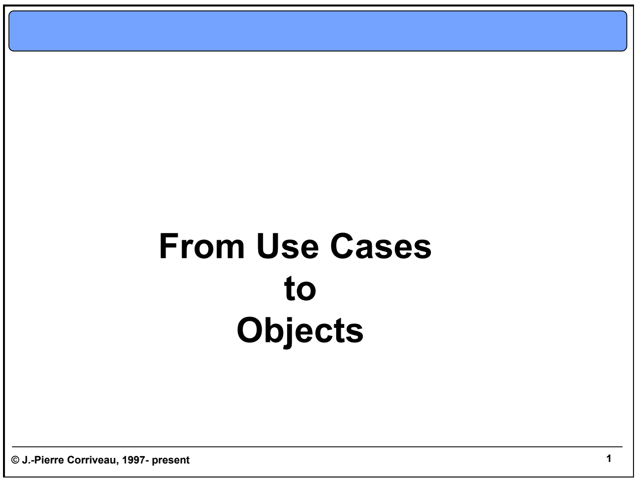
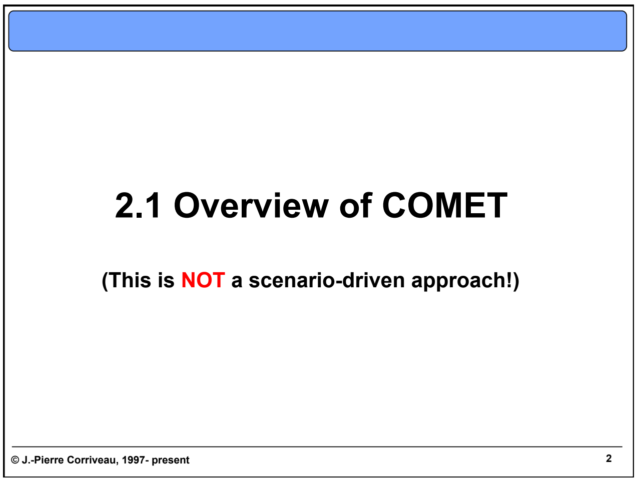
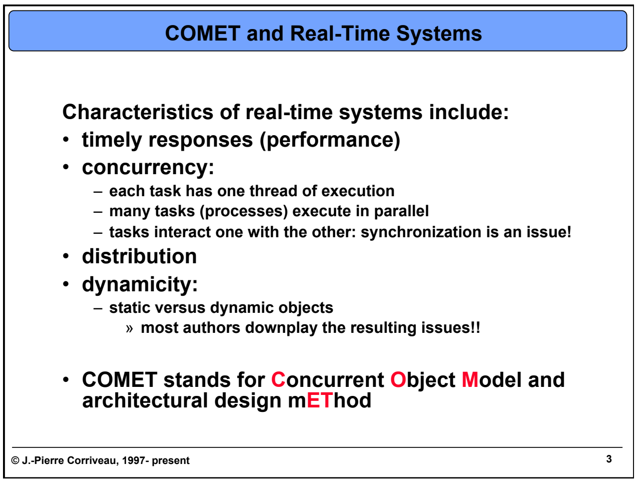


From Use Cases to Objects



2.1 Overview of COMET

(This is NOT a scenario-driven approach!)

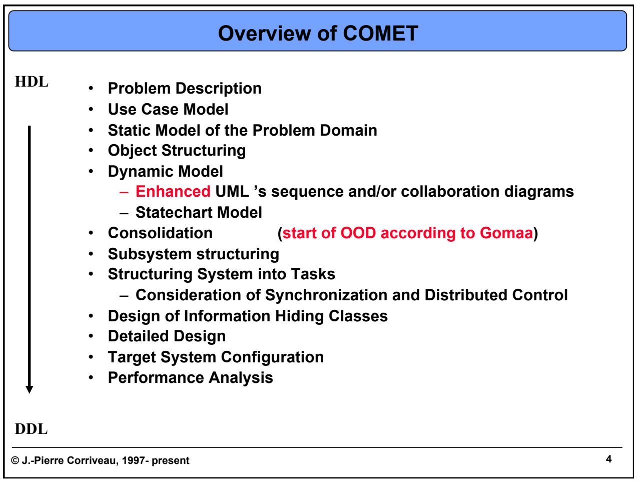


COMET and Real-Time Systems

Characteristics of real-time systems include:

- timely responses (performance)
- concurrency:
  - each task has one thread of execution
  - many tasks (processes) execute in parallel
  - tasks interact one with the other: synchronization is an issue!
- distribution
- dynamicity:
  - static versus dynamic objects
    - most authors downplay the resulting issues!!

- COMET stands for Concurrent Object Model and architectural design mETthod



Overview of COMET

HDL

- Problem Description
- Use Case Model
- Static Model of the Problem Domain
- Object Structuring
- Dynamic Model
  - Enhanced UML 's sequence and/or collaboration diagrams
  - Statechart Model
- Consolidation (start of OOD according to Gomaa)
- Subsystem structuring
- Structuring System into Tasks
  - Consideration of Synchronization and Distributed Control
- Design of Information Hiding Classes
- Detailed Design
- Target System Configuration
- Performance Analysis

DDL

OOA in COMET

- **Requirements Model:**
  - use cases
- **OOA models:**
  - static model of problem domain
  - object structuring:
    - » classes and their relationships
  - statecharts only for state dependent objects
    - » as opposed for all objects!
  - *Embellished* UML interaction diagram(s) for each use case

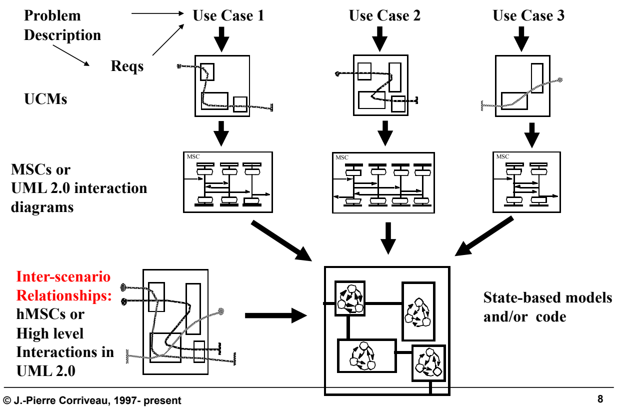
First Steps for the ATM

- Figure 19.1: use-case diagram
- Figure 19.2: conceptual static model
- Figure 19.3: context class diagram
- Figure 19.4: entity classes
- Figures 19.5, 19.6 and 19.7: class attributes (!!!)

- Bottom line: This is not a scenario-driven approach!!
  - While the use cases are used, OOD is first and foremost driven by the ‘magically’ chosen objects… Gomaa’s elevator case study is famous for this (going from 18.4 to 18.5…)

