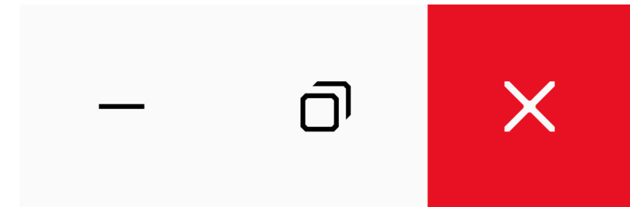
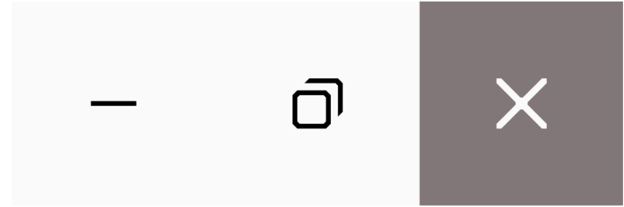


Undo-Redo

Principles and Concepts
Undo Patterns
Implementation

July 17



U CS 349

Principles and Concepts

Benefits of Undo / Redo

Undo lets you **recover from errors**

- input errors (human) and interpretation errors (computer)
- you can work quickly (“without fear”)

Undo enables **exploratory learning**

- try things you don’t know the consequences of (without fear or commitment)
- try alternative solutions (without fear or commitment)

Undo lets users **evaluate modifications**

- fast do-undo-redo cycle to evaluate last change to document

Unless stated otherwise, “undo” means “undo / redo” in these slides.

Checkpointing

A manual undo method

- you save the current state so you can rollback later (if needed)

Consider a video game …

- You kill a monster
- You save the game
- You try to kill the next monster
- You die
- You reload the saved game
- You try to kill the next monster
- You kill the monster
- You save the game

Undo Design Choices

As a designer, you need to consider the following:

- **Undoable Actions**: what actions should (or can) be undone?
- **State restoration**: what part of UI is restored after undo?
- **Granularity**: how much should be undone at a time?
- **Scope**: is undo global, local, or someplace in between?

Undoable Actions

Some actions may be omitted from undo:

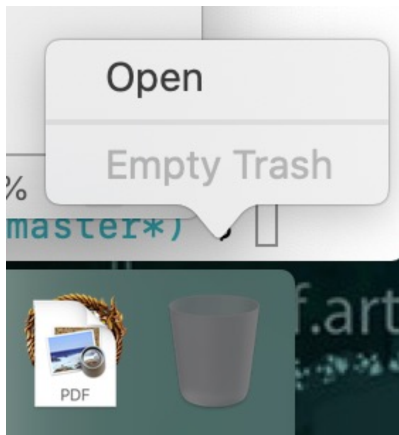
- Change to selection? Window resizing? Scrollbar positioning?

Some actions are destructive and not easily undone

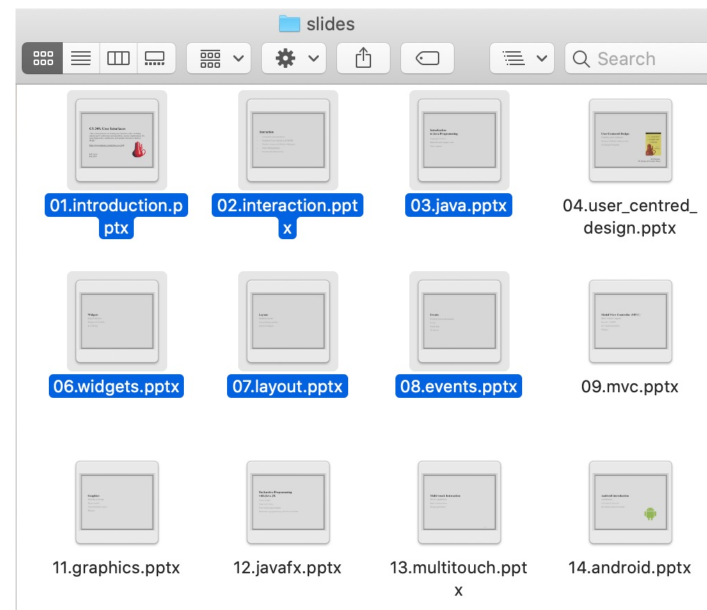
- Quitting program with unsaved data, emptying trash

Some actions cannot be undone

- Printing



Users typically cannot undo emptying the trash.



