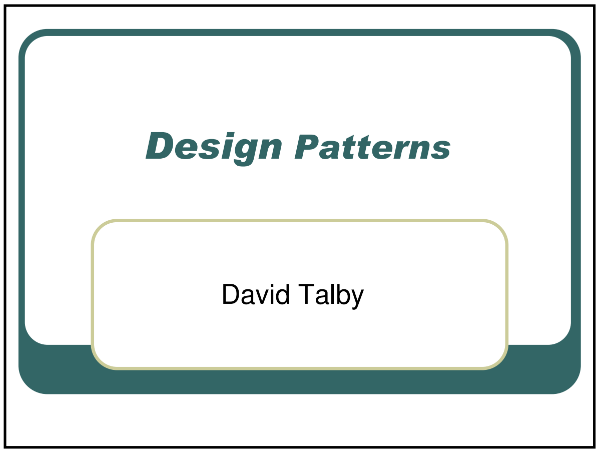
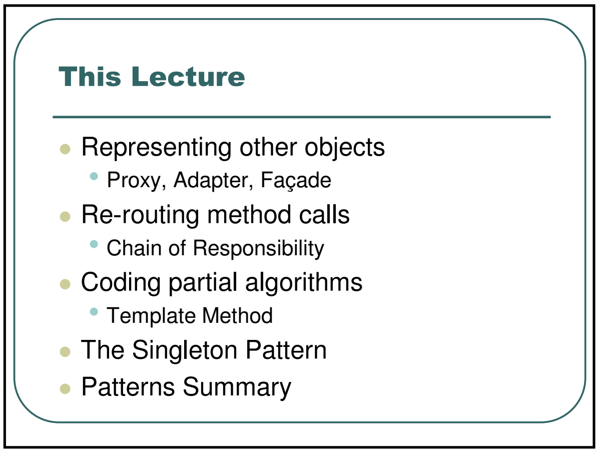


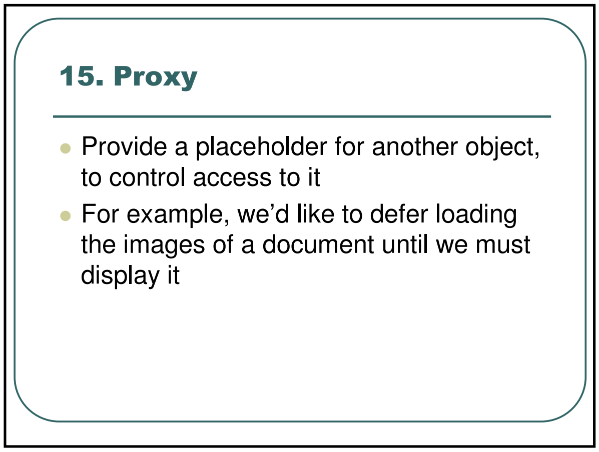
Design Patterns

David Talby



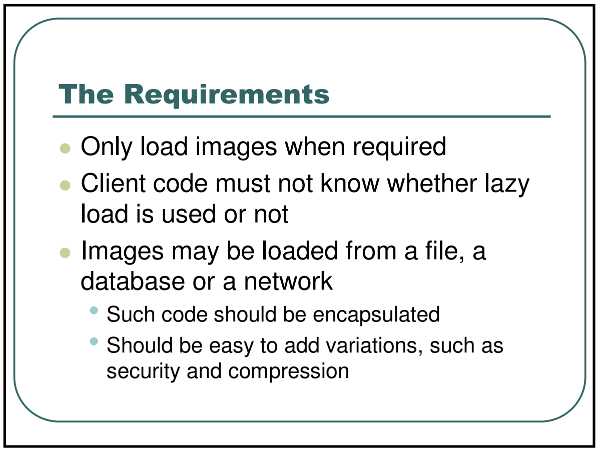
This Lecture

- Representing other objects
  - Proxy, Adapter, Façade
- Re-routing method calls
  - Chain of Responsibility
- Coding partial algorithms
  - Template Method
- The Singleton Pattern
- Patterns Summary