2.2 Requirements Engineering With Use Case Maps (ITU Z.151: URN-FR)

A Scenario-Driven Modeling Approach



Problem Description

Use Case 1

Use Case 2

Use Case 3

Reqs

UCMs

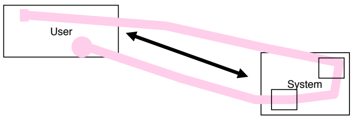
MSCs or UML 2.0 interaction diagrams

Inter-scenario Relationships: hMSCs or High level Interactions in UML 2.0

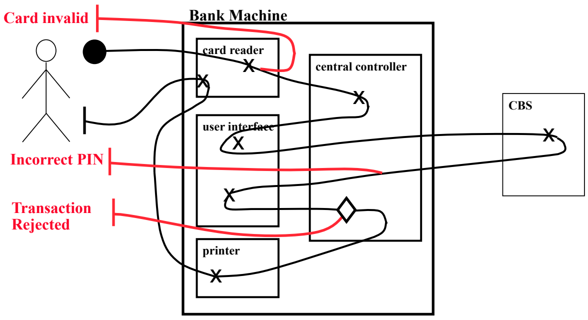
State-based models and/or code

Use-Case Maps

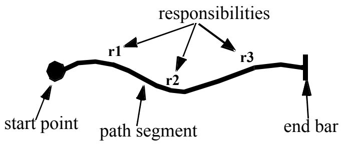
"Use-case maps trace the global dynamic flow of causality through the components of the system, that result from each use case." (Buhr and Casselman, 1996)



Bank Machine UCM

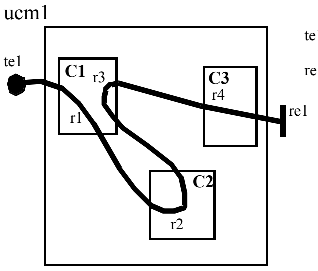


A UCM Path Segment



Formally a start point is defined by a pre-condition (if any) and a set of possible triggering events.

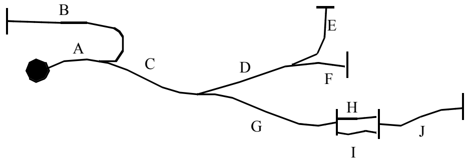
A Bound UCM Example



te1: triggering event for path P1
(precondition(te1): System is in the state S1)
re1: resulting event for path P1
(postcondition(re1): System is in the state S2)

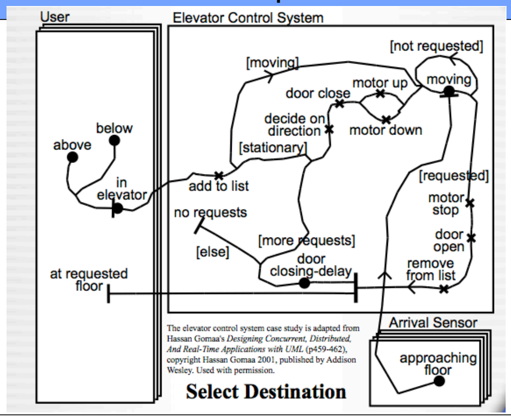
Superimposition of Scenarios

There can be several concurrent terminating paths in a UCM.



Each use case is to have a semantically equivalent UCM.

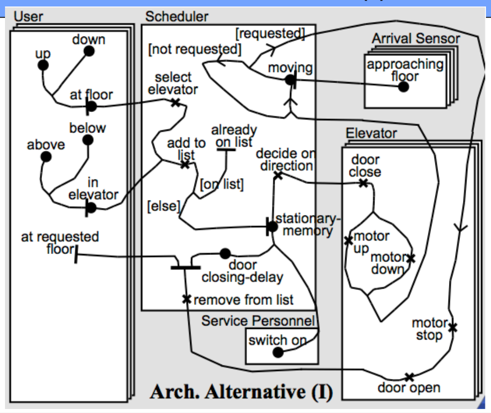
Example



The elevator control system case study is adapted from Hassan Gomaa's *Designing Concurrent, Distributed, And Real-Time Applications with UML* (p459-462), copyright Hassan Gomaa 2001, published by Addison Wesley. Used with permission.

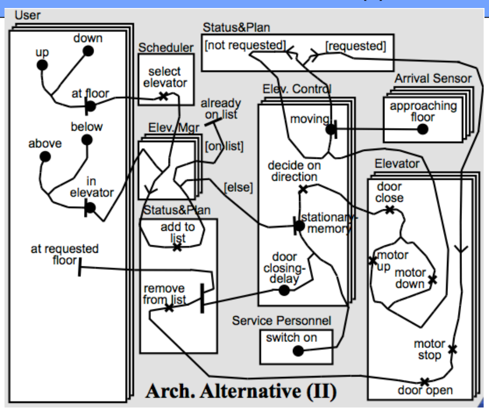
© D. Amyot

Choice of Architecture (1)