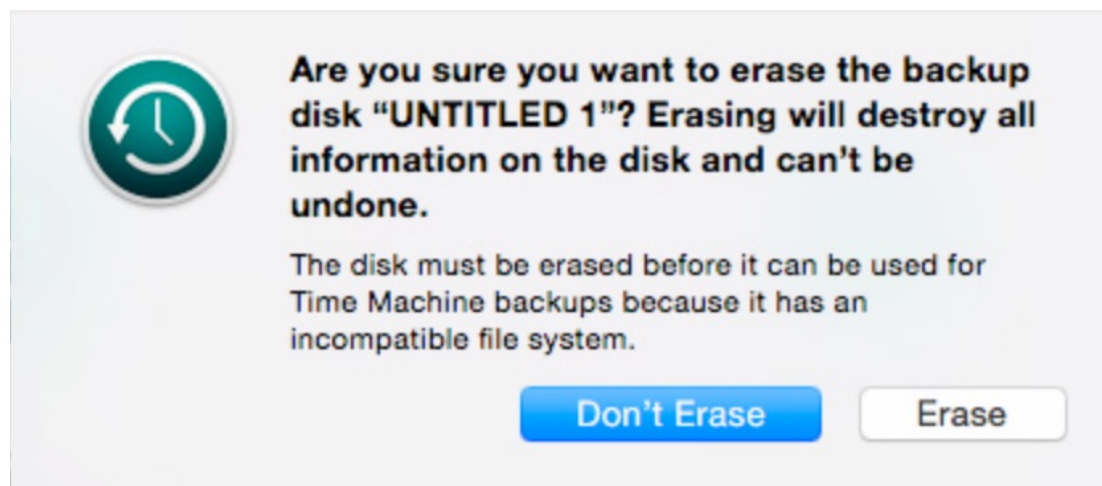
Can you undo selection of files in a window?

Undoable Actions

All changes to document (i.e., the Model) should be undoable.

Changes to the View(i.e., the document's interface) should be undoable only if they are tedious or require significant effort. Typically view changes are not undoable.

Ask for confirmation before doing a destructive action which cannot easily be undone.



State Restoration

What is the user interface state after an undo or redo?

- e.g., highlight text, delete, undo: is text highlighted?
- e.g., select file icon, delete, undo: is file icon selected?

User interface state should be meaningful after undo/redo action

- Change selection to object(s) changed as a result of undo/redo
- Scroll to show selection, if necessary
- Give focus to the control that is hosting the changed state

These provide additional undo feedback.

Granularity

Undo one chunk: the conceptual change from one document state to another.

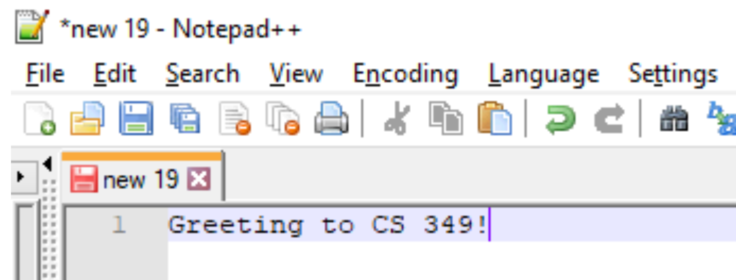
Granularity – Text

Undo one chunk:

- One typed character?
- One word? (Next problem: what is a word?)
- One time-interval? (Next problem: how long should that be?)
- How to deal with auto-correct?

  In Word: typed auto-correct undo

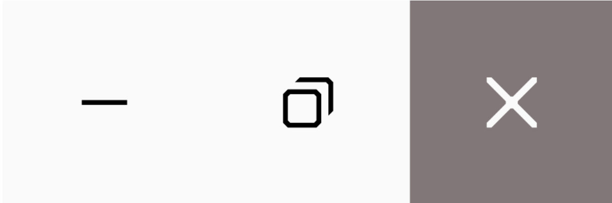
  This.·is·¶ This.·Is·¶ This.·is·¶



Granularity – Drawing

Undo one chunk:

- Per Pixel?
- Between strokes?
- Between `MousePress` and `MouseRelease`?
- How to deal with window switching?



Undo Patterns

U CS 349

Undo Patterns

A ***change record*** defines a single transformation to our model (state). Multiple operations will result in successive change records being applied. We have 2 main ways to apply *change records*:

Forward Undo:

- save complete baseline document state at some past point: S
- save change records to transform baseline document into current document state: $c\left(b\left(a(S)\right)\right)$
- to undo last action, build up state from the baseline to the current -1 change record. do not apply last change record: $S' = \mathrm{undo}\left(c\left(b\left(a(S)\right)\right)\right) = b\left(a(S)\right)$

Reverse Undo:

- save complete current document state: S
- save reverse change records to return to previous state: $\{c^{-1}, b^{-1}, a^{-1}\}$
- to undo last action, apply last reverse change records: $S' = \mathrm{undo}(S) = c^{-1}(S)$

Undo Patterns

Using either of these options requires two stacks:

- **Undo stack**: save all change records as you perform actions in the application – *limited to actions that are undoable*.
- **Redo stack**: move change records here after they have been undone (so you can reapply them if needed).