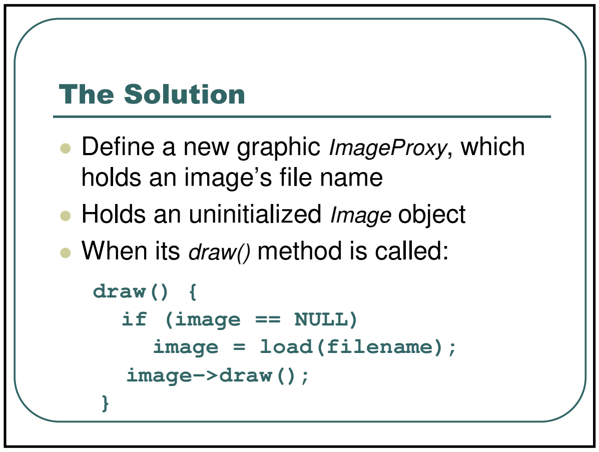
15. Proxy

- Provide a placeholder for another object, to control access to it
- For example, we'd like to defer loading the images of a document until we must display it



The Requirements

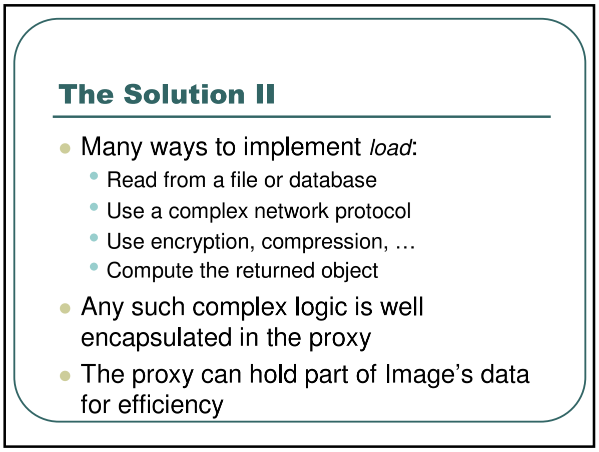
- Only load images when required
- Client code must not know whether lazy load is used or not
- Images may be loaded from a file, a database or a network
  - Such code should be encapsulated
  - Should be easy to add variations, such as security and compression



The Solution

- Define a new graphic *ImageProxy*, which holds an image's file name
- Holds an uninitialized *Image* object
- When its *draw()* method is called:

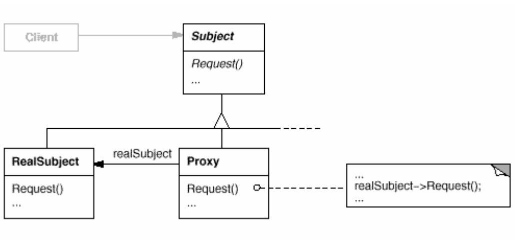
```
draw() {
   if (image == NULL)
     image = load(filename);
   image->draw();
 }
```



The Solution II

- Many ways to implement *load*:
  - Read from a file or database
  - Use a complex network protocol
  - Use encryption, compression, …
  - Compute the returned object
- Any such complex logic is well encapsulated in the proxy
- The proxy can hold part of Image's data for efficiency

The UML



The Fine Print

- The Proxy vocabulary
  - *Virtual Proxy* – creates expensive objects on demand
  - *Remote Proxy* – a local representative of an object in another address space
  - *Protection Proxy* – controls access to the original object
  - *Smart Pointers* – overload regular pointers with additional actions

The Fine Print II

- Uses of smart pointers
  - Reference counting
  - Synchronization (lock management)
  - Profiling and statistics
  - Copy-on-write
  - Cache coherence
  - Pooling
- Smart pointers are easy in C++ thanks to overloading = and –>

The Fine Print III

- Proxy is very much like Decorator
- Decorator = functional addition
- Proxy = technical addition

