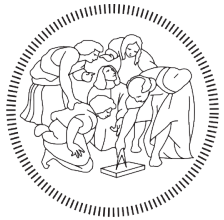


POLITECNICO
MILANO 1863

Exercise Lecture 5

Symbolic Execution: Exercise 1

POLITECNICO MILANO 1863

- Consider the function `foo`, written in a C-like language:

- Execute `foo` symbolically limiting the execution of the loop statement to exactly 2 iterations. Show, for each non-conditional statement:
  - <path condition, symbolic state>
- Define the pre-condition to the execution of `foo` s.t. the while loop is executed exactly twice
- Generate 3 possible test cases to run this path

```
0: int foo(int a, int b) {
1:   a++;
2:   while (a < b) {
3:     if (a != b)
4:       a++;
5:   }
6:   return a;
7: }
```

Symbolic Execution: Exercise 1

POLITECNICO MILANO 1863

- Limiting the execution of the loop to exactly 2 iterations.

```
0: int foo(int a, int b) {
1:   a++;
2:   while (a < b) {
3:     if (a != b)
4:       a++;
5:   }
6:   return a;
7: }
```

<0>

a	b	π
A	B	T

<0,1>

a	b	π
A+1	B	T

<0,1,2>

a	b	π
A+1	B	A+1 < B

<0,1,2,3>

a	b	π
A+1	B	A+1 < B
		A+1 ≠ B

<0,1,2,3,4>

a	b	π
A+2	B	A+1 < B

<0,1,2,3,4,2>

a	b	π
A+2	B	A+2 < B

<0,1,2,3,4,2,3>

a	b	π
A+2	B	A+2 < B
		A+2 ≠ B

<0,1,2,3,4,2,3,4>

a	b	π
A+3	B	A+2 < B

<0,1,2,3,4,2,3,4,2>

a	b	π
A+3	B	A+2 < B
		A+3 ≥ B

<0,1,2,3,4,2,3,4,2,6>

a	b	π
A+3	B	A+3 = B

Symbolic Execution: Exercise 1

POLITECNICO MILANO 1863

- Precondition to execute foo s.t. the loop is executed exactly 2 times

```
0: int foo(int a, int b) {
1:   a++;
2:   while (a < b) {
3:     if (a != b)
4:       a++;
5:   }
6:   return a;
7: }
```

<0,1,2,3,4,2,3,4,2,6>

a	b	π	SAT ✓
A+3	B	A+3 = B	

=> precondition: { b = a+ 3 }

- Three possible test cases
  - {a = 1, b = 4}, {a = 0, b = 3}, {a = -3, b = 0}

Symbolic Execution: Exercise 2

POLITECNICO MILANO 1863

- Consider the following function, written in a C-like language:

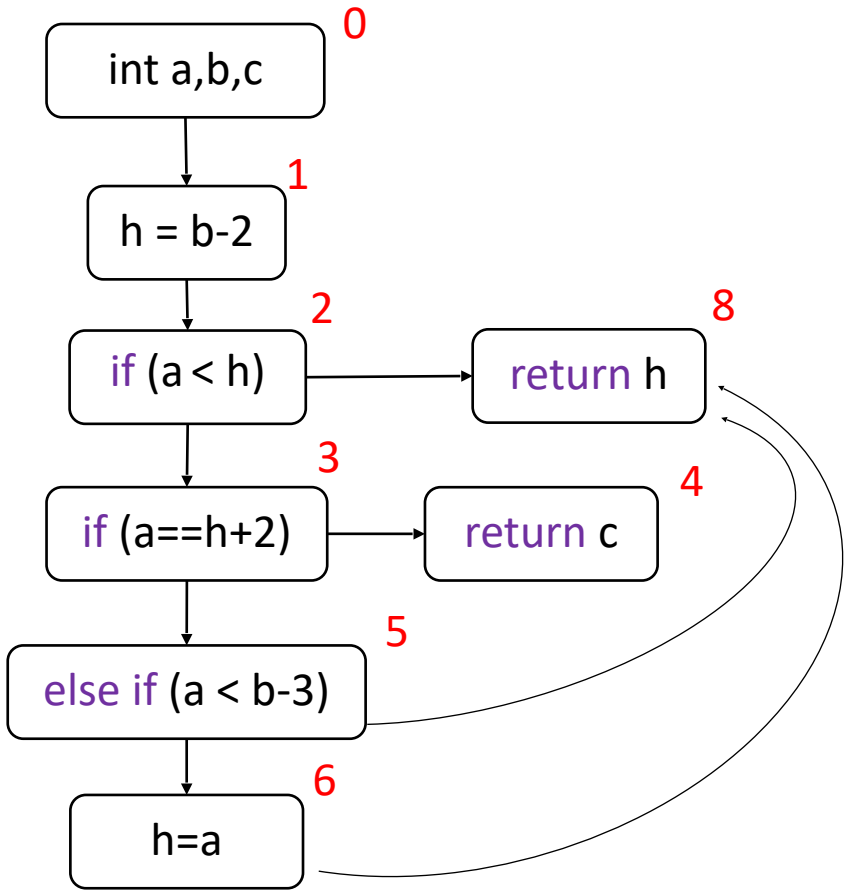
```
0:  int bar(int a, int b, int c) {
1:    int h = b-2;
2:    if (a < h) {
3:      if (a == h+2)
4:        return c;
5:      else if (a < b-3)
6:        h = a;
7:    }
8:    return h;
9:  }
```