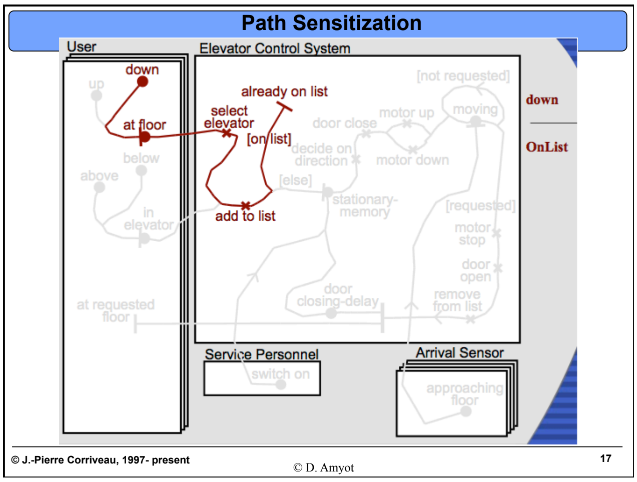
© D. Amyot

Choice of Architecture (2)

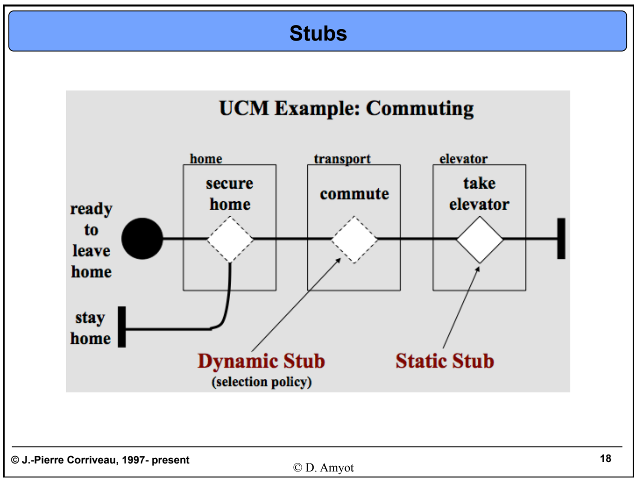


© D. Amyot

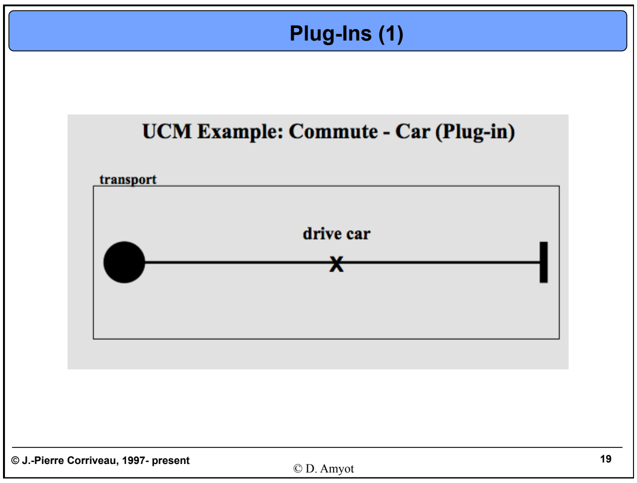
Path Sensitization



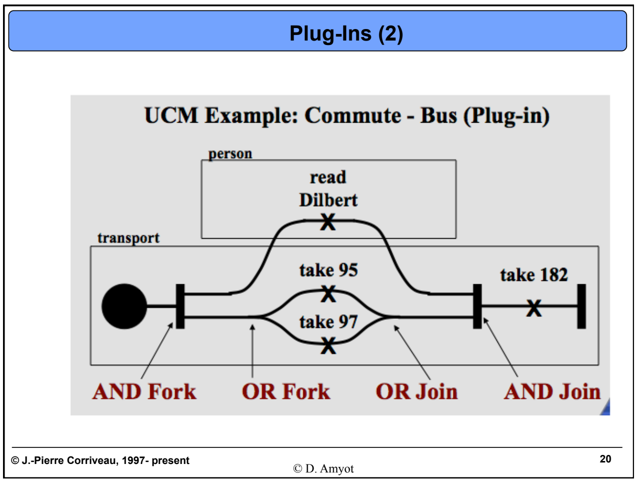
Stubs


UCM Example: Commuting

Plug-Ins (1)


