Components, interfaces, reentrance (Chapter 5)

Prof. Dr. Wolfgang Pree

Department of Computer Science
University of Salzburg
cs.uni-salzburg.at

UNIVERSITÄT SALZBURG

Components and interfaces (1)

- Semantics - actual meaning of things beyond their outer form
- Component technologies - interoperation of alien components
- Interfaces
  - means by which components connect
  - set of named operations that can be ivoked by clients
- The specification of the interface - mediating middle for two parties to work together

Components and interfaces (2)

- Interfaces
  - directly provided by a component - procedural interfaces of traditional libraries
  - provided indirectly by objects - Object interfaces
- There is a dualism between specifying components and specifying the interactions between components.
- Interfaces appear as the roles at the endpoints of an interaction specification.

Direct and indirect interfaces (1)

- Direct (procedural) and indirect (object) interfaces can be unified - use static objects that can be part of a component
- Object interface - dynamic method lookup
- The Object implementing an interface belongs to a component different from the component with which the client seems to interact through the interface
- Procedural interfaces are always direct
- Explicit procedural indirection is possible by means of procedure variables (also called function pointers).

Direct and indirect interfaces (2)

- Effective coupling of two dynamically selected parties via a well defined interface - late binding: right at the heart of object-oriented programming
- In a component system proper working of the resulting dynamic confituration is hard to control.
- The quality of the interface specification holds things together.

Versions (1)

- Versions of components can be prolific.
- Traditional version management assumes versions of a component evolve at a single source.
- Subtle aspect - Moving from direct to indirect interfaces.
- Direct interfaces check versions at bind time

- Versioned system - avoid indirect coupling of parties of incompatible versions

- Unless versions are checked in every place where an object reference crosses components boundaries, version checking is not sound.

Versions (2)

- Few component infrastructures address the component versioning problem properly.
- Goal - ensure that older and newer components are either compatible or clearly detected as incompatible.
- General strategy - support a sliding window of supported versions.
- Another possible approach - immutable interface specifications. Each interface, once published, is frozen.

  Supporting multiple versions = supporting multiple interfaces.
- More refined approach
  - Mutation of definitions from one version to the next.
  - Define precisely what changes are valid to retain backwards compatibility.
  - Multiple versions of a component can be loaded and instantiated in the same space.

Interfaces as contracts

- Interface specifications is contracts between a client of an interface and a provider of an implementation of the interface.
- Contract
  - what the client needs to do to use the interface
  - specifying pre- and postconditions for the operation
- Client
  - establishes the precondition before calling the operation
  - relies on the postcondition being met whenever the call to the operation returns
- Provider
  - relies on the preconditions being met whenever the operation is called.
  - has to establish the postcondition before returning to the client.
- Implementation can weaken preconditions or strengthen postconditions.

Contracts and extra-functional requirements

- Contemporary contracts often exclude precise performance requirements.
- Changing performance can break clients.
- Contractual specification
  - functional aspects (interface syntax and semantics with invariants, and pre- and postconditions)
  - service level - availability, mean time between failures, mean time to repair, throughput, latency, data safety for persistent state, capacity …
- Failure to fulfill the service level = Wrong result.

- The practice of including extra-functional specifications into a contract and monitoring them strictly will be used more in the future.

Undocumented „features“

- It is always possible to „observe“ behavior of an implementation beyond its specification.
- A way of exploring an implementation’s unspecified „features“ is called debugging.
- In practice, it is very difficult to avoid non-contractual dependencies.
- A contract needs to maintain a balance between being precise and not being to restrictive.

What belongs to a contract?

- Contracts as interface with pre- and postconditions and, perhaps, invariants are simple and practical.
- However, simple pre- and postconditions on operations merely specify partial correctness.
- The requirement that an operation should also eventually terminate leads to total correctness => default convention for all contracts.

Safety and progress

- The concept that something guaranteed by a precondition will lead to something guaranteed by a postcondition can be generalized.
- Common notation is the „leads-to“ operator – ex. the clause „model update leads-to notifier calls“
- Progress conditions often rely on some form of temporal logic.
- The notion of contracts can be formalized to capture safety and progress conditions, although still neglecting performace and resource consumption.
- Pre- and postconditions are widely accepted and usually combined with informal clauses to form complete contracts.
- The drawback of such semiformal approaches is the exclusion of formal verification.

Extra-functional requirements

- The specification techniques (conditions, contracts, histories, specification statements) are restricted to functional aspects.
- Violation of extra-functional requirements can break clients.
- Unless performance is regulated by contracts, it can be difficult or impossible to pinpoint the underperforming components.
- The provider also has to respect other resource limitations.