- Derive the CFG
- Derive the set of live variables at the exit of each block. Are there dead variables after definition at block 0?
- Use symbolic execution to explore all paths in the function

Symbolic Execution: Exercise 2

- CFG structure

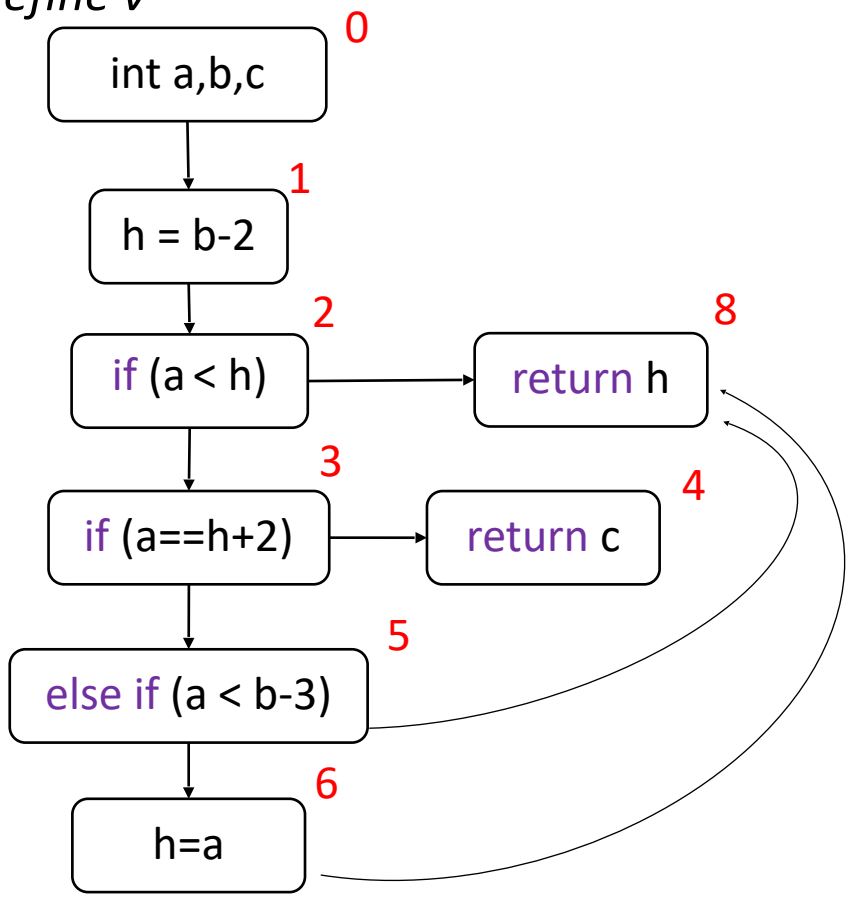
```
0:  int bar(int a, int b, int c) {
1:    int h = b-2;
2:    if (a < h) {
3:      if (a == h+2)
4:        return c;
5:      else if (a < b-3)
6:        h = a;
7:    }
8:    return h;
9:  }
```



