AARHUS UNIVERSITET

Software Engineering and Architecture

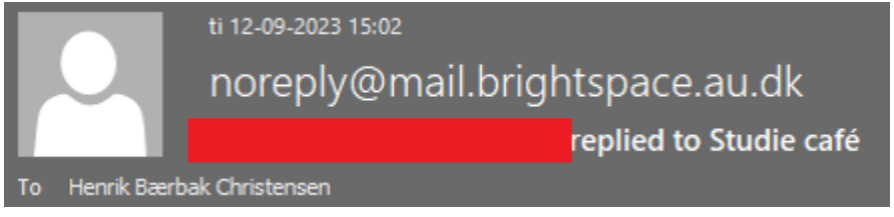
Observer Pattern

The ‘notification’ pattern

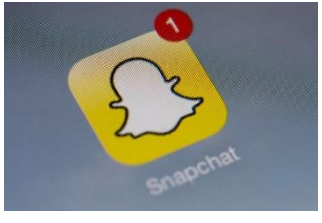
AARHUS UNIVERSITET

Notifications

- A recurring task
  - *When some object's state change, then notify those who subscribe/need to know*
- *Examples*
  - The BrightSpace forum/discussions



  - SoMe notifications



Reflecting State Changes

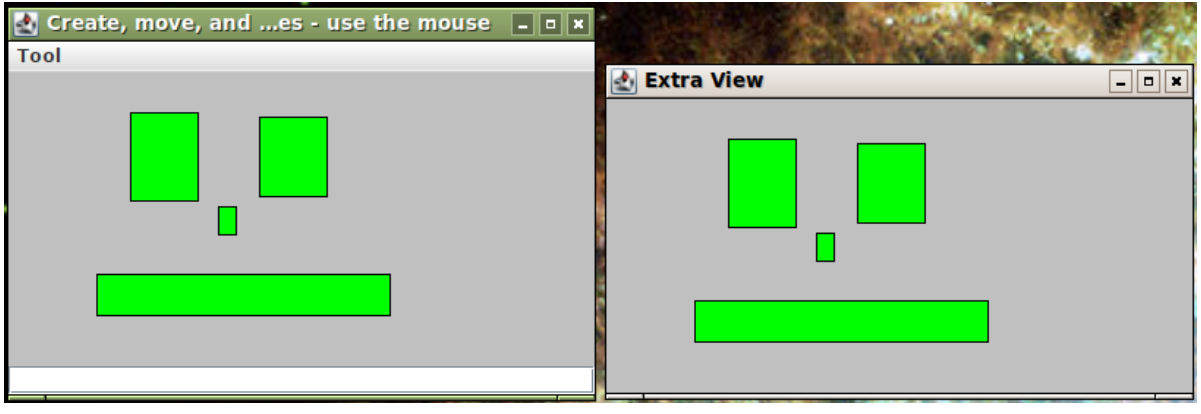
AARHUS UNIVERSITET

And Tons of it...

AARHUS UNIVERSITET

Original Problem

- Challenge when graphical screens were invented:
  - *writing programs with a graphical user interface*
  - multiple open windows showing the same data – keeping them consistent



