Design Patterns

Elements of Reusable OO Software Design

Design patterns are

- Design problems are rarely unique.
- Chances are that someone else has encountered a similar problem and come up with a good solution in the past.
- "Design Patterns" are reusable solutions to recurring problems
- Gang of 4 (GoF) book: Gamma, Helm, Johnson, & Vlissides, *Design Patterns: Elements of Reusable OO Software*, AW, 1994.

Pattern descriptions in GoF

Name, **Also Known As**. Crucial for building a common vocabulary among software designers.

Intent giving a short description

Motivation, giving example.

Applicability, explaining when to use the pattern.

Structure and **Participants**: How the classes relate.

Collaboration: How the objects use each other.

Consequences. The costs and benefits of the proposed solution.

Implementation, **Sample Code**: Details and variations of implementation.

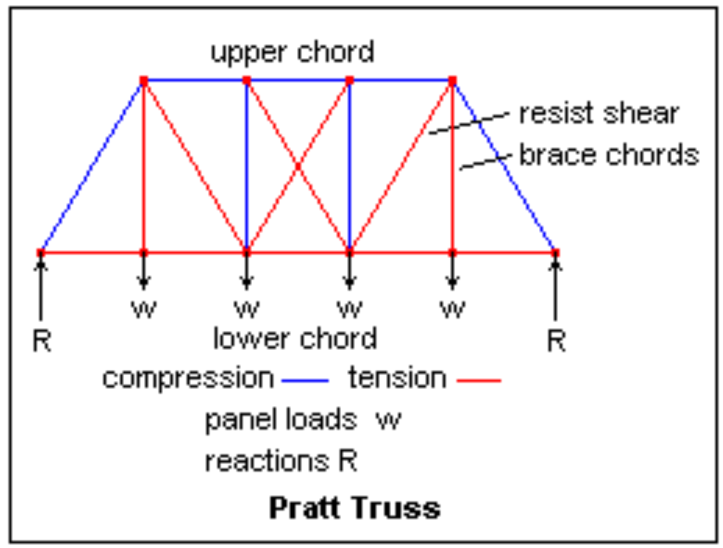
Known Uses. Examples from specific products.

Bridge Pattern

- (If GoF had been written for Civil Engineers.)
- Name: Bridge
- Intent: Allow a road to cross a body of water or other obstacle
- Motivation: It is hard to build a road on water, dangerous to build a road across a highway or railroad track, …
- Applicability: When an obstacle is not too wide or too high and going under or around are not options.

Bridge Pattern (cont)

- Structure:



[And so on.]

Kinds of patterns

- The gang of 4 book (GoF) divides patterns into 3 broad classes:
  - Creational Patterns. Deal with problems involving object creation.
  - Structural Patterns. Deal with problems involving composition and aggregation.
  - Behavioural Patterns. Deal with problems involving object behaviour.

Name: Observer (GoF, Behavioural)

- "**Intent**: Define a 1-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically." [Gamma et al 1994]
- **Also known as** "Listener", "Publish-Subscribe".
- **Motivation**: Need to maintain consistency between related objects without creating unwanted dependencies between classes.
  - Example: In GUI architecture views must be kept consistent with model, but:
    - We don't want the model classes to depend on the view classes so that model classes can be reused with other views.

Observer (GoF, Behavioural) cont.