Symbolic Execution: Exercise 2

- Live variables: *Given a CFG, a variable v is live at the exit of a block b if there is some path (on the CFG) from block b to a use of v that does not redefine v*

LV(0) = {a,b,c}
LV(1) = {a,b,c,h}
LV(2) = {a,b,c,h}
LV(3) = {a,b,c,h}
LV(4) = { }
LV(5) = {a,h}
LV(6) = {h}
LV(8) = { }



- All variables defined at block 0 may be live after 0

Symbolic Execution: Exercise 2

POLITECNICO MILANO 1863

- Symbolic execution path <0,1,2,3,4,8>

```
0:  int bar(int a, int b, int c) {
1:    int h = b-2;
2:    if (a < h) {
3:      if (a == h+2)
4:        return c;
5:      else if (a < b-3)
6:        h = a;
7:    }
8:    return h;
9:  }
```

<0>

a	b	c	π
A	B	C	T

<0,1>

a	b	c	h	π
A	B	C	B-2	T

<0,1,2>

a	b	c	h	π
A	B	C	B-2	A<B-2

<0,1,2,3,4>

a	b	c	h	π
A	B	C	B-2	A<B-2
				A=B-2+2

UNSAT ✘

Path <0,1,2,3,4> is the only one where c is used (with no redefinition) after definition at block 0, so, c is actually dead

Symbolic Execution: Exercise 2

POLITECNICO MILANO 1863

- Symbolic execution
  path <0,1,2,3,5,6,8>

```
0:  int bar(int a, int b, int c) {
1:    int h = b-2;
2:    if (a < h) {
3:      if (a == h+2)
4:        return c;
5:      else if (a < b-3)
6:        h = a;
7:    }
8:    return h;
9:  }
```

<0,1,2>

a	b	c	h	π
A	B	C	B-2	A<B-2

<0,1,2,3>

a	b	c	h	π
A	B	C	B-2	A<B-2 A≠B-2+2

<0,1,2,3,5>

a	b	c	h	π
A	B	C	B-2	A<B-2 A≠B A<B-3

It can be simplified

<0,1,2,3,5,6,8>

a	b	c	h	π	SAT ✓
A	B	C	A	A<B-3	

Symbolic Execution: Exercise 2

POLITECNICO MILANO 1863

- Symbolic execution
  path <0,1,2,3,5,8>

```
0:  int bar(int a, int b, int c) {
1:    int h = b-2;
2:    if (a < h) {
3:      if (a == h+2)
4:        return c;
5:      else if (a < b-3)
6:        h = a;
7:    }
8:    return h;
9:  }
```

<0,1,2>

a	b	c	h	π
A	B	C	B-2	A<B-2

<0,1,2,3>

a	b	c	h	π
A	B	C	B-2	A<B-2
				A≠B-2+2

<0,1,2,3,5>

a	b	c	h	π
A	B	C	B-2	A<B-2
				A≠B
				A≥B-3

<0,1,2,3,5,8>

a	b	c	h	π	SAT ✓
A	B	C	B-2	A=B-3	

- Symbolic execution
  path <0,1,2,8>

<0,1,2>

a	b	c	h	π
A	B	C	B-2	A≥B-2

<0,1,2,8>

a	b	c	h	π	SAT ✓
A	B	C	B-2	A≥B-2	

Concolic Execution: Exercise 1

POLITECNICO MILANO 1863

- Consider the function `foo`, written in a C-like language:

- Run a concolic execution starting from the following input {a = 1, b = 3} to find possible test cases that guarantee :
  - No execution of the loop;
  - Execution of the loop two times. Line 3 must be executed only in the second iteration

```
0: void foo(int a, int b) {
1:   for (int i=a; i<b; i++) {
2:     if (i % 5 == 0) {
3:       print(i)
4:   }
5: }
```

Exercise 1

- First execution from input {a = 1, b = 3}

```
0: void foo(int a, int b) {
1:   for (int i=a; i<b; i++) {
2:     if (i % 5 == 0) {
3:       print(i)
4:   }
5: }
```

