Modeling Interaction by Sequence Diagrams

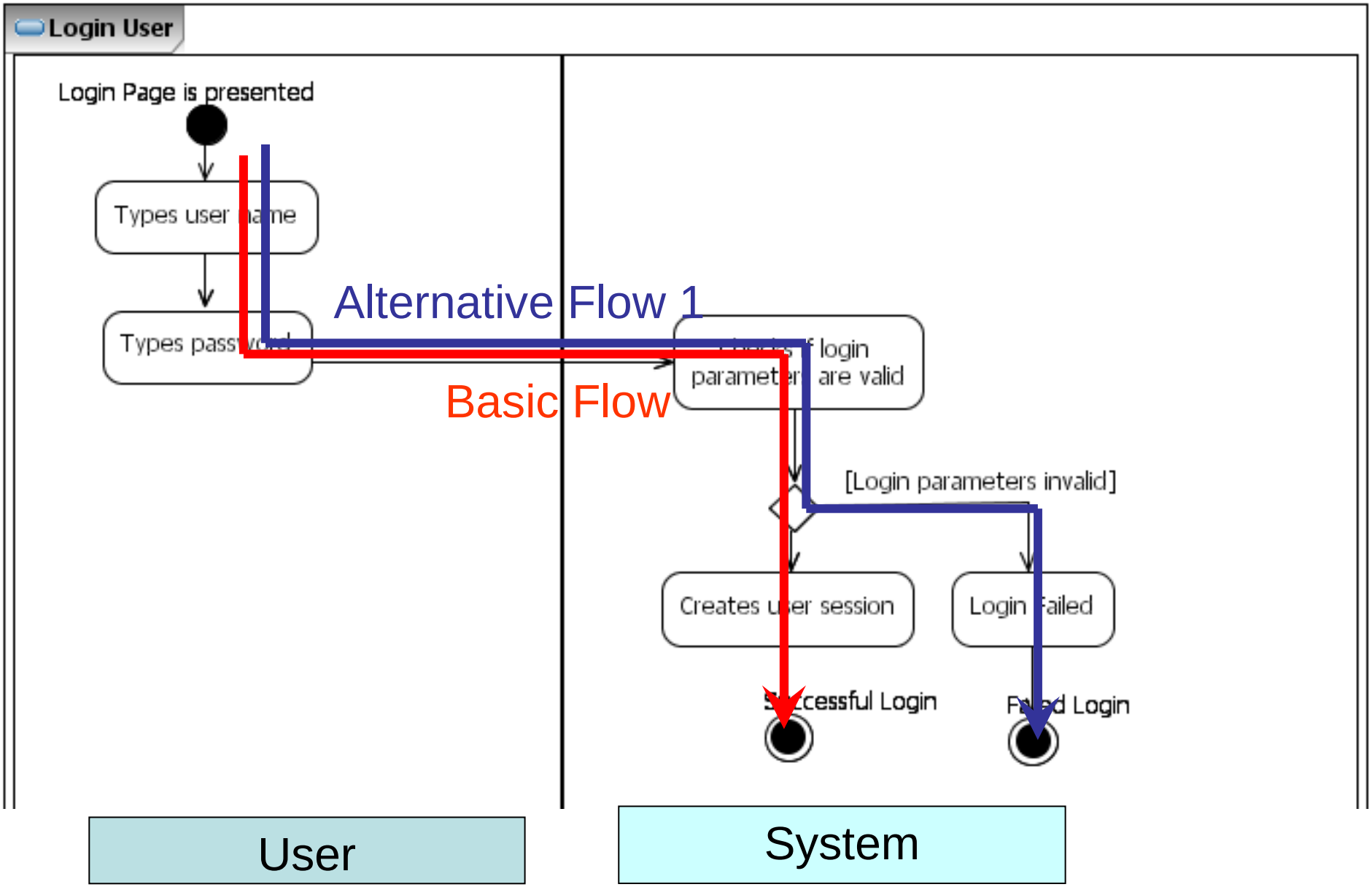
Sequence diagrams

Sequence diagrams

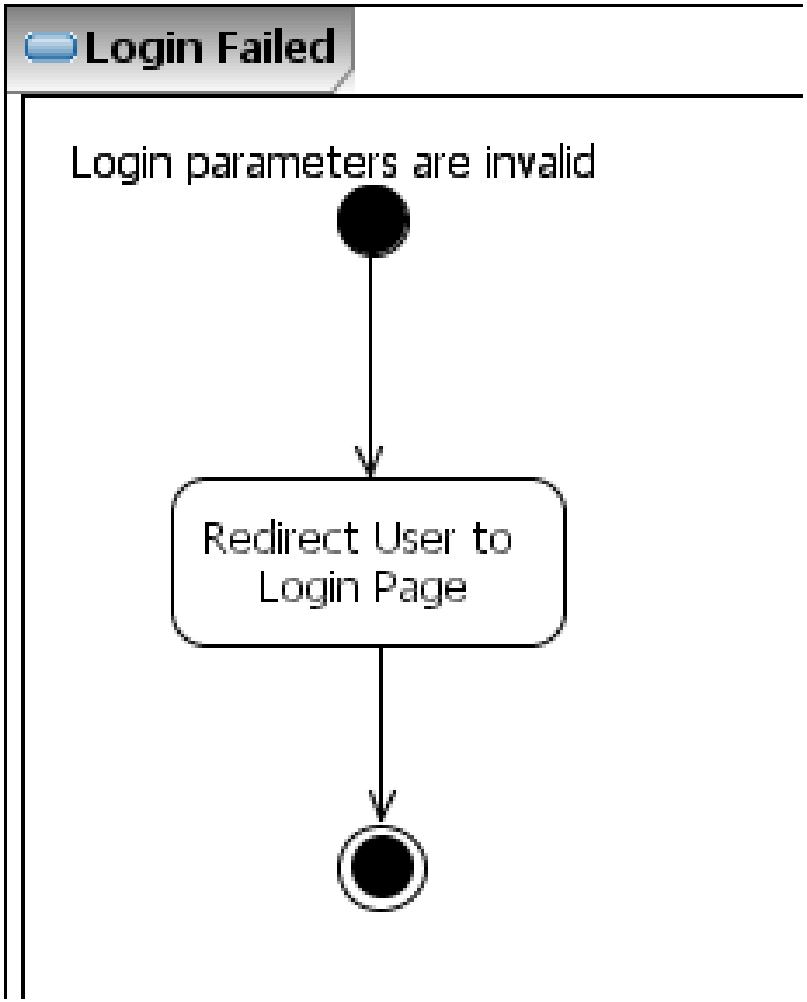
- Sequence diagram may specify:
  - a typical execution of a use case (scenario)
  - communication between components / objects by messages (analysis or design)
  - detailed trace of a statechart run
  - specification of a test case
  - overview of timing

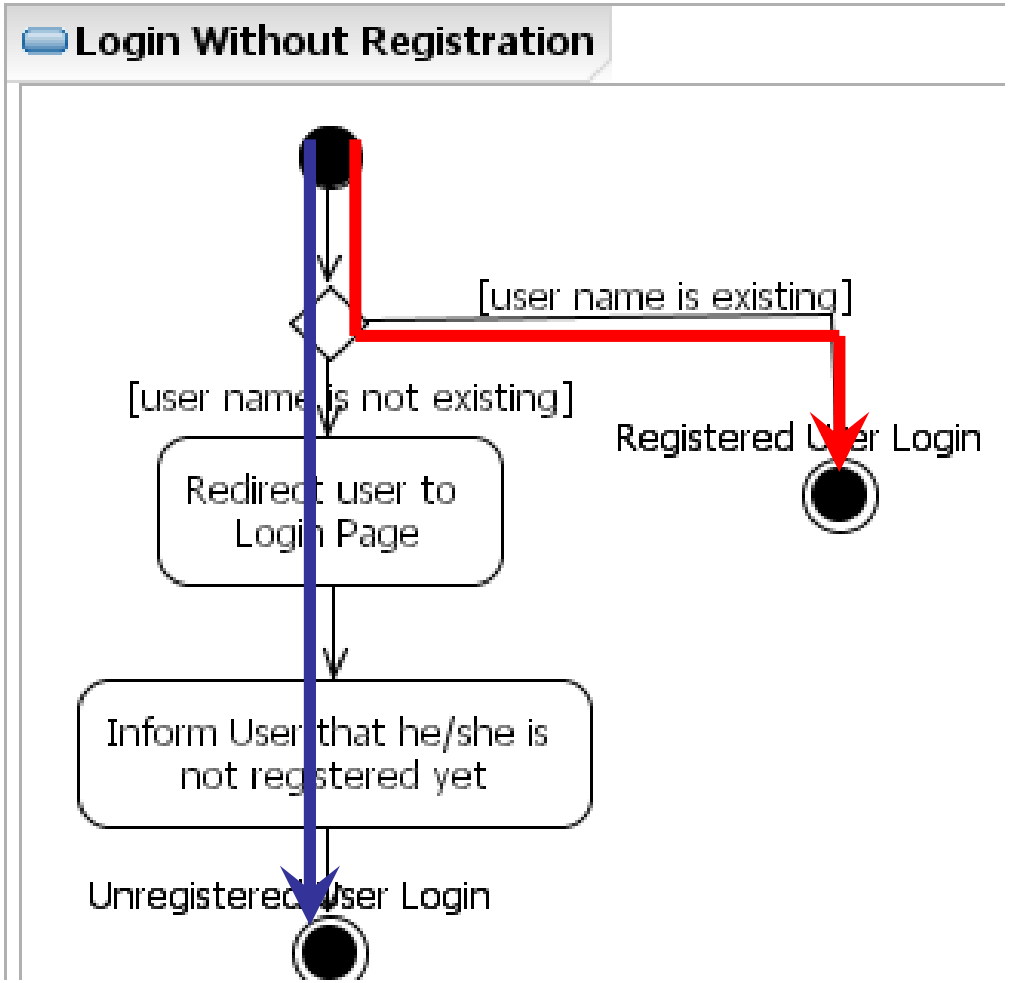
Example: Login User

Scenario: Login User I.



Scenario: Login User II.