Known Uses

- Every programming language
- Every middleware package
- Every database package

17. Adapter

- Convert the interface of a class into another that clients expect
- For example, We'd like to use advanced *Text* and *SpellCheck* component that we bought
- But *Text* doesn't inherit *Graphic* or supply iterators, and *SpellCheck* doesn't inherit *Visitor*
- We don't have their source code

The Requirements

- Convert the interface of a class into a more convenient one
- Without the class's source code
  - No compilation dependencies
- The class may be a module in a non-object oriented language

The Solution

- If you can't reuse by inheritance, reuse by composition:

```
class TextGraphic
   : public Graphic
{
 public:
    void draw() { text->paint(); }
    // other methods adapted...
 private:
    BoughtTextComponent *text;
}
```

The Requirements II

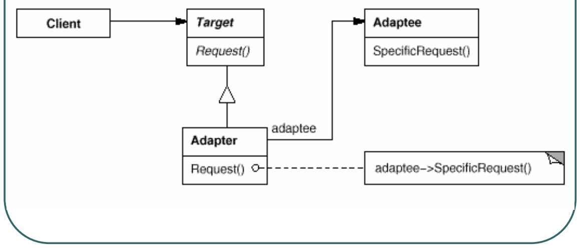
- Stacks and queues are kinds of lists, but they provide less functionality
- *LinkedQueue* is a linked list implementation of interface *Queue*
- We'd like to reuse *LinkedList* for it
- Inheritance can't be used if children offer less than their parents

The Solution II

- Object Adapter
  - Class LinkedQueue will hold a reference to a LinkedList and delegate requests to it
- Class Adapter
  - Class LinkedQueue will inherit from both Queue and LinkedList
  - Method signatures in both classes must match
- In C++ class adapters are safer thanks to private inheritance

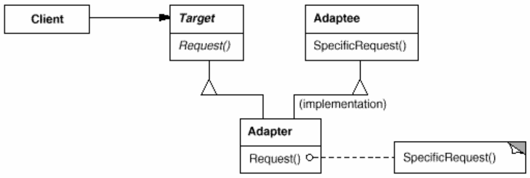
The UML

- Object Adapter:



The UML II

- Class Adapter:



Known Uses

- Using external libraries
- Reusing non O-O code
- Limiting access to classes

18. Facade

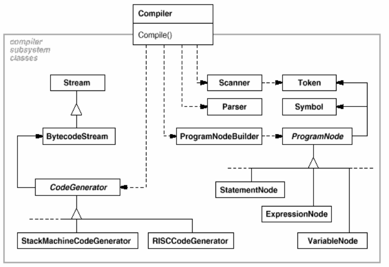
- Provide a unified interface to a set of interfaces of subsystems
- For example, a compiler is divided into many parts
  - Scanner, parser, syntax tree data structure, optimizers, generation, …
- Most clients just compile files, and don't need to access inner parts

The Requirements

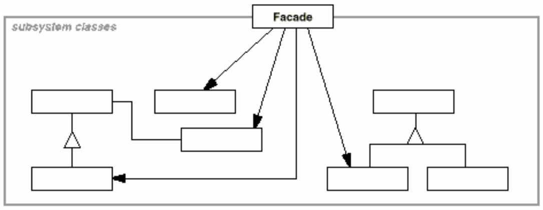
- Provide a simple, easy to use and remember interface for compilation
- Keep the flexibility to tweak inner parts when needed

The Solution

- Define a façade Compiler class as the entry point to the system



The UML



The Fine Print

- Advantages of a façade:
  - Most users will use a very simple interface for the complex system
  - Clients are decoupled from the system
  - Makes it easier to replace the entire system with another
- Packages (Java) and namespaces (C++) are ways to define "systems" of classes and decide which classes are visible to the system's clients

Known Uses

- A Compiler or XML Parser
- Browsing objects at runtime
- The Choices O-O operating system
  - The File and Memory systems

16. Chain of Responsibility

- Decouple the sender and receiver of a message, and give more than one receiver a chance to handle it
- For example, a context-sensitive help system returns help on the object currently in focus
- Or its parent if it has no help
- Recursively