Xerox Parc in the 1980ies

AARHUS UNIVERSITET

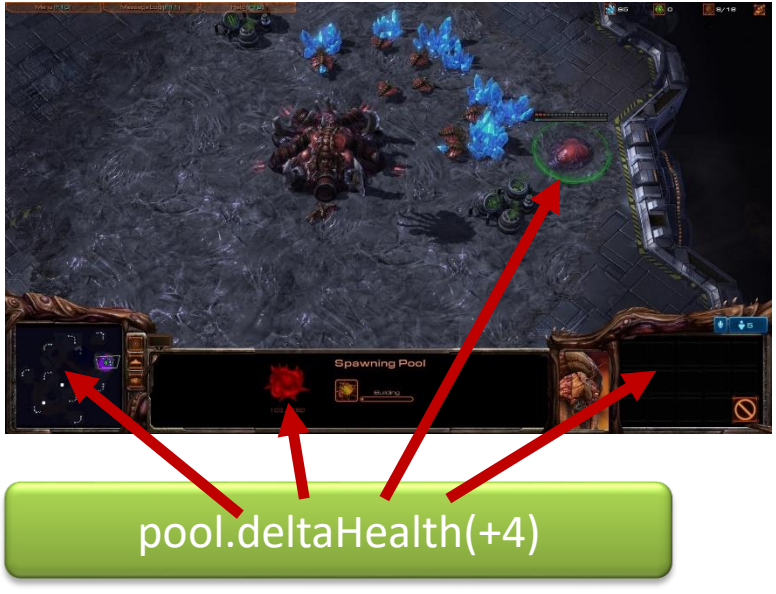
The Observer Pattern

AARHUS UNIVERSITET

Observer

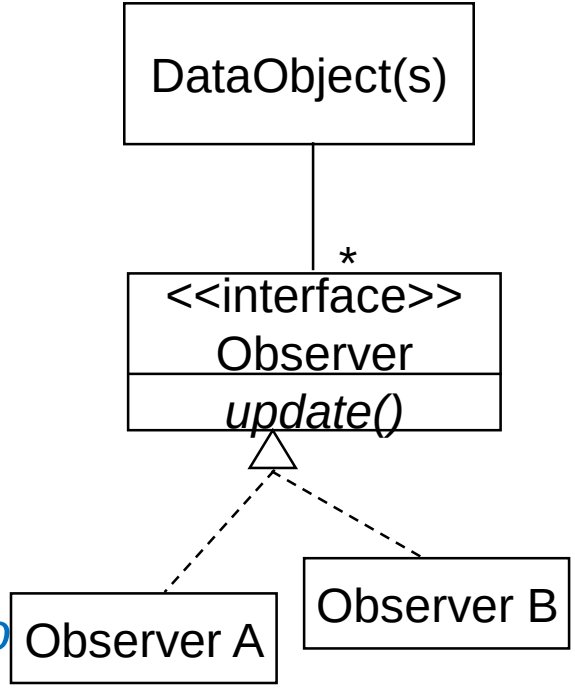
- *Intent*
  - *Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically*.

- Example
  - Spawning pool's health increases
  - 4 UI elements are notified and can update accordingly