Specifying time and space requirements

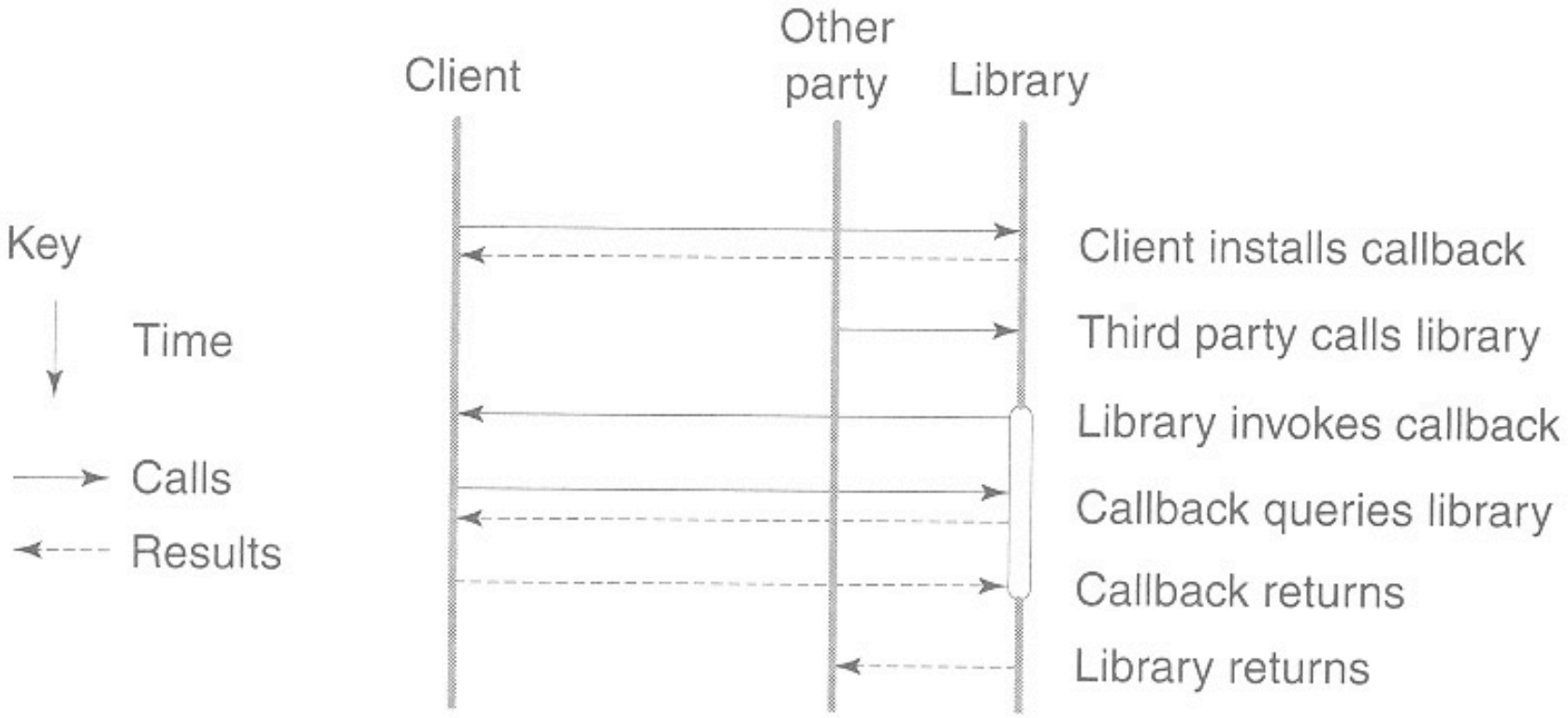
- Ideally, a contract should cover all essential functional and extra-functional aspects. It is not yet clear how this can best be achieved, leaving room for ongoing future research.

  Ex.: The time complexity of legal implementations is bounded.
- It is possible to determine absolute time and space bounds for a given provider on a given platform.
- The gap between complexity bounds in a contract and absolute bounds (in seconds or bytes) on a specific platform needs to be bridged.
- Automated composer, could compute absolute bounds for a given platform.
- The specification of worst-case bounds may not be the best choice.
  Ex.: Quicksort vs Heapsort

Dress code – formal or informal?

- Interface contracts should be as formal as possible to enable verification.
- In practice the formal specifications are rarely used because of their complexity.
- Pre- and postconditions – a semiformal style.
- None of the real-world laws are formal.
- Formalizing contracts, where possible and agreeable, is a good idea.
- Attempting to formalize everything can easily lead to totally unapproachable, and therefore unsolvable, situations.

Callbacks and contracts



UNIVERSITÄT SALZBURG

Callbacks and contracts

- Callback
  - procedure that is passed to a library at one point
  - registered with the library
  - reverses the direction of the flow control
- Callback invocations = up-calls
- In a strict procedural library model always the client calls the library.
- Library has to establish a “valid” state before invoking any callbacks.
- A callback may cause the library’s state to change through direct or indirect calls to the library.
- Validity of the library state is specified as part of a contract.

Examples of callbacks and contracts

- Callbacks introduce subtle dependencies that lead to unexpected complexities and error-prone situations.
- Common practice to demonstrate pre- and postconditions by using effectively flat abstract data types (ADTs) examples.

  Ex.: A directory service with two versions of a client
- Major points are missed when using trivial examples

A directory service

- Directory service as part of a simple file system supports callbacks to notify clients of changes in the managed directory.
- Directory service interface is grouped into two parts
  - file lookup and addition or removal of named files
  - registration and unregistration of callbacks

A directory service

```
DEFINITION Directory;
  IMPORT Files;   (* details of no importance for this example *)
  TYPE
    Name = ARRAY OF CHAR;
    Notifier = PROCEDURE (IN name: Name);   (* callback *)
  PROCEDURE ThisFile (n: Name): Files.File;
    (* pre  n ≠ "" *)
    (* post  result = file named n  or  (result = NIL and no such file) *)
  PROCEDURE AddEntry (n: Name; f: Files.File);
    (* pre  n ≠ "" and f ≠ NIL *)
    (* post  ThisFile(n) = f *)
  PROCEDURE RemoveEntry (n: Name);
    (* pre  n ≠ "" *)
    (* post  ThisFile(n) = NIL *)
  PROCEDURE RegisterNotifier (n: Notifier);
    (* pre  n ≠ NIL *)
    (* post  n registered, will be called on AddEntry and RemoveEntry *)
  PROCEDURE UnregisterNotifier (n: Notifier);
    (* pre  n ≠ NIL *)
    (* post  n unregistered, will no longer be called *)
END Directory.
```

A client of the directory service

- Simple client that uses directory callbacks to maintain visual display of the directory's content.

