

protected:
  Spell* CastSpell() const;
};
```

Behavioral Patterns

Command

Command


depeche
MODE
MASTER
and
servant
Erstauflage
in
rotem Vinyl

Command (Object Behavioral)

Intent Reify a request to make it manageable

AKA Action, Transaction

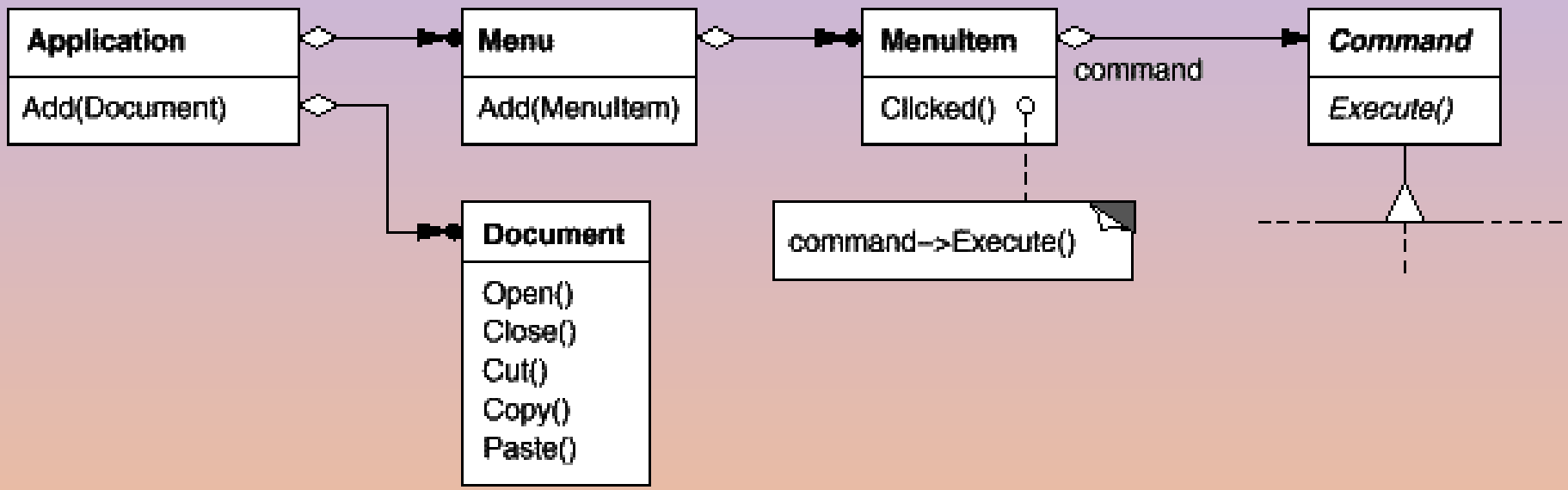
Motivation Reified callbacks for GUI

Applicability

- to parameterize objects by an action to perform
- to specify, queue, and execute requests at different times
- to support undo
- to log changes (to reapply in case of a system crash)
- to structure a system around high-level operations built on primitives operations

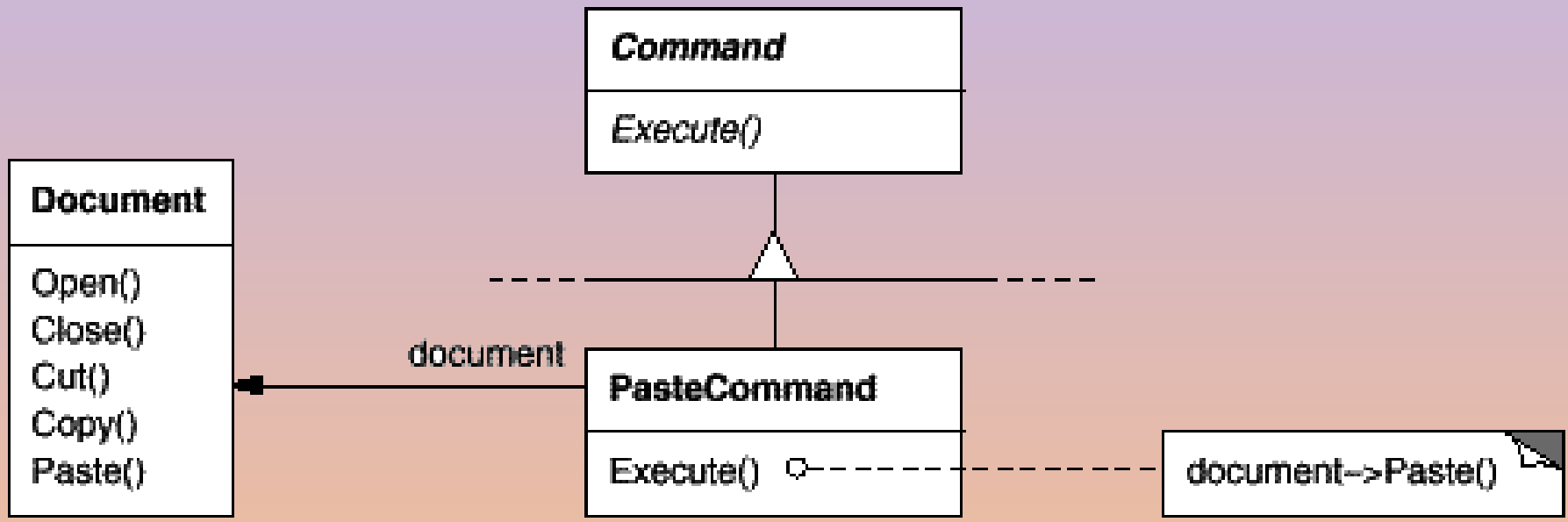
Command (Object Behavioral)

Motivation



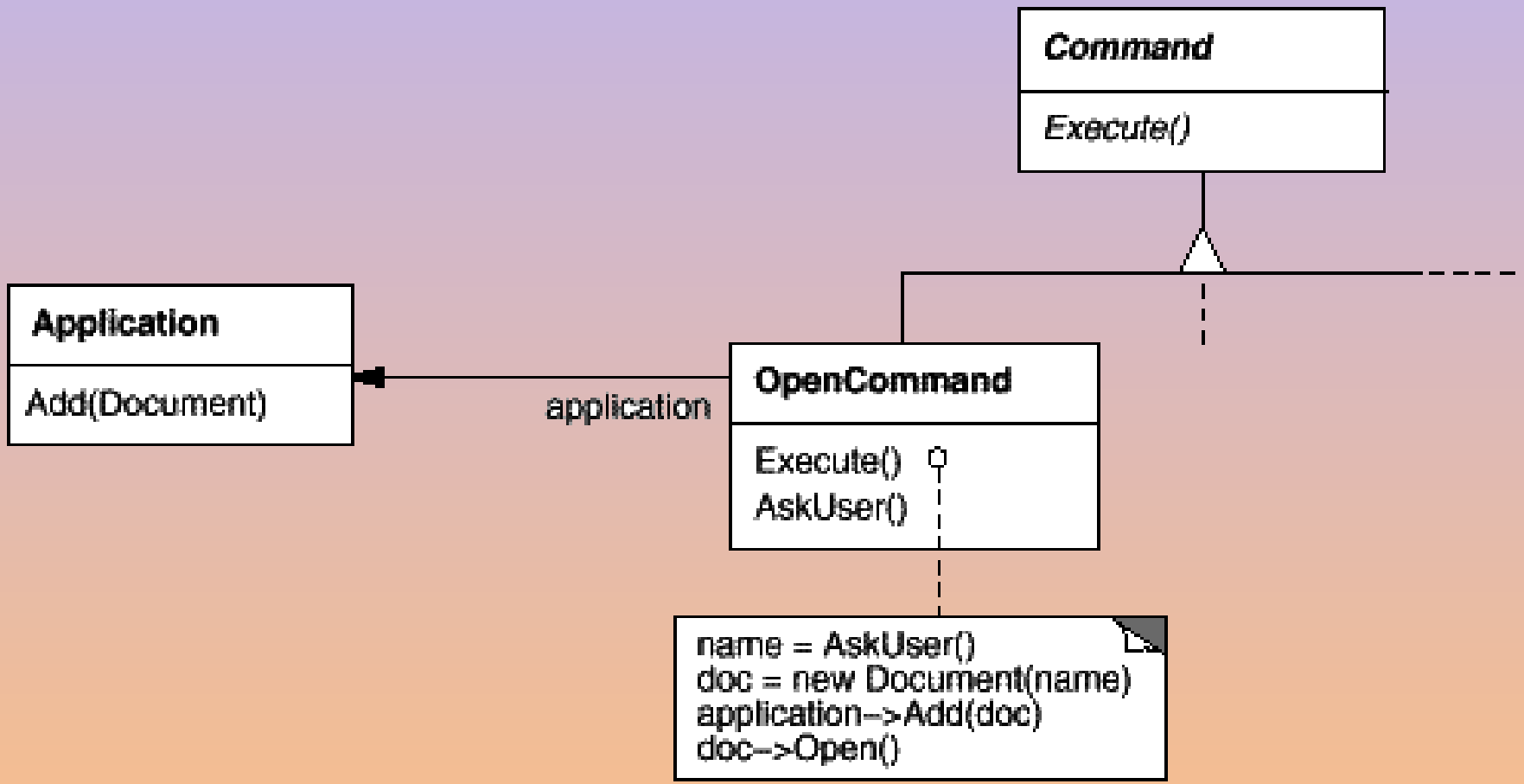
Command (Object Behavioral)

Motivation



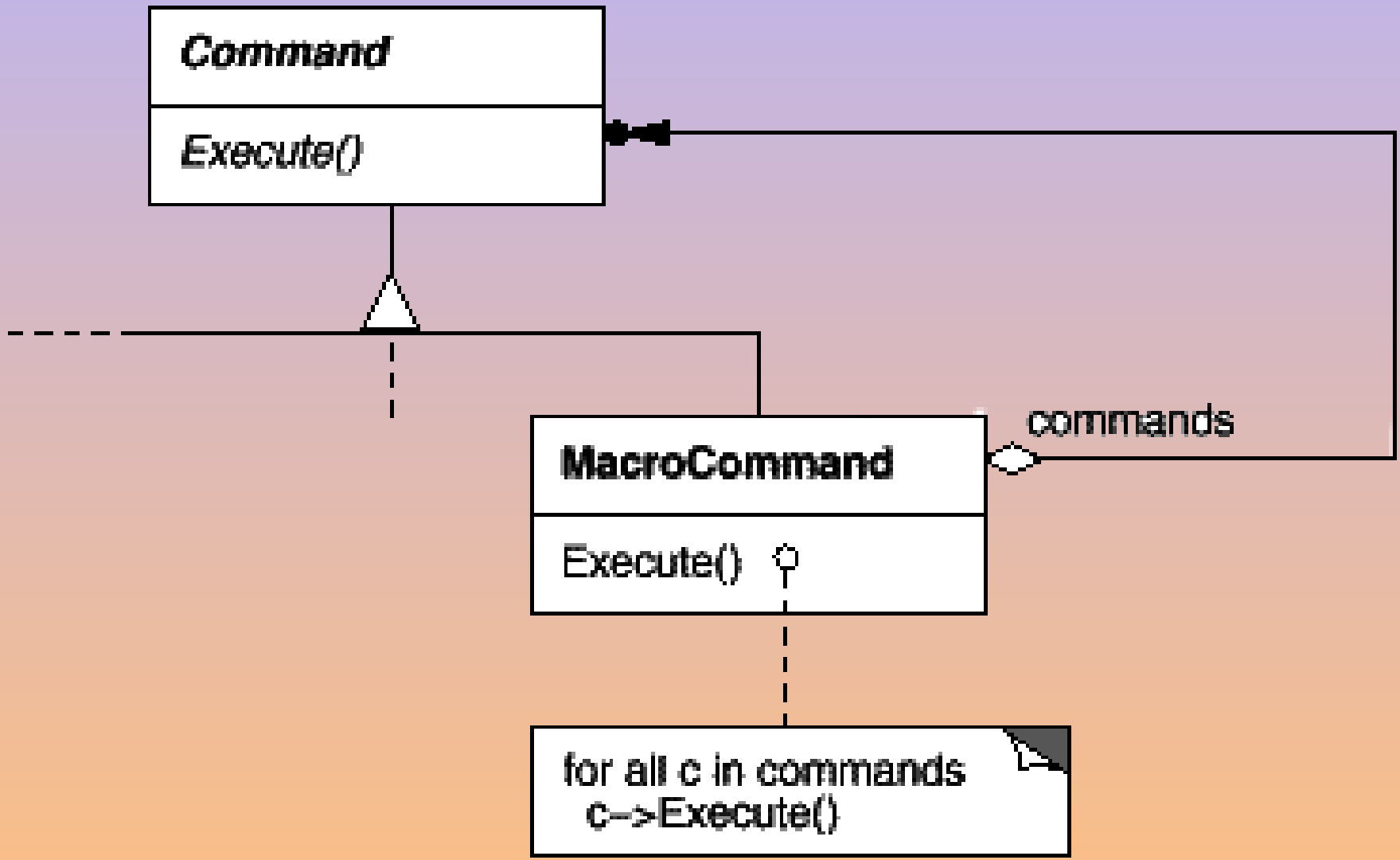