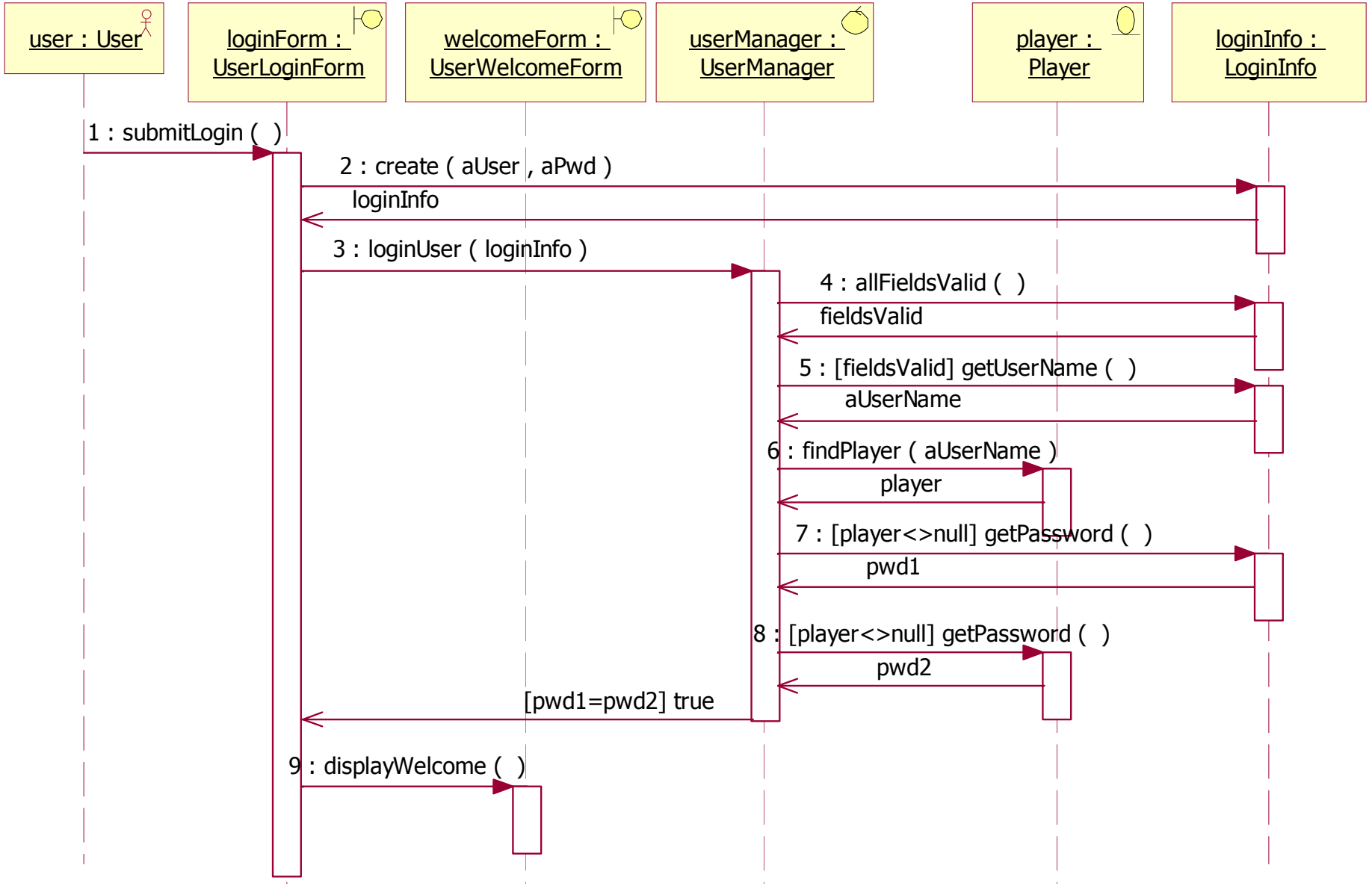




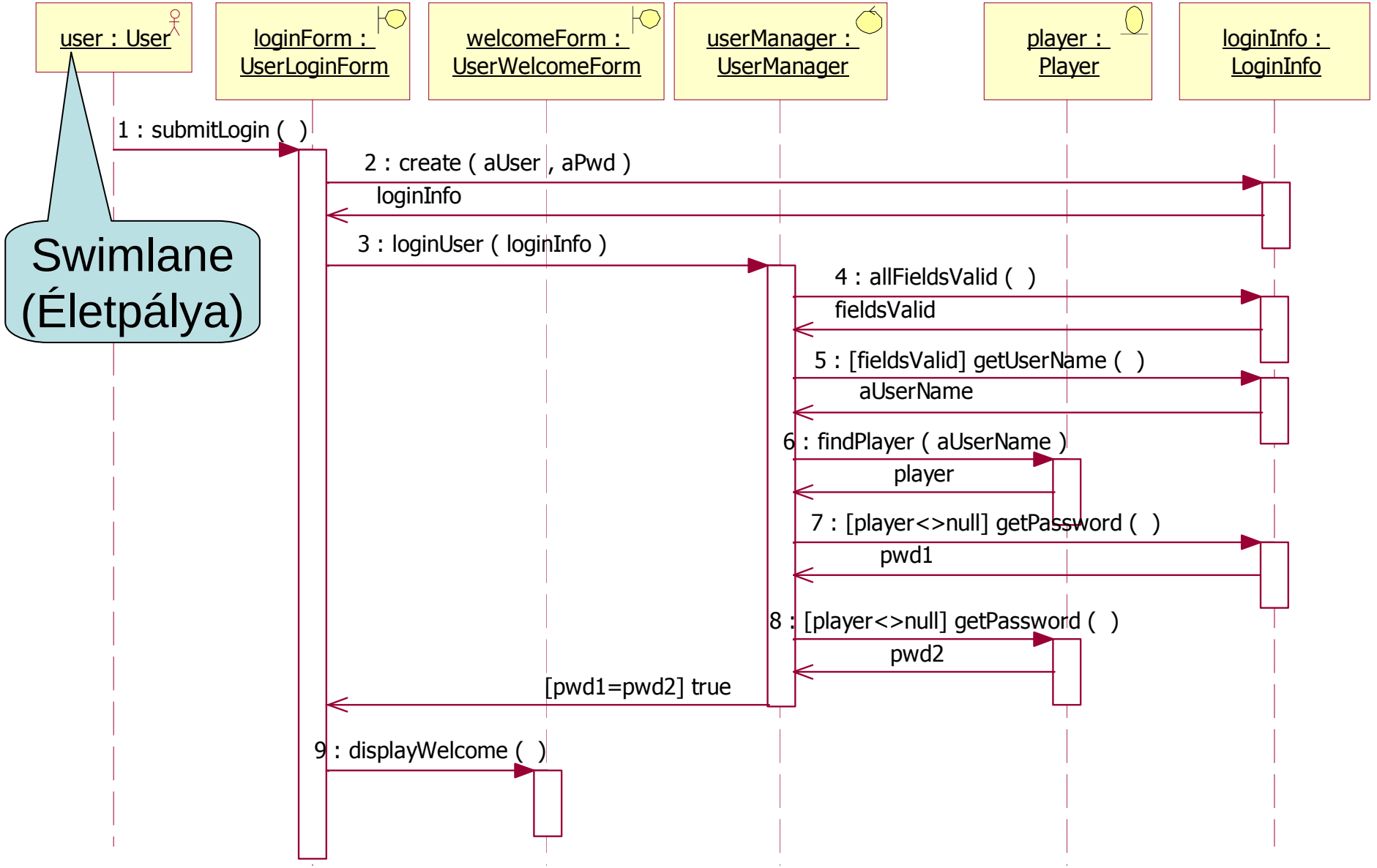
Prototypes of Sequence Diagrams

- Basic Flow: Main Success Scenario
  - Actor initiates interaction by submitting data
  - Boundary Class: collects user data
  - Control class performs
    - validity checks on data
    - business operation
  - Entity Classes:
    accessed exclusively by Control classes
- Alternative Flows:
  - UC-driven: At least one for each extension UC
  - Basic flow driven:
    At least one for each negated guard