How do you support undo-redo with multiple windows?

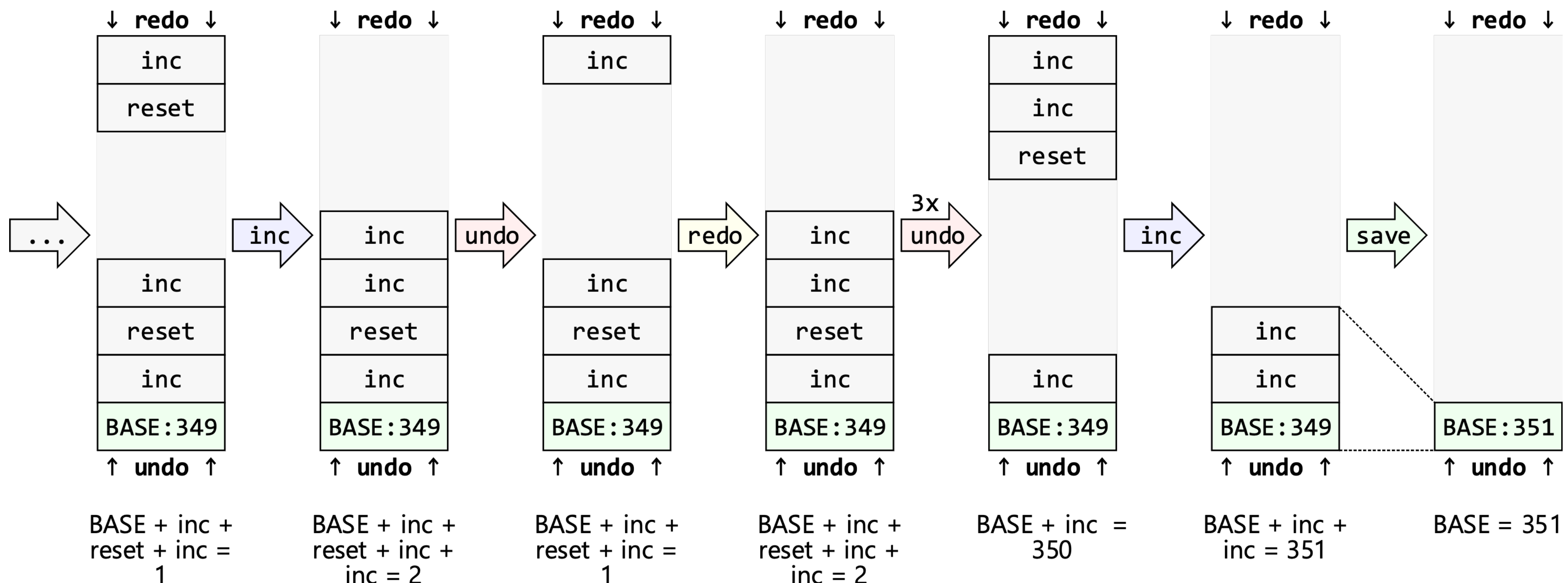
- Treat all windows as a single model, with a single set of actions that can be undone. If you do this, you need to switch focus when you undo/redo any actions.
- Alternatively, you can also choose to have multiple undo-redo stacks, one for each window, and use whichever has focus.