Command (Object Behavioral)

Motivation



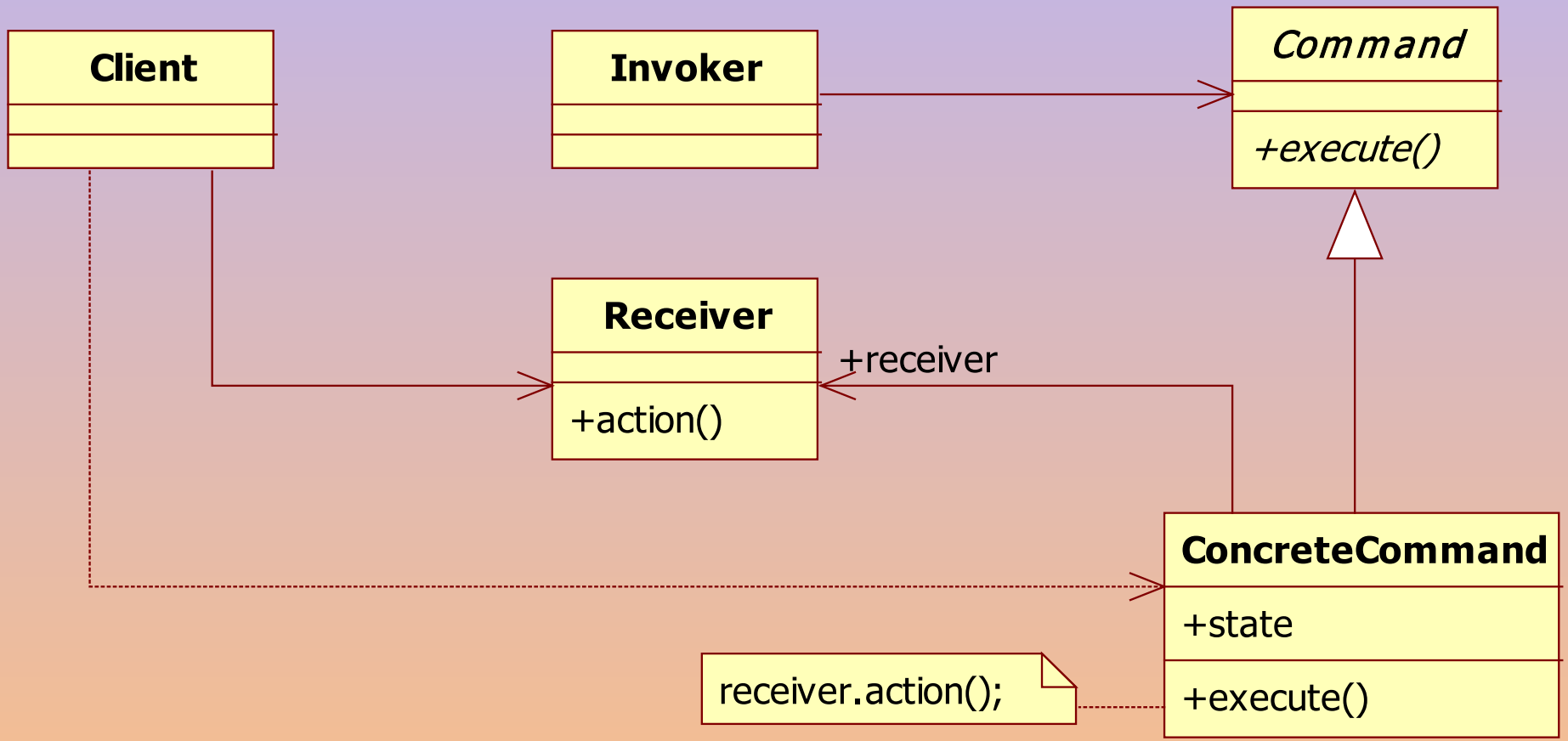
Command (Object Behavioral)

Motivation



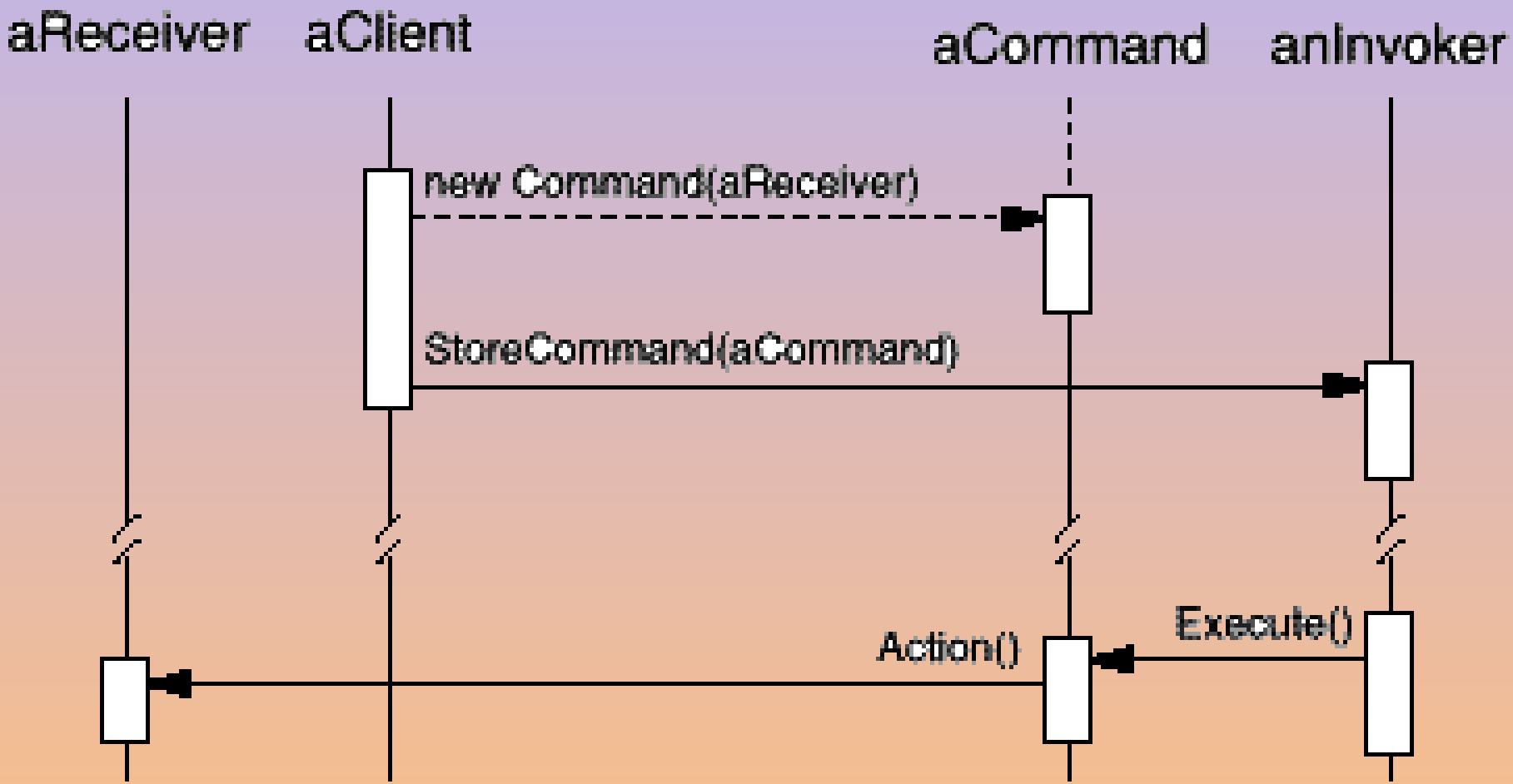
Command (Object Behavioral)

Structure



Command (Object Behavioral)

Collaborations



Command (Object Behavioral)

Consequences

1. decouples invoker from performer
2. commands are first-class objects
3. can assemble commands into a composite command
4. adding new Commands is easy

Implementation

1. what grain?
2. support undo/redo
3. C++ templates

Visitor

Visitor

A Simple Hierarchy

Simple Hierarchy

Described with a Grammar

$$\langle \text{exp} \rangle ::= \langle \text{exp} \rangle \ (\text{'+'} \mid \text{'-'} \mid \text{'*'} \mid \text{'/'}) \ \langle \text{exp} \rangle \mid \langle \text{num} \rangle$$

Expressions