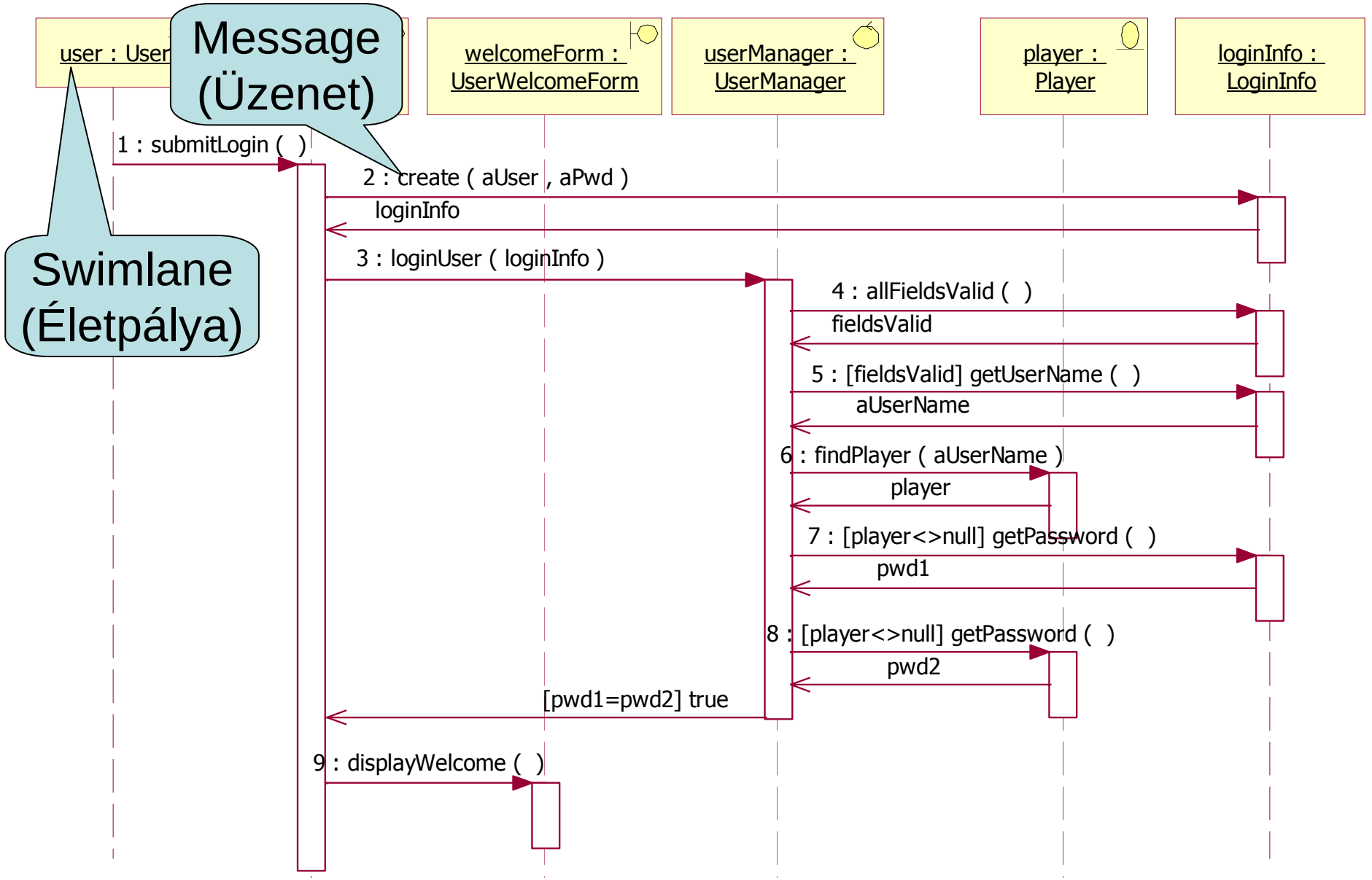
Login User – Basic Flow



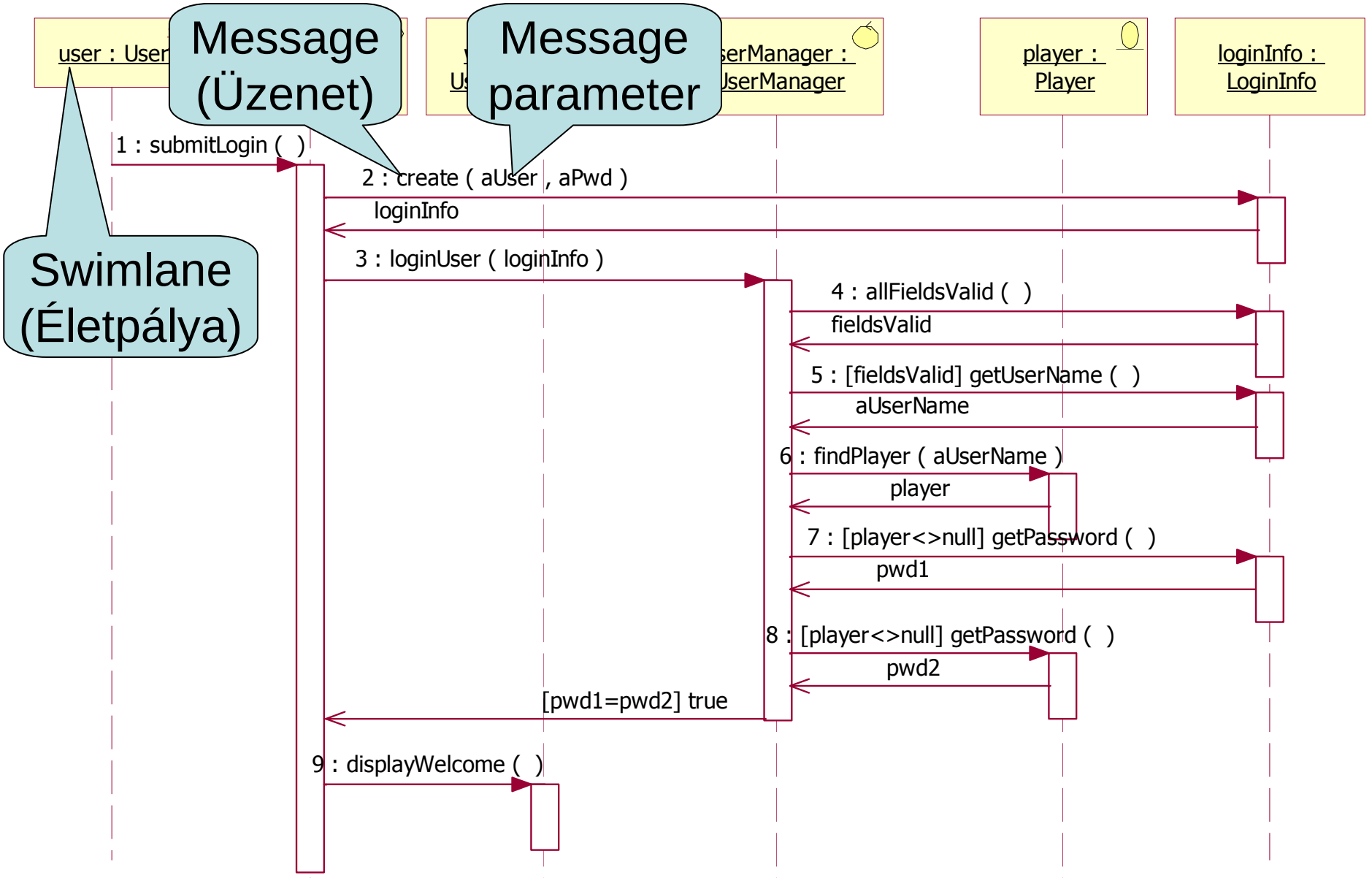
Login User – Basic Flow (Syntax)



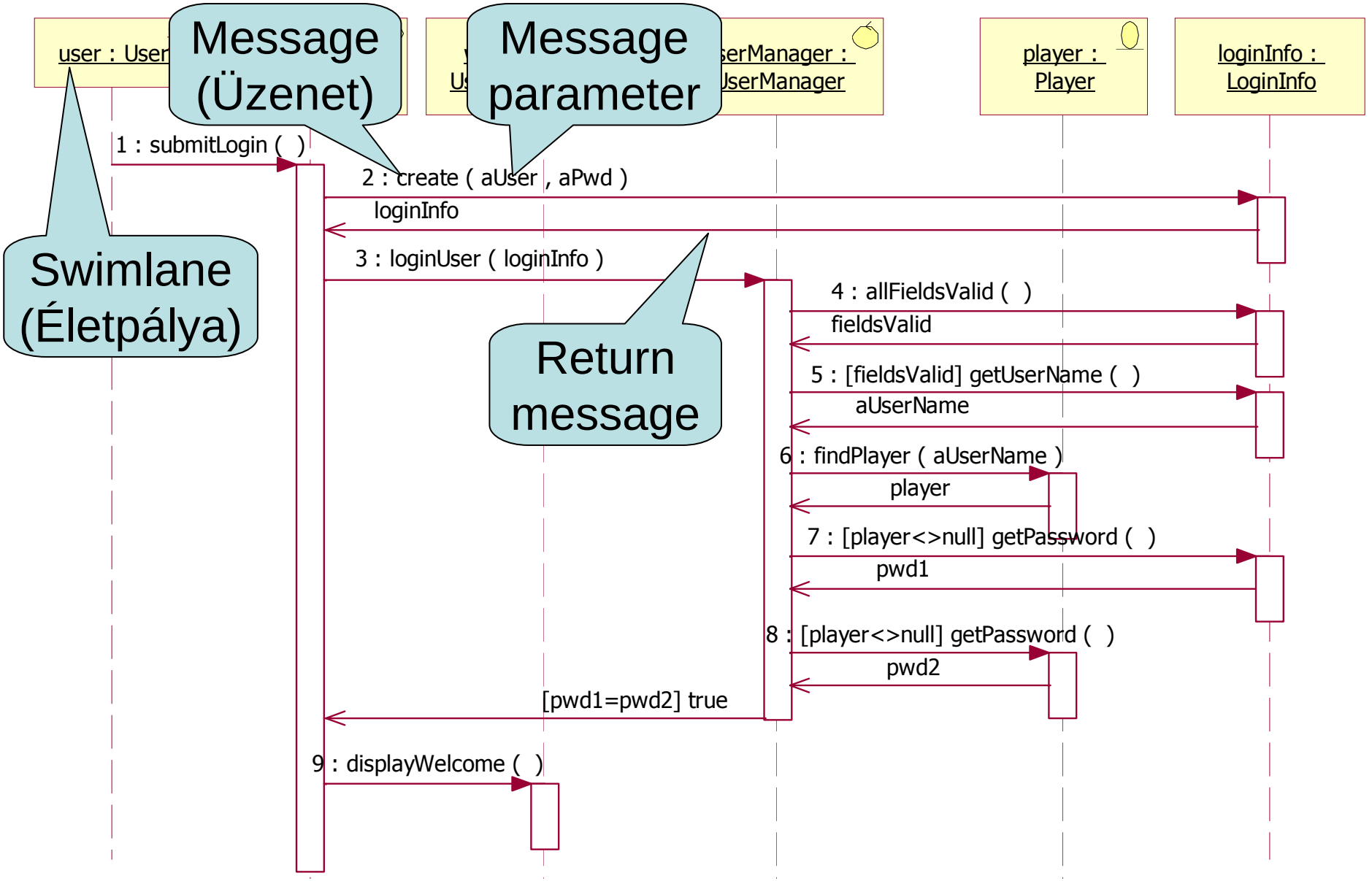
Login User – Basic Flow (Syntax)



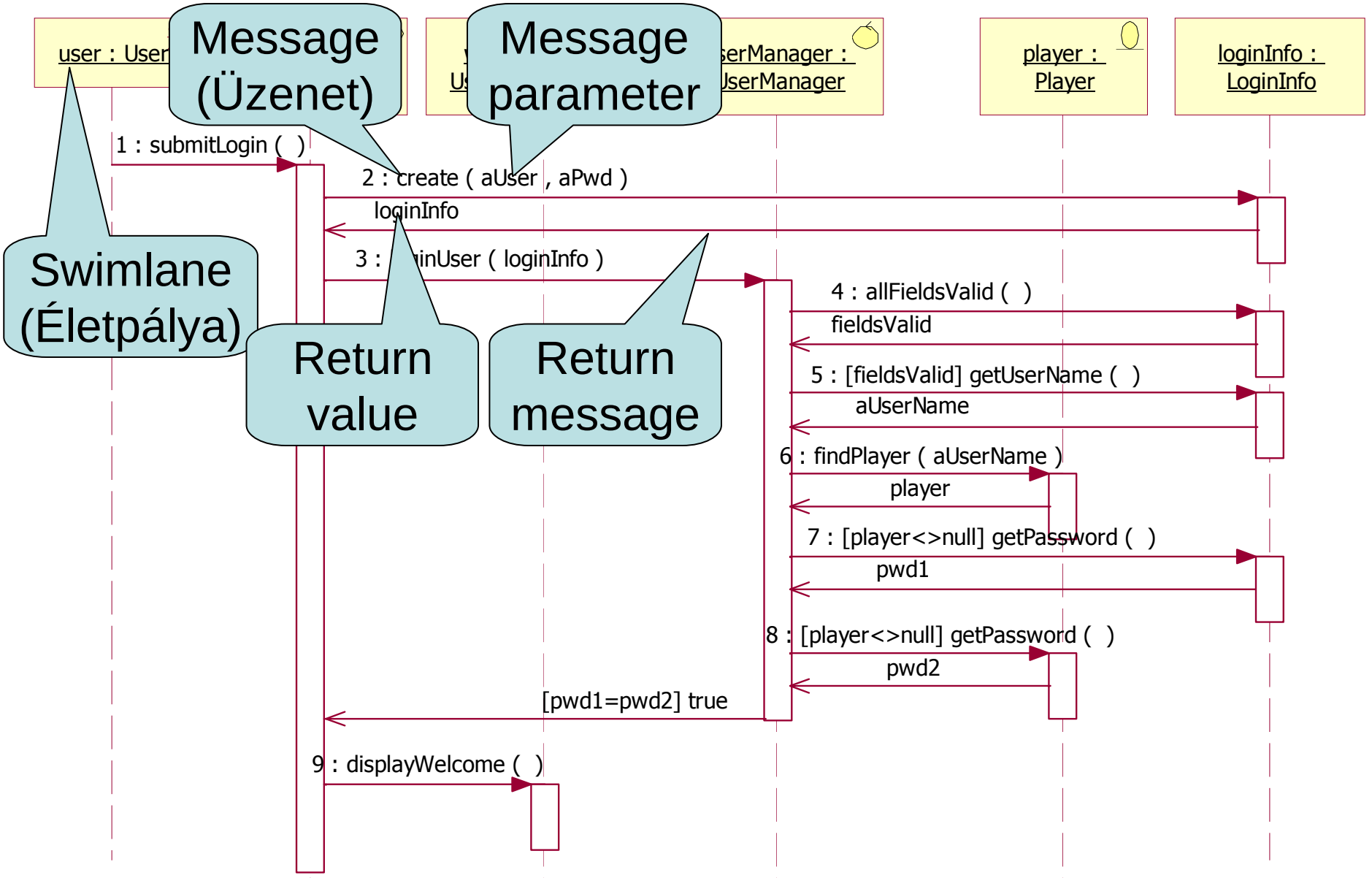
Login User – Basic Flow (Syntax)