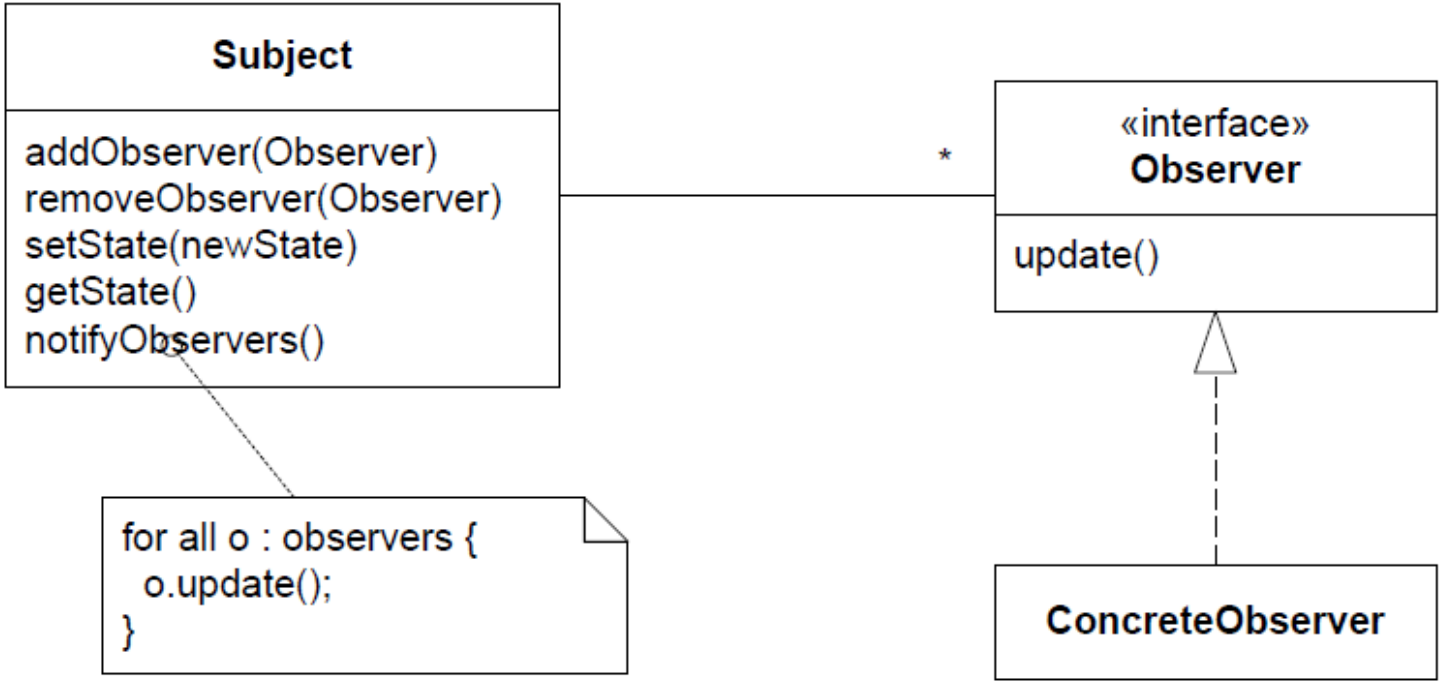
3-1-2

- Our 3-1-2 process?
  - Goal: *Keeping multiple (visual) objects updated (consistent) when state changes*
- Analysis:
  - **Data** is shared but **visualization** is variable!
  - ③ Data **visualization** is variable behavior
  - ① Responsibility to visualize/update data is expressed in interface: ***Observer***
  - ② Instead of data object itself is responsible for updating graphics it *lets someone else do the job: the Observers*



Observer: Structure

Structure:



The Roles

- Subject / or **Observable**

Subject

- Must handle storage, access, and manipulation of state
- Must maintain a set of observers and allow adding and removing observers to this set
- Must notify every observer in the set of any state change by invoking each observer's update method