- A callback recursively invokes operations on the calling service.
- Several points left unclear in the directory contract:
  - is the notifier called at all, called at once, or called twice?
  - to call the notifier before or after the directory itself has been updated?

A client of the directory service

```
MODULE DirectoryDisplay;   (* most details deleted *)
  IMPORT Directory;
  PROCEDURE Notifier (IN n: Directory.Name);
  BEGIN
    IF Directory.ThisFile(n) = NIL THEN
      (* entry under name n has been removed – delete n in display *)
    ELSE
      (* entry has been added under name n – include n in display *)
    END
  END Notifier;
BEGIN
  Directory.RegisterNotifier(Notifier)
END DirectoryDisplay.
```

Same client, next release

- Variation of DirectoryDisplay that uses a pseudo name “Untitled” for anonymous files that have not been registered with the directory, but have been announced to the display service.

  Assumption: no registered file will ever be called “Untitled”.

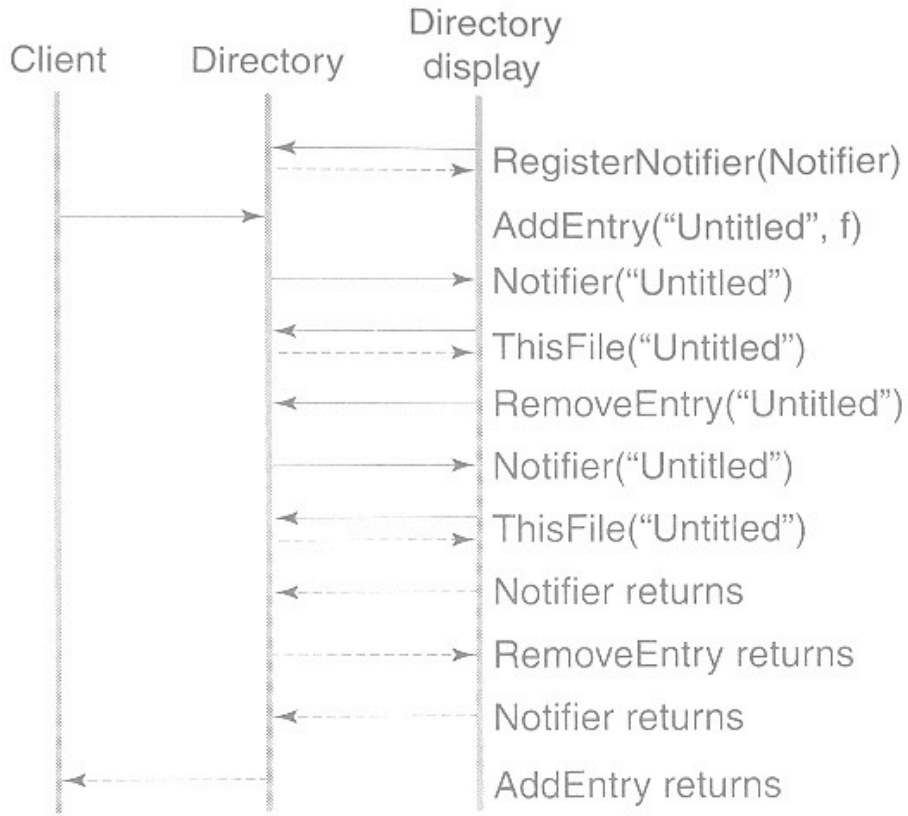
Same client, next release

```
MODULE DirectoryDisplay;   (* Version 1.0.1 *)
  IMPORT Directory;
  PROCEDURE Notifier (IN n: Directory.Name);
  BEGIN
    IF Directory.ThisFile(n) = NIL THEN
      (* entry under name n has been removed – delete n in display *)
    ELSE
      IF n = "Untitled" THEN
        (* oops – you shouldn't do that ...*)
        Directory.RemoveEntry(n)   (* ... gotcha! *)
      ELSE
        (* entry has been added under name n – include n in display *)
      END
    END
  END Notifier;
BEGIN
  Directory.Register(Notifier)
END DirectoryDisplay.
```

```
MODULE Directory;
  IMPORT Files, Strings;
  TYPE
    Name* = ARRAY OF CHAR;   (* trailing * exports definition *)
    Notifier* = PROCEDURE (IN name: Name);
    EntryList = POINTER TO RECORD   (* linked list of entries *)
      next: EntryList;
      name: POINTER TO Name; file: Files.File
        (* an entry is a pair of a name and a file *)
    END;
    NotifierList = POINTER TO RECORD   (* linked list of notifiers *)
      next: NotifierList;
      notify: Notifier
    END;
  VAR
    entries: EntryList;   (* list of registered entries *)
    notifiers: NotifierList;   (* list of registered notifiers *)
  PROCEDURE AddEntry* (n: Name; f: Files.File);
    VAR e: EntryList; u: NotifierList;
    BEGIN
      ASSERT(n # ""); ASSERT(f # NIL);
      e := entries;   (* search for entry under name n *)
      WHILE (e # NIL) & (e.name # n) DO e := e.next END;
      IF e = NIL THEN   (* not found: prepend new entry *)
        NEW(e); e.name := Strings.NewFromString(n);   (* create new entry *)
        e.next := entries; entries := e   (* prepend new entry to list *)
      END;
      e.file := f;   (* fill in file – replaces old binding if entry already existed *)
      u := notifiers;   (* invoke all registered notifiers *)
      WHILE u # NIL DO u.notify(n); u := u.next END
    END AddEntry;
    (* other procedures not shown *)
END Directory.
```