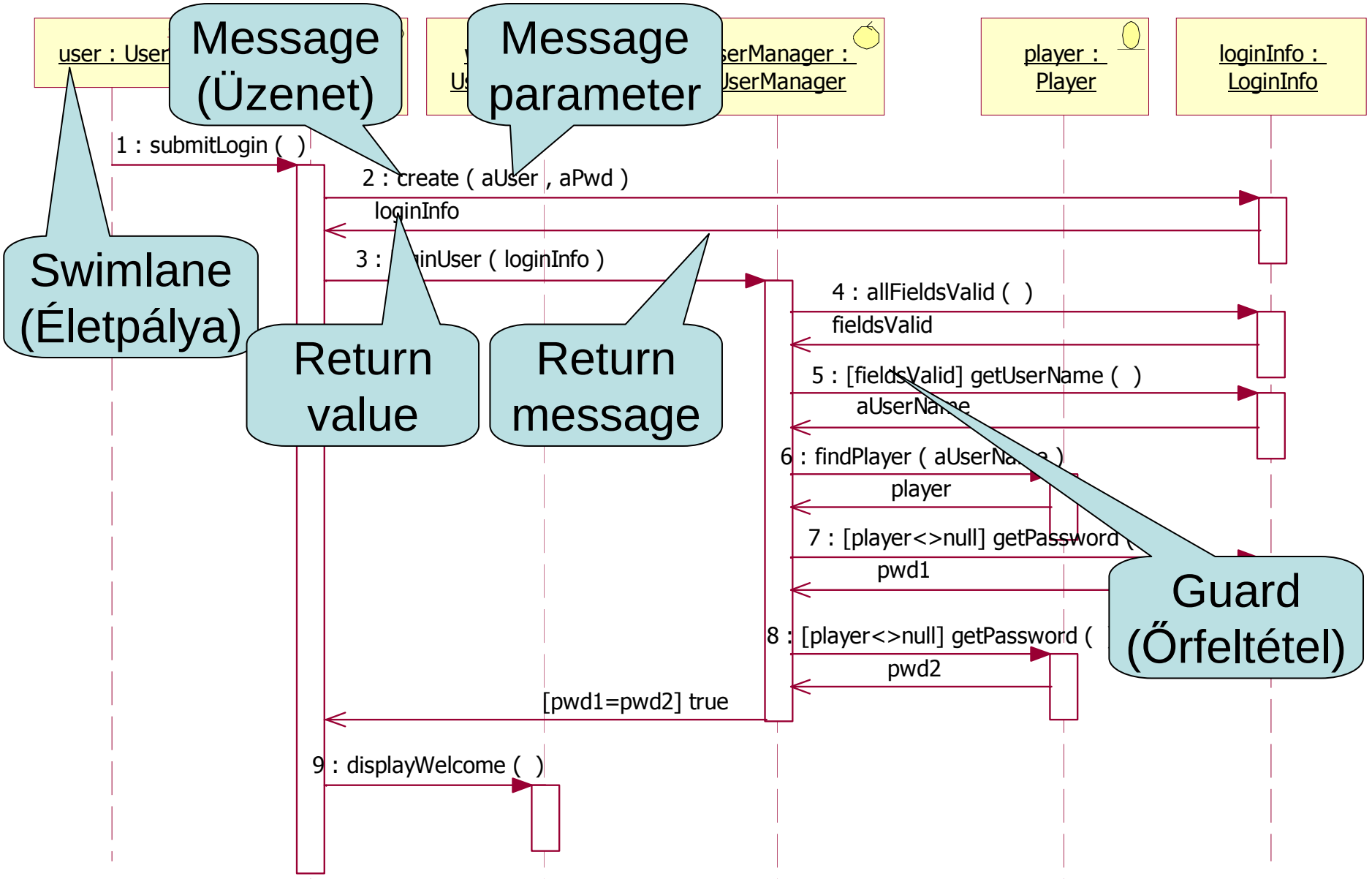
Login User – Basic Flow (Syntax)



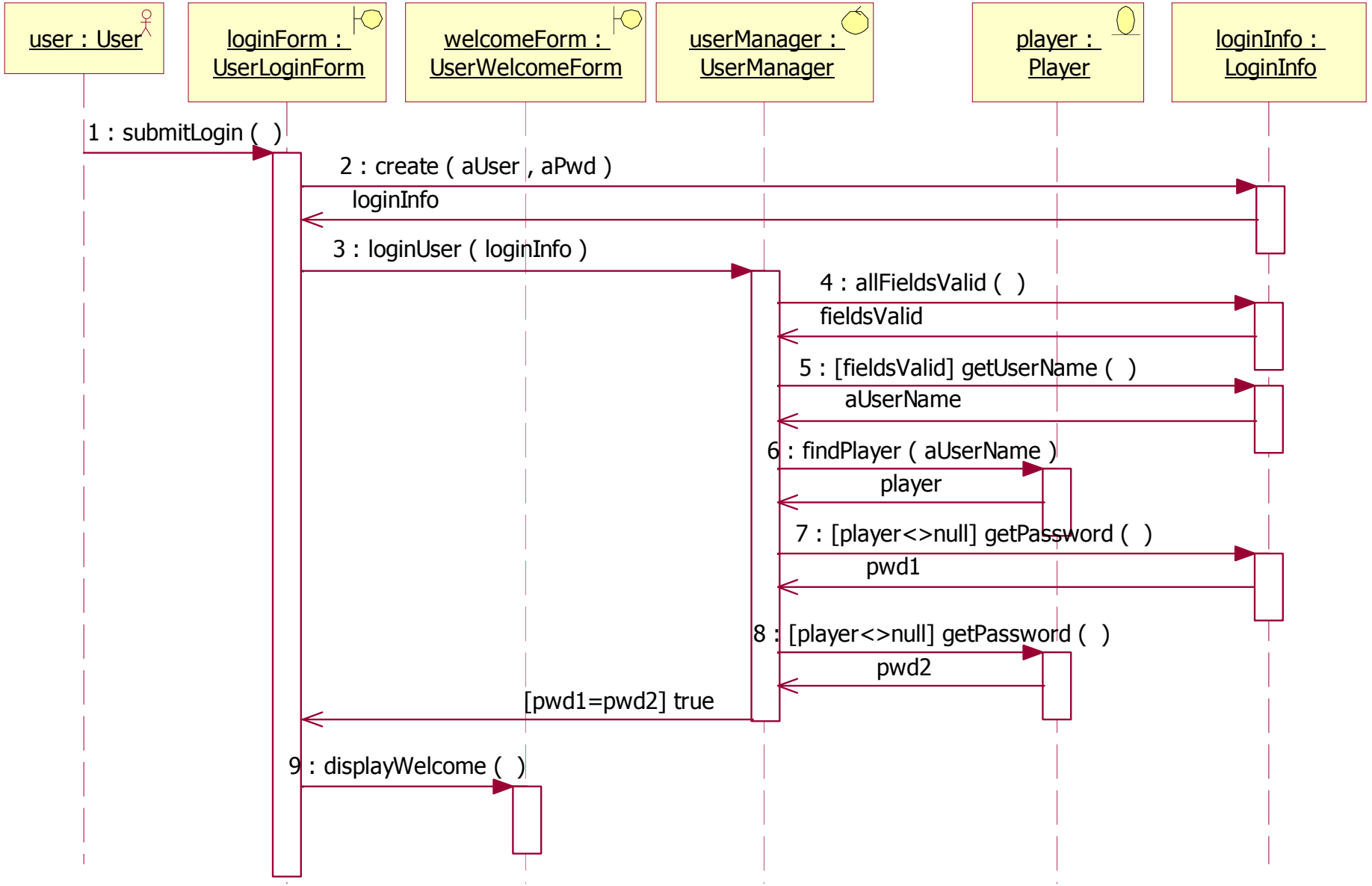
Login User – Basic Flow (Syntax)



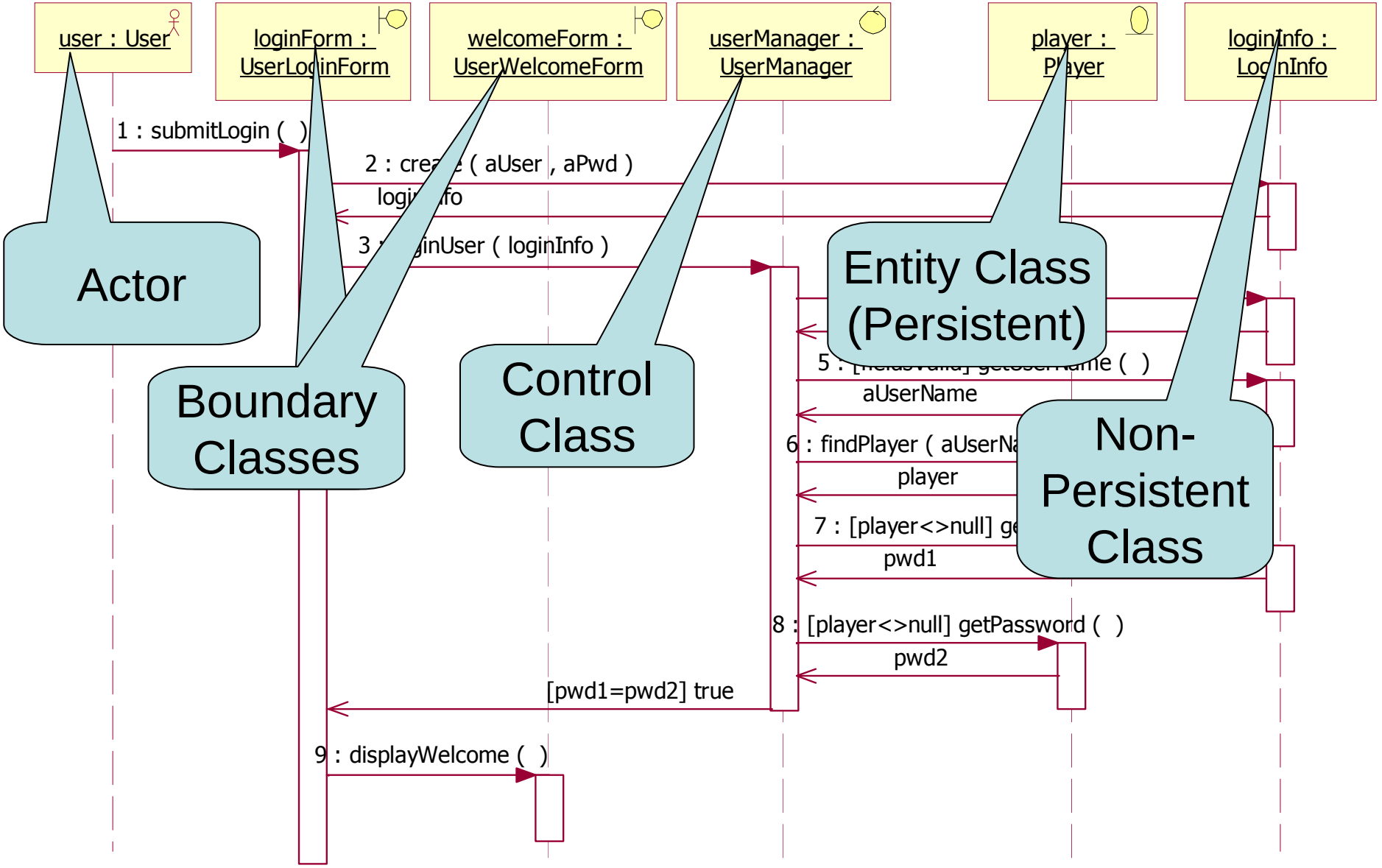
Login User – Basic Flow (Syntax)