The Requirements

- Allow calling for context-sensitive help from any graphical object
- If the object can't handle the request (it doesn't include help),
  it knows where to forward it
- The set of possible handlers is defined and changed dynamically

The Solution

- Define a *HelpHandler* base class:

```
class HelpHandler
{
   handleHelp() {
     if (successor != NULL)
       successor->handleHelp();
   }
   HelpHandler* successor = NULL;
}
```

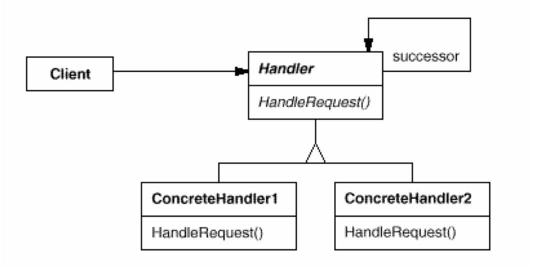
The Solution II

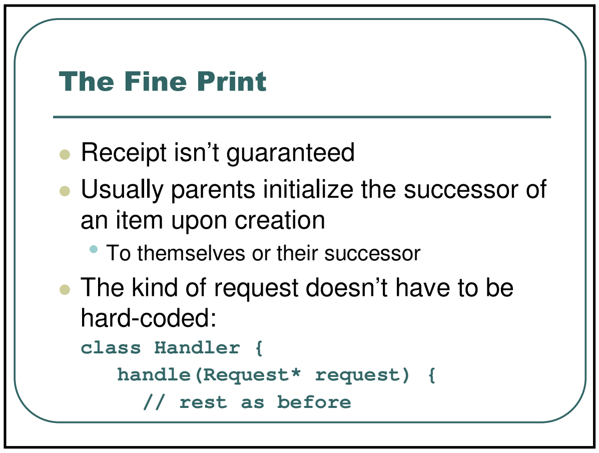
- Class *Graphic* inherits *HelpHandler*
- Graphic descendants that have help to show redefine *handleHelp*:

```
handleHelp() {
   ShowMessage("Buy upgrade");
}
```

- Either the root *Graphic* object or *HelpHandler* itself can redefine *handleHelp* to show a default

The UML

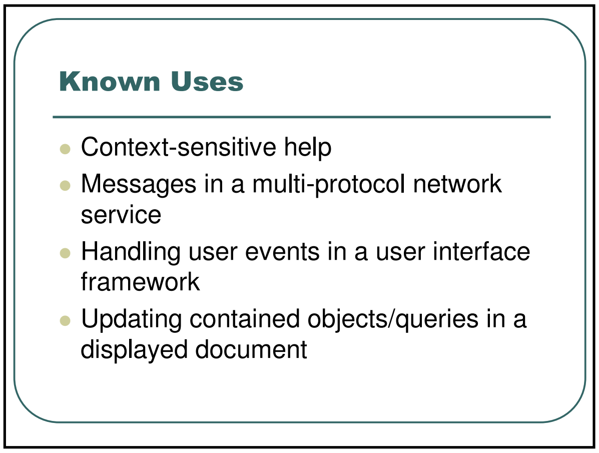




The Fine Print

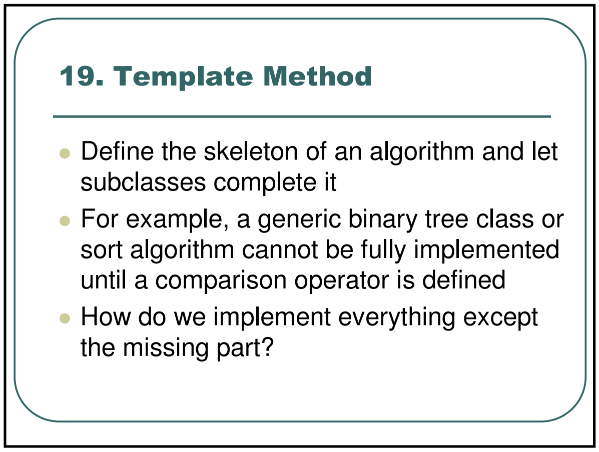
- Receipt isn't guaranteed
- Usually parents initialize the successor of an item upon creation
  - To themselves or their successor
- The kind of request doesn't have to be hard-coded:

```
class Handler {
   handle(Request* request) {
     // rest as before
```



