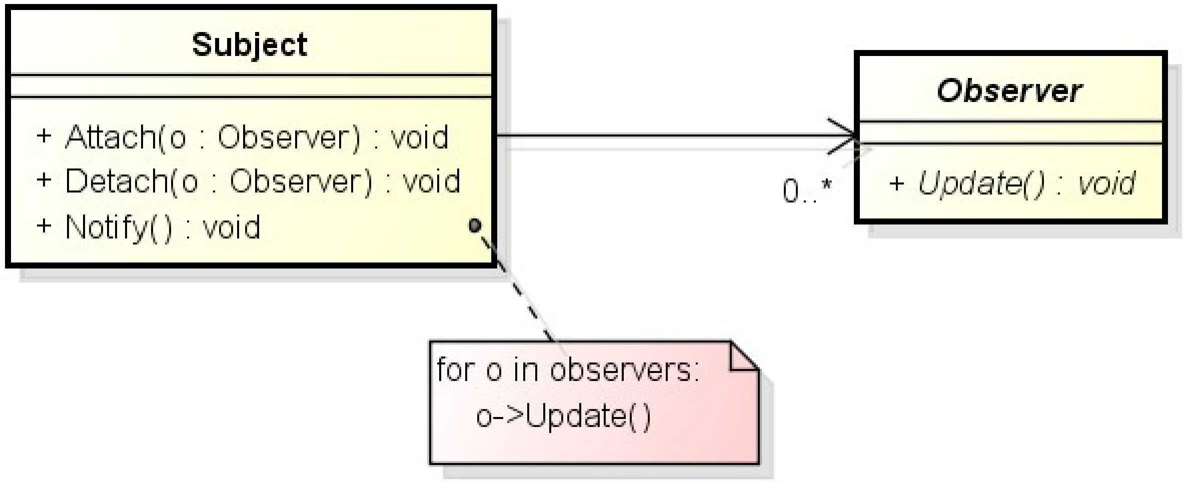
Observer Pattern

Adam Gu, 2015/7/16

Introduction

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



Implementation

Who triggers the update?

What object actually calls the Notify to trigger the update?

a) Have state-setting operations on Subject call Notify after they change the subject's state.

b) Make clients responsible for calling Notify at the right time.

When to detach an observer?

Deleting a subject should not produce dangling references in its observers.

a) Make the subject notify its observers as it is deleted so that they can reset their reference to it.

b) ?

Implementation (cont.)

Subject state should be self-consistent before notify.

Notify after added, before remove, before destroy, etc.

Push or pull?

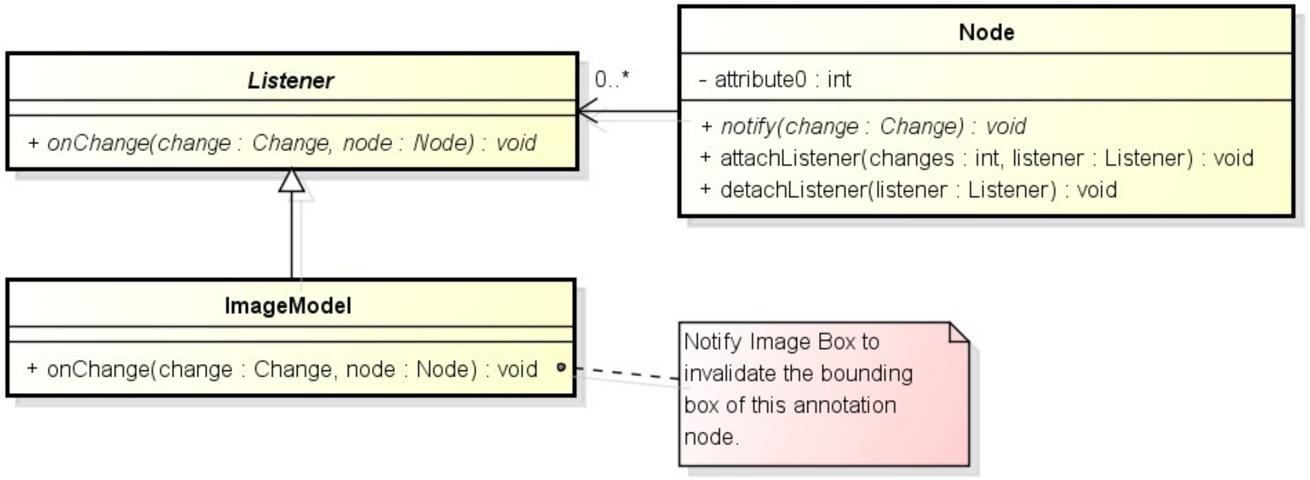
The subject passes additional information about the change as an argument to Update. The amount of information may vary widely.