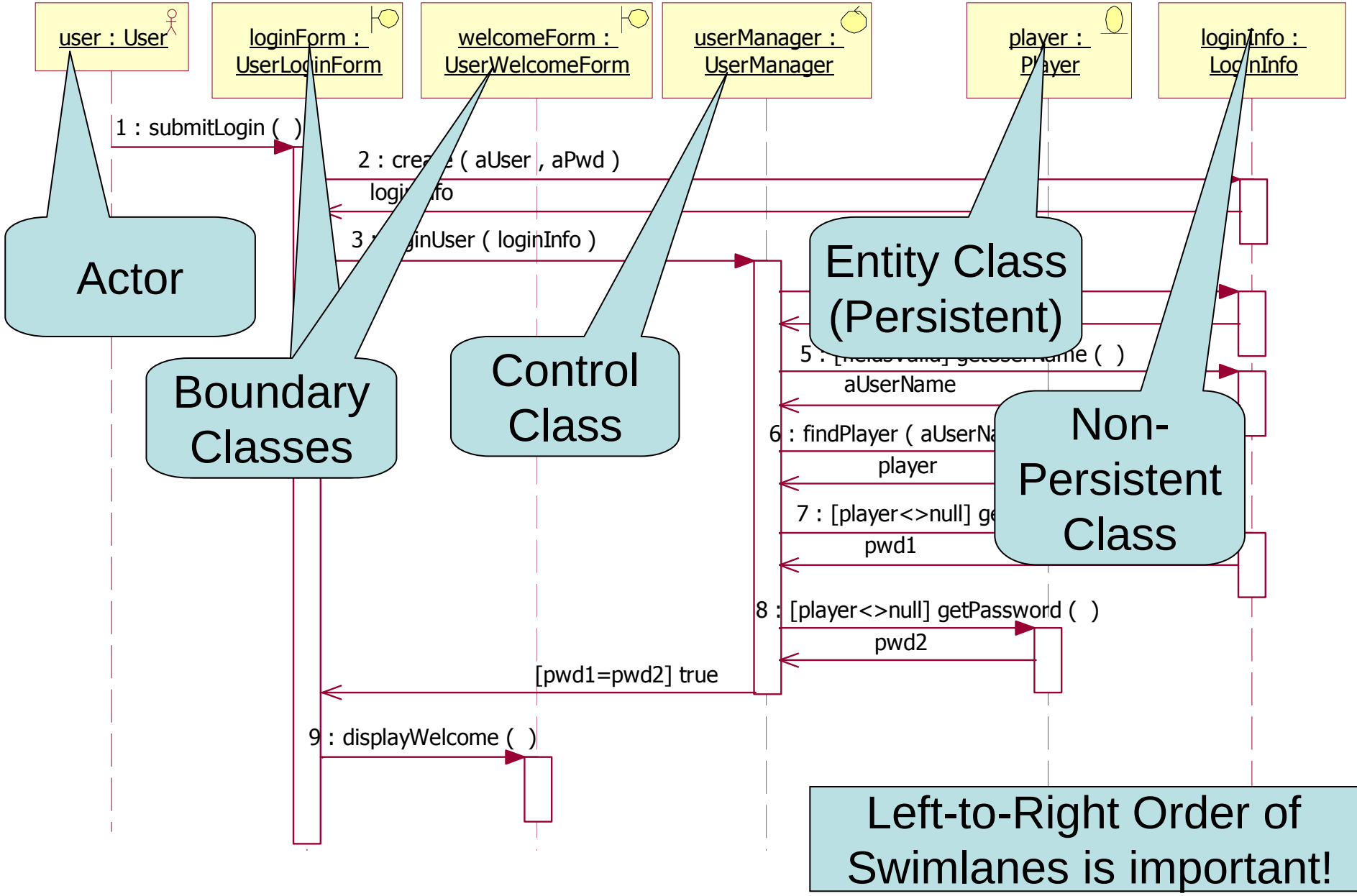
Login User – Basic Flow (Syntax)



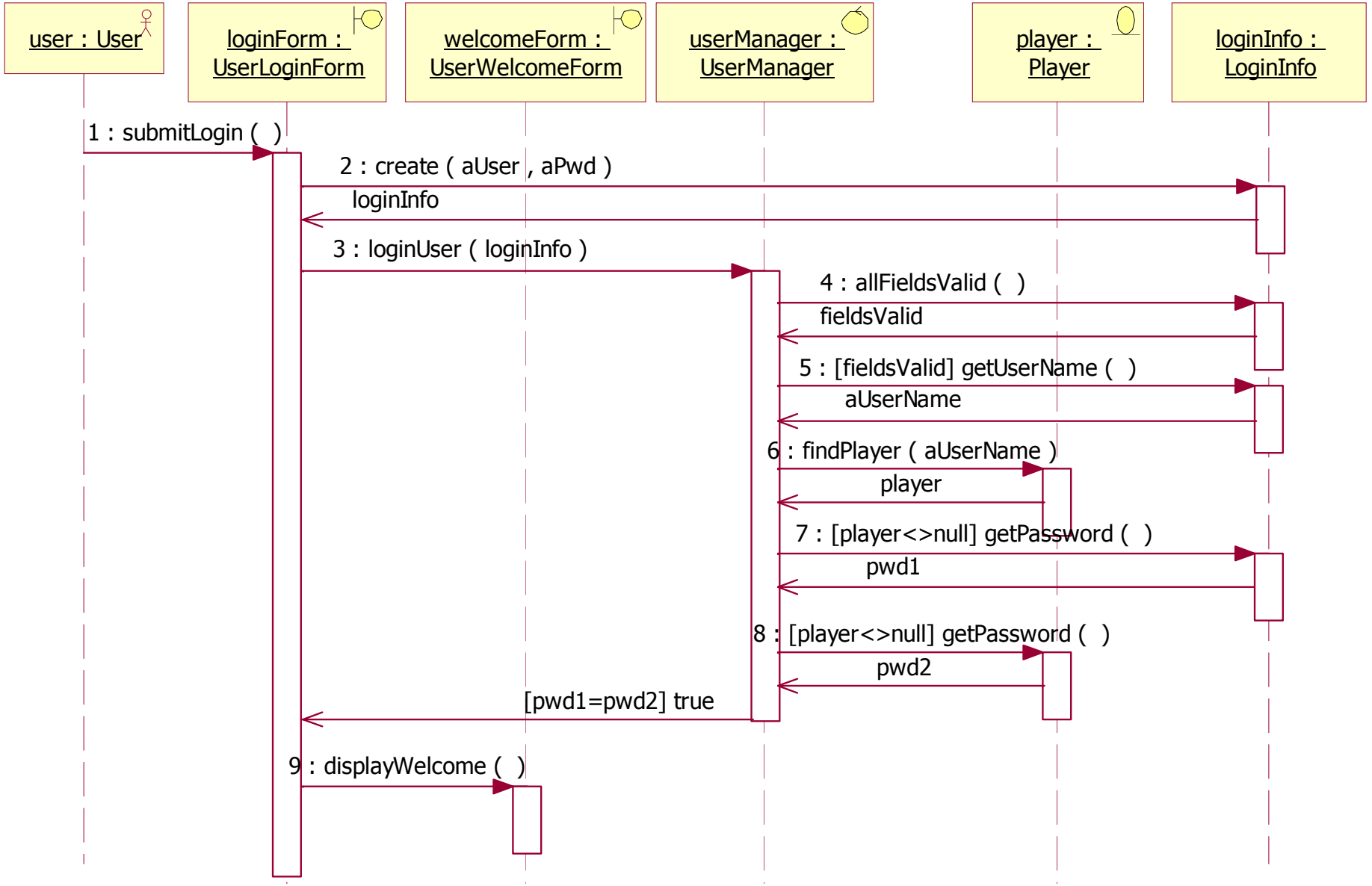
Login User – Basic Flow (Syntax)



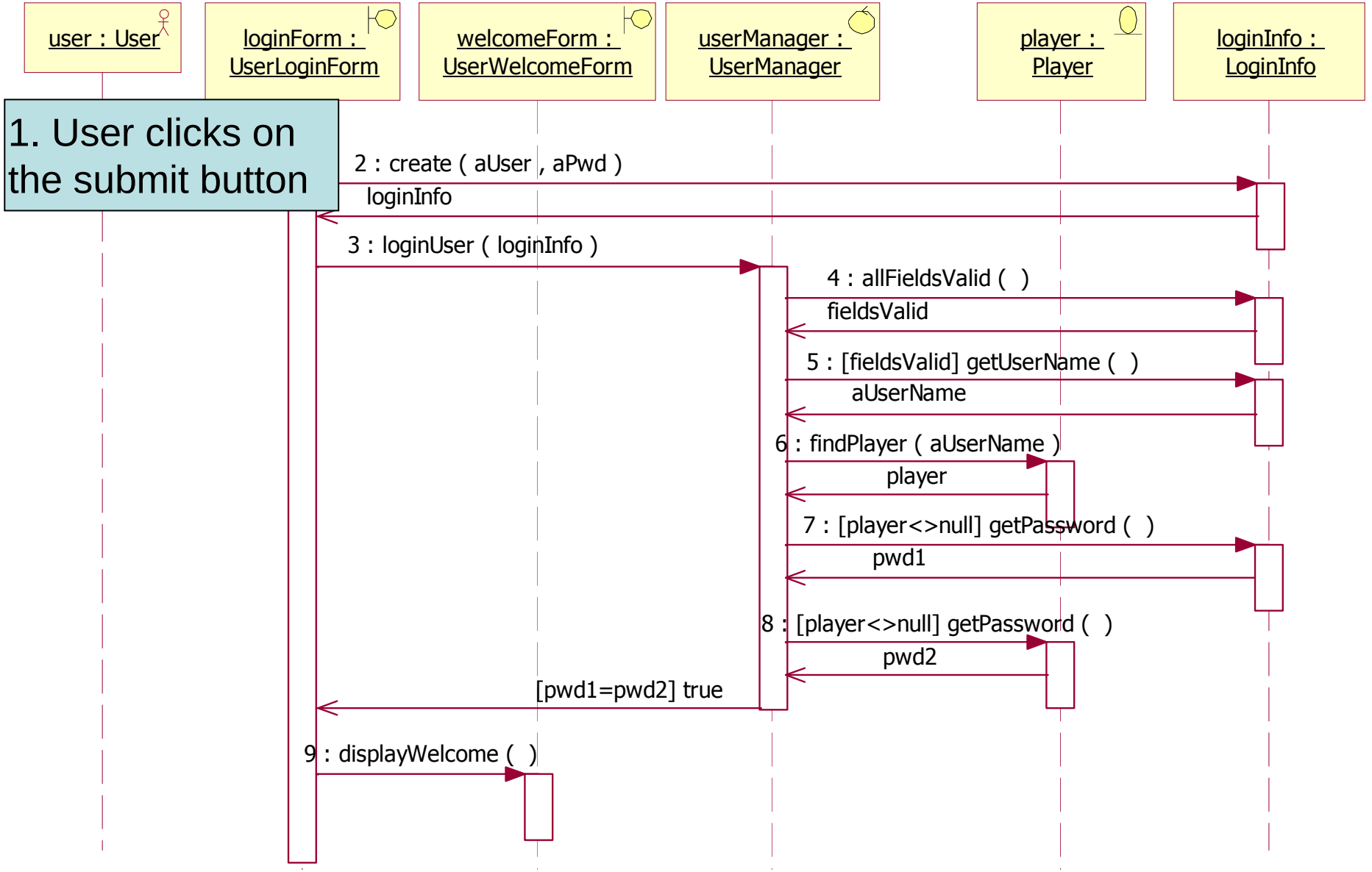
Login User – Basic Flow (Syntax)