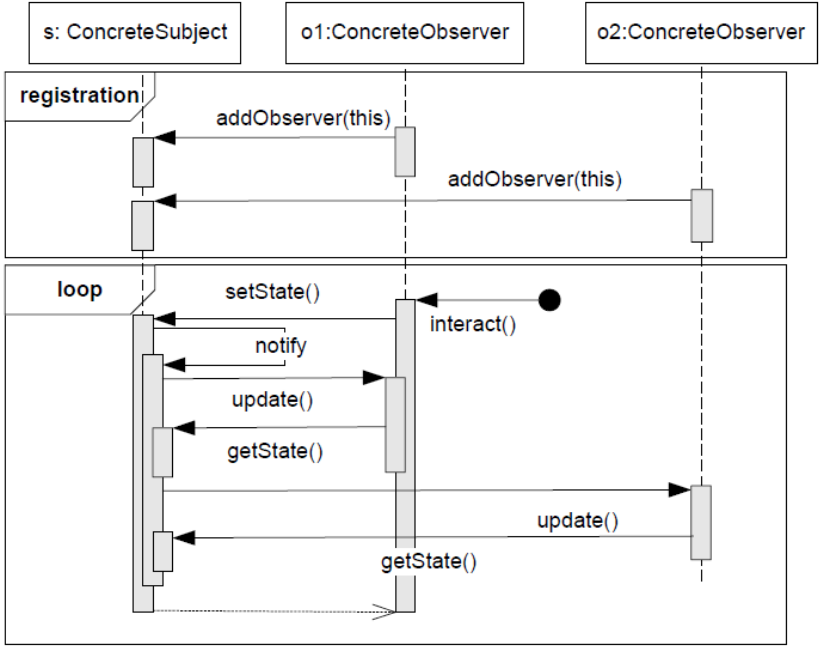
- Observer / or **Listener**

Observer

- Must register itself in the subject
- Must react and process subject state changes every time a notification arrives from the subject, that is, the update method is invoked

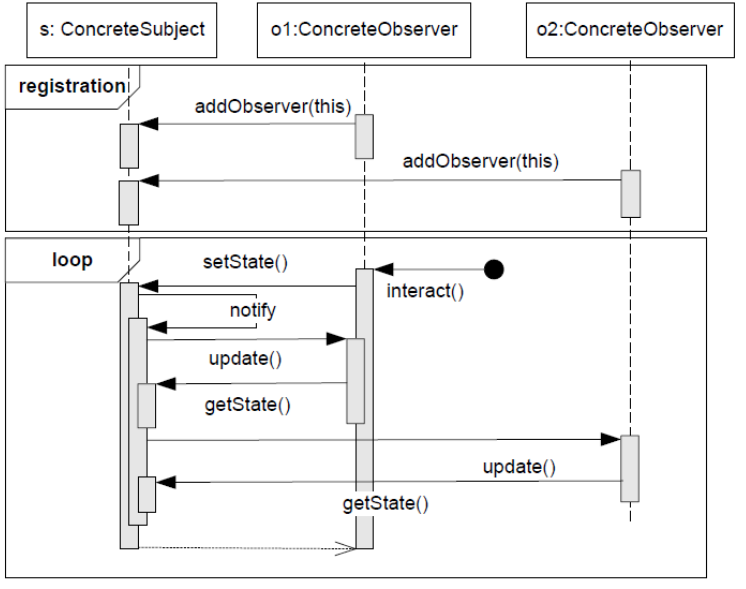
Observer Protocol

- **Protocol:**
  - *A convention detailing the expected sequence of interactions or actions expected by a set of roles.*



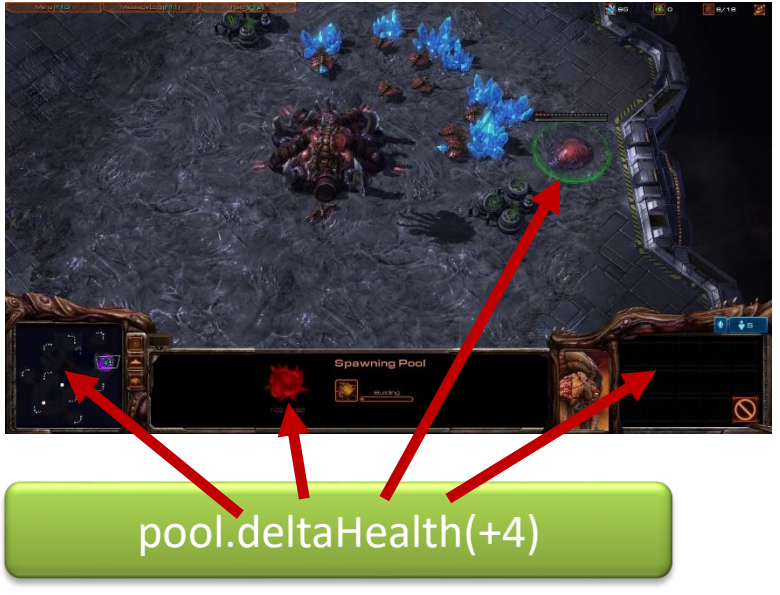
That is

- When a **state change** happens in **Subject**
  - Then it loops over all registered **Observers**
  - … and for each
    - … calls its **update()** method (= the notification)



Exercise

- Objects
  - The spawning pool
  - The detail map
  - The overview map
  - The object detail UI
  - The object command UI
- Which are subject(s)?
- Which are observer(s)?