Same client, next release

- Unlike a contract, an implementation answers all questions about the what, when, and where – assuming the semantics of the programming language are known.
- It is not appropriate to rely only on implementation details when designing or implementing a new client.



Same client, next release

- Result: Stack overflow during a test insertion of an entry “Untitled”.

A broken contract

PROCEDURE AddEntry(IN n: Name; f: Files.File);

(pre n≠”” and f≠NIL *)*

(post ThisFile(n) = f *)*

- The postcondition ThisFile (“Untitled”) = someFile does not hold on return from AddEntry. A client relying on this postcondition would break.
- The notifier is “out of loop”.

Prevention is better than cure

- Histories of possible state - specify the behavior of asynchronous systems.
- Resulting contracts - far less manageable than simple pre- and postconditions.
- A notified observer must not change the notifying observed object - transitive nature restriction.
- A notifier may invoke any other operation that internally, for whatever reason, uses the service that originally notified.
- Middle ground - equip a library using callbacks with state test functions.

Proofing the directory service

- Addition of a test function InNotifier and its proper use in the preconditions of AddEntry and RemoveEntry would solve the problem.

```
DEFINITION Directory;   (* refined *)
  ...   (* unaffected parts omitted *)
  TYPE
    Notifier = PROCEDURE (IN name: Name);   (* callback *)
      (* pre  InNotifier() *)
  PROCEDURE InNotifier (): BOOLEAN;
    (* pre  true *)
    (* post  a notifier call is in progress *)
  PROCEDURE AddEntry (IN n: Name; f: Files.File);
    (* pre  (not InNotifier()) and n ≠ "" and f ≠ NIL *)
    (* post  ThisFile(n) = f *)
  PROCEDURE RemoveEntry (IN n: Name);
    (* pre  not InNotifier() *)
    (* post  ThisFile(n) = NIL *)
END Directory.
```

Proofing the directory service

- Test functions are a mixed blessing. It would be preferable to resort to a more declarative form in the contract, rather than relying on an executable function that inspects state.
- Test functions can be invoked by any client of a service and thus solve the problem of transitive restrictions.
- As the precondition can be checked at runtime, clients can be implemented to behave correctly.

Test functions in action

- Test functions in interfaces are not commonplace today.
- Test functions are nevertheless used sometimes.

  Ex. - Java security manager - an object that protects critical services from being called by untrusted code.
- The indirect recursion across abstractions caused by callbacks in a good basis from which to understand the similar difficulties introduced by webs of interacting objects.

From callbacks to objects

- Objects references introduce linkage across arbitrary abstraction domains.
- Proper layering of system architectures lies somewhere between the challenging and the impossible certainly not as natural as with procedural abstractions.
- Object reference can be used at one time in a layer above that of the referenced object and at another time in a layer below that of the referenced object.
- With object reference every method invocation is potentially an up-call, every method potentially a callback.