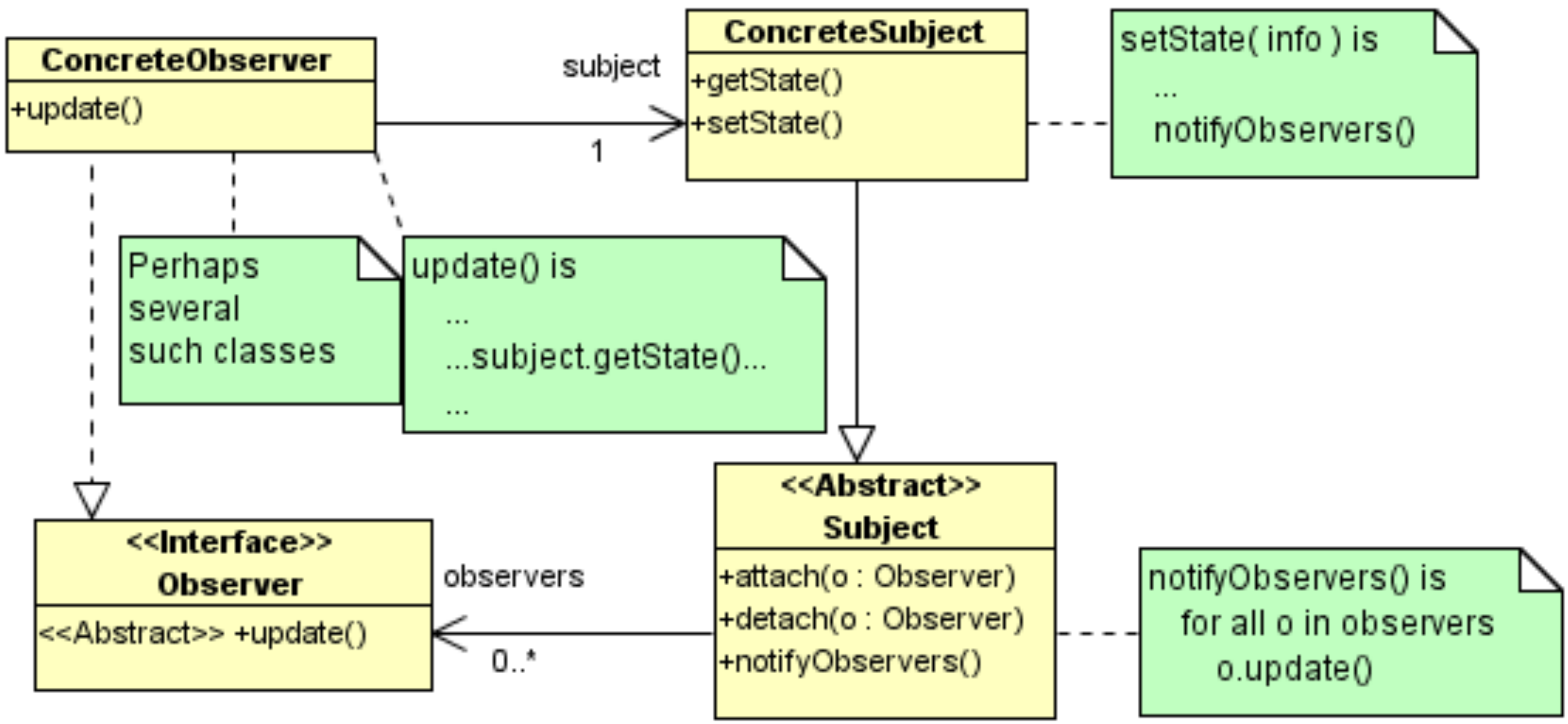
- Applicability:

  Use the Observer pattern:

