Implementation of Dynamic Binding

Dynamic Binding with Single Inheritance

each object holds reference to dynamic type

type contains array of references to methods
(VFT = virtual function table)

internal representation of method = VFT index,
internal representation of variable = offset from object address

single inheritance: take VFT and variable offsets from superclass,
append additional methods and variable offsets at the end,
inherited method = copied entry in VFT,
overridden method = changed entry in VFT

Dynamic Binding with Interfaces

simple implementation not useable for multiple inheritance
because different VFT indexes inherited for same method

hence, use separate VFT for each implemented interface (except one),
several references to VFTs in object or in primary VFT,
declared type determines the VFT to be used

single inheritance by appending additional methods,
multiple inheritance by adding further VFTs

Dynamic Binding with Multiple Inheritance

same techniques as for interfaces, but object layouts have to be considered

C++: subclass takes over object layouts of all superclasses as object parts (necessary for simple up-casts)

up-cast changes object reference to start of object part by adding **delta**

each object part has its own VFT and remembers its delta

delta must be subtracted before comparing object references for identity

Method Invocation and Thunks

method invocation with multiple inheritance (in general):

```
1.   load [objectReg + #VFToffset], tableReg
2.   load [tableReg + #deltaOffset], deltaReg
3.   load [tableReg + #selectorOffest], methodReg
4.   add objectReg, deltaReg, objectReg
5.   call methodReg
```

delta $=0$ in most cases $\rightarrow$ **thunks** for optimization:
omit statements 2 and 4 and jump to thunk ($=$ overridden method if delta $\neq 0$):

```
thunk: add objectReg, #delta, objectReg
       jump #method
```

Optimizations

if method is statically known

> directly jump to method instead of using the VFT,
> **method inlining** possible to enable further optimizations

methods more likely statically known if compiler knows the whole type hierarchy

partial inlining (specialization) through thunks

thunks (with delta $\neq 0$) are almost always avoidable
if VFT grows in both directions and compiler knows type hierarchy

dynamic caching (mostly in dynamic languages):

> first jump directly to same method as in last invocation,
> then compare classes and jump again if in wrong class

Observer

The Observer Design Pattern

defines 1-to-n relation between objects;
keeps dependent objects informed about state changes

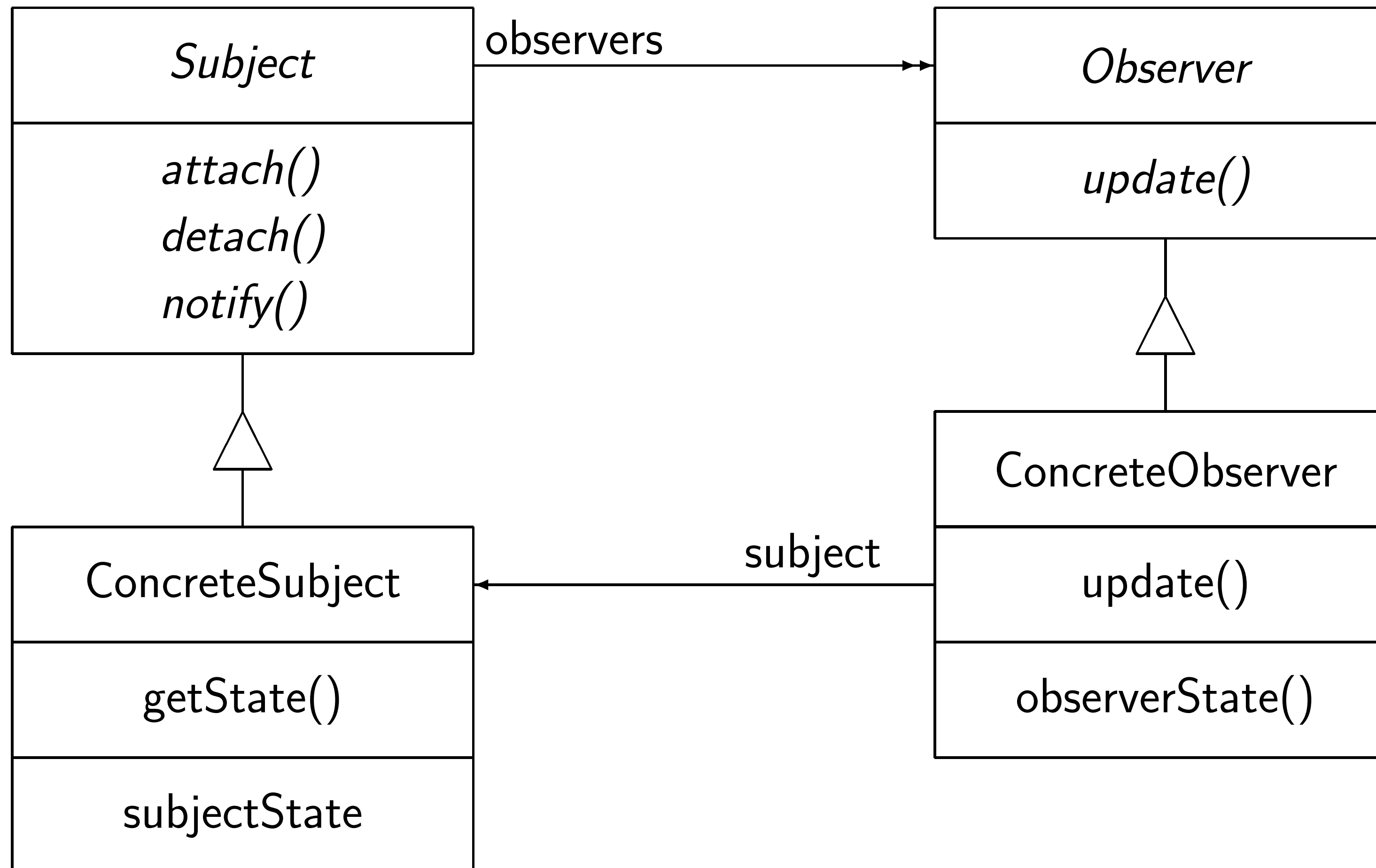
use cases:

an abstraction with two aspects, one dependent on the other;
encapsulation in different objects $\longrightarrow$ separate change and reuse of aspects

a state change causes further state changes
without static knowledge of the objects to be changed

objects shall be informed about something,
but these objects are not statically known (loose coupling)

Structure of Observer



Consequences of Observer

abstract coupling between subject and observers

$\longrightarrow$ can belong to different layers

broadcast happens automatically

$\longrightarrow$ observers can be added and removed at any time

unexpected updates possible because of missing information

$\longrightarrow$ reasons difficult to find

$\longrightarrow$ costs of updates hardly predictable (sometimes high)

Implementation of Observer

subject as argument of "update" (for differentiation)

"notify" triggered by

client of subject	$\longrightarrow$	error-prone
state-changing operations of subject	$\longrightarrow$	unnecessary updates

state of subject must be consistent before "notify"

consider references in observers before deleting a subject

"update" with much or little information in parameters (push or pull model)

observers register only for some aspects