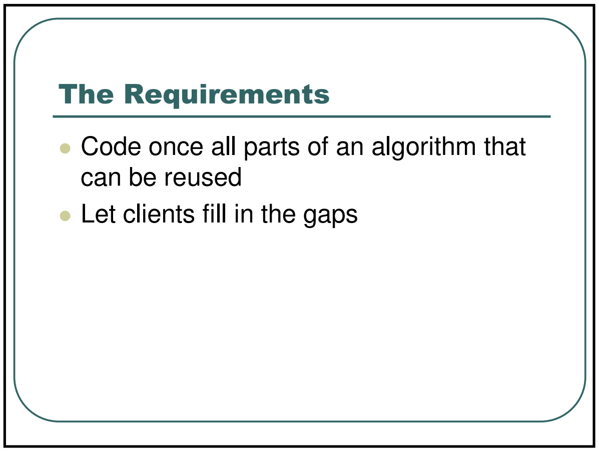
Known Uses

- Context-sensitive help
- Messages in a multi-protocol network service
- Handling user events in a user interface framework
- Updating contained objects/queries in a displayed document



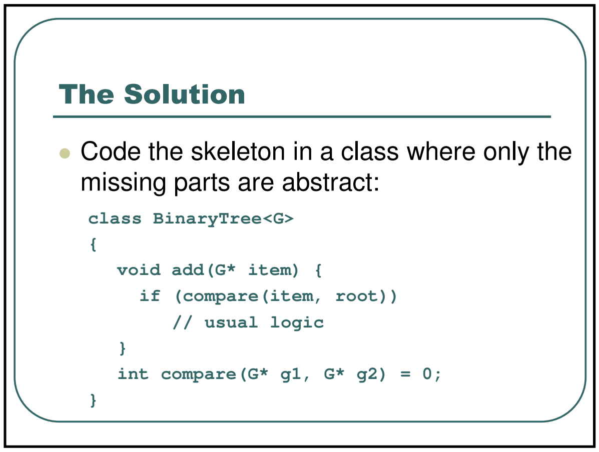
19. Template Method

- Define the skeleton of an algorithm and let subclasses complete it
- For example, a generic binary tree class or sort algorithm cannot be fully implemented until a comparison operator is defined
- How do we implement everything except the missing part?



The Requirements

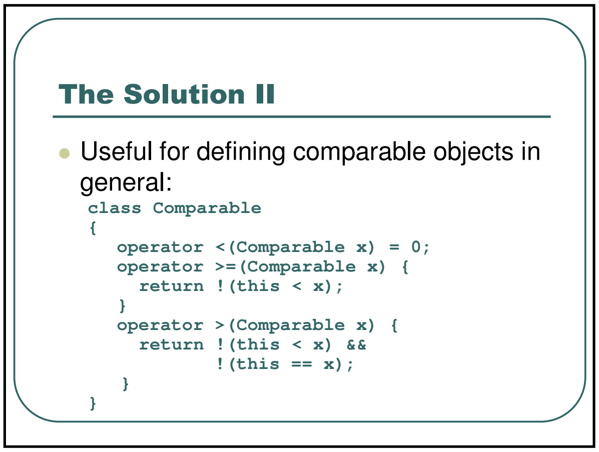
- Code once all parts of an algorithm that can be reused
- Let clients fill in the gaps



The Solution

- Code the skeleton in a class where only the missing parts are abstract:

```
class BinaryTree<G>
{
   void add(G* item) {
     if (compare(item, root))
        // usual logic
   }
   int compare(G* g1, G* g2) = 0;
}
```



The Solution II

- Useful for defining comparable objects in general:

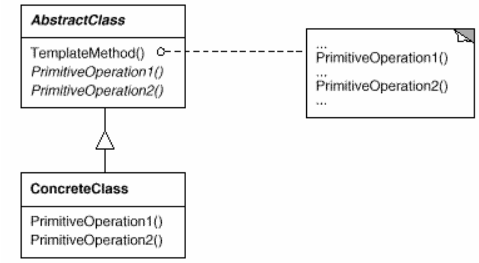
```
class Comparable
{
   operator <(Comparable x) = 0;
   operator >=(Comparable x) {
     return !(this < x);
   }
   operator >(Comparable x) {
     return !(this < x) &&
            !(this == x);
    }
}
```

The Solution III

- A very common pattern:

```
class HelpHandler
{
   handleHelp() {
     if (successor != NULL)
       successor->handleHelp();
   }
   HelpHandler* successor = NULL;
}
```

The UML



The Fine Print

- The template method is public, but the ones it calls should be protected
- The called methods can be declared with an empty implementation if this is a common default
- This template can be replaced by passing the missing function as a template parameter
- Java sometimes requires more coding due to single inheritance

Known Uses

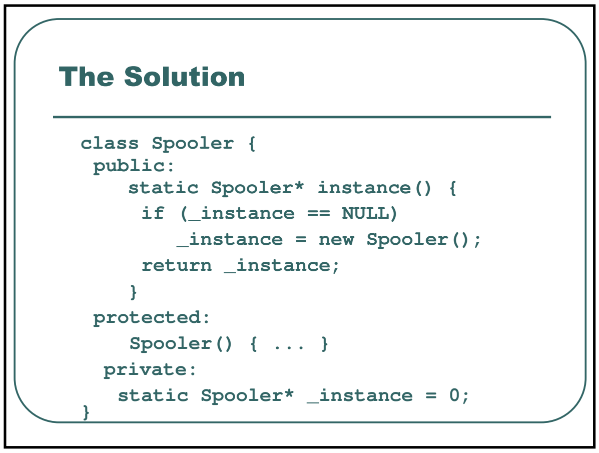
- So fundamental that it can be found almost anywhere
- Factory Method is a kind of template method specialized for creation

20. Singleton

- Ensure that only one instance of a class exists, and provide a global access point to it
- For example, ensure that there’s one *WindowManager*, *FileManager* or *PrintSpooler* object in the system
- Desirable to encapsulate the instance and responsibility for its creation in the class

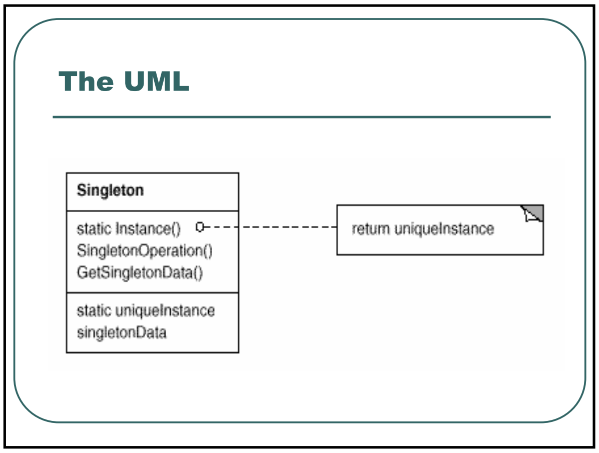
The Solution

- O-O languages support methods shared by all objects of a class
  - *static* in C++ and Java
  - *class methods* in SmallTalk, Delphi
- The singleton class has a reference to its single instance
- The instance has a getter method which initializes it on the first request
- The class’s constructor is protected to prevent creating other instances