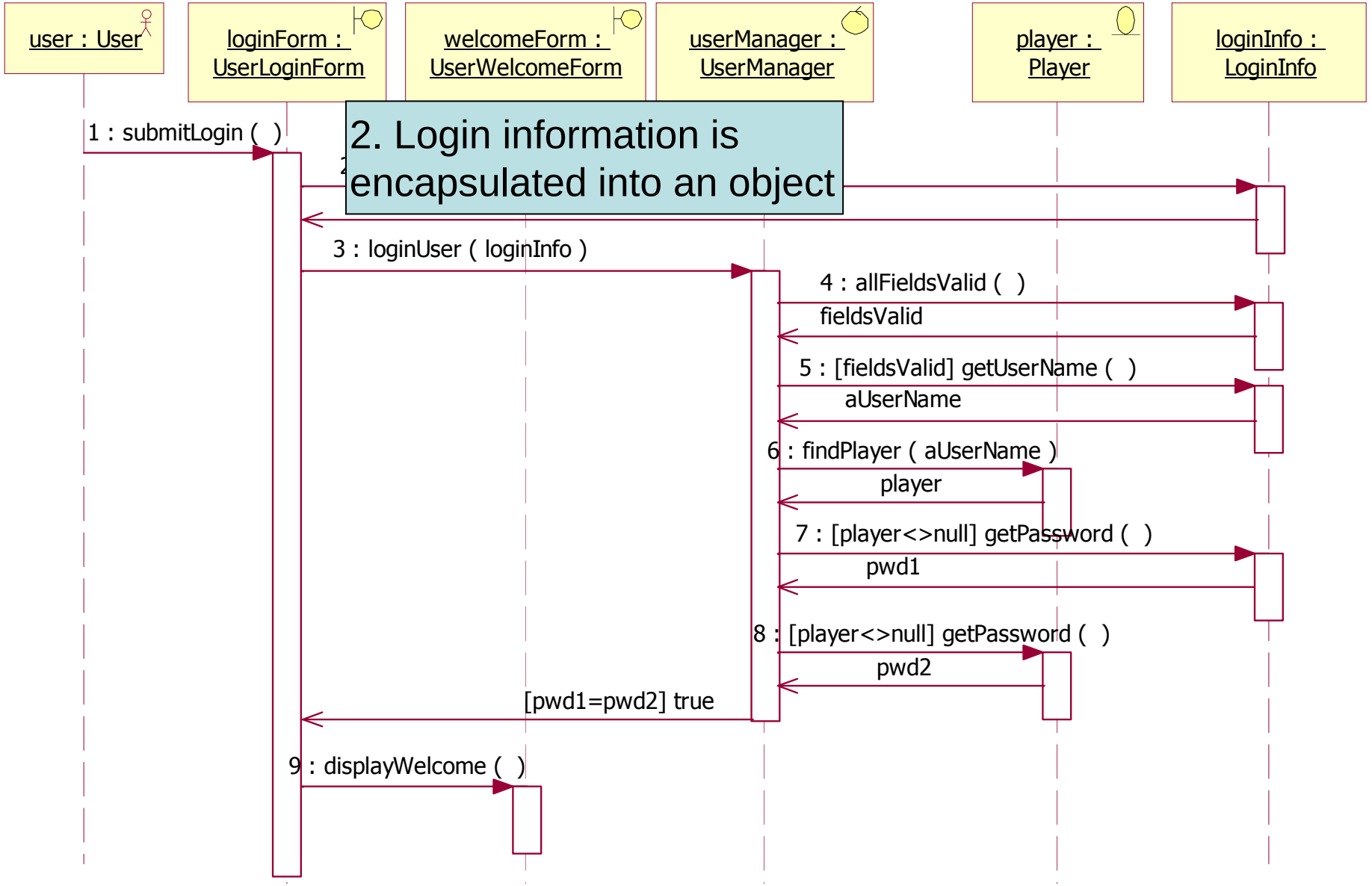
Login User – Basic Flow (Semantics)



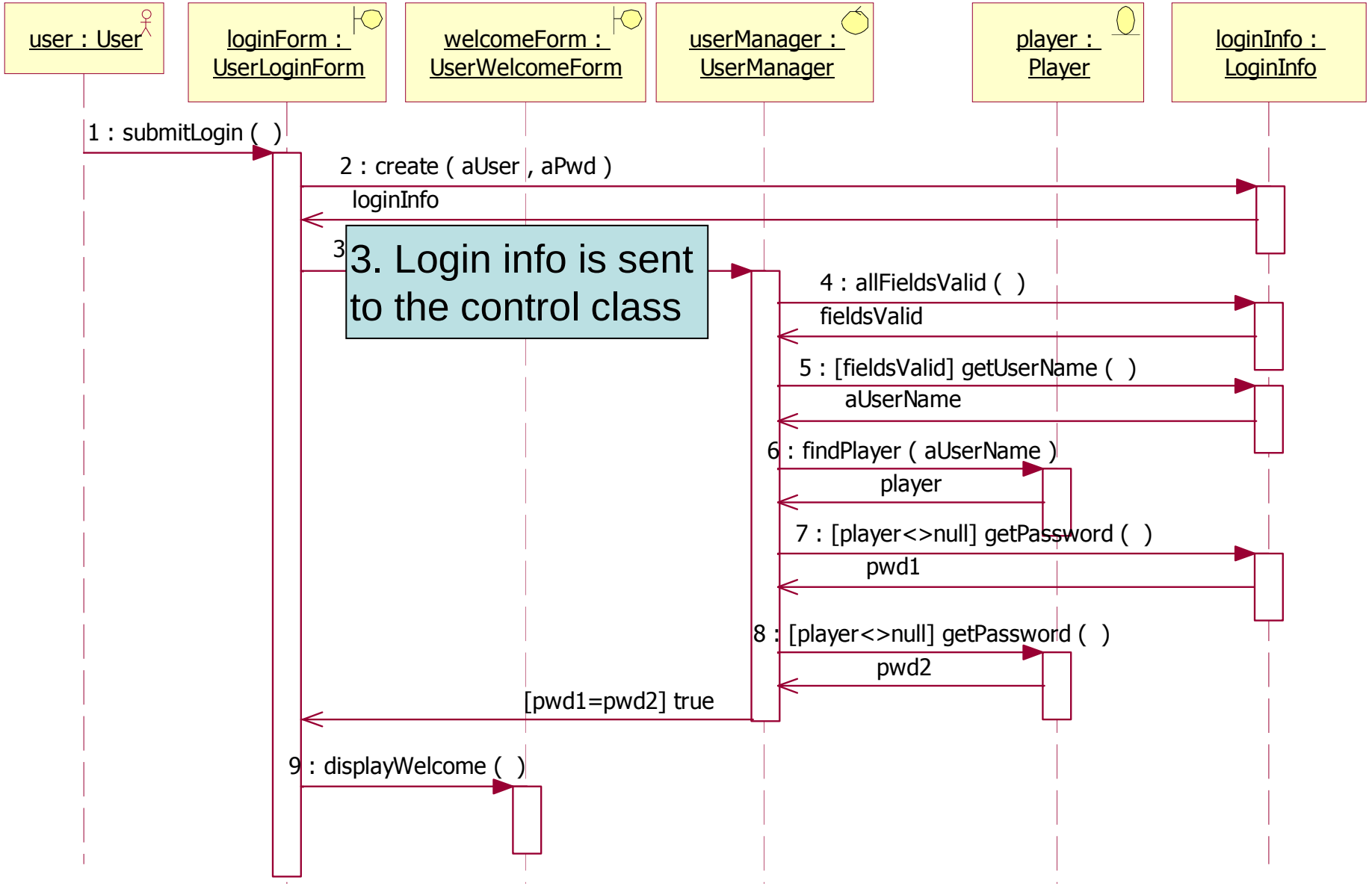
Login User – Basic Flow (Semantics)



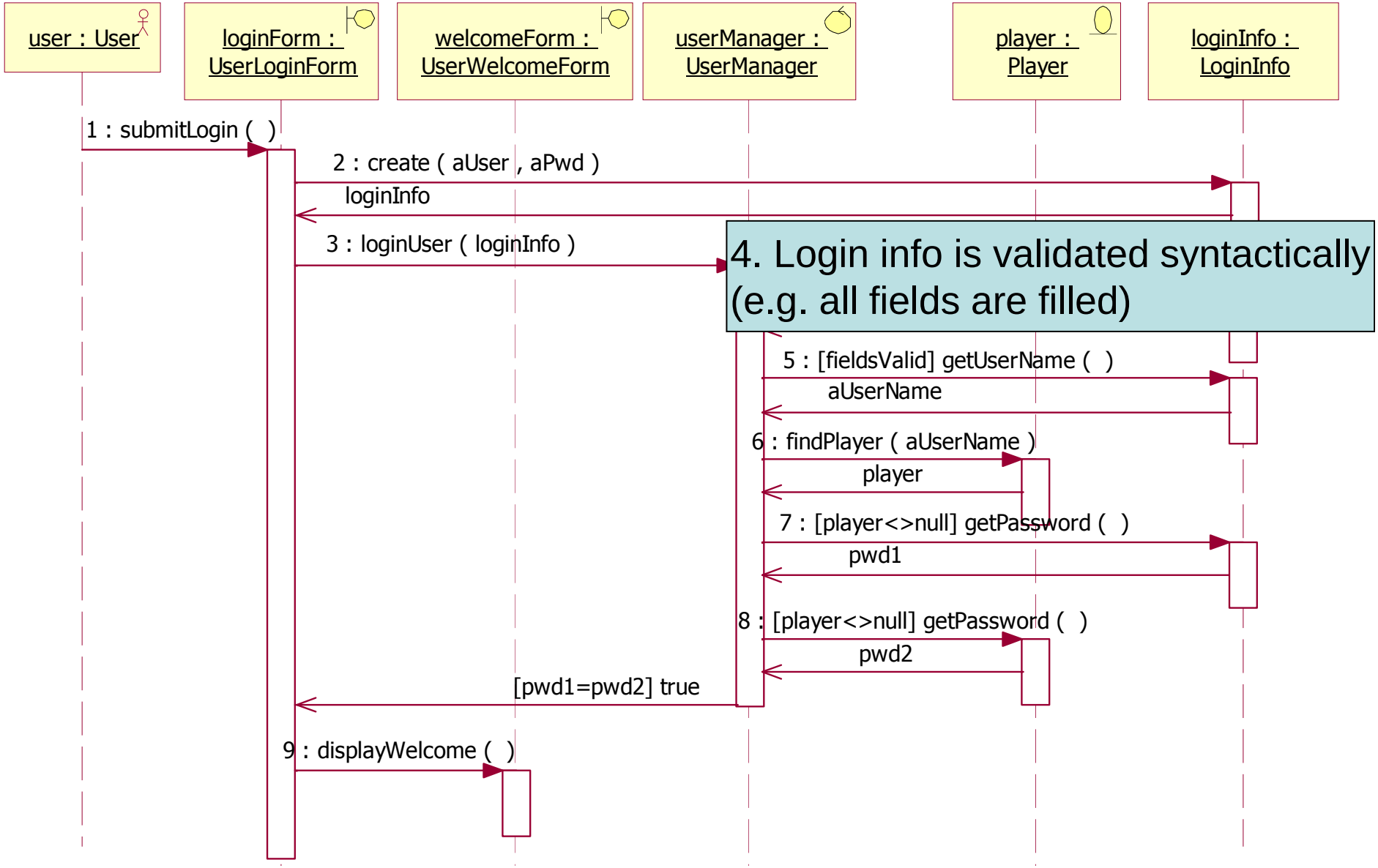
Login User – Basic Flow (Semantics)