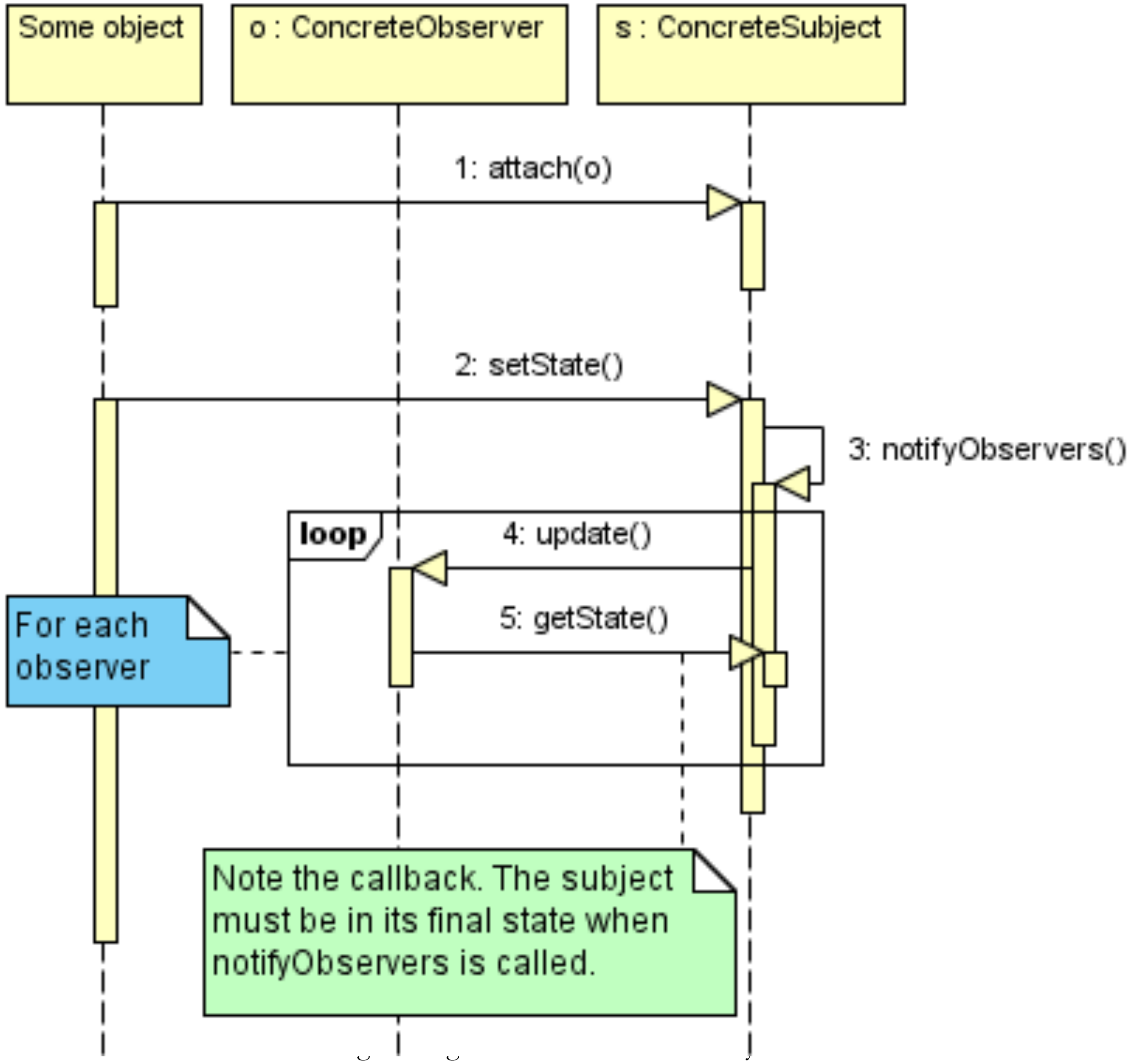
  - "When an abstraction has two aspects, one dependant on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently." [Gamma et al 1994]
  - "When a change to one object requires changing others and you don't know how many objects need to be changed." [Gamma et al 1994]
  - "When an object should be able to notify other objects without making assumptions about who [sic] these objects are." [Gamma et al 1994]

Observer: Structure (from GoF)