Observer

- Benefits
  - open ended number of viewer types (run-time binding)
  - need not be known at develop time
    - change by addition, not by modification...
  - any number of views open at the same time when executing
  - all guarantied to be synchronized
    - (if responding correctly to their ‘update()’ calls)

- Liabilities
  - update sequence can become cyclic or costly to maintain

AARHUS UNIVERSITET

Push / Pull Variants

- Observer is implemented in two variants
  - **Pull variant**
    - Update() method with **no parameters/state details**
      - Observer needs to 'pull' the relevant information
  - **Push variant**
    - Update(….) method(s) with **parameters/details about state change**
      - Often in the form of a specific **event class**
      - Relevant information is 'pushed' to observer

Predominant variant today

- Exercise:
  - What is benefits of each?
    - Hint: Consider a subject with 56 different independent state changes?

AARHUS UNIVERSITET

Observer Terminology

- Observer pattern is used in so many places that a special *vocabulary* is often used, as well as naming conventions on the methods.
  - Observer often called **Listeners** (Java/Swing)
    - 'I listen to the events that occur in the "subject"'
  - Observer methods often called **Callback functions**
  - The subject emits **Events**
    - 'I did this state change'

```
JButton okButton = new JButton("OK");
okButton.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
      statusLabel.setText("Ok button is clicked here");
   }
});
```

AARHUS UNIVERSITET

Observer Terminology

- Observer pattern is used in so many places that a special *vocabulary* is often used, as well as naming conventions on the methods.
  - The methods that receives the events are often named '**onX**()'
    - In the Observer/Listener class
  - The methods emitting the events are often named '**notifyX()**'
    - In the Subject class

```
public void notifyPlayCard(Player who, Card card) {
```

Example: Android

LocationListener

The LocationListener interface, which is part of the Android Locations API is used for receiving notifications from the **LocationManager** when the location has changed. The **LocationManager** class provides access to the systems location services. The LocationListener class needs to implement

- **onLocationChanged(Location location)** : Called when the location has changed.
- **onProviderDisabled(String provider)** : Called when the provider is disabled by the user.
- **onProviderEnabled(String provider)** : Called when the provider is enabled by the user.
- **onStatusChanged(String provider, int status, Bundle extras)** : Called when the provider status changes.

AARHUS UNIVERSITET

Mandatory Note

- It is quite easy to encode a new hero power ala
  - … US Chef, that adds +7 health to all own minions, whose health is below 3 and whose names begins with a consonant…
- But, how do we update all the right elements of the UI, without redrawing everything from scratch all the time???
- Answer:
  - **Let Game emit Events for every detailed state change**
    - A card has been played; hero has been attacked; card drawn; …
  - **Let the UI listen for these events**
    - **And update the corresponding UI element accordingly**

AARHUS UNIVERSITET

Mandatory Note

- **Let Game emit Events for every state change**

```
public interface GameObserver {
  void onPlayCard(Player who, Card card, int atIndex);
  void onChangeTurnTo(Player playerBecomingActive);
  void onAttackCard(Player playerAttacking, Card attackingCard, Card defendingCard);
  void onAttackHero(Player playerAttacking, Card attackingCard);
  void onUsePower(Player who);

  void onCardDraw(Player who, Card drawnCard);
  void onCardUpdate(Card card);
  void onCardRemove(Player who, Card card);
  void onHeroUpdate(Player who);
  void onGameWon(Player playerWinning);
}
```

- US Chef Hero power used in a Game
  - adds +7 health to all own minions, whose health is below 3 and whose names begins with a consonant
    - First *onHeroUpdate()* event emitted (argue why)
      - **Update the Hero Graphics on the UI**
    - Next a series of *onCardUpdate()* events emitted, one for each affected card
      - **Update the UI representations of those minions (only)**