Register observers only for specific events of interest.

```
void Subject::Attach(Observer*, Aspect& interest);
```

Example: Annotation

2D graphic annotation library for CSI.


Node
- attribute0 : int
+ notify(change : Change) : void
+ attachListener(changes : int, listener : Listener) : void
+ detachListener(listener : Listener) : void
0..*
Listener
+ onChange(change : Change, node : Node) : void
ImageModel
+ onChange(change : Change, node : Node) : void
Notify Image Box to invalidate the bounding box of this annotation node.

Node (Subject)

```cpp
class Node {
public:
  void attachListener(int changes, Listener* listener);
  void detachListener(Listener* listener);

  virtual void notify(Change change);

private:
  // Listener -> changes
  typedef std::map<Listener*, int> listeners_t;
  listeners_t listeners;
};
```

Listener (Observer)

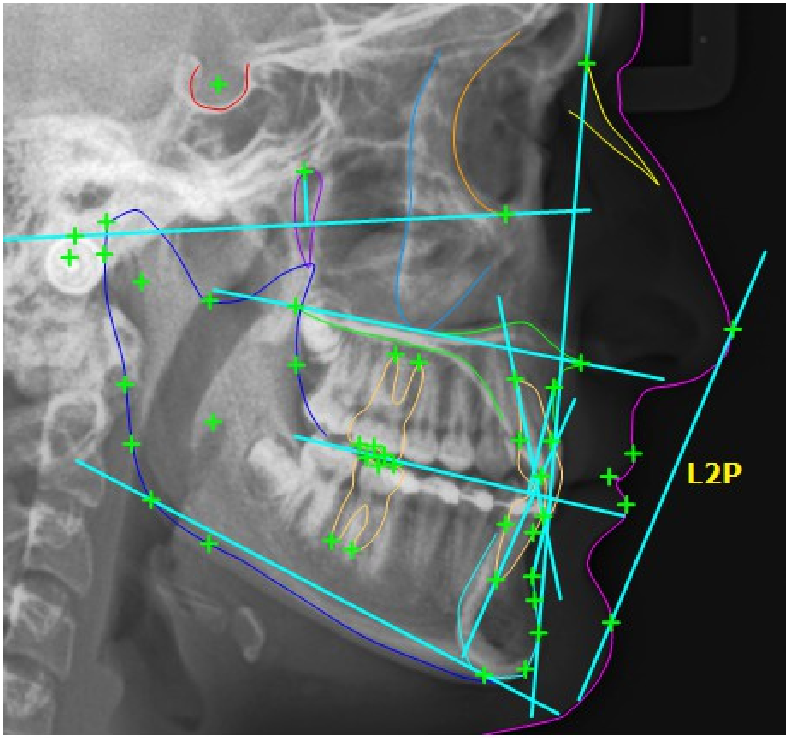
```
class Listener {
public:
  virtual void onChange(Change change, Node* node) = 0;
  virtual void onChange(Change change, const NodeList& nodes);
};
```

Changes (Interest)

> MatrixChange / MatrixHasChanged: Move, rotate
> VisibleChange / VisibleHasChanged: Show, hide.
> StateChange / StateHasChanged: Select, normal, etc.
> StyleChange / StyleHasChanged: Pen, brush, thickness, etc.
> ShapeChange / ShapeHasChanged:
>
> AddedChange: After added to the tree
> RemoveChange: Before removed from the tree
> DestroyChange: Before deleted

Dependency L2P

A line determined by two points.