- **Structure**

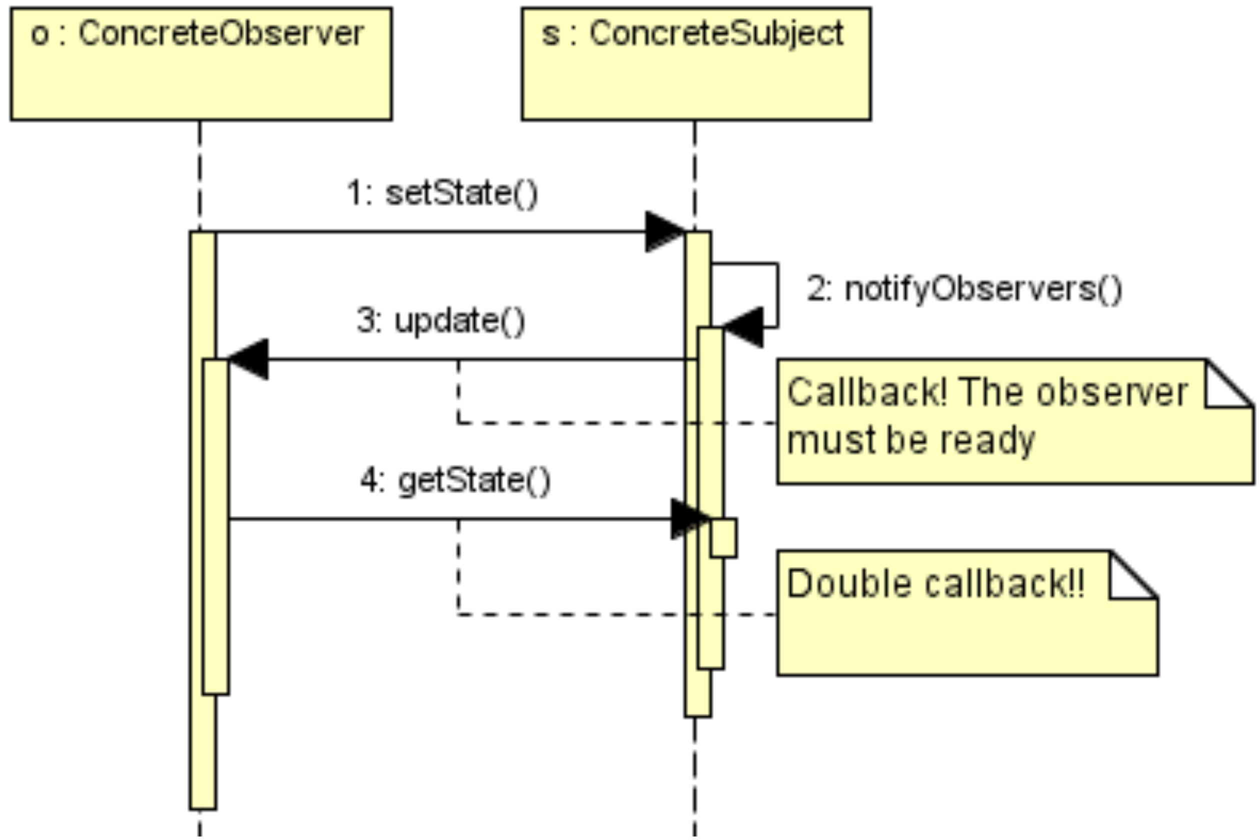


Observer: Behaviour



Observer: Behaviour

- **Collabor-ations**
- **Where an observer calls setState.**



Consequences (+)