Performing any undoable action will typically *clear the redo stack.*

- *Why?*

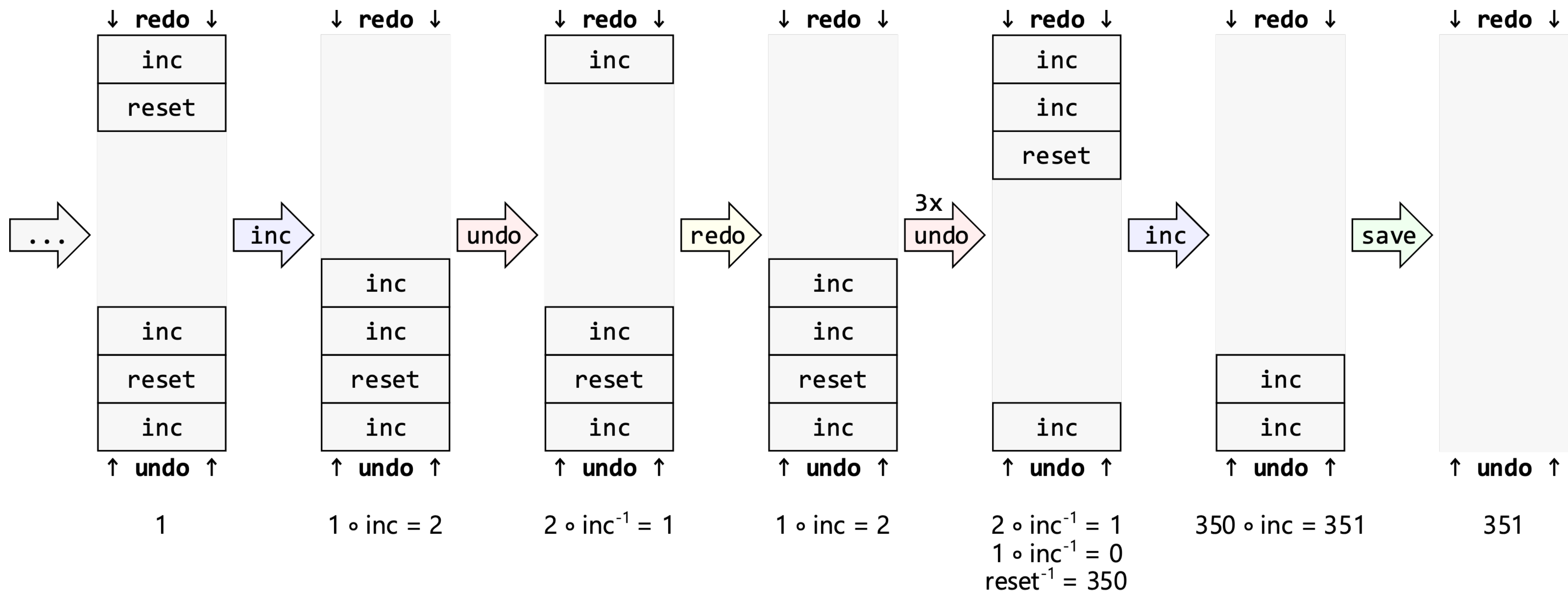
Undo Stacks

Forward Undo: calculate the current state from a base value and the "Do" commands on the undo-stack. **Undo** moves commands from the undo to the redo stack and forces re-calculation. **Save** (i.e., creating a memento) consolidates the undo and redo stacks.



Undo Stacks

Reverse Undo: calculating the current value based on the previous value combined with the “Do”- or “Undo”-action of the latest command.



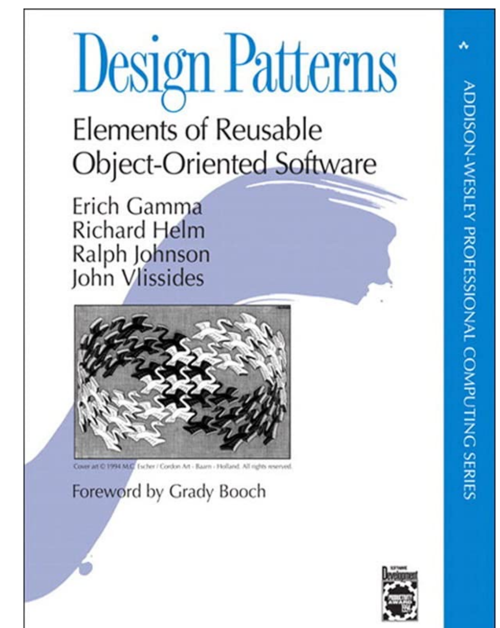
Implementing Undo

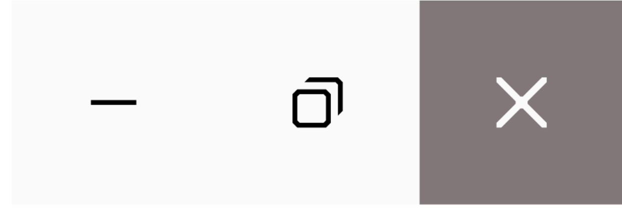
Forward Undo: GoF Command & Memento patterns

- Save one (or more) snapshots
- Apply “Do” commands until desired state has been reached

Backward Undo: GoF Command pattern

- Apply “Undo” commands until desired state has been reached
- Caveat: an action must be “undo-able”.