Dependency L2P: Attach Listener

```
DependencyL2P(Landmark* point1_, Landmark* point2_, Line* line_)
    : point1(point1_), point2(point2_) {
  point1->attachListener(PointChanges, this);
  point2->attachListener(PointChanges, this);

  line->attachListener(LineChanges, this);
}
```

Dependency L2P: onChange

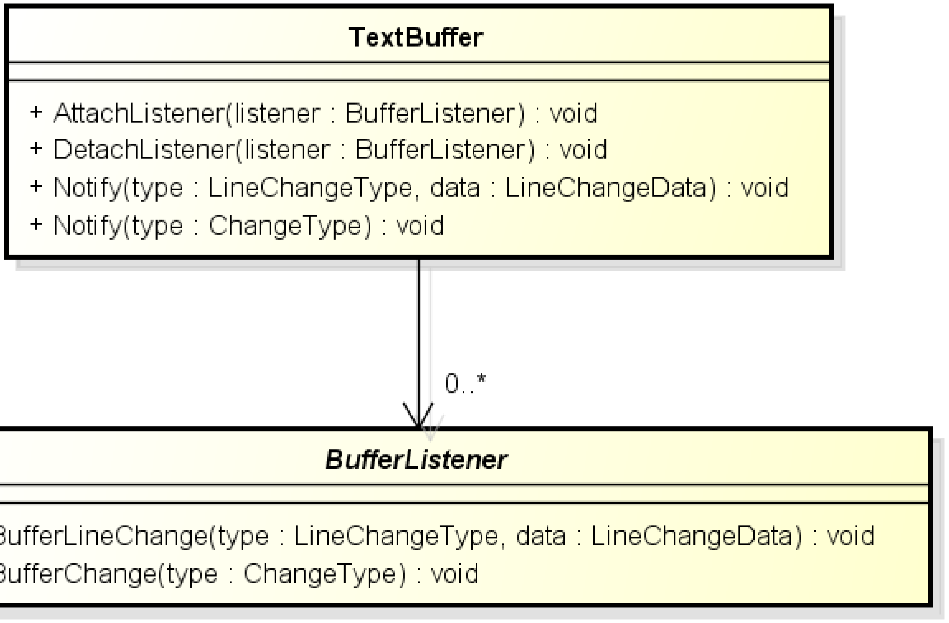
```
void DependencyL2P::onChange(Change change, Node* node) {
  if (node == line) {
    if (change == DestroyChange) {
      detachSelf();
      deleteSelf();
    } else if (change == ShapeHasChanged) {
      if (getEnabled()) {
        DependencyLxP::adjustLine();
      }
    }
  } else { // Point changes.
    ...
```

Dependency L2P: onChange (cont.)

```
} else { // Point changes.
    if (change == DestroyChange) {
      detachSelf();
      line->destroySelf();
      line = NULL;
      deleteSelf();
    } else {
      if (getEnabled()) {
        apply();
      }
    }
  }
}
```

Example: Text Editor

A programmer's text editor.


TextBuffer
+ AttachListener(listener : BufferListener) : void
+ DetachListener(listener : BufferListener) : void
+ Notify(type : LineChangeType, data : LineChangeData) : void
+ Notify(type : ChangeType) : void
0..*
BufferListener
+ OnBufferLineChange(type : LineChangeType, data : LineChangeData) : void
+ OnBufferChange(type : ChangeType) : void

Text Buffer (Subject)

Text buffer represents a text file in memory.

```
class TextBuffer {
public:
  void AttachListener(BufferListener* listener);
  void DetachListener(BufferListener* listener);

  void Notify(LineChangeType type, const LineChangeData& data);
  void Notify(ChangeType type);

private:
  std::vector<BufferListener*> listeners_;
};
```

Buffer Listener (Observer)

When text buffer is changed, buffer listeners will be notified.

```
class BufferListener {
public:
  virtual void OnBufferLineChange(LineChangeType type, const
LineChangeData& data) = 0;

  virtual void OnBufferChange(ChangeType type) = 0;
};
```

