Selected Design Patterns

- Design Patterns are recurrent solutions to design problems
- They are pros and cons

- We already saw:
  - Factory, Hook, Templates

- Singleton
- Composite

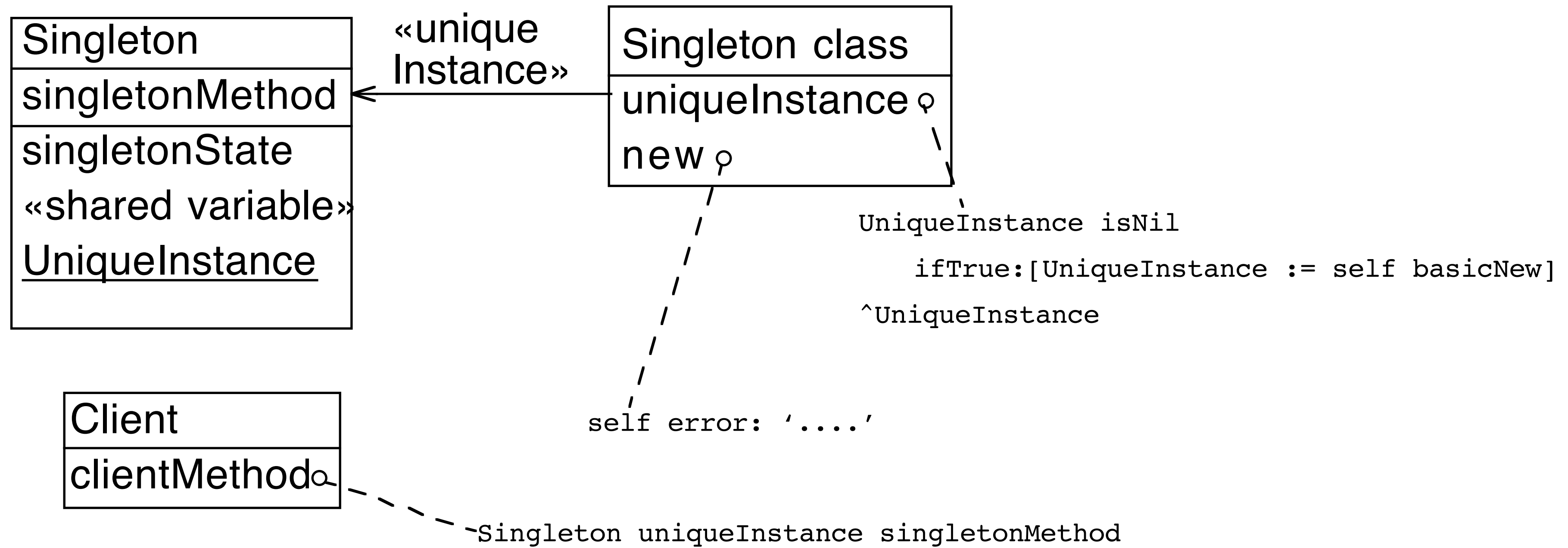
Alert!!! Design Patterns are invading…

- Design Patterns may be a real plague!
- Do not apply them when you do not need them
- Applying too much or badly design patterns makes software rot
- Design Patterns make the software more complex
  - More classes
  - More indirections, more messages
- Try to understand when NOT applying them!

The Singleton Pattern

- *Intent*: Ensure that a class has only one instance, and provide a global point of access to it
- *Problem*: We want a class with a unique instance.
- *Solution*: We specialize the #new class method so that if one instance already exists this will be the only one. When the first instance is created, we store and return it as result of #new.

Possible Design



The Singleton Pattern

```
|aLan|
aLan := NetworkManager new
aLan == LAN new -> true
aLan uniqueInstance == NetworkManager new -> true

NetWorkManager class
  instanceVariableNames: 'uniqueInstance '
NetworkManager class>>new
  self error: 'should use uniqueInstance'

NetworkManager class>>uniqueInstance
  uniqueInstance isNil
        ifTrue: [ uniqueInstance := self basicNew initialize].
  ^uniqueInstance
```

The Singleton Pattern

- Providing access to the unique instance is not always necesssary.
- It depends on what we want to express. The difference between #new and #uniqueInstance is that #new potentially initializes a new instance, while #uniqueInstance only returns the unique instance (there is no initialization)
- Do we want to communicate that the class has a singleton?

Implementation Issues

Singletons may be accessed via a global variable (ex: NotificationManager uniqueInstance notifier).

```
SessionModel>>startupWindowSystem
        "Private - Perform OS window system startup"
        Notifier initializeWindowHandles.
        ...
        oldWindows := Notifier windows.
        Notifier initialize.
        ...
        ^oldWindows
```

- Global Variable or Class Method Access
  - Global Variable Access is dangerous: if we reassign Notifier we lose all references to the current window.
  - Class Method Access is better because it provides a single access point. This class is responsible for the singleton instance (creation, initialization,...).