UCM Example: Commute - Car (Plug-in)

Plug-Ins (2)


UCM Example: Commute - Bus (Plug-in)

The Bottom Line

- UCMs are typically useful for obtaining and/or verifying the ***responsibilities*** of objects:
  - expressing use-cases as paths of responsibilities helps tremendously in enforcing traceability between requirements and the more detailed sequence diagrams:
    - » UC -> UCMs -> sequence diagrams
  - knowing which responsibilities of an object participate in which scenarios helps with concurrency analysis, scheduling, and regression testing.
  - a UCM documents the relationships between different path segments. So inter-scenario relationships should be captured in the UCM associated with each use case.
  - the information of the use case diagram must not be forgotten! It gives the overall map for inter-UC processing.
- A public domain Eclipse plugin exists for UCM drawing:
  - See www.usecasemaps.org

About the Examples

We want to try to avoid the magic found in Gomaa!

- Poker:
  - Notice the discussion of design decisions in these documents but also the absence of UCMs!
- Alarm System:
  - Older document reorganized to have uUCMs and bUCMs before class diagram (and CRCs) and then and only then MSCs
- 2 Groceries:
  - More recent examples

2.3 Packaging Responsibilities: The Watch Example Revisited

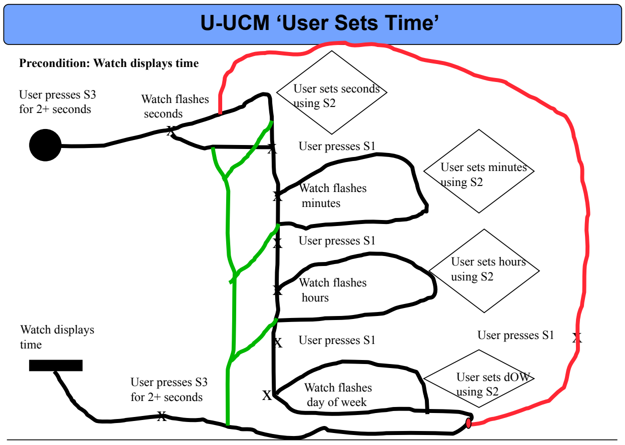
From Use Cases to UCMs

- From the requirements and use-cases, identify *all* the responsibilities of the system:
  - identify all inputs and outputs and infer all interface responsibilities
  - identify all the information that must be kept by the system
  - for each step of each use-case ask what the system needs to do to carry out that step (update data, interact with environment, etc.)
  - obtain a sequence of responsibilities for each scenario of each use-case (assuming a UC is written as an event-processing grammar…)
    - » UCMs are designed to capture this information!
  - verify the consistency and completeness of the responsibilities with respect to requirements and use cases

System Responsibilities

- From requirements and/or use-cases:
  - store:
    - » seconds, minutes, hours, am/pm, day-of-week, ticks
    - » day, month, year?
    - » current display, current mode (setting/displaying)
  - update:
    - » seconds, minutes, hours, am/pm, day-of-week, ticks
    - » day, month
    - » current display, current mode (setting/displaying)
      - do we really need both variables???
  - interaction:
    - » detect pressing/releasing S1, S2, S3
      - detect long S3 (no need for long S1 yet)
    - » display seconds, minutes, hours, am/pm, day-of-week, day, month
    - » flash any field of the watch
- Next step: getting unbound UCMs!