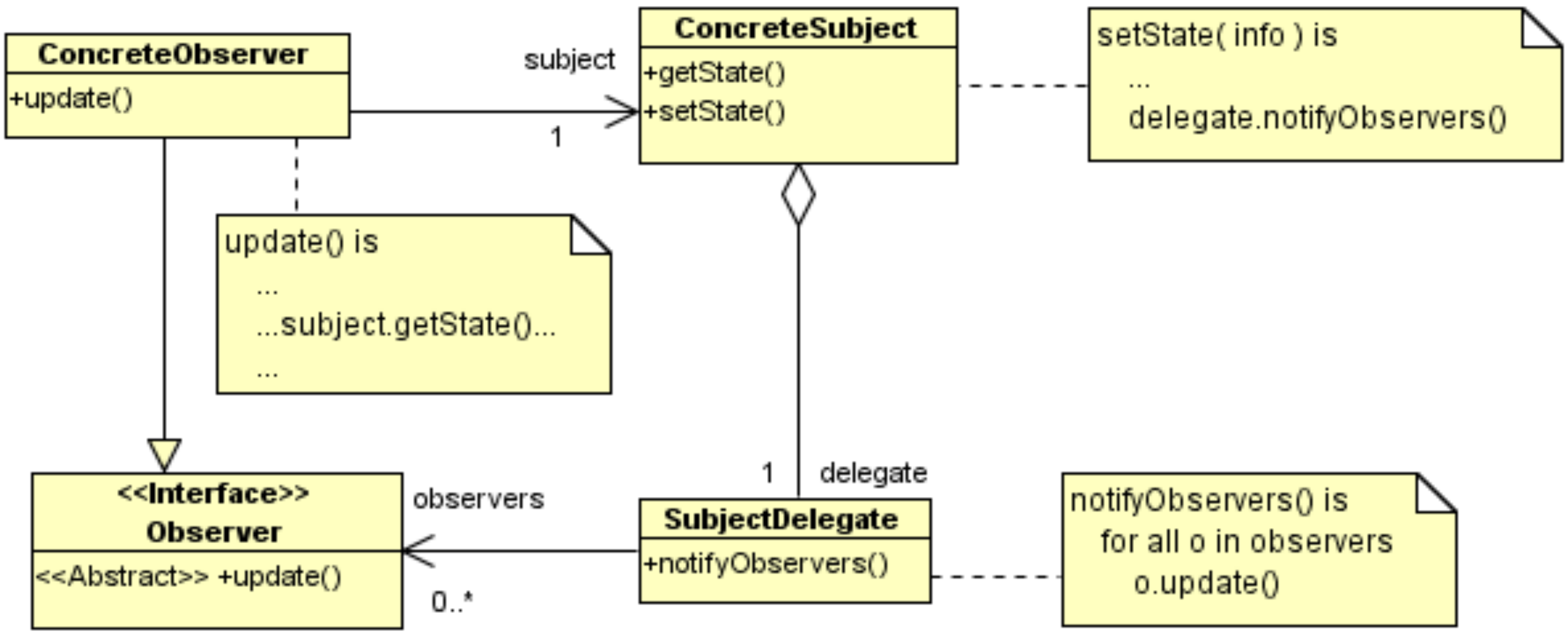
- ConcreteSubject may be reused without reusing observers
- Observer classes may be added or removed without modifying ConcreteSubject or other observer classes.
- Observers may belong to higher level in a layered system.
- Supports broadcast to many observers

Consequences (-)

- Cost of update is hidden from subject.
- No indication of how subject has changed, may lead to costly unneeded updates. (Not usually a problem in GUIs.)
- Subject must be consistent when it calls notifyObservers.
- Observer must be consistent if it calls setState.
- Too many notifications. Every change causes notifications. This may be too many.

Implementations and variations.

- Abstract subject may be replaced by a delegate (delegation rather than inheritance)



Implementations and variations.

- Deleting subject creates dangling references. Observers should be informed of detachment and then detached.
- Multiple subjects for single observer.
- Third party may notify the observers rather than subject. This can avoid problem of too many notifications. (See Teaching Machine example later in these notes.)

Implementations and variations.

- What changed: Subject can inform observer of how it changed. This is the “push model”. Parameterless update() is “pull model”.
  - Push implies observer has some knowledge of what information observers require.
  - We can combine push and pull. The subject pushes a “change event” that gives the Observer enough information to know if the change is interesting to it. Then the Observer pulls the details.
- Observer might know only an abstract subject (or a subject interface). This makes observers reusable with other concrete subjects.

Known Uses

- “observer” package. Multiple views of lists being sorted are presented.
- “Turtle talk”.
  - The `Maze` is the subject. `MazePanel` is the concrete observer.
  - Delegation, rather than inheritance, is used to decouple observer list management from model representation. The delegate is of class

    `javax.swing.event.SwingPropertyChangeSupport`

Known Uses (cont.)

- The Teaching Machine:
  - The Subject represents machine state.
  - Observers display the state to the user.
  - An Executive mediates all user interaction and knows an object that knows the Observers.
  - The Observers are only updated at the end of a user interaction.
  - Why: In response to each user action, there are potentially 1000s of small changes to the state. Updating the displays on every change would be costly and have no benefit.

Known uses (cont.)