Change Types

```
enum LineChangeType {
  kLineUpdated       = 1,
  kLineAdded         = 2,
  kLineDeleted       = 4,
  kLineRefresh       = 8,
};
```

```
enum ChangeType {
  kEncodingChange    = 1,
  kFileNameChange    = 2,
  kModifiedChange    = 4,
  kFileFormatChange = 8,
};
```

Text Window

```
class TextWindow : public wxScrolledWindow, public BufferListener {
public:
  virtual void OnBufferLineChange(LineChangeType type, const
LineChangeData& data);
  virtual void OnBufferChange(ChangeType type);
};
```

Handle line update change:

- Update text size and client virtual width.
- Refresh the lines specified by the line change data.

Example: CSI Save Image

Without Observer

CmdSave & DentalArchPanel:

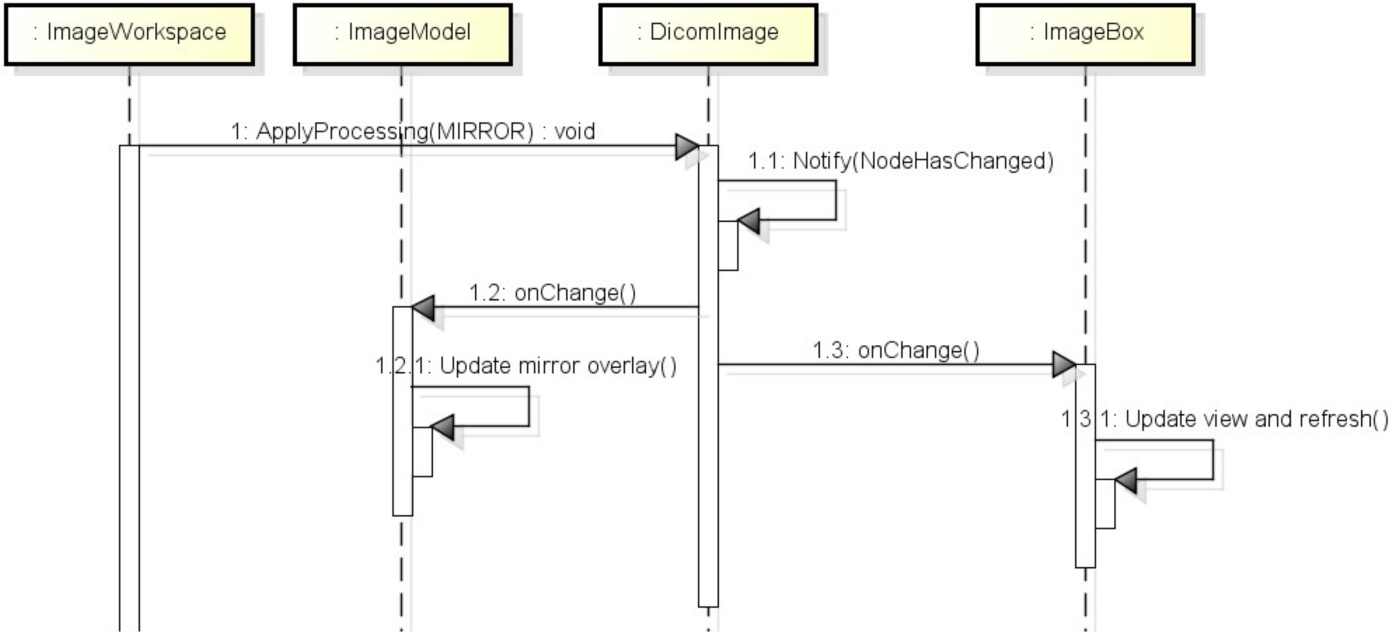
```
BasicImageInfo* imgInfoOverlay = img->GetBasicImageInfoNode();
if (imgInfoOverlay != NULL && img->GetDicomImage()-
>GetImageStatus() == kStatusFull) {
  imgInfoOverlay->Update();
  img->GetDicomImage()->notify(at::RefreshChange);
}
```