```
class Exp
{
protected:
  virtual ~Exp() = 0;
};
```

```
class Num : public Exp
{
public:
  Num(int val)
    : Exp(), val_(val)
  {}

private:
  int val_;
};
```

```
class Bin : public Exp
{
public:
  Bin(char oper, Exp* lhs, Exp* rhs)
    : Exp(), oper_(oper)
    , lhs_(lhs), rhs_(rhs)
  {}

  virtual ~Bin()
  {
    delete lhs_;
    delete rhs_;
  }

private:
  char oper_; Exp* lhs_; Exp* rhs_;
};
```

Building an Expression

```
int
main()
{
  Exp* tree = new Bin{'+', new Num{42}, new Num{51});
  delete tree;
}
```

Traversals?

Building an Expression

```
int
main()
{
  Exp* tree = new Bin{'+', new Num{42}, new Num{51});
  delete tree;
}
```

Traversals?

Expressions: Exp

```
#include <iostream>

class Exp
{
protected:
  virtual ~Exp() = 0;
};

std::ostream&
operator<<(std::ostream& o, const Exp& tree)
{
  return o << "Uh oh...";
}
```

Binary Expressions: Bin

```
class Bin : public Exp
{
public:
  Bin(char oper, Exp* lhs, Exp* rhs)
    : Exp(), oper_(oper), lhs_(lhs), rhs_(rhs)
  {}
  virtual ~Bin()
  {
    delete lhs_;
    delete rhs_;
  }

  friend std::ostream& operator<<(std::ostream& o, const Bin& tree);
  {
    return o << '(' << *tree.lhs_ << tree.oper_ << *tree.rhs_ << ')';
  }

private:
  char oper_; Exp* lhs_; Exp* rhs_;
};
```

Numbers: Num

```
class Num : public Exp
{
public:
  Num(int val)
    : Exp(), val_(val)
  {}

  friend std::ostream& operator<<(std::ostream& o, const Num& tree)
  {
    return o << tree.val_;
  }

private:
  int val_;
};
```

Invoking and Printing

```
int
main()
{
  Bin* bin = new Bin{'+', new Num{42}, new Num{51});
  Exp* exp = bin;
  std::cout << "Exp: " << *exp << std::endl;
  std::cout << "Bin: " << *bin << std::endl;
  delete bin;
}
```

Using operator<<

```
% ./bin2
Exp: Uh oh...
Bin: (Uh oh...+Uh oh...)
```

- compile time selection (*static binding*)
  based on the containing/variable type
- We need it at run time (*dynamic binding*)
  based on the contained/object type
  - also called *inclusion polymorphism*
  - provided by `virtual` in C++

Using operator<<

```
% ./bin2
Exp: Uh oh...
Bin: (Uh oh...+Uh oh...)
```

- compile time selection (*static binding*)
  based on the containing/variable type
- We need it at run time (*dynamic binding*)
  based on the contained/object type
  - also called *inclusion polymorphism*
  - provided by `virtual` in C++

Using operator<<

```
% ./bin2
Exp: Uh oh...
Bin: (Uh oh...+Uh oh...)
```

- compile time selection (*static binding*)
  based on the containing/variable type
- We need it at run time (*dynamic binding*)
  based on the contained/object type
  - also called *inclusion polymorphism*
  - provided by `virtual` in C++

Expressions: Exp

```
#include <iostream>

class Exp
{
public:
  virtual std::ostream& print(std::ostream& o) const = 0;
};
```

Binary Expressions: Bin

```
class Bin : public Exp
{
public:
  Bin(char op, Exp* l, Exp* r)
    : Exp(), oper_(op), lhs_(l), rhs_(r)
  {}

  virtual ~Bin() {
    delete lhs_; delete rhs_;
  }

  virtual std::ostream& print(std::ostream& o) const {
    o << '('; lhs_->print(o); o << oper_;
    rhs_->print(o); return o << ')';
  }

private:
  char oper_; Exp* lhs_; Exp* rhs_;
};
```

Numbers: Num

```
class Num : public Exp
{
public:
  Num(int val)
    : Exp(), val_(val)
  {}

  virtual std::ostream&
  print(std::ostream& o) const
  {
    return o << val_;
  }

private:
  int val_;
};
```

Using this Tree

```
std::ostream&
operator<<(std::ostream& o, const Exp& e)
{
  return e.print(o);
}

int
main()
{
  Bin* bin = new Bin{'+', new Num{42}, new Num{51});
  Exp* exp = bin;
  std::cout << "Exp: " << *exp << std::endl;
  std::cout << "Bin: " << *bin << std::endl;
  delete bin;
}
```

Discussion

It works...

```
% ./exp3
Exp: (42+51)
Bin: (42+51)
```

but Bin::print is obfuscated.

```
std::ostream&
Bin::print(std::ostream& o) const
{
  o << '(';
  lhs_->print(o);
  o << oper_;
  rhs_->print(o);
  o << ')';
  return o;
}
```

Discussion

It works...

```
% ./exp3
Exp: (42+51)
Bin: (42+51)
```

but `Bin::print` is obfuscated.

```
std::ostream&
Bin::print(std::ostream& o) const
{
  o << '(';
  lhs_->print(o);
  o << oper_;
  rhs_->print(o);
  o << ')';
  return o;
}
```

Making operator<< Polymorphic

Just use the operator<< in print!

```
class Exp
{
public:
  virtual std::ostream& print(std::ostream& o) const = 0;
};

std::ostream& operator<<(std::ostream& o, const Exp& e)
{
  return e.print(o);
}

std::ostream& Bin::print(std::ostream& o) const
{
  return o << '(' << *lhs_ << oper_ << *rhs_ << ')';
}
```

Cuter, but you cannot pass additional arguments to print.

Making operator<< Polymorphic

Just use the operator<< in print!

```
class Exp
{
public:
  virtual std::ostream& print(std::ostream& o) const = 0;
};

std::ostream& operator<<(std::ostream& o, const Exp& e)
{
  return e.print(o);
}

std::ostream& Bin::print(std::ostream& o) const
{
  return o << '(' << *lhs_ << oper_ << *rhs_ << ')';
}
```

Cuter, but you cannot pass additional arguments to print.

Separate processing and dispatching

- In the previous code, `operator<<` processes and dispatches
- Additional operations will require processing and dispatching

Processing

- Keep it external
- Add new easily

Dispatching

- Keep it internal
- Once for all: Factor it!

Separate processing and dispatching

- In the previous code, `operator<<` processes and dispatches
- Additional operations will require processing and dispatching

Processing

- Keep it external
- Add new easily

Dispatching

- Keep it internal
- Once for all: Factor it!

Separate processing and dispatching

- In the previous code, `operator<<` processes and dispatches
- Additional operations will require processing and dispatching

Processing

- Keep it external
- Add new easily

Dispatching

- Keep it internal
- Once for all: Factor it!

Separate processing and dispatching

- In the previous code, `operator<<` processes and dispatches
- Additional operations will require processing and dispatching

Processing

- Keep it external
- Add new easily

Dispatching

- Keep it internal
- Once for all: Factor it!

operator<< to process

```
std::ostream& operator<<(std::ostream& o, const Bin& e)
{
  return o << '(' << *e.lhs_ << oper_ << *e.rhs_ << ')';
}

std::ostream& operator<<(std::ostream& o, const Num& e)
{
  return o << e.val;
}
```

```
std::ostream& operator<<(std::ostream& o, const Exp& e)
{
  return e.print(o);
}
```

operator<< to process

```
std::ostream& operator<<(std::ostream& o, const Bin& e)
{
  return o << '(' << *e.lhs_ << oper_ << *e.rhs_ << ')';
}

std::ostream& operator<<(std::ostream& o, const Num& e)
{
  return o << e.val;
}
```

```
std::ostream& operator<<(std::ostream& o, const Exp& e)
{
  return e.print(o);
}
```

print to dispatch

```
class Exp  {
public:
  virtual std::ostream& print(std::ostream& o) const = 0;
};

class Bin : public Exp  {
public:
  virtual std::ostream& print(std::ostream& o) const
    { return o << *this; }
  ...
};

class Num : public Exp  {
public:
  virtual std::ostream& print(std::ostream& o) const
    { return o << *this; }
  ...
};
```

Separate processing and dispatching

- Now `operator<<` processes
- `print` dispatches
- Each processing requires its dispatching
- Pass pointers to functions to factor the dispatching?

Separate processing and dispatching

- Now `operator<<` processes
- `print` dispatches
- Each processing requires its dispatching
- Pass pointers to functions to factor the dispatching?

Separate processing and dispatching

- Now `operator<<` processes
- `print` dispatches
- Each processing requires its dispatching
- Pass pointers to functions to factor the dispatching?

Visitors

Multimethods

- Polymorphism over any argument, not just on the object:

```
ostream& operator<<(ostream& o, virtual const Exp& e);
ostream& operator<<(ostream& o, virtual const Bin& e);
ostream& operator<<(ostream& o, virtual const Num& e);
```

- This is called multimethods
- CLOS, Common Lisp Object System

Multimethods in C++

- No multimethods in C++ 03 (nor 11)
- Simulate via a callback

```
ostream& operator<<(ostream& o, const Exp& e)
{
  return e.print(o);
}
virtual ostream& Exp::print(ostream& o) = 0;
virtual ostream& Bin::print(ostream& o) { ... };
virtual ostream& Num::print(ostream& o) { ... };
```

- Ask the hierarchy to perform the dispatch
- Additional work on the hierarchy
- The concept is spread in several files
- Requires the ability to edit the hierarchy

Multimethods in C++

- No multimethods in C++ 03 (nor 11)
- Simulate via a callback

```
ostream& operator<<(ostream& o, const Exp& e)
{
  return e.print(o);
}
virtual ostream& Exp::print(ostream& o) = 0;
virtual ostream& Bin::print(ostream& o) { ... };
virtual ostream& Num::print(ostream& o) { ... };
```

- Ask the hierarchy to perform the dispatch
- Additional work on the hierarchy
- The concept is spread in several files
- Requires the ability to edit the hierarchy

Multimethods

- Support for indentation: a new argument is needed
- Similarly if we want to return a value
- Introduce structures carried in the traversals

```
struct stick_t
{
  std::ostream& ostr;
  int res;
  unsigned tab;
};
void traverse(stick_t& s, const Exp& e);
void traverse(stick_t& s, const Bin& e);
void traverse(stick_t& s, const Num& e);
```

- Better yet: make them objects

Multimethods

- Support for indentation: a new argument is needed
- Similarly if we want to return a value
- Introduce structures carried in the traversals

```
struct stick_t
{
  std::ostream& ostr;
  int res;
  unsigned tab;
};
void traverse(stick_t& s, const Exp& e);
void traverse(stick_t& s, const Bin& e);
void traverse(stick_t& s, const Num& e);
```

- Better yet: make them objects

Visitors

Visitors encapsulate the traversal data and algorithm.

```
class PrettyPrinter
{
public:
  void visitBin(const Bin& e)
  {
    ostr_ << '('; ...
  }
  void visitNum(const Num& e)
  {
    ostr_ << e.val_;
  }

private:
  ostream& ostr_;
  unsigned tab_;
};
```

Class Visitor

```
// fwd.hh.
class Exp;
class Bin;
class Num;
```

```
#include <iostream>
#include <fwd.hh>

class Visitor
{
public:
  virtual void visitBin(const Bin& exp) = 0;
  virtual void visitNum(const Num& exp) = 0;
};
```

Classes Exp and Num

```
class Exp {
public:
  virtual void accept(Visitor& v) const = 0;
};
```

```
class Num : public Exp {
public:
  Num(int val)
    : Exp(), val_(val)
  {}

  virtual void accept(Visitor& v) const
    { v.visitNum(*this); }

private:
  int val_;
};
```

Class Bin

```
class Bin : public Exp
{
public:
  Bin(char op, Exp* l, Exp* r)
    : Exp(), oper_(op), lhs_(l), rhs_(r)
  {}

  virtual ~Bin()
    { delete lhs_; delete rhs_; }

  virtual void accept(Visitor& v) const
    { v.visitBin(*this); }

private:
  char oper_; Exp* lhs_; Exp* rhs_;
};
```

Class PrettyPrinter

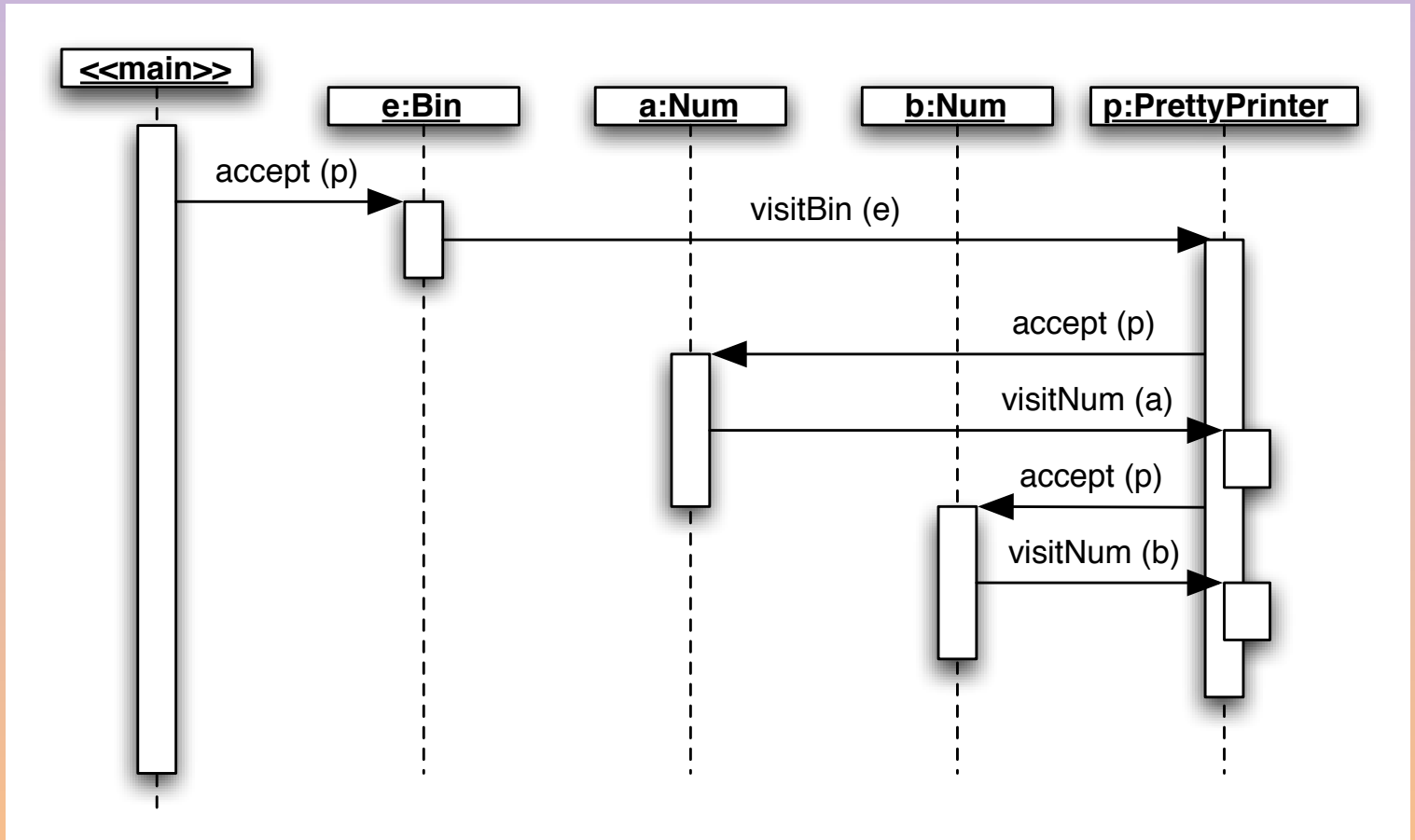
```
class PrettyPrinter : public Visitor
{
public:
  PrettyPrinter(std::ostream& ostr)
    : ostr_(ostr) {}

  virtual void visitBin(const Bin& e) {
    ostr_ << '('; e.lhs_->accept(*this);
    ostr_ << e.oper_; e.rhs_->accept(*this); ostr_ << ')';
  }

  virtual void visitNum(const Num& e) {
    ostr_ << e.val_;
  }

private:
  std::ostream& ostr_;
};
```

operator<< and main

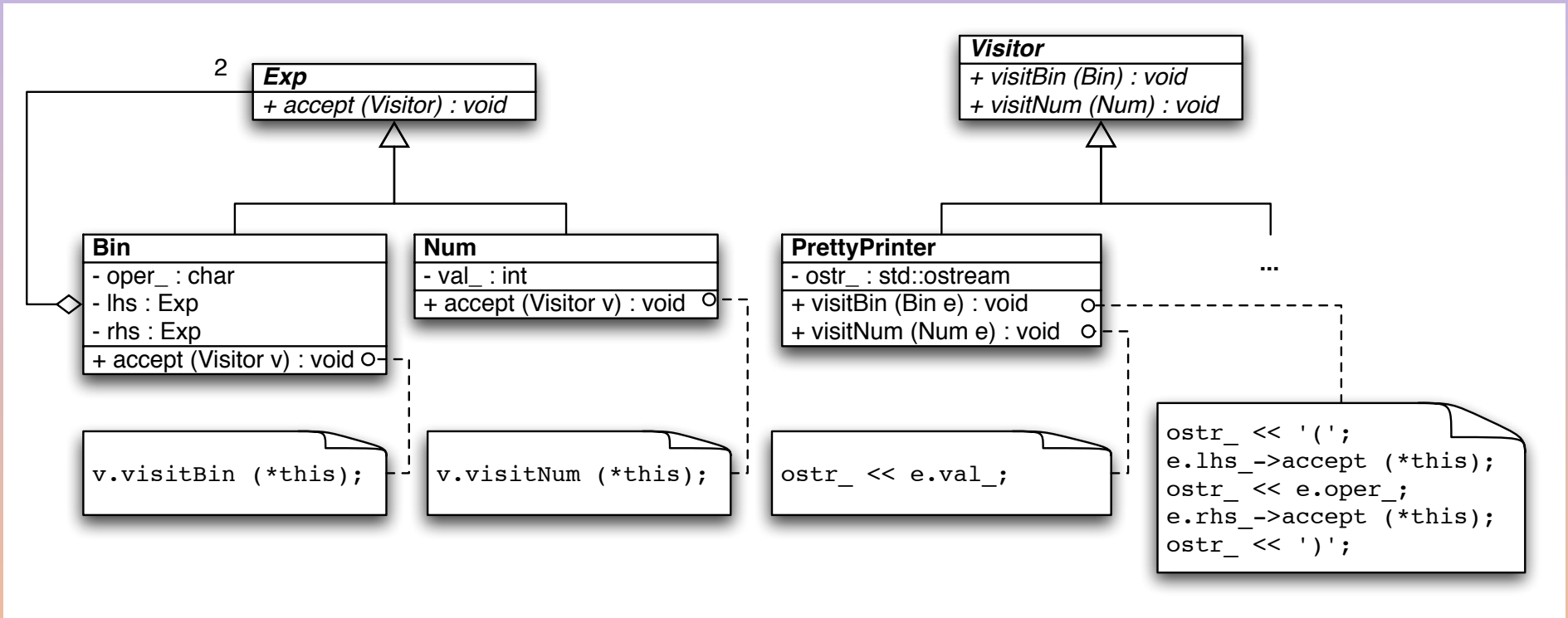
```
std::ostream&
operator<<(std::ostream& o, const Exp& e)
{
  PrettyPrinter printer(o);
  e.accept(printer);
  return o;
}