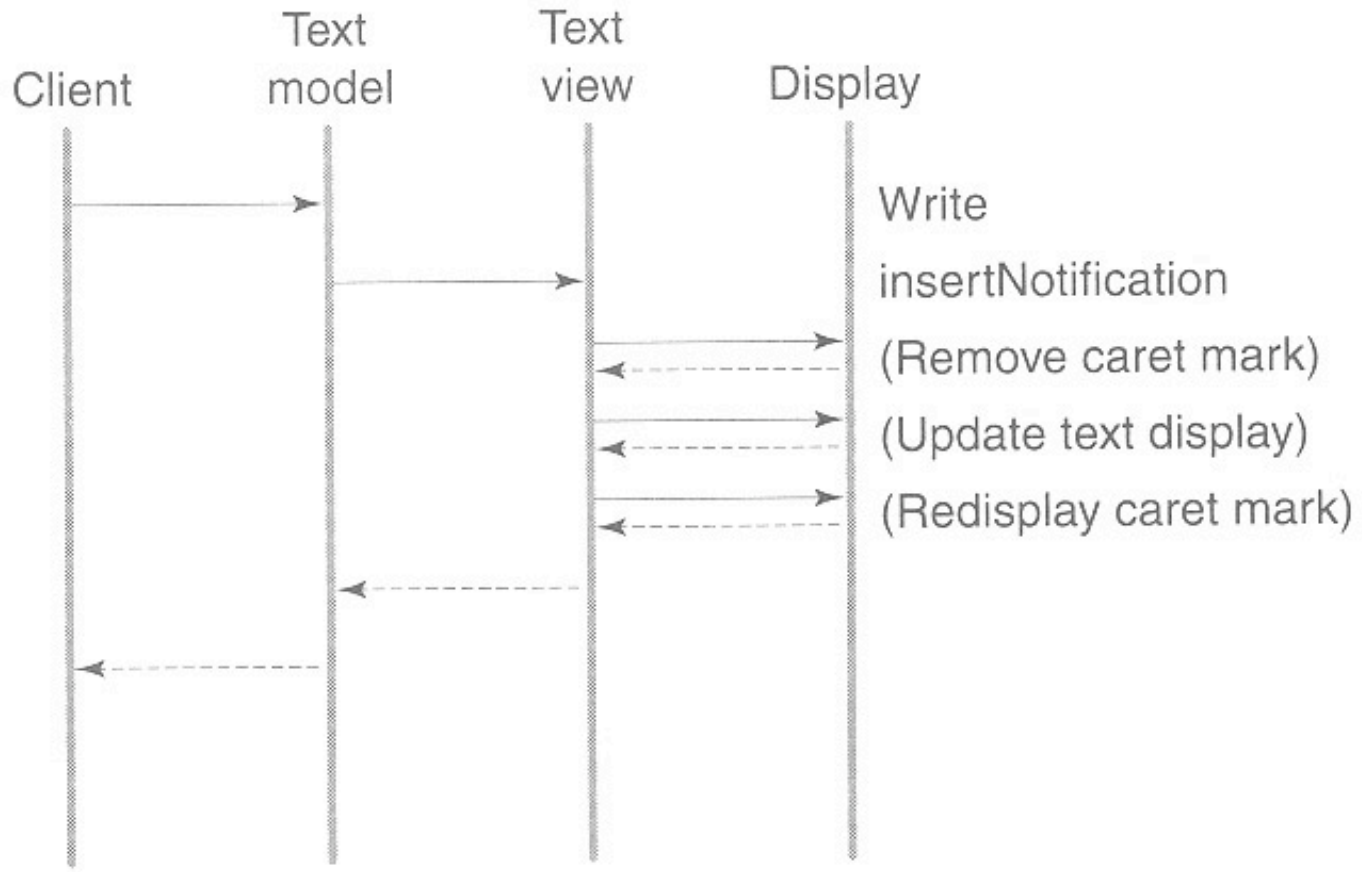
```
interface TextModel {
  int max ();
    // pre  true
    // post  result = maximum length this text instance can have
  int length ();
    // pre  true
    // post  0 ≤ result ≤ this.max()  and  result = length of text
  char read (int pos);
    // pre  0 ≤ pos < this.length()
    // post  result = character at position pos
  void write (int pos, char ch);
    // [ len: int; txt: array of char •
    //   pre   len := this.length(); (all i: 0 ≤ i < len: txt[i] := this.read(i)) :
    //           len < this.max()  and  0 ≤ pos ≤ len
    //   post  this.length() = len + 1
    //     and  (all i: 0 ≤ i < pos: this.read(i) = txt[i])
    //     and  this.read(pos) = ch
    //     and  (all i: pos < i < this.length(): this.read(i) = txt[i - 1])
    // ]
  void delete (int pos);
    // [ len: int; txt: array of char •
    //   pre   len := this.length(); (all i: 0 ≤ i < len: txt[i] := this.read(i)) :
    //           0 ≤ pos < len
    //   post  this.length() = len - 1
    //     and  (all i: 0 ≤ i < pos: this.read(i) = txt[i])
    //     and  (all i: pos ≤ i < this.length(): this.read(i) = txt[i + 1])
    // ]
  void register (TextObserver x);
  void unregister (TextObserver x);
}
```