<0>

a	b	i	π
A	B		T
1	3		

→

<0,1>

a	b	i	π
A	B	A	A<B
1	3	1	

→

<0,1,2>

a	b	i	π
A	B	A	A<B
			A%5≠0
1	3	1	

→

<0,1,2,1>

a	b	i	π
A	B	A+1	~~A<B~~
			A%5≠0
			A+1<B
1	3	2	

→

<0,1,2,1,2>

a	b	i	π
A	B	A+1	A%5≠0
			A+1<B
			A+1%5≠0
1	3	2	

→

<0,1,2,1,2,1>

a	b	i	π
A	B	A+2	A%5≠0
			~~A+1<B~~
			A+1%5≠0
			~~A+2≥B~~ A+2 = B
1	3	3	

The test case executes the loop 2 times without executing the location 3

Exercise 1

- Let's find a test case that runs the loop zero times…

```
0: void foo(int a, int b) {
1:   for (int i=a; i<b; i++) {
2:     if (i % 5 == 0) {
3:       print(i)
4:   }
5: }
```

<0>

a	b	i	π
A	B		T
1	3		

<0,1>

a	b	i	π
A	B	A	A<B
1	3	1	

Negation: ¬(A < B) => A ≥ B

Solve: A ≥ B

{a = 3, b = 1}

<0>

a	b	i	π
A	B		T
3	1		

<0,1>

a	b	i	π
A	B	A	A≥B
3	1	3	

The new test case {a = 3, b = 1} does not execte the loop

Exercise 1

- Let's find a test case for the other path: <0,1,2,1,2,3,1>

```
0: void foo(int a, int b) {
1:   for (int i=a; i<b; i++) {
2:     if (i % 5 == 0) {
3:       print(i)
4:   }
5: }
```

<0>

a	b	i	π
A	B		T
1	3		

→ … →

a	b	i	π
A	B	A+2	A%5≠0 ~~A+1<B~~ A+1%5≠0 ~~A+2≥B~~ A+2=B
1	3	3	

Partial negation:

$A\%5\neq0 \wedge \neg(A+1\%5\neq0) \wedge A+2=B$

Solve:

$A\%5\neq0 \wedge A+1\%5=0 \wedge A+2=B$

<0>

a	b	i	π
A	B		T
4	6		

{a = 4, b = 6}

Exercise 1

- Let's find a test case for the other path: <0,1,2,1,2,3,1>

```
0: void foo(int a, int b) {
1:   for (int i=a; i<b; i++) {
2:     if (i % 5 == 0) {
3:       print(i)
4:   }
5: }
```

<0>

a	b	i	π
A	B		T
4	6		

→ <0,1>

a	b	i	π
A	B	A	A<B
4	6	4	

→ <0,1,2>

a	b	i	π
A	B	A	A<B
			A%5≠0
4	6	4	

→ <0,1,2,1>

a	b	i	π
A	B	A+1	~~A<B~~
			A%5≠0
			A+1<B
4	6	5	

→

<0,1,2,1,2>

→

a	b	i	π
A	B	A+1	A%5≠0
			A+1<B
			A+1%5=0
4	6	5	

→ <0,1,2,1,2,3,1>

a	b	i	π
A	B	A+2	A%5≠0
			A+1<B
			A+1%5=0
			~~A+2≥B~~ A+2 = B
4	6	6	

The new test case {a = 4, b = 6} executes the loop 2 times and runs location 3 the second time only.

Exercise 1

POLITECNICO MILANO 1863

- Summary of test cases we found:
  - { a=3, b=1 }: no loop
  - { a=1, b=3 }: loop 2 times without executing location 3
  - { a=4, b=6 }: loop 2 times executing location 3 (2^{nd} iteration only)