int
main()
{
  Bin* bin = new Bin{'+', new Num{42}, new Num{51}};
  Exp* exp = bin;
  std::cout << "Bin: " << *bin << std::endl;
  std::cout << "Exp: " << *exp << std::endl;
  delete bin;
}
```

A pretty-printing sequence diagram

```
Exp* a = new Num{42}; Exp* b = new Num{51};
Exp* e = new Bin{'+', a, b}; std::cout << *e << std::endl;
```



A class diagram: Visitor and Composite design patterns



Visitor Design Pattern

Visitor (Object Behavioral)

Intent represent an operation to be performed on the elements of an object structure.
Define new operations extrusively.

AKA

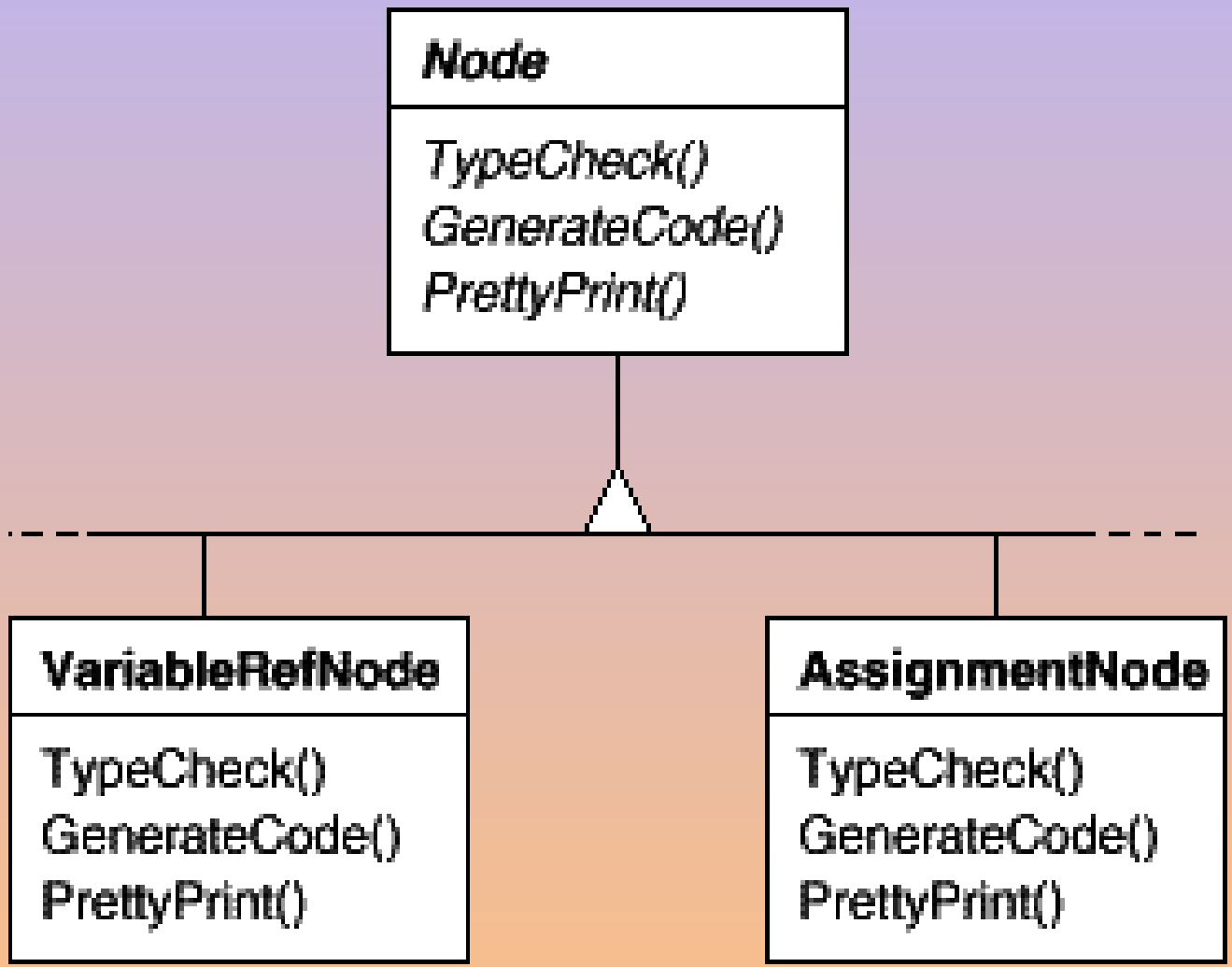
Motivation a compiler that represents programs as Abstract Syntax Trees (ASTs)

Applicability

- operations on many classes with differing interfaces
- external definition of polymorphic operations
- the visited hierarchy rarely changes

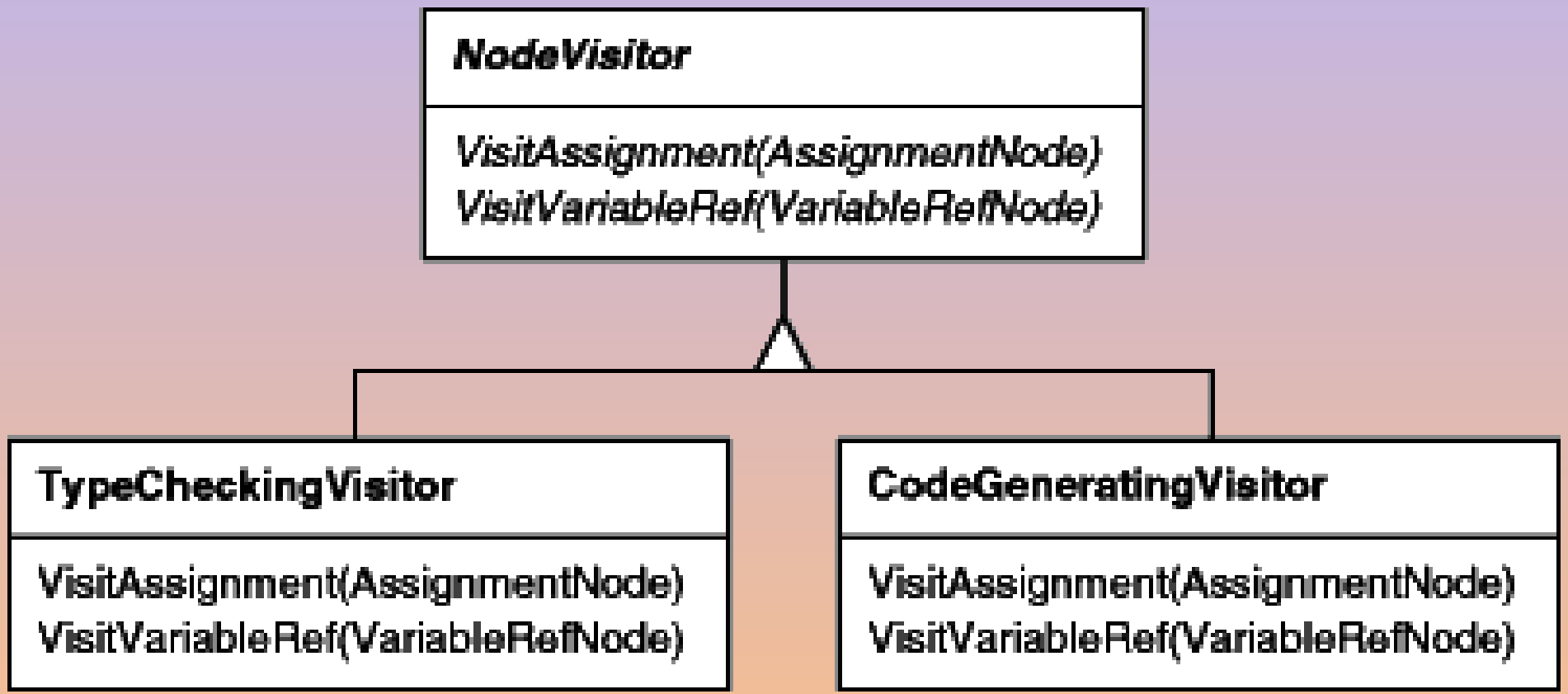
Visitor (Object Behavioral)

Motivation



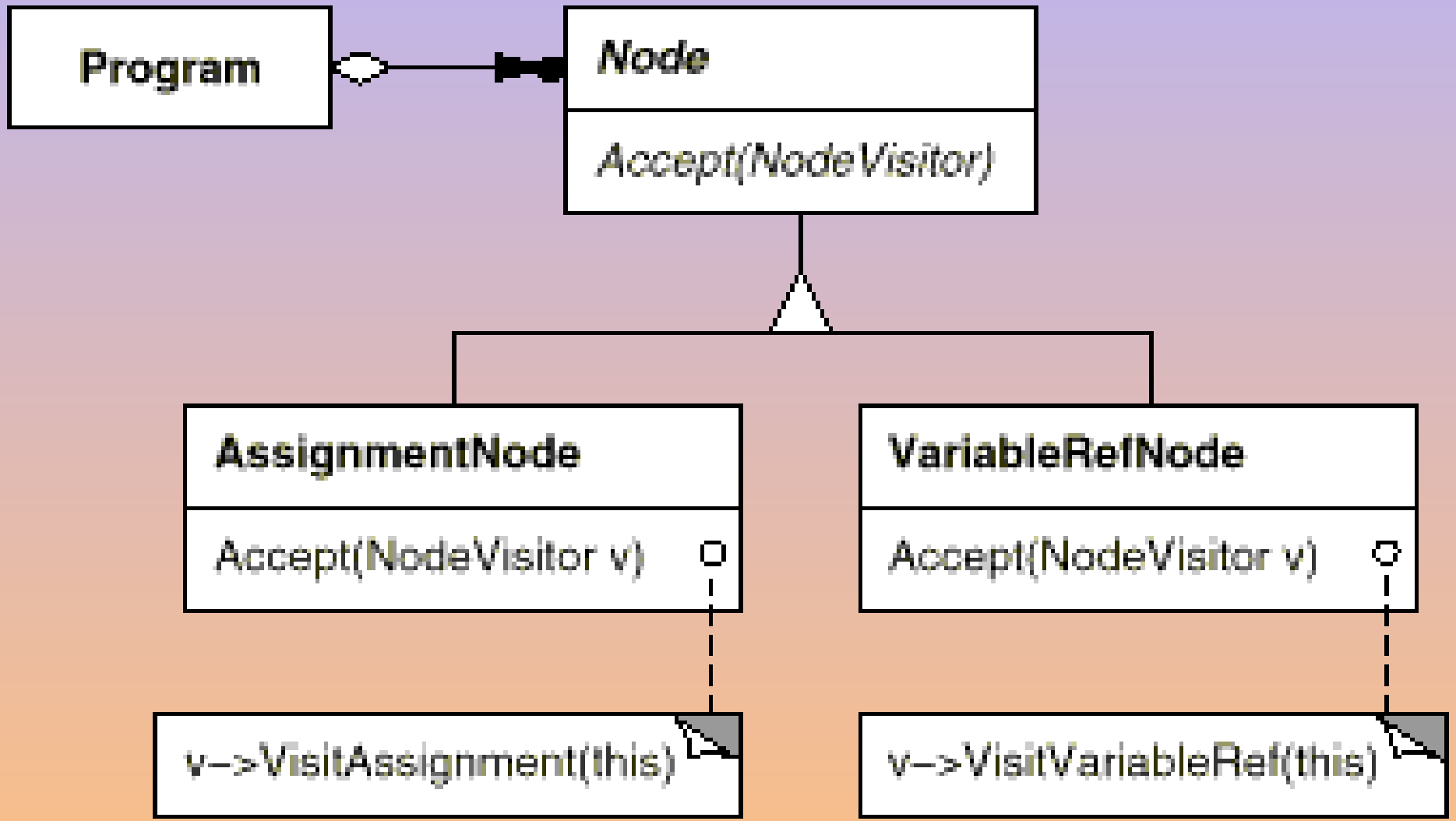
Visitor (Object Behavioral)

Motivation



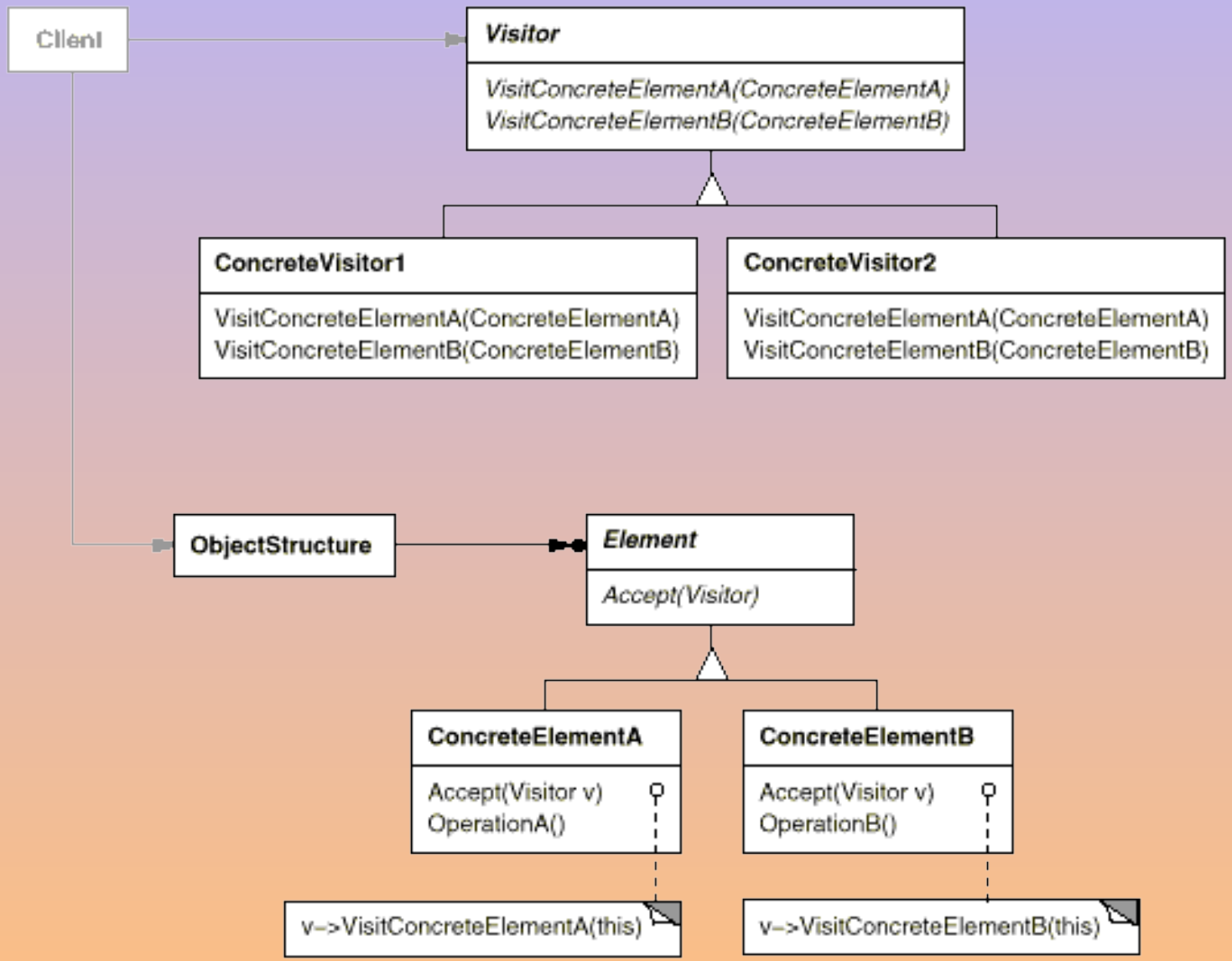
Visitor (Object Behavioral)

Motivation



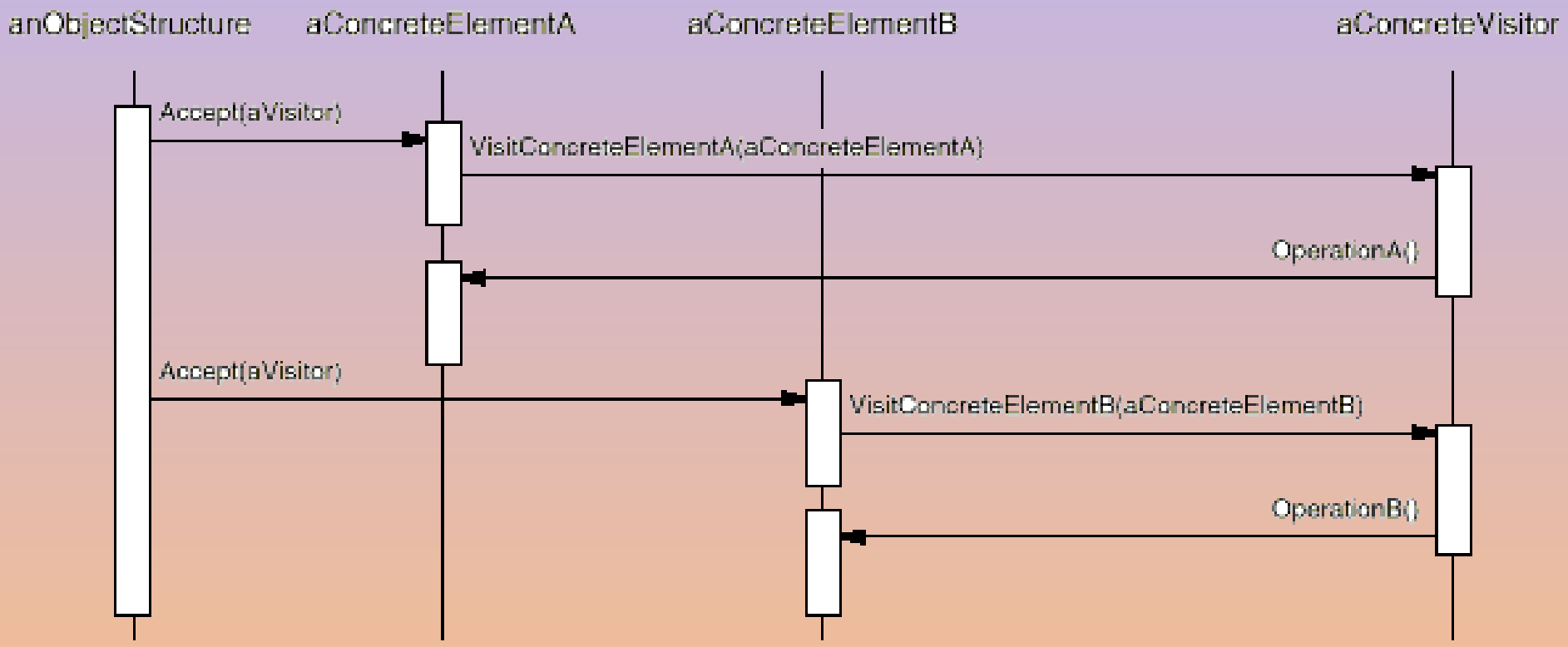
Visitor (Object Behavioral)

Structure



Visitor (Object Behavioral)

Collaborations



Command (Object Behavioral)

Consequences

1. adding new operations is easy
2. gathers related operations and separates unrelated ones
3. adding new ConcreteElement classes is hard
4. visiting *across* class hierarchies (e.g., on variants)
5. accumulating state
6. breaks encapsulation

Implementation