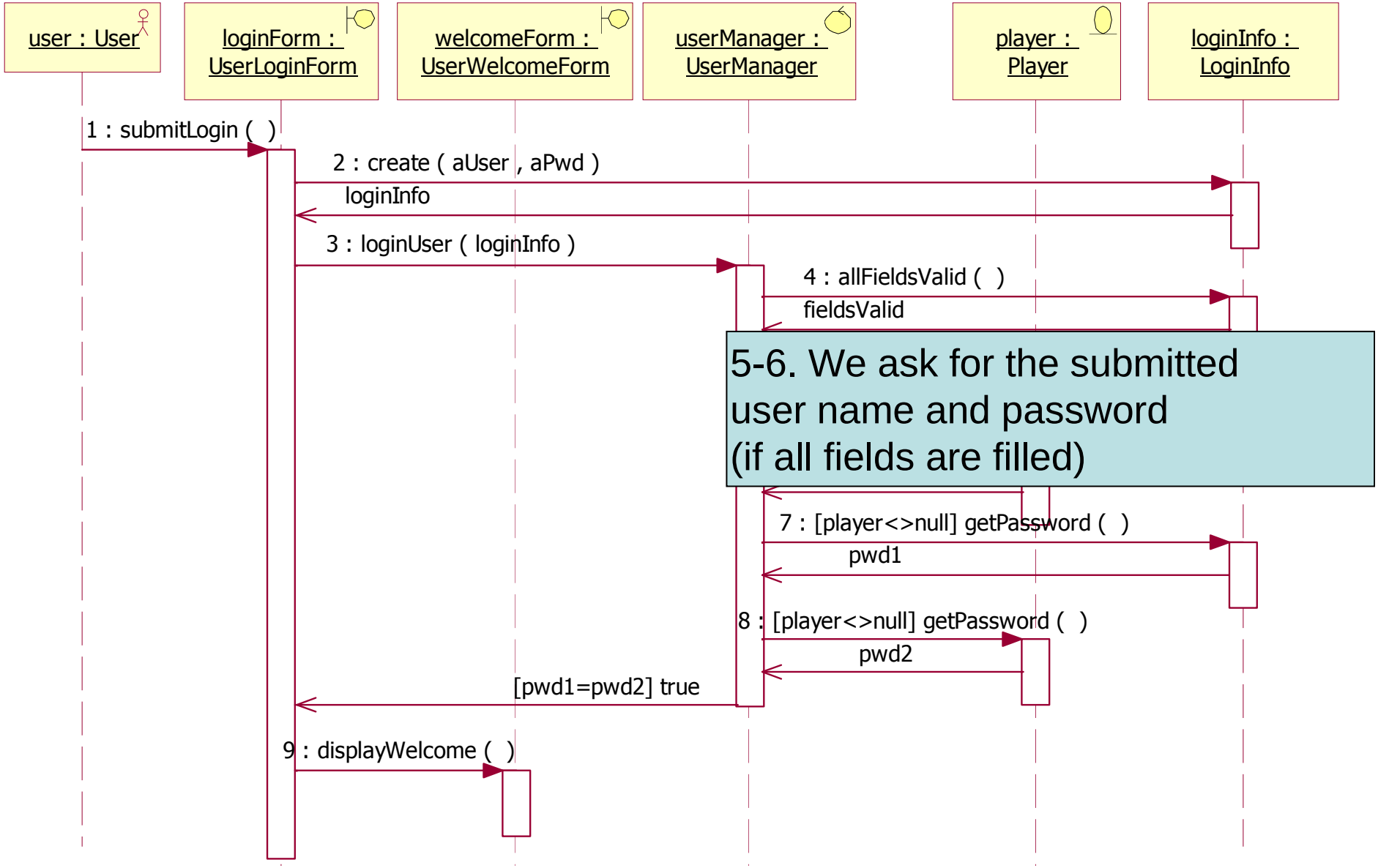
Login User – Basic Flow (Semantics)



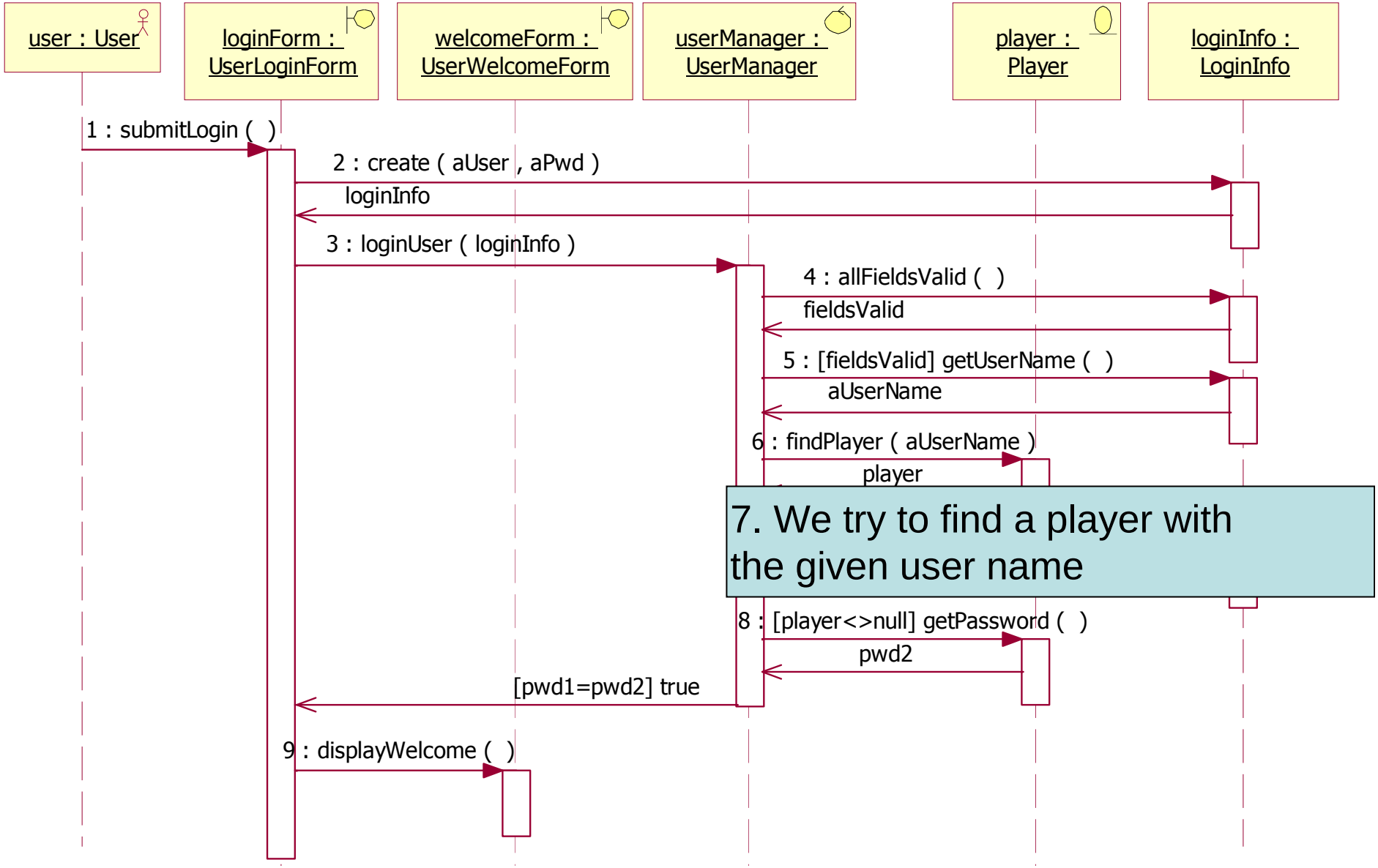
Login User – Basic Flow (Semantics)



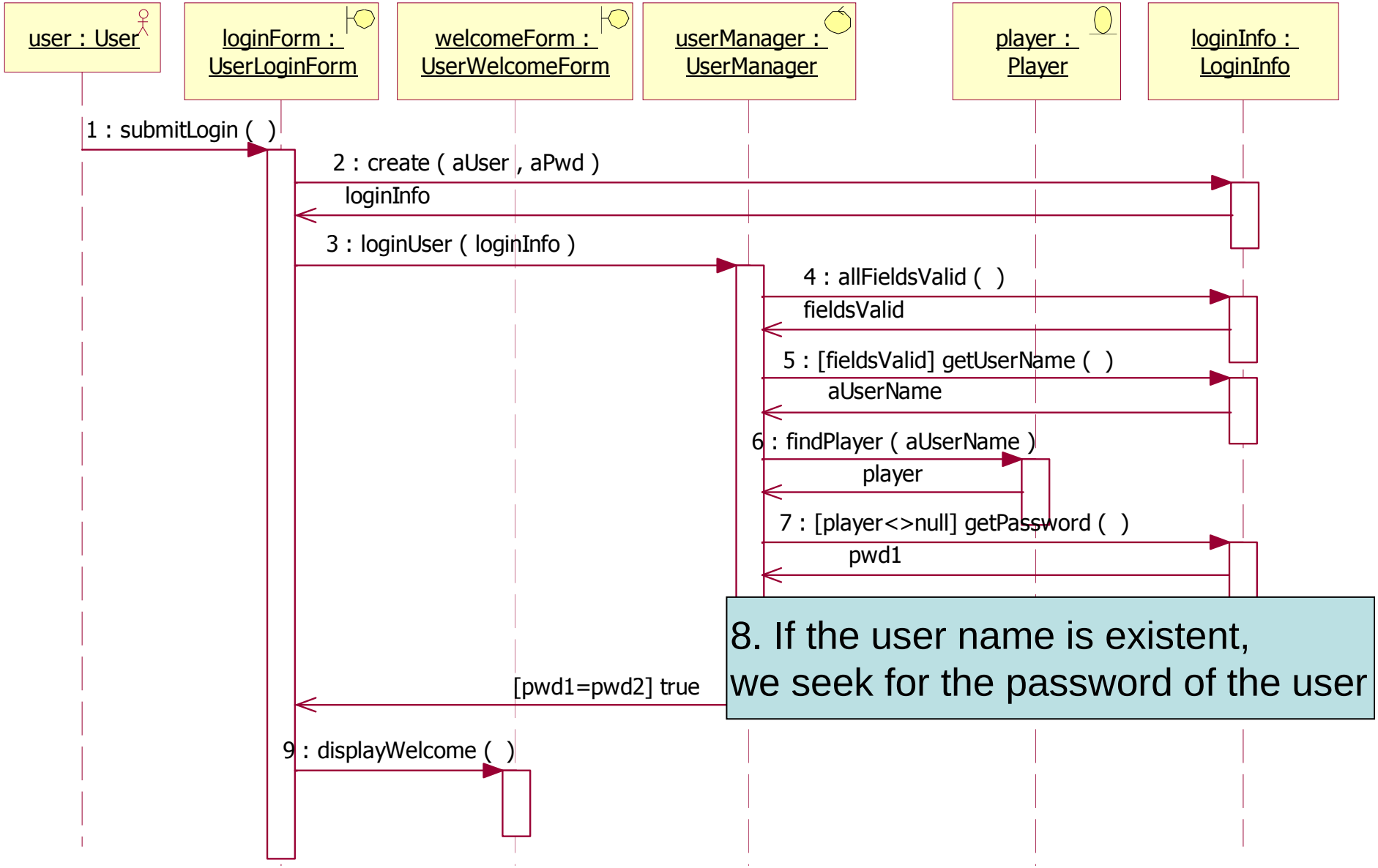
Login User – Basic Flow (Semantics)