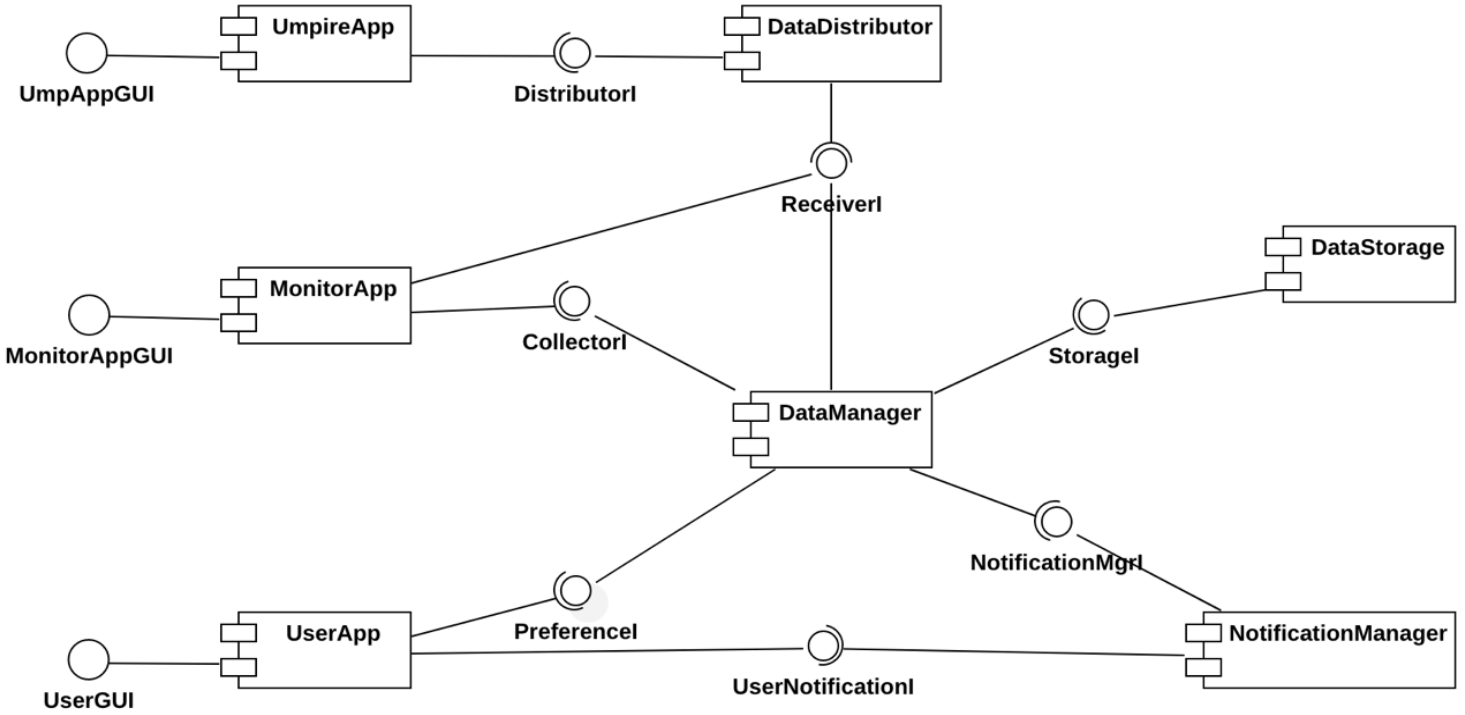
Integration Testing: Exercise 1

POLITECNICO MILANO 1863

- TennisMatchTracker (TMT) is a system that keeps track of scores of tennis matches. Every time that the chair umpire (i.e., the referee) of a tennis match updates the score of the match because a player won a point, the information is collected and stored by the system. Each point starts with a serve; if the first serve is out, a second serve is played; if a serve hits the net and stays in, it is repeated; if both the first and the second serve are out, the player who served loses the point through a “double fault”.
- The system relies on human monitors, who can tag each point with information about how the point was won (“ace”, “service winner”, “forehand winner”, “double fault”, etc.), and about the serves in each point (e.g., “first serve out”, “net first serve”); there is a standardized, finite set of tags that can be applied by human monitors. The information gathered by the system is made available to users, who can follow the status of matches in real time. Users can also indicate to TMT that they want to be notified when, after a point, certain matches (e.g., their favorite matches or the matches of their favorite players) they are interested in reach a crucial event (e.g., when a player is serving for the set, when there is set point, when there is match point).

Integration Testing: Exercise 1

- The development team has defined the following high level architecture for TMT.