1. who is responsible for traversing the object structure?
2. constness
3. hierarchy of visitors

Command (Object Behavioral)

Consequences

1. adding new operations is easy
2. gathers related operations and separates unrelated ones
3. adding new ConcreteElement classes is hard
4. visiting *across* class hierarchies (e.g., on variants)
5. accumulating state
6. breaks encapsulation

Implementation

1. who is responsible for traversing the object structure?
2. constness
3. hierarchy of visitors

Further with Visitors

Visitors in C++

- `Visitor` and `ConstVisitor`
  similar to `iterator` and `const_iterator`
- Use C++ templates to factor
  (e.g., `Visitor` and `ConstVisitor`)
- Use C++ overloading
  only `visit` instead of `visitBin` and `visitNum`

Visitors in C++

- `Visitor` and `ConstVisitor`
  similar to `iterator` and `const_iterator`
- Use C++ templates to factor
  (e.g., `Visitor` and `ConstVisitor`)
- Use C++ overloading
  only `visit` instead of `visitBin` and `visitNum`

Visitors in C++

- `Visitor` and `ConstVisitor`
  similar to `iterator` and `const_iterator`
- Use C++ templates to factor
  (e.g., `Visitor` and `ConstVisitor`)
- Use C++ overloading
  only `visit` instead of `visitBin` and `visitNum`

Object Functions

- How about `operator()` instead of `visit`?
- But then, we can improve this

provided

- Derive from `std::unary_function<Exp, void>` to ease "functional programming" (`#include <functional>`)

Object Functions

- How about `operator()` instead of `visit`?
- But then, we can improve this