From callbacks to objects

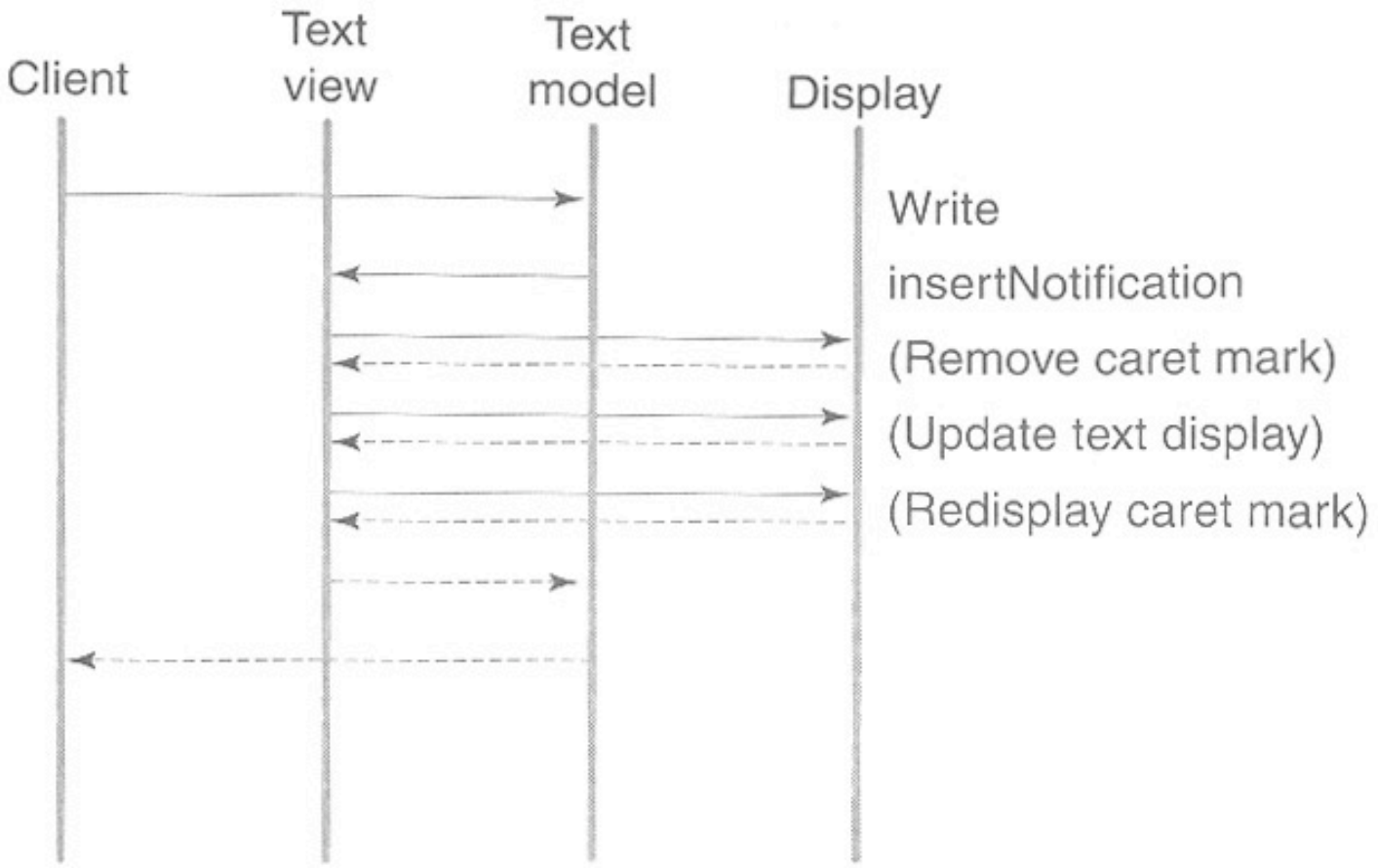
```
interface TextObserver {
  void insertNotification (int pos);
    // pre  character at position pos has just been inserted
  void deleteNotification (int pos);
    // pre  character that was at position pos has been deleted
}
```

```
interface TextView extends TextObserver {
  TextModel text ();
    // pre  true
    // post  result ≠ null
  int caretPos ();
    // pre  true
    // post  0 ≤ result ≤ this.text().length()
  void setCaret (int pos);
    // pre  0 ≤ pos ≤ this.text().length()
    // post  this.caretPos() = pos
  int posToXCoord (int pos);
    // pre  0 ≤ pos ≤ this.text().length()
    // post  result = x-coordinate corresponding to text position pos
  int posToYCoord (int pos);
    // pre  0 ≤ pos ≤ this.text().length()
    // post  result = y-coordinate corresponding to text position pos
  int posFromCoord (int x, int y);
    // pre  (x, y) is valid screen coordinate
    // post  this.posToXCoord(result) = x and this.posToYCoord(result) = y
  void type (char ch);
    // [ caret: int •
    //   pre  caret := this.CaretPos() : this.text().length() < this.text().max()
    //   equiv  this.text().write(caret, ch)
    //   post  this.caretPos() = caret + 1
    // ]
  void rubout ();
    // [ caret: int •
    //   pre  caret := this.CaretPos() : caret > 0
    //   equiv  this.text().delete(caret - 1)
    //   post  this.caretPos() = caret - 1
    // ]
  void insertNotification (int pos);
    //   inherited from TextObserver
    //   repeated here for strengthened postcondition
    // pre  character at position pos has just been inserted
    // post  display updated  and  this.caretPos() = pos + 1
  void deleteNotification (int pos);
    //   inherited from TextObserver
    //   repeated here for strengthened postcondition
    // pre  character that was at position pos has been deleted
    // post  display updated  and  this.caretPos() = pos
```