Implementation

Implementing Undo – Command Pattern

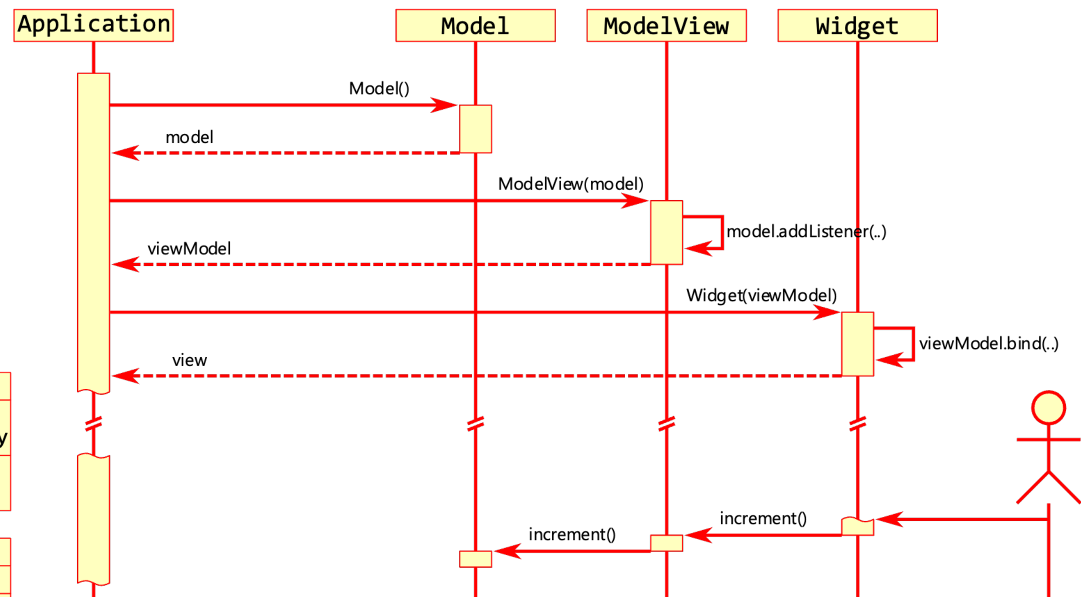
The command pattern uses an object to encapsulate all information needed to perform an action or trigger an event.

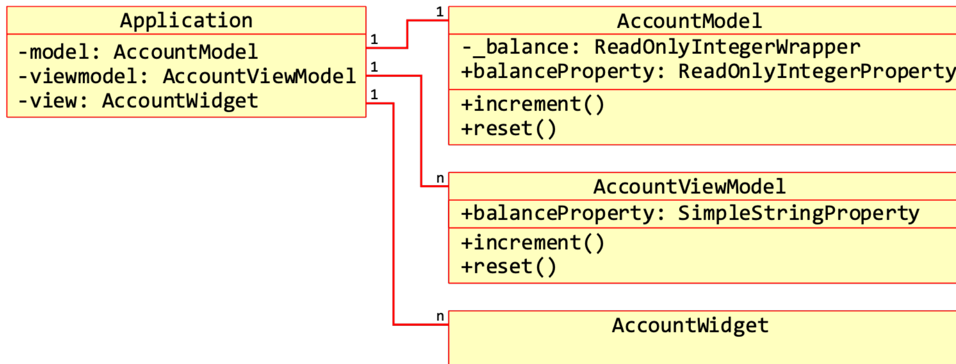
This information can include the business logic / function pointer to the business logic, the owner / target / receiver, and additional parameters.

```
/* An UndoableCommand interface typically defines methods
 * execute(): perform this command (first time, or to redo it)
 * undo(): go back to the previous state
 */

class IncrementCommand : UndoableCommand {

    override fun execute(value: Int) : Int {
        return value + 1
    }

    override fun undo(value: Int) : Int {
        return value - 1
    }
}
```