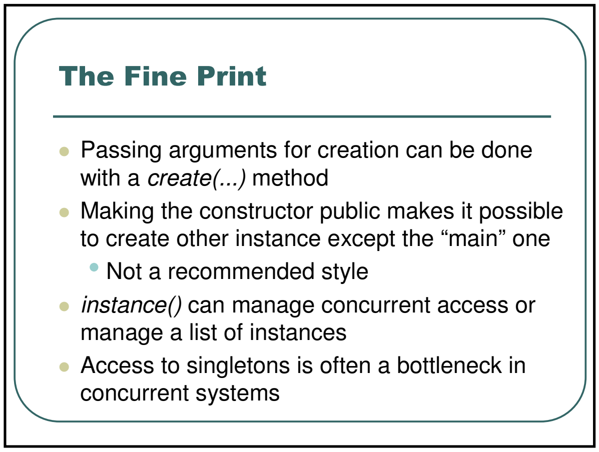


The Solution

```
class Spooler {
 public:
    static Spooler* instance() {
     if (_instance == NULL)
        _instance = new Spooler();
     return _instance;
    }
 protected:
    Spooler() { ... }
  private:
   static Spooler* _instance = 0;
}
```

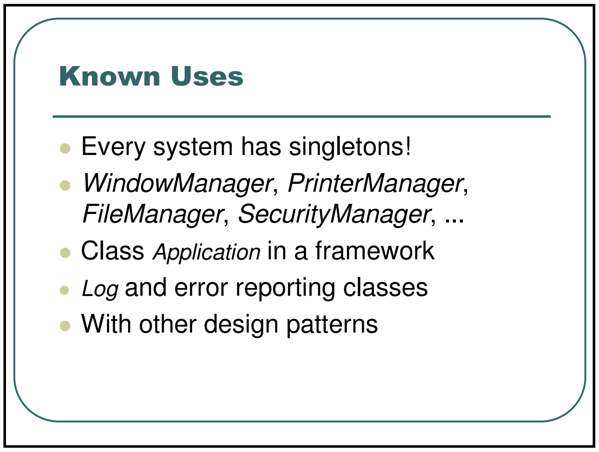


The UML



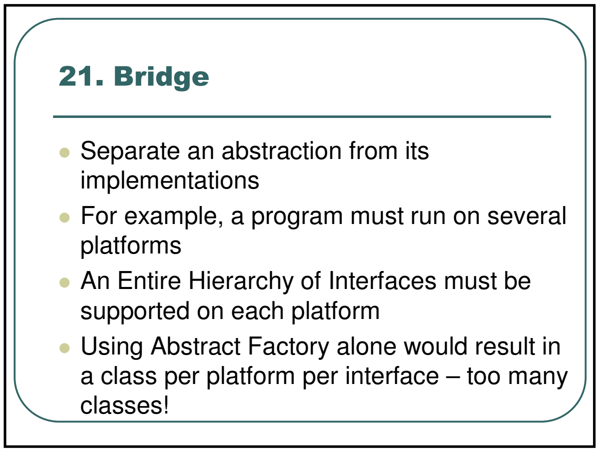
The Fine Print

- Passing arguments for creation can be done with a *create(...)* method
- Making the constructor public makes it possible to create other instance except the "main" one
  - Not a recommended style
- *instance()* can manage concurrent access or manage a list of instances
- Access to singletons is often a bottleneck in concurrent systems



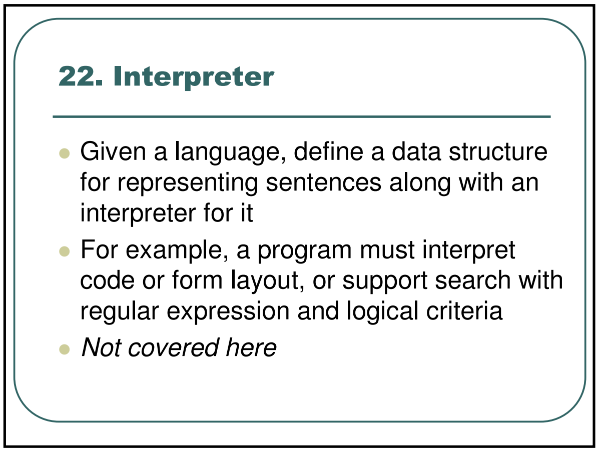
Known Uses

- Every system has singletons!
- *WindowManager*, *PrinterManager*, *FileManager*, *SecurityManager*, ...
- Class *Application* in a framework
- *Log* and error reporting classes
- With other design patterns