Integration Testing: Exercise 1

POLITECNICO MILANO 1863

- The system includes three front-end applications: one (UmpireApp) for chair umpires, which allows them (through the UmpAppGUI interface) to indicate that a new point has been won by a player; one (MonitorApp) for human monitors, which allows them (through MonitorAppGUI) to tag points; and one (UserApp) for general users, which allows them (through UserGUI) to view the results of tennis matches (both ongoing and already completed), indicate their preferred players, matches, etc. The information about points inserted by chair umpires is sent (through the DistributorI interface) to a DataDistributor component, which uses the ReceiverI interface to notify both MonitorApp components and the DataManager component when a new point has been won (you can assume that the DataDistributor component is initialized with the information about the components that need to be notified of the completion of a point).
- The tags added by human monitors are sent to the DataManager component through the CollectorI interface. DataManager uses (through interface StorageI) the DataStorage component to persist information about points, tags, etc. UserApp uses interface PreferenceI to set user preferences (e.g., the matches to be followed, the favorite players). Finally, DataManager uses the NotificationMgrI interface of NotificationManager to distribute information about points (when a point is won, when tags are added, etc.) and crucial events (set points, match points, etc.); this information is passed by NotificationManager to users through the UserNotificationI interface. Notice that NotificationManager also keeps track of the users who must be notified of the various events. To do so, DataManager uses interface NotificationMgrI also to inform NotificationManager when a user changes one of its notification preferences.

Integration Testing: Exercise 1

POLITECNICO
MILANO 1863

- Identify a suitable integration testing strategy explaining why you think such strategy is appropriate for the TMTcase.

Integration Testing: Exercise 1

POLITECNICO MILANO 1863

- The integration testing strategy we adopt depends on the integration sequence we plan to execute. We have mentioned the bottom-up, top-down, thread based, and critical modules strategies.

- In this particular case, we can argue that DataManager is the most critical component offering three different interfaces and using other two. We could then adopt a critical module strategy centered around this component.

Integration Testing: Exercise 1

POLITECNICO MILANO 1863

- Define an integration test plan for TMT that follows the identified strategy

Integration Testing: Exercise 1

- Following the critical module strategy, after completing unit testing of DataManager and of DataStorage, we could focus on the integration and testing of these two components. We could reuse the stub implementing the NotificationMgrI interface and the DataManager driver, and run again the tests used for unit testing of DataManager verifying that the various pieces of data are correctly stored by DataStorage.
- After this, we could focus on the interaction between DataManager and DataDistributor (of course, after unit testing DataDistributor). In this step, we will still use a driver for DataManager but this will be focused on exercising only the operations offered through the CollectorI and PreferenceI interfaces. Data will be persistently stored through the DataStorage component and the stub implementing NotificationMgrI interface will still be used. Moreover, DataDistributor will be exercised by a driver simulating the UmpireApp.
- The next step could be the replacement of DataDistributor driver with the actual UmpireApp component. All other stubs and drivers will remain.
- At this point, we could focus on incorporating in the system the MonitorApp. This is used by DataDistributor through the ReceiveI interface and uses DataManager. We could, first, focus on the integration with DataDistributor and then focus on the integration with DataManager.
- As soon as this integration is finalized, the only missing components are NotificationManager and UserApp. We could focus on the integration between these two components in isolation, using a driver simulating DataManager for the NotificationManager and checking whether UserApp receives notifications properly. In the case we can allocate some team members to this task, such integration could take place in parallel with the other steps described above.
- Finally, we could connect NotificationManager and UserApp to DataManager and check whether UserApp is receiving the correct notifications based on the preferences that have been stated by the end user.

Integration Testing: Exercise 2

POLITECNICO MILANO 1863

- PubCoReader is a software system conceived to be used in courses with multiple instructors. It allows groups of students to read (or watch – for simplicity, in the following the two terms are used as synonyms) and annotate publications (books, papers, reports, videos…) in a collaborative way. PubCoReader should support the creation of multiple groups of readers that read the same or different publications independently one from the other. Group members collaborate online.
- For instance, at a certain point in time, we can observe the following scenario, where three groups of students are using the system.
- Group 1 is studying the book titled “Software Architectures” and is currently annotating Chapter 2 by complementing the book content with a picture from the lecture slides and some textual notes. Each group member sees the same book page and the identity of the group mate that has added an annotation to the book, as well as the exact position where the annotation has been added.