U-UCM 'User Sets Time'



From UCMs to Objects

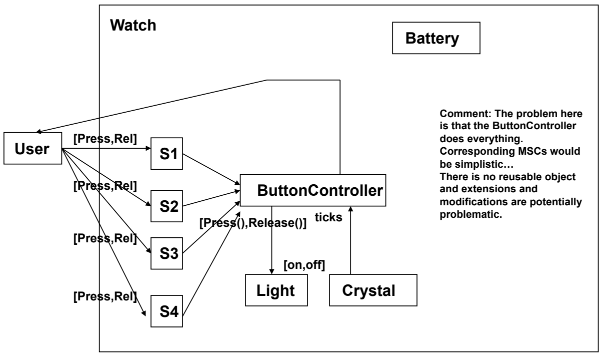
Package system responsibilities into classes:

- obey the heuristics of the next slide in order to package responsibilities into instances, of which you will infer the corresponding classes:
  - » don't prematurely turn responsibilities into operations!
- for each class, produce a CRC card (or something equivalent)
  - » Introduced shortly
- for each UC (and UCM), then develop a corresponding (set of) interaction diagram(s) and use these diagrams to scrutinize your choice of classes:
  - » All instances of a same class must behave consistently across all the diagrams in which they appear! This is crucial, especially for statechart design.
  - » Interaction diagrams in UML 2 will be our next topic.

Packaging Responsibilities

- If an overall architecture has been chosen, identify its components
- Review domain objects and try to assign responsibilities to them: some may turn out to merely be containers
  - <<Interface>> objects are to be kept separate from the rest of the system
  - <<Entity>> objects keep persistent data
  - <<Control>> objects process messages between other objects
    - » Two flavors: coordinators and state-dependent control
- In order to minimize the complexity of interactions and maximize decoupling, consider the use of <<coordinator>> objects.
  - But avoid 'god' objects, who set the state of other objects
- Try to avoid duplication of responsibilities over several classes
  - But turning an operation into an object is always controversial…
  - A display object regroups similar operations!
- Consideration of inheritance and of specific implementation details is almost unavoidably premature and detrimental
  - Unless you are selecting design patterns
- Group together the responsibilities that store/update the same data. Then identify the procedures coupled to such data.

Too Few Objects

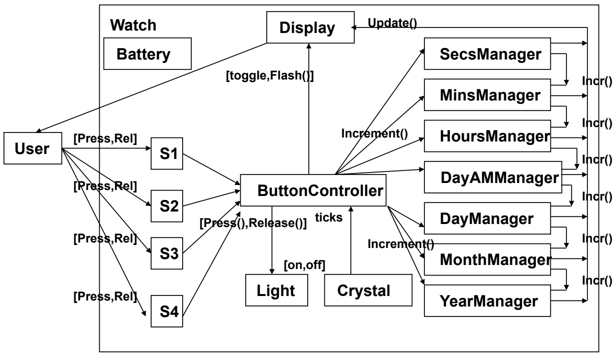

Watch
Battery
User
[Press,Rel]
S1
[Press,Rel]
S2
[Press,Rel]
S3
[Press,Rel]
S4
ButtonController
[Press(),Release()]
ticks
[on,off]
Light
Crystal

Comment: The problem here is that the ButtonController does everything. Corresponding MSCs would be simplistic… There is no reusable object and extensions and modifications are potentially problematic.

Too Few Objects

- Too-few objects typically entails ‘big’ objects with lots of responsibilities. This results in:
  - simplified patterns of interactions (as there are fewer objects) and, typically, better performance than with excessive messaging.
  - fewer referential attributes (again, because there are fewer objects)
  - poor *cohesion* within an object, which entails that:
    - » class synthesis is complicated as an object participates in most requirements and has a complex state (i.e., lots of data members)!
    - » reusability is greatly reduced as it is not sufficiently fine-grained.
    - » maintenance is complicated as within such an object, one change can have dramatic repercussions.
    - » overrides in subclasses may be more frequent.
  - trickier scheduling and development :
    - » multi-developers objects *are* problematic and greatly increase the risk of redundant procedures/data within a same object.
- Question: Would the watch as a display and a control system be acceptable?