Implementation Issues

- Persistent Singleton: only one instance exists and its identity does not change (ex: NotifierManager in Visual Smalltalk)
- Transient Singleton: only one instance exists at any time, but that instance changes (ex: SessionModel in Visual Smalltalk, SourceFileManager, Screen in VisualWorks)
- Single Active Instance Singleton: a single instance is active at any point in time, but other dormant instances may also exist. Project in VisualWorks, ControllerManager.

Implementation Issues

- classVariable or class instance variable
- classVariable
  - One singleton for a complete hierarchy
- Class instance variable
  - One singleton per class

Access?

- In Smalltalk we cannot prevent a client to send a message (protected in C++). To prevent additional creation we can redefine new/new:

```
Object subclass: #Singleton
  instanceVariableNames: 'uniqueInstance'
  classVariableNames: ''
  poolDictionaries: ''

Singleton class>>new
  self error: 'Class ', self name, ' cannot create new
  instances'
```

Access using new: not so good idea

```
Singleton class>>new
	^self uniqueInstance
```

- The intent (uniqueness) is not clear anymore! New is normally used to return newly created instances. The programmer does not expect this:

```
|screen1 screen2|
screen1 := Screen new.
screen2 := Screen uniqueInstance
```

Favor Class Behavior

- When a class should only have one instance, it could be tempting to define all its behavior at the class level. But this is not good:
- Class behavior represents behavior of classes: “Ordinary objects are used to model the real world. MetaObjects describe these ordinary objects”
- Do not mess up this separation and do not mix domain objects with metaconcerns.
- What’s happens if later on an object can have multiple instances? You would have to change a lot of client code!

The Composite Pattern

- A Case study: Queries. We want to be able to
- Specify different queries over a repository

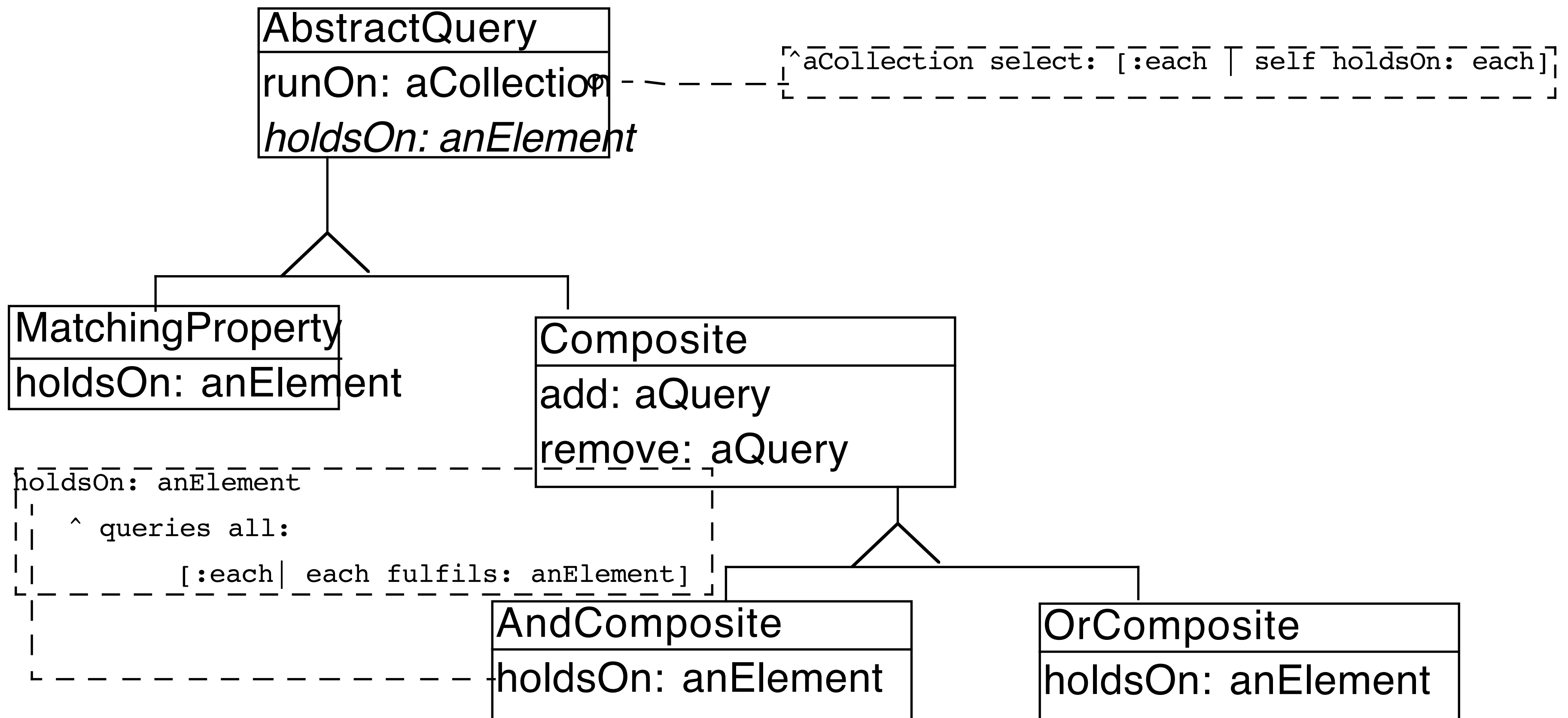
```
q1 := PropertyQuery property: #HNL with: #< value: 4.
q2 := PropertyQuery property: #NOM with: #> value: 10.
q3 := MatchName match: '*figure*'
```