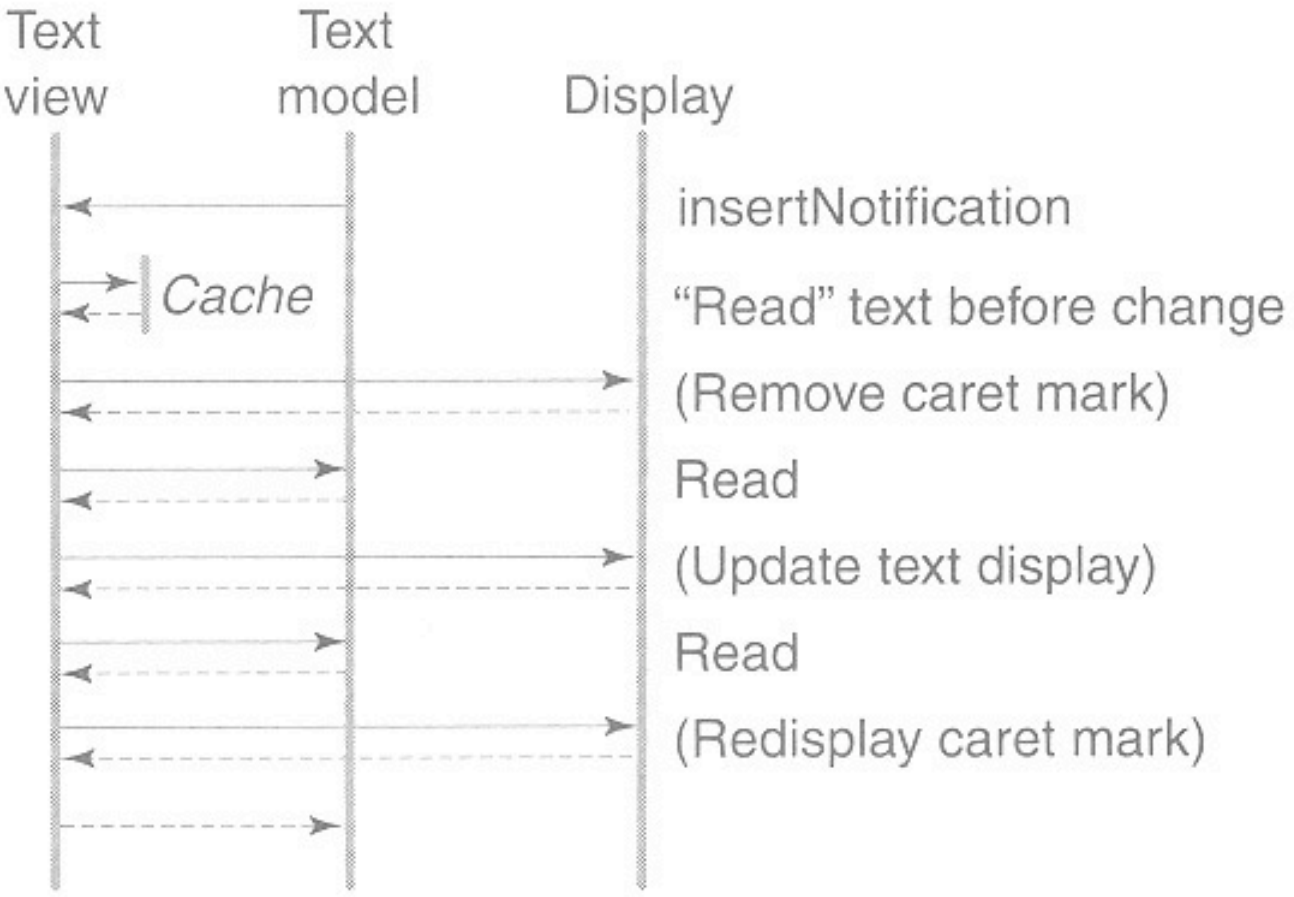
From callbacks to objects



From callbacks to objects



From callbacks to objects



From callbacks to objects

- The flow of control against the layers of abstraction (up-call) clearly exposes inconsistencies to arbitrary other objects – a circumstance that should be reflected in the interfaces of text models and views but usually is not.

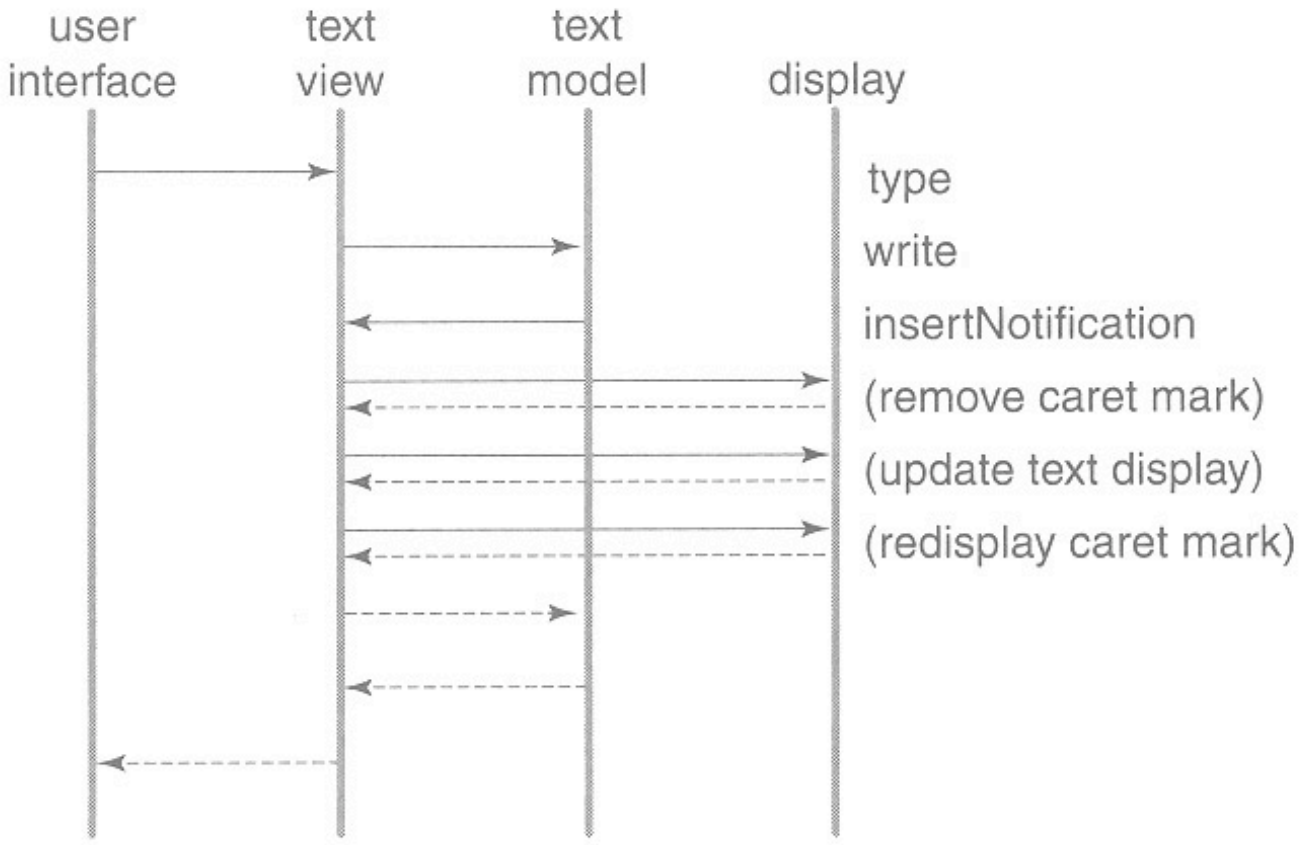
From interobject consistency to object re-entrance