```
int eval(const Exp& e) {
  Evaluator eval;
  e.accept(eval);
  return eval.value;
}
```

```
int eval(const Exp& e) {
  Evaluator eval;
  eval(e);
  return eval.value;
}
```

provided

- Derive from `std::unary_function<Exp, void>` to ease "functional programming" (`#include <functional>`)

Object Functions

- How about `operator()` instead of `visit`?
- But then, we can improve this

```
int eval(const Exp& e) {
  Evaluator eval;
  e.accept(eval);
  return eval.value;
}
```

```
int eval(const Exp& e) {
  Evaluator eval;
  eval(e);
  return eval.value;
}
```

provided

- Derive from `std::unary_function<Exp, void>` to ease "functional programming" (`#include <functional>`)

Object Functions

- How about `operator()` instead of `visit`?
- But then, we can improve this

```
int eval(const Exp& e) {
  Evaluator eval;
  e.accept(eval);
  return eval.value;
}
```

```
int eval(const Exp& e) {
  Evaluator eval;
  eval(e);
  return eval.value;
}
```

provided

```
void Evaluator::operator()(const Exp& e) {
  e.accept(*this);
}
```

- Derive from `std::unary_function<Exp, void>` to ease "functional programming" (`#include <functional>`)

Object Functions

- How about `operator()` instead of `visit`?
- But then, we can improve this

```
int eval(const Exp& e) {
  Evaluator eval;
  e.accept(eval);
  return eval.value;
}
```

```
int eval(const Exp& e) {
  Evaluator eval;
  eval(e);
  return eval.value;
}
```

provided

```
void Evaluator::operator()(const Exp& e) {
  e.accept(*this);
}
```

- Derive from `std::unary_function<Exp, void>` to ease "functional programming" (`#include <functional>`)

Sugaring Visitors 1

```
struct Evaluator : public ConstVisitor {
  virtual void operator()(const Exp& e) {
    e.accept(*this);
  }
  virtual void operator()(const Bin& e) {
    e.lhs_->accept(*this); int lhs = value;
    e.rhs_->accept(*this); int rhs = value;
    ... value = lhs + rhs; ...
  }
  virtual void operator()(const Num& e) {
    value = e.val;
  }
  int value;
};

int eval(const Exp& e) {
  Evaluator eval;
  eval(e);
  return eval.value;
}
```

Sugaring Visitors 2

```
struct Evaluator : public ConstVisitor
{
  ...
  virtual void
  operator()(const Bin& e) {
    ...
    value = eval(e.lhs_)
          + eval(e.rhs_);
    ...
  }

  virtual void
  operator()(const Num& e) {
    value = e.val;
  }

  int value;
};
```

One visitor per `eval` invocation

- A useless cost
- Easy automatic variables
- Harder for shared data (no `static` please!)

Sugaring Visitors 2

```
struct Evaluator : public ConstVisitor
{
  ...
  virtual void
  operator()(const Bin& e) {
    ...
    value = eval(e.lhs_)
          + eval(e.rhs_);
    ...
  }

  virtual void
  operator()(const Num& e) {
    value = e.val;
  }

  int value;
};
```

One visitor per `eval` invocation

- A useless cost
- Easy automatic variables
- Harder for shared data (no `static` please!)

Sugaring Visitors 3

```
struct Evaluator : public ConstVisitor
{
  virtual int eval(const Exp& e) {
    e.accept(*this); return value;
  }

  virtual void operator()(const Exp& e) {
    e.accept(*this);
  }
  virtual void operator()(const Bin& e) {
    ...
    value = eval(e.lhs_) + eval(e.rhs_);
    ...
  }
  virtual void operator()(const Num& e) {
    value = e.val;
  }

  int value;
};
```

Sugaring the PrettyPrinter

```
virtual void
PrettyPrinter::operator()(const Bin& e)
{
  ostr_ << '(';
  e.lhs_->accept(*this);
  ostr_ << e.oper_;
  e.rhs_->accept(*this);
  ostr_ << ')';
}
```

- We could insert a `print` method
- But that's not nice
- We can use the `operator<<`
- But we no longer can pass additional arguments
- Unless...

Sugaring the PrettyPrinter

```
virtual void
PrettyPrinter::operator()(const Bin& e)
{
  ostr_ << '(';
  e.lhs_->accept(*this);
  ostr_ << e.oper_;
  e.rhs_->accept(*this);
  ostr_ << ')';
}
```

- We could insert a print method
- But that's not nice
- We can use the operator<<
- But we no longer can pass additional arguments
- Unless...

Sugaring the PrettyPrinter

```
virtual void
PrettyPrinter::operator()(const Bin& e)
{
  ostr_ << '(';
  e.lhs_->accept(*this);
  ostr_ << e.oper_;
  e.rhs_->accept(*this);
  ostr_ << ')';
}
```

- We could insert a `print` method
- But that's not nice
- We can use the `operator<<`
- But we no longer can pass additional arguments
- Unless...

Sugaring the PrettyPrinter

```
virtual void
PrettyPrinter::operator()(const Bin& e)
{
  ostr_ << '(';
  e.lhs_->accept(*this);
  ostr_ << e.oper_;
  e.rhs_->accept(*this);
  ostr_ << ')';
}
```

- We could insert a `print` method
- But that's not nice
- We can use the `operator<<`
- But we no longer can pass additional arguments
- Unless...

Sugaring the PrettyPrinter

```
virtual void
PrettyPrinter::operator()(const Bin& e)
{
  ostr_ << '(';
  e.lhs_->accept(*this);
  ostr_ << e.oper_;
  e.rhs_->accept(*this);
  ostr_ << ')';
}
```

- We could insert a `print` method
- But that's not nice
- We can use the `operator<<`
- But we no longer can pass additional arguments
- Unless...

Sugaring the PrettyPrinter

```
virtual void
PrettyPrinter::operator()(const Bin& e)
{
  ostr_ << '(';
  e.lhs_->accept(*this);
  ostr_ << e.oper_;
  e.rhs_->accept(*this);
  ostr_ << ')';
}
```

- We could insert a `print` method
- But that's not nice
- We can use the `operator<<`
- But we no longer can pass additional arguments
- Unless...

Sugaring the PrettyPrinter

```
virtual void
PrettyPrinter::operator()(const Bin& e)
{
  ostr_ << '(';
  e.lhs_->accept(*this);
  ostr_ << e.oper_;
  e.rhs_->accept(*this);
  ostr_ << ')';
}
```

- We could insert a `print` method
- But that's not nice
- We can use the `operator<<`
- But we no longer can pass additional arguments
- Unless... we can put data in the stream

Visitors Hierarchies

- Implement default behaviors (`DefaultVisitor`, `DefaultConstVisitor`)
- Use C++ templates to factor
- Overloaded virtual method must be imported

```
class Renamer : public DefaultVisitor
{
public:
  typedef DefaultVisitor super_type;
  using super_type::operator();
  //...
}
```

Visitors Hierarchies

- Implement default behaviors (`DefaultVisitor`, `DefaultConstVisitor`)
- Use C++ templates to factor
- Overloaded virtual method must be imported

```
class Renamer : public DefaultVisitor
{
public:
  typedef DefaultVisitor super_type;
  using super_type::operator();
  //...
}
```

Visitors Hierarchies

- Implement default behaviors (`DefaultVisitor`, `DefaultConstVisitor`)
- Use C++ templates to factor
- Overloaded virtual method must be imported