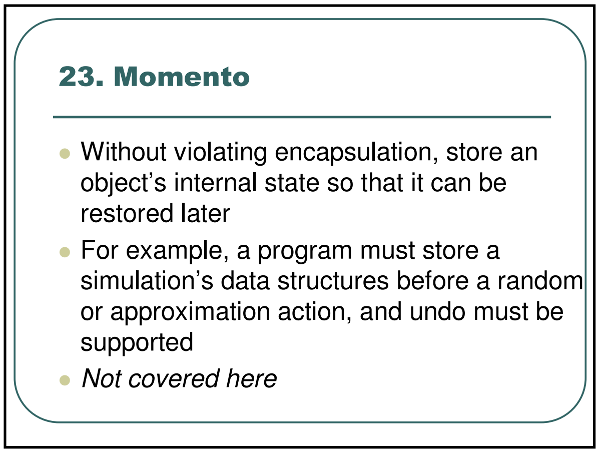
21. Bridge

- Separate an abstraction from its implementations
- For example, a program must run on several platforms
- An Entire Hierarchy of Interfaces must be supported on each platform
- Using Abstract Factory alone would result in a class per platform per interface – too many classes!



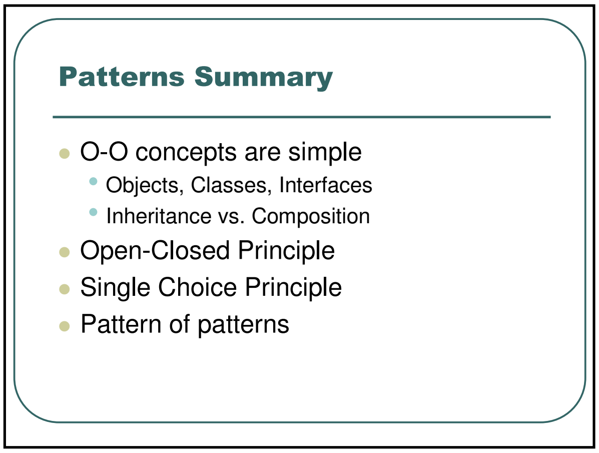
22. Interpreter

- Given a language, define a data structure for representing sentences along with an interpreter for it
- For example, a program must interpret code or form layout, or support search with regular expression and logical criteria
- *Not covered here*



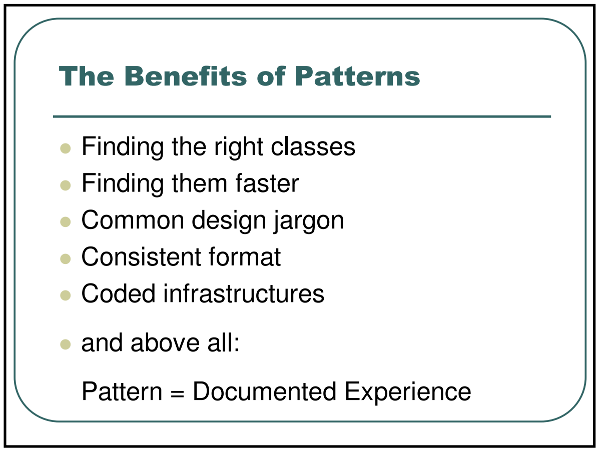
23. Momento

- Without violating encapsulation, store an object's internal state so that it can be restored later
- For example, a program must store a simulation's data structures before a random or approximation action, and undo must be supported
- *Not covered here*



Patterns Summary

- O-O concepts are simple
  - Objects, Classes, Interfaces
  - Inheritance vs. Composition
- Open-Closed Principle
- Single Choice Principle
- Pattern of patterns



The Benefits of Patterns

- Finding the right classes
- Finding them faster
- Common design jargon
- Consistent format
- Coded infrastructures
- and above all:

Pattern = Documented Experience