- Model view scenario shows how multiple objects can be subject to consistency constraints. Objects arranged in strict layers, with messages being sent only from objects located in higher layers to objects located in lower layers.
- For object systems strict layering is rarely performed. Passing object references “down” and abstractly dealing with upper objects in lower layers one of the most powerful aspects of object orientation. Harnessing this potential is a major challenge.
- The real problem observation of an object undergoing a state transition with inconsistent intermediate states becoming visible. Inconsistencies can only be observed by entering an object’s method.
- If the intermediate state is observed by means of re-entrance, maintaining correctness becomes difficult.

FIGURE 5.7

- Object reentrance poses difficult problems even in a relatively simple example.

From interobject consistency to object re-entrance



- Object reentrance poses difficult problems even in a relatively simple example.

Self-interference and object re-entrance: a summary (1)

- Re-entrance of methods can pose problems, as objects normally have state and are not automatically re-entrant.
- Simple solution - require all invariants of an object to be established before calling any method. All state needs to be made consistent before calling a method.
- Invariants covering multiple objects in a cycle cannot always be maintained. The observation of intermediate state can be difficult to prevent.
- One way to address re-entrance problems - weaken invariants conditionally and make the conditions available to clients by means of test functions.
- On receipt of a notification that the model has changed, a view cannot rely on the model to compute further.
- It either has to cache all required information or the marks have to be removed before a model can be changed.

Self-interference and object re-entrance: a summary (2)

- A significant number of design and implementation errors go back to unexpected recursive re-entrance of objects. The recursion leading to such re-entrances is obscured either by subtle interactions of classes in an inheritance hierarchy or subtle interactions of objects.
- Contracts based on pre- and postconditions can capture the conditions to allow for safe interactions, even in the presence of recursive re-entrance.
- Dealing with recursion and re-entrance is difficult in situations where the recursion is explicit and part of the design. Self-recursion within a single object can be affected by class inheritance and leads to recursive re-entrance patterns that are neither explicitly specified nor necessarily expected.
- Recursion and re-entrance become a more pressing problem when crossing the boundaries of components. Each component must be independently verifiable based on the contractual specifications of the interface requires and those it provides.

Processes and multithreading

- The problems of recursive re-entrance of objects and concurrent interaction of processes are similar. The idea can be taken to the extreme, by assigning full process semantics to every object.
- “Actors”
  - go further and turn every object invocation into a separate process!
  - doing so is not efficient. Objects and processes are kept separate and processes are populated by multiple threads.
- Understanding and properly addressing the issues of re-entrance in object systems does not become any simpler by introducing processes or threads.

Histories

- Another approach to capturing the legal interactions among objects is the specification of permissible histories.
- Valid state transitions can be specified by restricting the set of permissible traces. The specification is a formally captured set of permissible traces.
- In most history-based specification techniques, permissible traces or histories are specified indirectly.
- An object is substitutable for another if all traces of the new object, projected to the states and operations of the old object, are explicable in terms of the old object.
- If traces were used to specify an interface, then any class implementing this interface would have to satisfy the substitutability criterion.
- Coordination language
  - describe the composition processes, yielding composite components,
  - describe connections between processes external to these processes similar to connection-oriented programming and architecture description languages.

Specification statements

- Specification statements – similar to that of statement-oriented imperative languages. It is possible to insert specification statements into a sequence of normal statements.
- Specification statements state as much about an implementation as needs to be stated to capture the specification.

```
class TextModel;
  var text: seq char; observers: set Observer; max: N •
  procedure Max (result m: N) =
    m: [true, m ≥ max]; max := m;
  procedure Length (result l: N) =
    l := #text;
  procedure Read (pos: N; result ch: char) =
    ch := text[pos];
  procedure Write (pos: N; ch: char) =
    {max > #text} ;
    text := text[0 ... pos - 1] + <ch> + text[pos ... #text - 1];
    for o ∈ observers do o.InsertNotification(pos) od;
  procedure Delete (pos: N) =
    text := text[0...pos - 1] + text[pos + 1...#text - 1];
    for o ∈ observers do o.DeleteNotification(pos) od;
  procedure Register (obs: Observer) =
    observers := observers + {obs} ;
  procedure Unregister (obs: Observer) =
    observers := observers - {obs} ;
    initially text = <>; observers = { } ; max = 0
  end
class Observer;
  var text: TextModel •
  procedure InsertNotification (pos: N) =
    keep text^ ;
  procedure DeleteNotification (pos: N) =
    keep text^
end
invariant
  o: Observer; t: TextModel • o ∈ t.observers ⇔ o.text = t
```

Specification statements

- The specification of notifiers prohibits the notifier from modifying its notifying text.