Integration Testing: Exercise 2

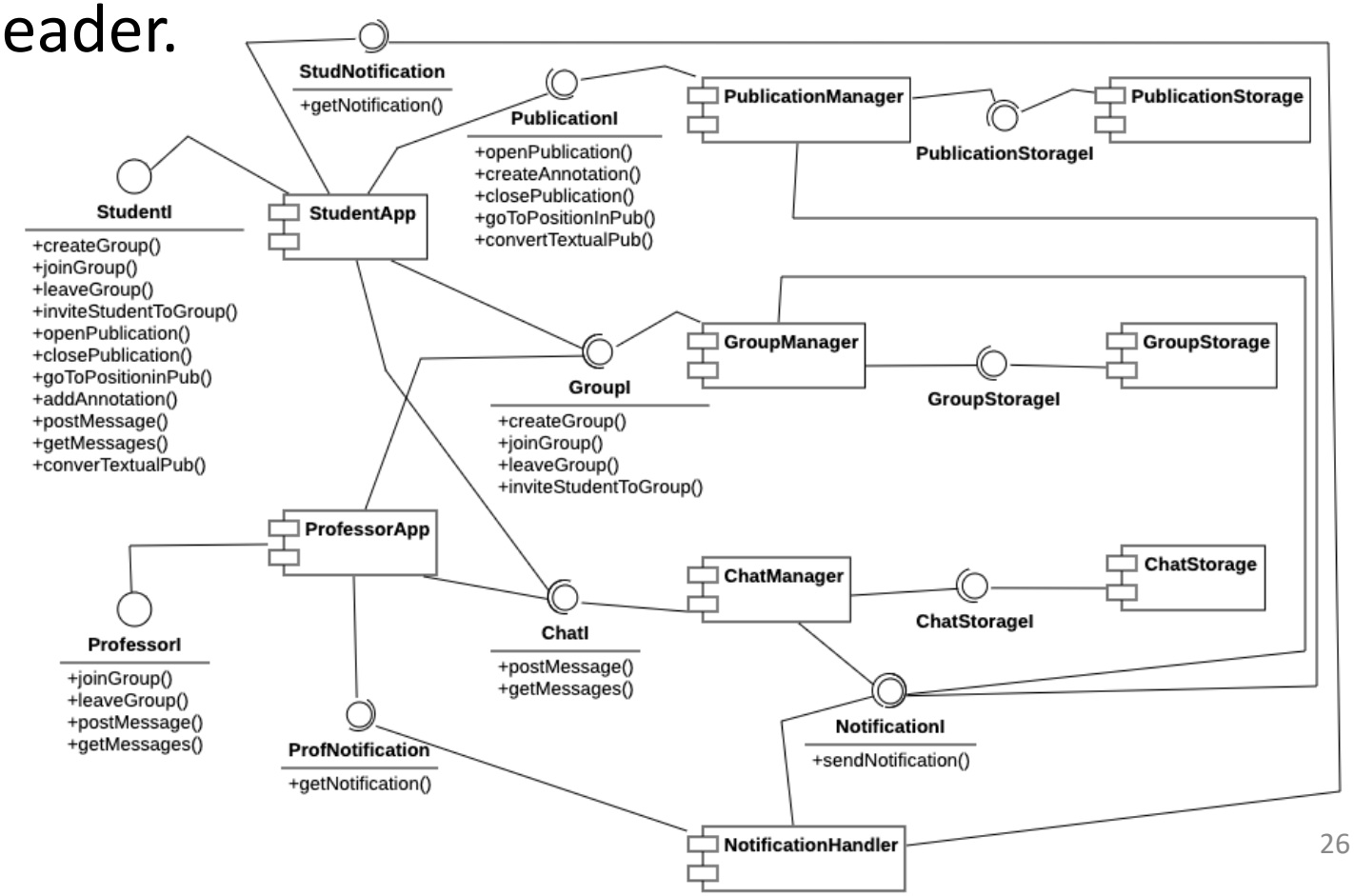
POLITECNICO MILANO 1863

- Group 2 is watching the video of the latest lecture in the course. The group members are discussing through the chat implemented as part of PubCoReader about a point that is not clear to any of them and add the following annotation to the specific point under discussion “@professor: please help us understand this point”.
- Group 3 has run the “translate to audio” feature on Chapter 3 of the “Software Architectures” book and is currently listening to the generated audio. Each time a group member stops the audio and inserts an annotation, this is associated with the corresponding part of text in the book.
- Two professors are monitoring the students’ activities. One of them notices the @professor message and temporarily joins the corresponding group to discuss the misunderstanding with them.