Too Many Objects

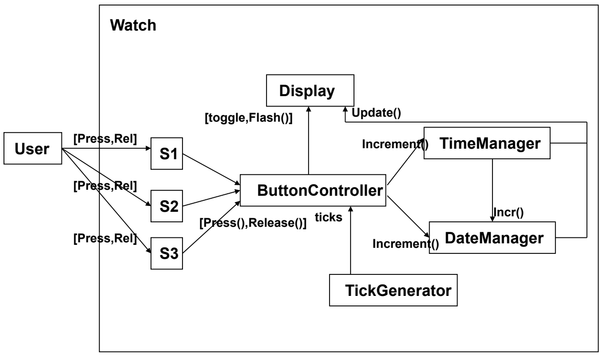

Watch
Display
Update()
Battery
[toggle,Flash()]
SecsManager
MinsManager
HoursManager
DayAMManager
DayManager
MonthManager
YearManager
Incr()
Incr()
Incr()
Incr()
Incr()
Incr()
Increment()
Increment()
User
[Press,Rel]
S1
[Press,Rel]
S2
[Press,Rel]
S3
[Press,Rel]
S4
ButtonController
[Press(),Release()]
ticks
[on,off]
Light
Crystal

N.B. We could collapse Display and ButtonController into the different managers.

Too Many Objects

- Small reusable objects are typically easier to modify and test per se than objects with lots of (possibly large) procedures.
- Too-many objects typically entails very specialized interfaces and complex interactions. This may lead to:
  - lots of coupling between these objects
  - performance problems:
    - » procedure calls and access *within* a class is generally cheaper than across objects.
    - » more time is required for creation, initialization and destruction of all these objects.
  - software evolution problems:
    - » the more objects, the likelier a change in the requirements will affect several!

A Better Object Model?



N.B. This is meant to be an instance, not a class, diagram!

A CRC Card

Class: Telephone

Responsibilities	Collaborators
• Acts as subscriber interface or agent	*customers:*
• Receives subscriber actions and translates them into signals for the network	Network *suppliers:* Dialer,
• Receives signal from the network and translates them to user audio signals	Hand-Set, Hook-Switch
• Acts as a transducer to send and receive voice and data	

Initial CRCs (1)

- Buttons S1 through S4:
  - Sending press(aButton) and release(aButton) messages to ButtonController
- ButtonController
  - Receiving messages from buttons
  - Measuring delay between press and release of S3
  - Storing/updating the current ‘mode’ and current display of the watch
  - Sending flash(anItem) and toggle messages to Display
  - Sending update(anItem) message to Time and Date managers
- Display
  - Receives flash(anItem) from ButtonController and flashes accordingly
  - Receives Toggle messages from ButtonController and changes display
  - Receives display(anItem) from Time and Date managers and updates accordingly
- TickGenerator
  - Sends a tick to ButtonController every 1/50th of a second

Initial CRCs (2)

- TimeManager
  - Storing count for seconds, minutes, hours, d-o-w, and am/pm
  - Updating these data members upon reception of an update message from the Button Controller
  - Requesting Display to update accordingly
- DateManager
  - Storing count for days and months, year
  - Updating these data members upon reception of an update message from the Button Controller or the Time manager
  - Requesting Display to update accordingly