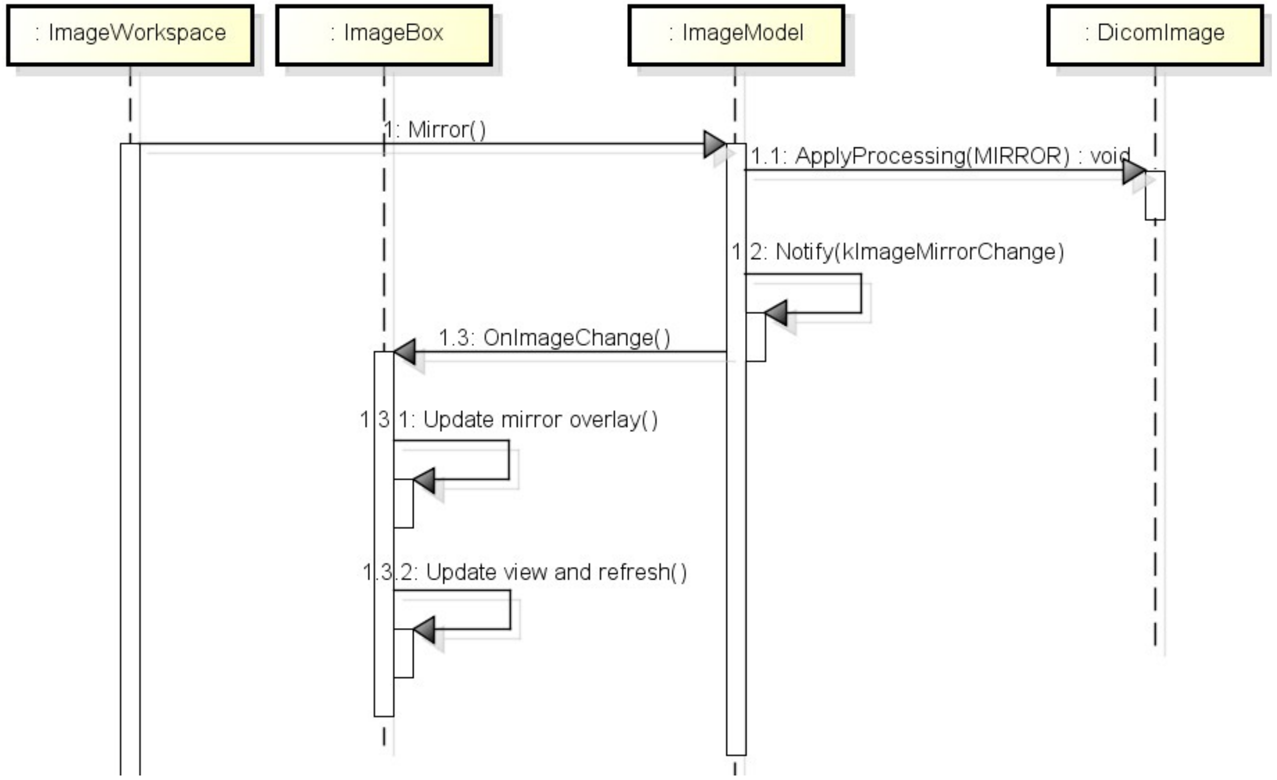
Observer

CmdSave & DentalArchPanel:

```
image_model->Notify(ImageModel::kInfoChange);
```

Example: CSI Mirror


: ImageWorkspace
: ImageModel
: DicomImage
: ImageBox
1: ApplyProcessing(MIRROR) : void
1.1: Notify(NodeHasChanged)
1.2: onChange()
1.2.1: Update mirror overlay()
1.3: onChange()
1.3.1: Update view and refresh()


: ImageWorkspace
: ImageBox
: ImageModel
: DicomImage
1: Mirror()
1.1: ApplyProcessing(MIRROR) : void
1.2: Notify(kImageMirrorChange)
1.3: OnImageChange()
1.3.1: Update mirror overlay()
1.3.2: Update view and refresh()