- Compose these queries and treat composite queries as one query

  (e1 e2 e3 e4 ... en)((q1 and q2 and q4) or q3) -> (e2 e5)

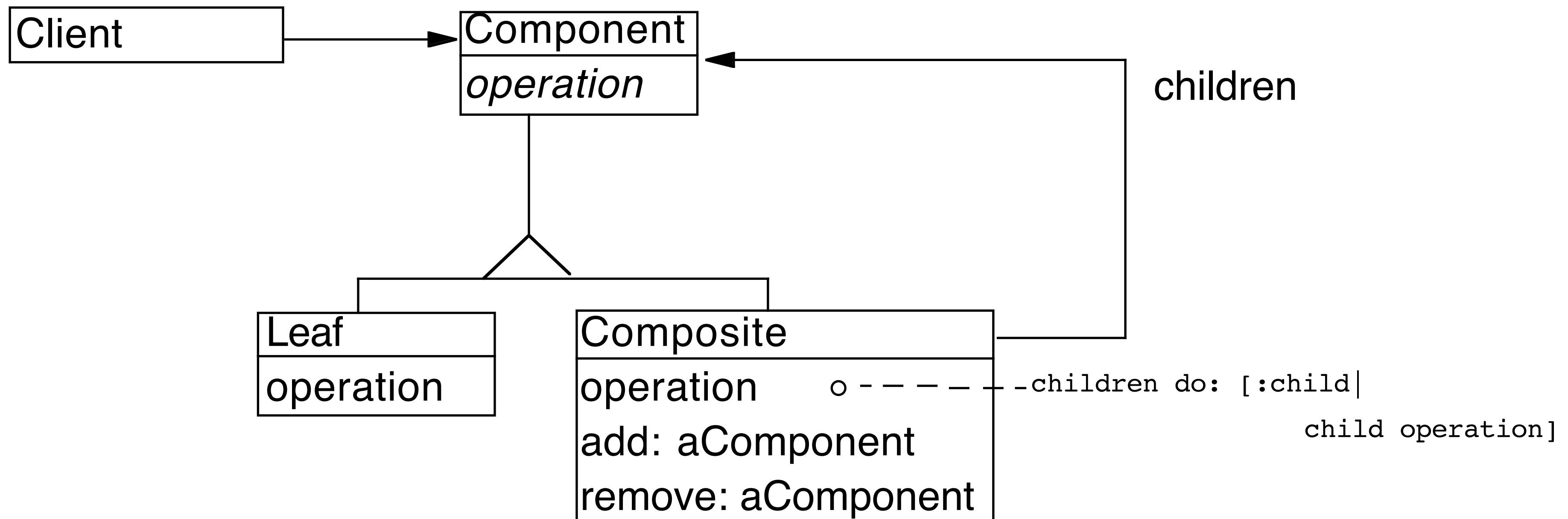
  ```
  composer := AndComposeQuery with: (Array with: q1 with: q2 with: q3)
  ```

A Possible Solution



Composite

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



In Smalltalk

- Composite not only groups leaves but can also contain composites
- In Smalltalk add:, remove: do not need to be declared into Component but only on Composite. This way we avoid to have to define dummy behavior for Leaf

Composite Variations

- Use a Component superclass (To define the interface and factor code there)
- Consider implementing abstract Composite and Leaf (in case of complex hierarchy)
- Only Composite delegates to children
- Composites can be nested
- Composite sets the parent back-pointer (add:/remove:)

Composite Variations

- Can Composite contain any type of child? (domain issues)
- Is the Composite's number of children limited?
- Forward
  - Simple forward. Send the message to all the children and merge the results without performing any other behavior
  - Selective forward. Conditionally forward to some children
  - Extended forward. Extra behavior
  - Override. Instead of delegating