```
class Renamer : public DefaultVisitor
{
public:
  typedef DefaultVisitor super_type;
  using super_type::operator();
  //...
}
```

Visitors Hierarchies

Specialize behaviors

`Flower <: Analyzer <: Cloner <: DefaultConstVisitor <: ConstVisitor`

```
void Flower::visit(const ast::Try* code)
{
  Finally finally(scoped_set(has_general_catch_, false));
  super_type::visit(code);
}

void Flower::visit(const ast::Catch* code)
{
  if (has_general_catch_)
    err(code->location_get(),
        "catch: exception already caught by a previous clause");
  has_general_catch_ = !code->match_get();

  Finally finally(scoped_set(in_catch_, true));
  super_type::visit(code);
}
```

Visitor Combinators

- Work and traversal are still too heavily interrelated

→ Create visitors from basic traversal bricks: *combinators* [Visser, 2001].

Combinator	Description
Identity	Do nothing.
Sequence(v_1, v_2)	Sequentially run visitor v_1 then v_2.
Fail	Raise an exception.
Choice(v_1, v_2)	Try visitor v_1; if v_1 fails, try v_2.
All(v)	Apply visitor v sequentially to every immediate subtree.
One(v)	Apply visitor v sequentially to the immediate subtrees until it succeeds.

Visitor Combinators (cont.)

- Combine them to create visiting strategies.

$$
\begin{aligned}
\mathrm{Twice}(v) &=_{def} \mathrm{Sequence}(v, v) \\
\mathrm{Try}(v) &=_{def} \mathrm{Choice}(v, \mathit{Identity}) \\
\mathrm{TopDown}(v) &=_{def} \mathrm{Sequence}(v, \mathrm{All}(\mathrm{TopDown}(v))) \\
\mathrm{BottomUp}(v) &=_{def} \mathrm{Sequence}(\mathrm{All}(\mathrm{BottomUp}(v)), v)
\end{aligned}
$$

Visitors Don't Require Composite

```
#include <boost/variant.hpp>
#include <iostream>

struct my_visitor : boost::static_visitor<int>
{
  int operator()(int i)                const { return i; }
  int operator()(const std::string& s) const { return s.length(); }
};

int main()
{
  boost::variant<int, std::string> u("hello world");
  std::cout << u; // output: hello world

  int result = boost::apply_visitor(my_visitor(), u);
  std::cout << result; // output: 11
}
```

Design Patterns

- Extremely powerful means to communication
- Therefore, fine mental tool
- A nice way to glorify weaknesses of programming languages [Norvig, 1998, Ford, 2009]
- Or POO in itself
- Adjustments needed for C++ 11

Design Patterns

- Extremely powerful means to communication
- Therefore, fine mental tool
- A nice way to glorify weaknesses of programming languages [Norvig, 1998, Ford, 2009]
- Or POO in itself
- Adjustments needed for C++ 11

Design Patterns

- Extremely powerful means to communication
- Therefore, fine mental tool
- A nice way to glorify weaknesses of programming languages [Norvig, 1998, Ford, 2009]
- Or POO in itself
- Adjustments needed for C++ 11

Design Patterns

- Extremely powerful means to communication
- Therefore, fine mental tool
- A nice way to glorify weaknesses of programming languages [Norvig, 1998, Ford, 2009]
- Or POO in itself
- Adjustments needed for C++ 11

Design Patterns

- Extremely powerful means to communication
- Therefore, fine mental tool
- A nice way to glorify weaknesses of programming languages [Norvig, 1998, Ford, 2009]
- Or POO in itself
- Adjustments needed for C++ 11

Bibliography I

Alexander, C., Ishikawa, S., and Silverstein, M. (1977).
A pattern language: towns, buildings, construction.
New York: Oxford University Press.
Companion volume to The timeless way of building and The Oregon experiment.

Alexandrescu, A. (2001).
Modern C++ Design: Generic Programming and Design Patterns Applied.
Addison-Wesley.

Bibliography II

Bacon, D. F., Bloch, J., Bogda, J., an Paul Haahr, C. C., Lea, D., May, T., Maessen, J.-W., Manson, J., Mitchell, J. D., Nilsen, K., Pugh, B., and Sirer, E. G.
The "Double-Checked Locking is Broken" Declaration.
http://www.cs.umd.edu/~{}pugh/java/memoryModel/DoubleCheckedLocking.html.

Bouhours, C. (2012).
GOPROD — GOod PRactices in Object oriented Designs.
http://www.goprod.bouhours.net.

Bibliography III

Ford, N. (2009).
"design patterns" in dynamic languages.
In *Proceedings of OSCON 2009, Open Source Convention*, San Jose, California.
`http://www.oscon.com/oscon2009/public/schedule/detail/7754`.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995).
Design Patterns: Elements of Reusable Object-Oriented Software.
Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY.

Bibliography IV


Intel (2011).
Intel Threading Building Blocks Documentation.
Intel.
http://software.intel.com/sites/products/documentation/doclib/tbb_sa/help/index.htm.


Meyers, S. (1996).
More Effective C++.
Addison-Wesley Professional.


Meyers, S. and Alexandrescu, A. (2004).
C++ and the Perils of Double-Checked Locking.
Dr. Dobb's Journal.
http://www.aristeia.com/Papers/DDJ_Jul_Aug_2004_revised.pdf.

Bibliography V

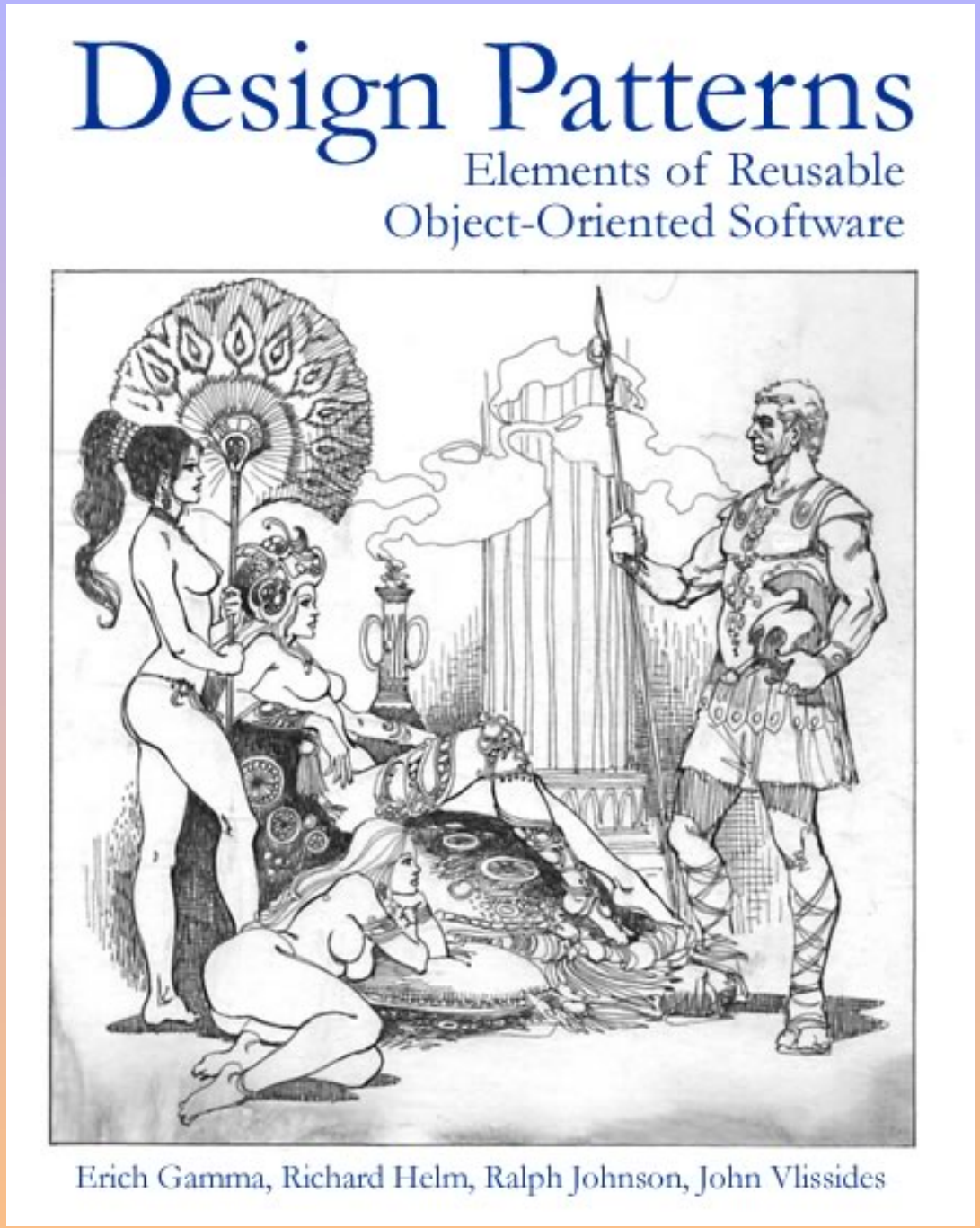

Norvig, P. (1998).
Design patterns in dynamic languages.
http://norvig.com/design-patterns/.


Schmidt, D. (1996).
Reality check.
C++ Report, 8(3).
http://www.cs.wustl.edu/~schmidt/editorial-3.html.


Visser, J. (2001).
Visitor combination and traversal control.
ACM SIGPLAN Notices, 36(11):270–282.
OOPSLA 2001 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications.

Bibliography VI


Yegge, S. (2004).
Singleton considered stupid.
https://sites.google.com/site/steveyegge2/singleton-considered-stupid.



http://www.globalnerdy.com/2007/09/14/reimagining

Questions?

1. Design Patterns
   - Definition
   - The Original Design Patterns
2. Creational Patterns
   - Singleton
   - Factory Method
   - Abstract Factory
3. Behavioral Patterns
   - Command
   - Visitor