Integration Testing: Exercise 2

- The development team has defined the following high level architecture for PubCoReader.

The interfaces offered by StudentApp and ProfessorApp are aiming at representing the features the two applications make available to the two users of PubCoReader, students and professors.



Integration Testing: Exercise 2

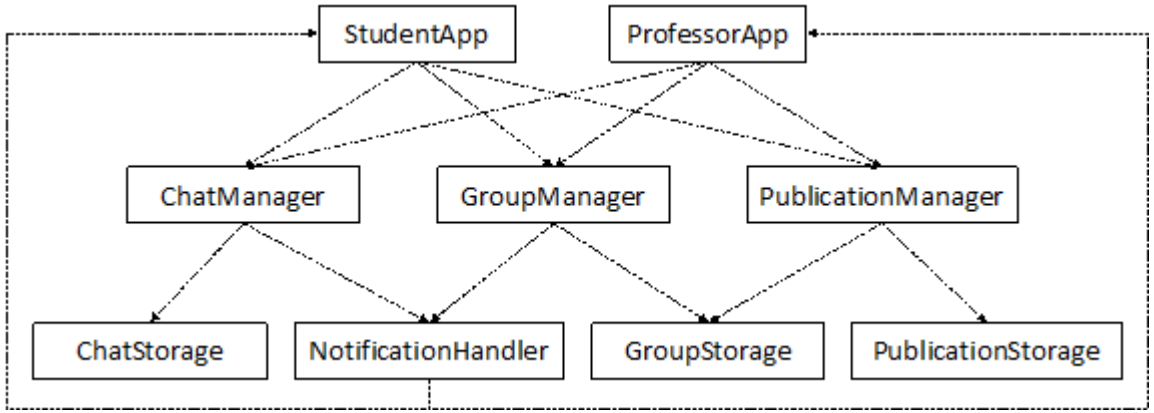
POLITECNICO MILANO 1863

- Define an integration testing strategy explaining why you think it is appropriate for the case at hand.

Integration Testing: Exercise 2 Solution

POLITECNICO MILANO 1863

- The integration testing strategy we adopt depends on the integration sequence we plan to execute. We have mentioned the bottom-up, top-down, thread based, and critical modules strategies.
- In this particular case, applying a bottom-up strategy could be the simplest approach. However, we need to consider that, in this case, the usage relationships lead to a cycle due to the fact that NotificationManager is using ProfessorApp and StudentApp that, for all the other features, are actually the top of the usage relationship (see the usage relationships representation in the figure below).



Integration Testing: Exercise 2 Solution

POLITECNICO MILANO 1863

So, we could proceed by excluding from the initial integration iteration the implementation of the NotificationHandler that could be replaced by a stub till the last step. More specifically, we could start from the integration and testing of the following elements:

- PublicationManager and PublicationStorage, using a stub for NotificationHandler and a driver for PublicationManager
- GroupManager and GroupStorage using a stub for NotificationHandler and a driver for GroupManager
- ChatManager and ChatStorage using a stub for NotificationHandler and a driver for ChatManager.

These three integration steps could be performed even in parallel if we can organize the development team in three disjoint subteams.

Then, we could proceed as follows:

- Integration of StudentApp with PublicationManager using a driver to replace the NotificationHandler
- Integration of StudentApp with GroupManager using a driver to replace the NotificationHandler
- Integration of StudentApp with ChatManager using a driver to replace the NotificationHandler

At this point we could focus on the integration between the NotificationHandler and the StudentApp and, finally, we could proceed with the integration and testing of the ProfessorApp and PublicationManager, GroupManager, and ChatManager, respectively, followed by the integration between the NotificationHandler and the ProfessorApp.