Undo Implementation

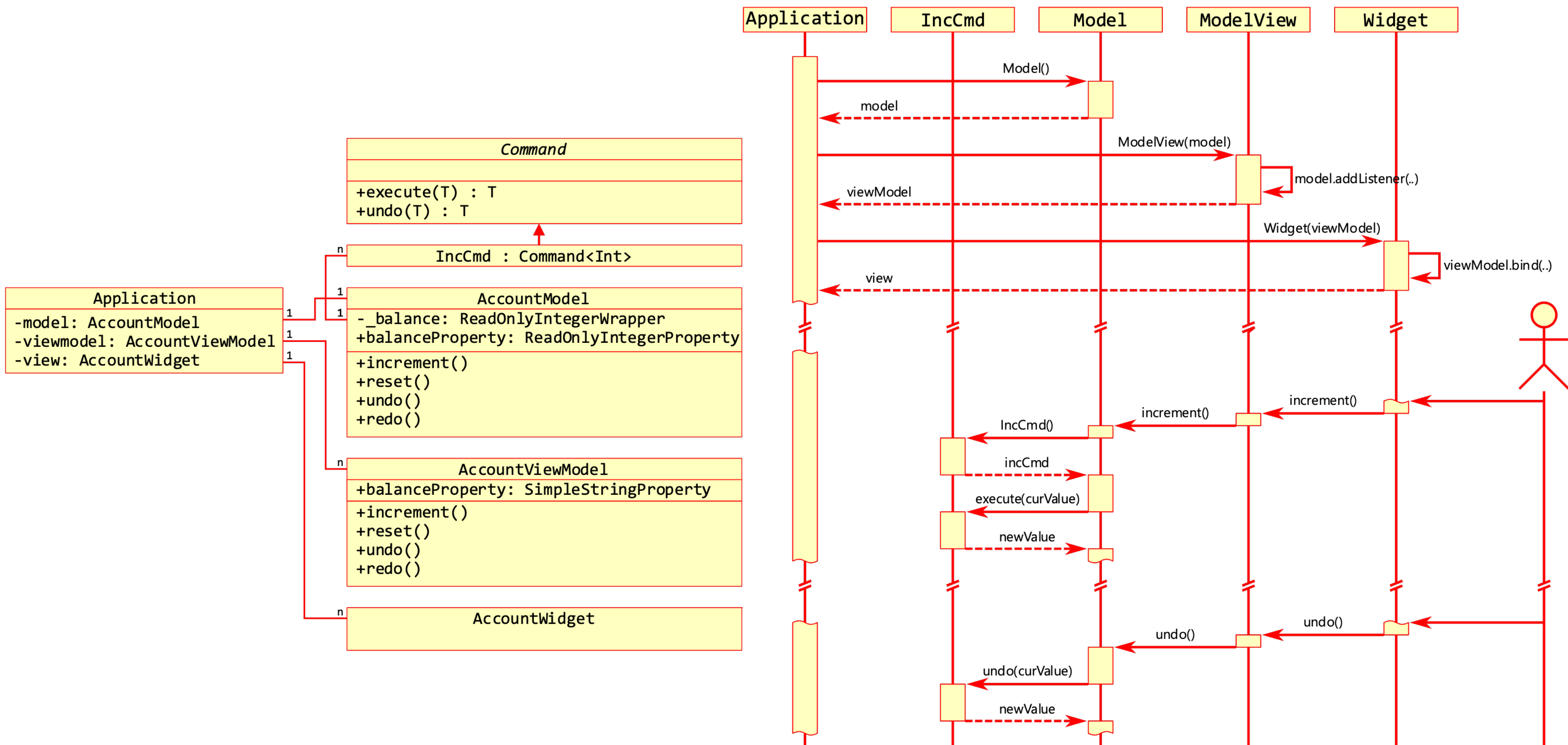
Application with MVVM, without undo / redo:

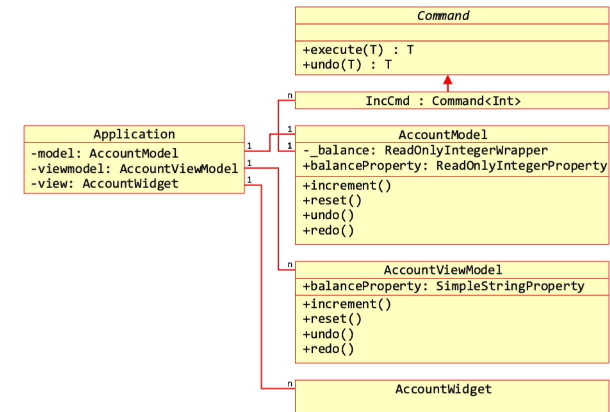




Undo Implementation

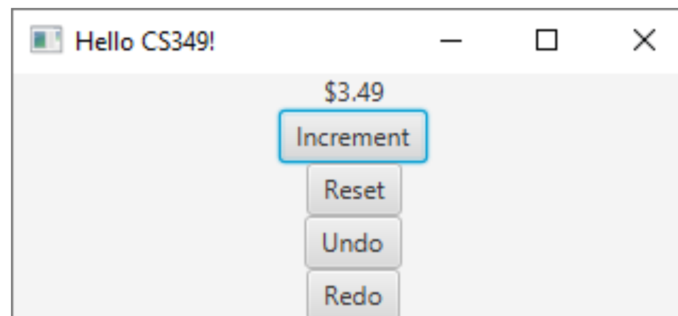
Application with MVVM, Command pattern and undo / redo:



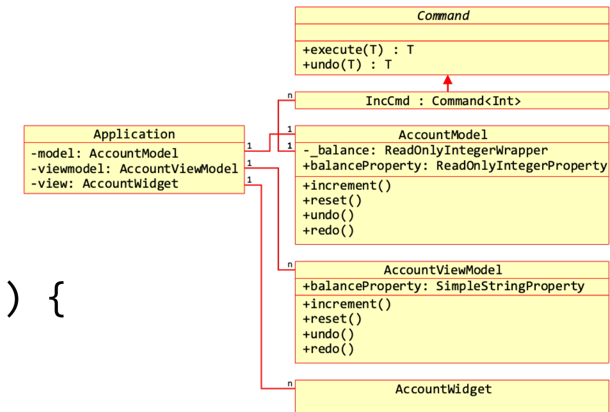


Undo Implementation – View

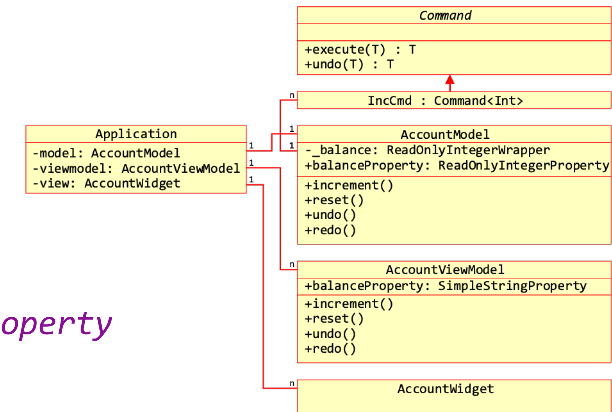
```
class BalanceWidget(viewModel : NumberViewModel): VBox() {
    init {
        children.addAll(
            Label(viewModel.numberProperty.value).apply {
                textProperty().bind(viewModel.numberProperty)
            },
            Button("Increment").apply {
                onAction = EventHandler { viewModel.increment() }
            },
            Button("Reset").apply {
                onAction = EventHandler { viewModel.reset() }
            },
            Button("Undo").apply {
                onAction = EventHandler { viewModel.undo() }
            },
            Button("Redo").apply {
                onAction = EventHandler { viewModel.redo() }
           })
        alignment = Pos.CENTER
    }
}
```



Undo Implementation – ViewModel

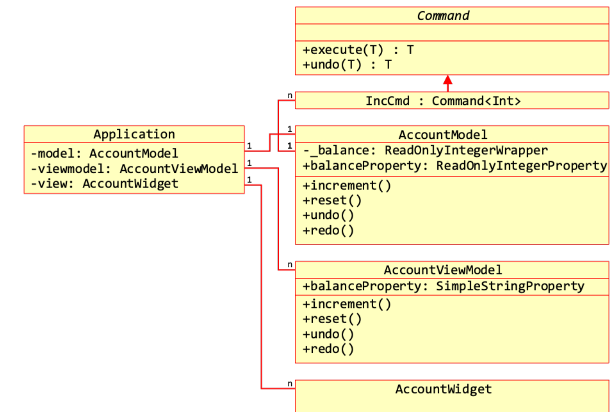



```
class BalanceViewModel(private val numberModel: BalanceModel) {

    val balanceProperty = SimpleStringProperty()

    init {
        balanceProperty.bind(Bindings.createStringBinding(
            { "$ ${model.balanceProperty.value / 100}." +
              "${model.balanceProperty.value % 100 / 10}" +
              "${model.balanceProperty.value % 10}" },
        model.balanceProperty)
    )}

    fun increment() { numberModel.increment() }
    fun reset() { numberModel.reset() }
    fun undo() { numberModel.undo() }
    fun redo() { numberModel.redo() }
}
```



Reverse Undo Implementation – Model

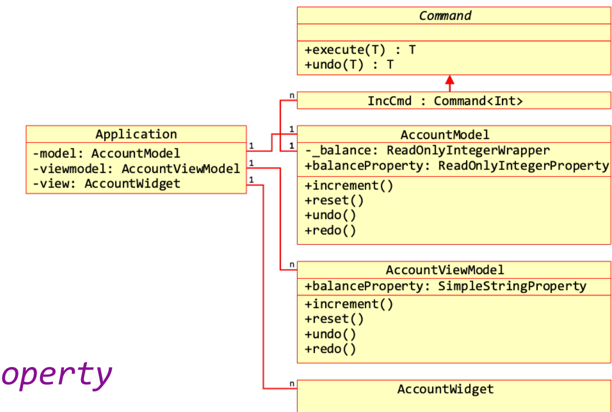
```kotlin
class AccountModel {
    private val _balance = ReadOnlyIntegerWrapper(0)
    val balanceProperty: ReadOnlyIntegerProperty = _balance.readOnlyProperty
    private val undoCommands = mutableListOf<Any>()
    private val redoCommands = mutableListOf<Any>()

    fun increment () {
        IncrementCommand().apply {
            undoCommands.add(this)
            _balance.value = execute(_balance.value)
        }
        redoCommands.clear()
    }
    fun reset() {
        ResetCommand().apply {
            undoCommands.add(this)
             _balance.value = execute(_balance.value)
        }
        redoCommands.clear()
    }
    fun undo() {
        undoCommands.removeLastOrNull()?.apply {
            redoCommands.add(this)
            _balance.value = (this as UndoableCommand<Int>).undo(_balance.value)
        }
    }
    fun redo() {
        redoCommands.removeLastOrNull()?.apply {
            undoCommands.add(this)
            _balance.value = (this as UndoableCommand<Int>).execute(_balance.value)
        }
    }
}
```