AARHUS UNIVERSITET

Mandatory Note

- The UI is then a listener on the game events

```
public class HotStoneDrawingSolution implements Drawing, GameObserver {
```

- And implement the *onXEvent()* methods ala

```
@Override
public void onCardUpdate(Card card) {
  HotStoneActorFigure actor = actorMap.get(card);
  // Opponent cards may not have an associated actor
  // for instance if they are in the hand.
  if (actor != null) {
    actor.updateStats();
  }
}
```

AARHUS UNIVERSITET

Mandatory Pitfall

- *Intent*
  - *Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically*.
- Exercise
  - Can there ever be fired an update event when an *accessor method* has been called on the Subject?

- Morale:
  - All *notifyX()* calls are always in (exercise solution) methods!

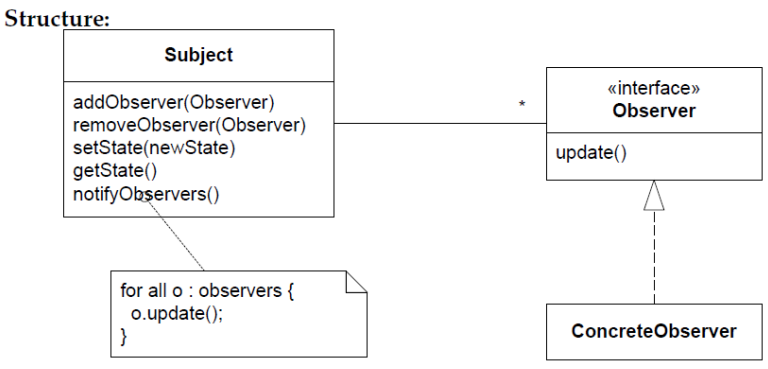
WarStory

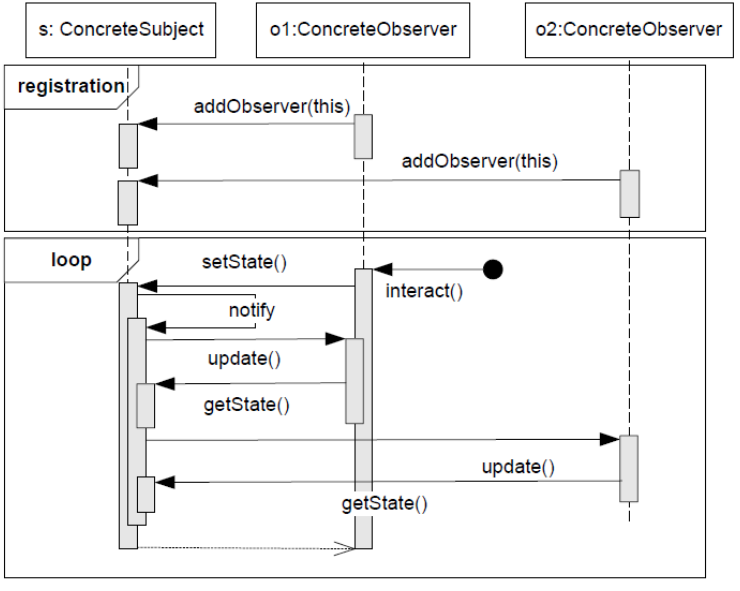
- A previous SWEA student group had
  - *notifyGameWon()*
- Called in their game's *getWinner()* method
- Which means
  - UI is notified, which called… (guess)

- What was their problem?

Summary

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





And Sidebar

- No, neither LoL nor StarCraft II uses the observer pattern for UI updates…
- **GameEngine** architecture
  - **Loop 60+ times a second**
    - **Redraw every visible element from a scratch based upon the state in the underlying game model**
- No wonder we need hefty graphics cards ☺