Modeling Issues & Design Decisions

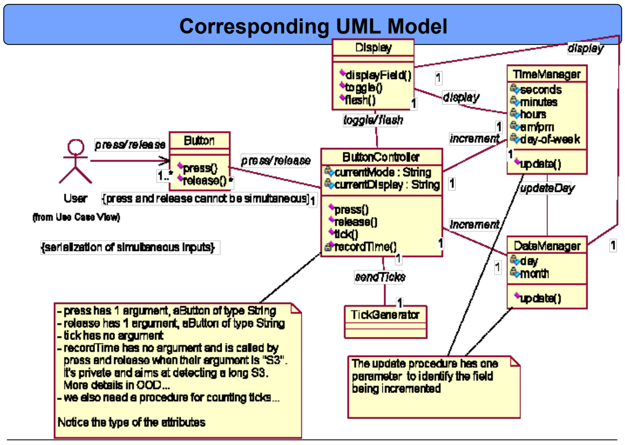
Modeling issues:

- Consistency of message names
  - update or increment, press/rel or press/release?
- Relevance of operation parameters and of attributes in UML model (see in next slide)
- Relevance of sending/receiving responsibilities in CRC cards
- Consistency between UML models and CRC cards

Design Decisions:

- Separation of the two managers
- Existence of ButtonController
- Existence of Display
- Consistency of messaging strategy
  - did we decouple as much as we could?
  - do we end up having a coordinator that does not coordinate….

Corresponding UML Model



Display
displayField()
toggle()
flash()

TimeManager
seconds
minutes
hours
am/pm
day-of-week
update()

DateManager
day
month
update()

Button
press()
release()

ButtonController
currentMode : String
currentDisplay : String
press()
release()
tick()
recordTime()

TickGenerator

User
(from Use Case View)

press/release
press/release
toggle/flash
display
display
increment
increment
updateDay
sendTicks
1..*
*
1
{press and release cannot be simultaneous}
{serialization of simultaneous inputs}

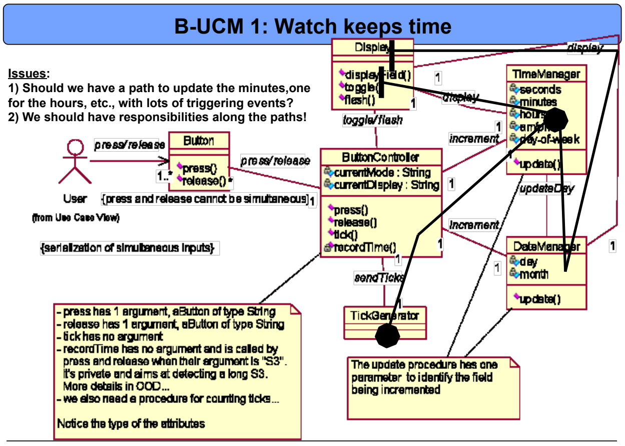
- press has 1 argument, aButton of type String
- release has 1 argument, aButton of type String
- tick has no argument
- recordTime has no argument and is called by press and release when their argument is "S3". It's private and aims at detecting a long S3. More details in OOD...
- we also need a procedure for counting ticks...

Notice the type of the attributes

The update procedure has one parameter to identify the field being incremented

B-UCM 1: Watch keeps time

Issues:
1) Should we have a path to update the minutes,one for the hours, etc., with lots of triggering events?
2) We should have responsibilities along the paths!



Display
displayField()
toggle()
flash()

TimeManager
seconds
minutes
hours
am/pm
day-of-week
update()

DateManager
day
month
update()

Button
press()
release()

ButtonController
currentMode : String
currentDisplay : String
press()
release()
tick()
recordTime()

TickGenerator

User
(from Use Case View)

press/release
press/release
toggle/flash
display
display
increment
increment
updateDay
sendTicks
1..*
*
1
{press and release cannot be simultaneous}
{serialization of simultaneous inputs}

- press has 1 argument, aButton of type String
- release has 1 argument, aButton of type String
- tick has no argument
- recordTime has no argument and is called by press and release when their argument is "S3". It's private and aims at detecting a long S3. More details in OOD...
- we also need a procedure for counting ticks...

Notice the type of the attributes

The update procedure has one parameter to identify the field being incremented