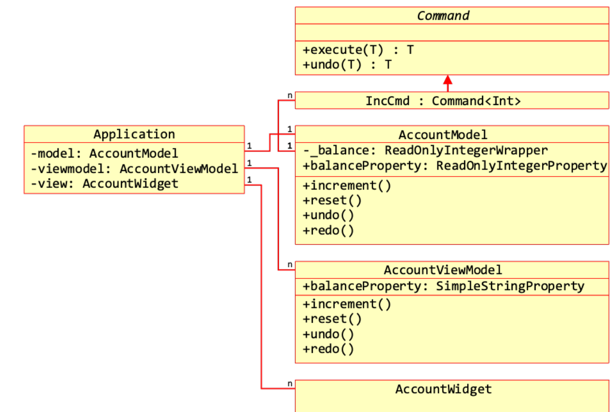
Reverse Undo Implementation – Model

```
interface UndoableCommand<T> {
    fun execute(value: T) : T
    fun undo(value: T) : T
}

class IncrementCommand : UndoableCommand<Int> {
    override fun execute(value: Int) : Int {
        return value + 1
    }
    override fun undo(value: Int) : Int {
        return  value - 1
    }
}

class ResetCommand() : UndoableCommand<Int> {
    private var oldVal = 0
    override fun execute(value: Int) : Int {
        oldVal = value
        return 0
    }
    override fun undo(value: Int) : Int {
        return  oldVal
    }
}
```

Forward Undo Implementation – Model




```
class AccountModel {
    private val _balance = ReadOnlyIntegerWrapper(0)
    val balanceProperty: ReadOnlyIntegerProperty = _balance.readOnlyProperty

    private val undoCommands = mutableListOf<Any>(Memento(_balance.value)) // baseline
    private val redoCommands = mutableListOf<Any>()

    fun increment () { ... }
    fun reset () { ... }

    fun undo() {
        if (undoCommands.size > 1) {
            undoCommands.removeLast().apply {
                redoCommands.add(this)
                _balance.value = undoCommands.fold((undoCommands[0] as Memento).execute(0)) {
                    acc, cur -> (cur as Command<Int>).execute(acc) // accumulate after baseline
                }
            }
        }
    }
    fun redo() { ... }
}
```



Forward Undo Implementation – Model

```
interface Command<T> {
    fun execute(value: T) : T
}

class IncrementCommand : Command<Int> {
    override fun execute(value: Int) : Int {
        return value + 1
    }
}

class ResetCommand() : Command<Int> {
    override fun execute(value: Int) : Int {
        oldVal = value
        return 0
    }
}

class Memento(private val savedState: Int) : Command<Int> {
    override fun execute(value: Int): Int {
        return savedState
    }
}
```

Example: Text Editor Undo/Redo Commands

Available Commands:

- `insert(string, start)`
- `delete(start, end)`
- `style(start, end, FontWeight.NORMAL | FontWeight.BOLD)`

Change	Command	Result
<start>	`insert("Quick brown", 0)`	Quick brown
<command>	`style(6, 10, FontWeight.BOLD)`	Quick **brown**
<command>	`insert (" fox", 11)`	Quick **brown** fox
<undo>	`delete(11, 14)`	Quick **brown**
<undo>	`style(6, 10, FontWeight.NORMAL)`	Quick brown
<redo>	`style(6, 10, FontWeight.BOLD)`	Quick **brown**
<command>	`insert(" dog", 11)`	Quick **brown** dog

Reverse Undo Command Problems

Consider a bitmap paint application

- stroke(points, thickness, colour)
- erase(points, thickness)

<start>

<command>

```
stroke(points, 10, black)
```

<undo>

```
erase(points, 10, black)
```

Solutions for “Destructive” Commands

Use forward command undo.

Use backward command undo for “non-destructive” commands, and mementos for destructive commands.

End of the Chapter

Any further questions?