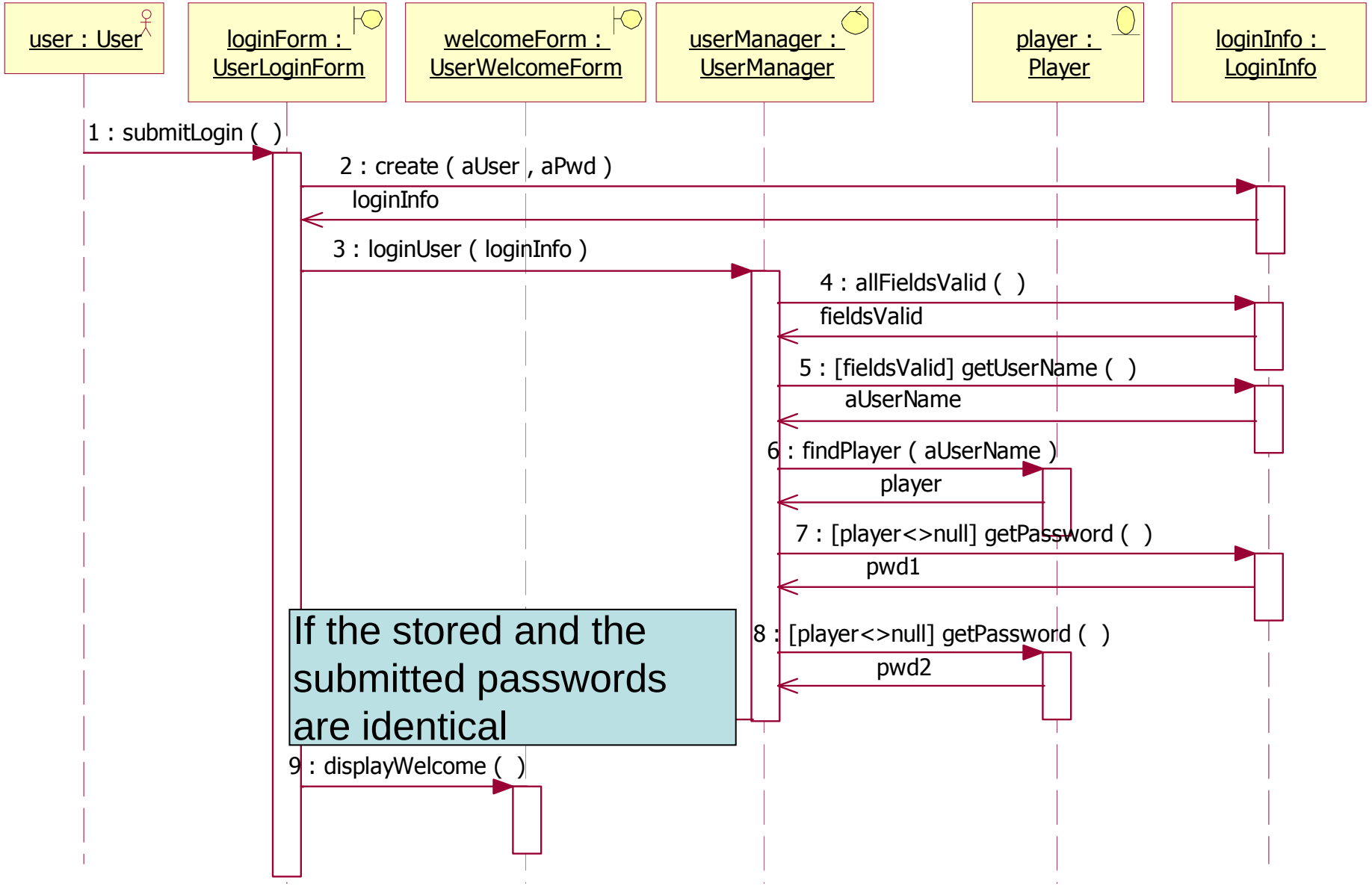
Login User – Basic Flow (Semantics)



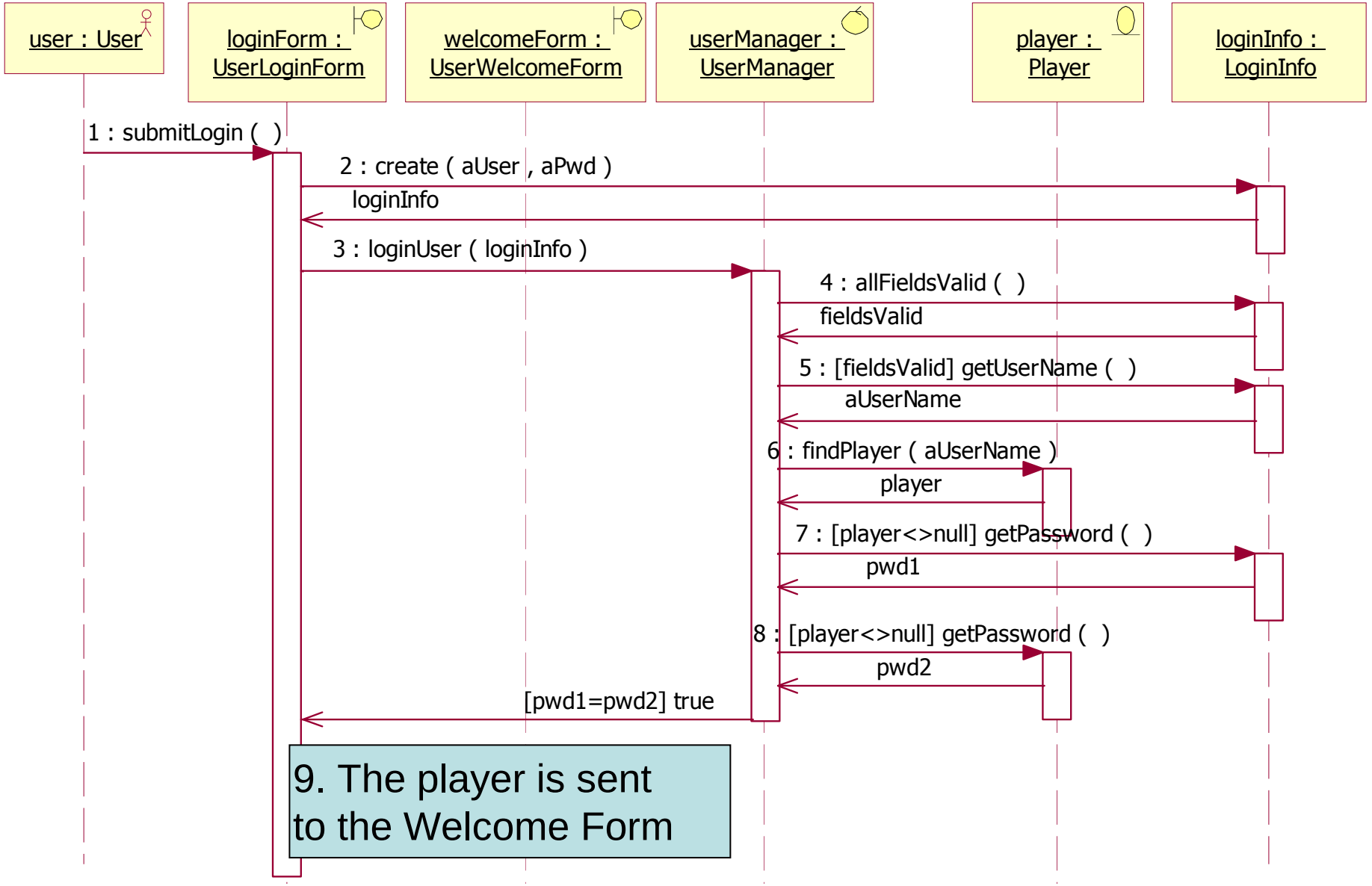
Login User – Basic Flow (Semantics)



Login User – Basic Flow (Semantics)

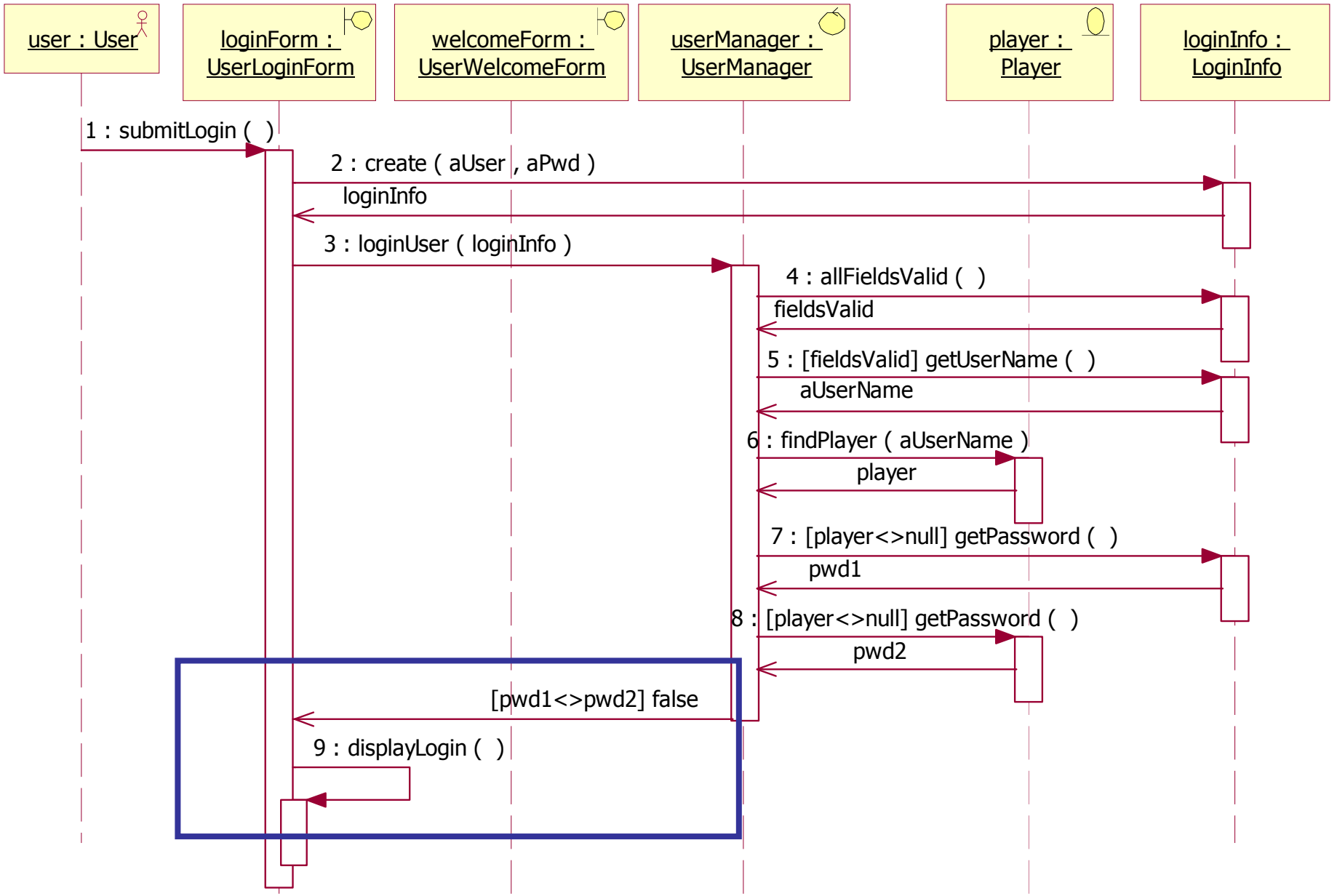


Login User – Basic Flow (Semantics)

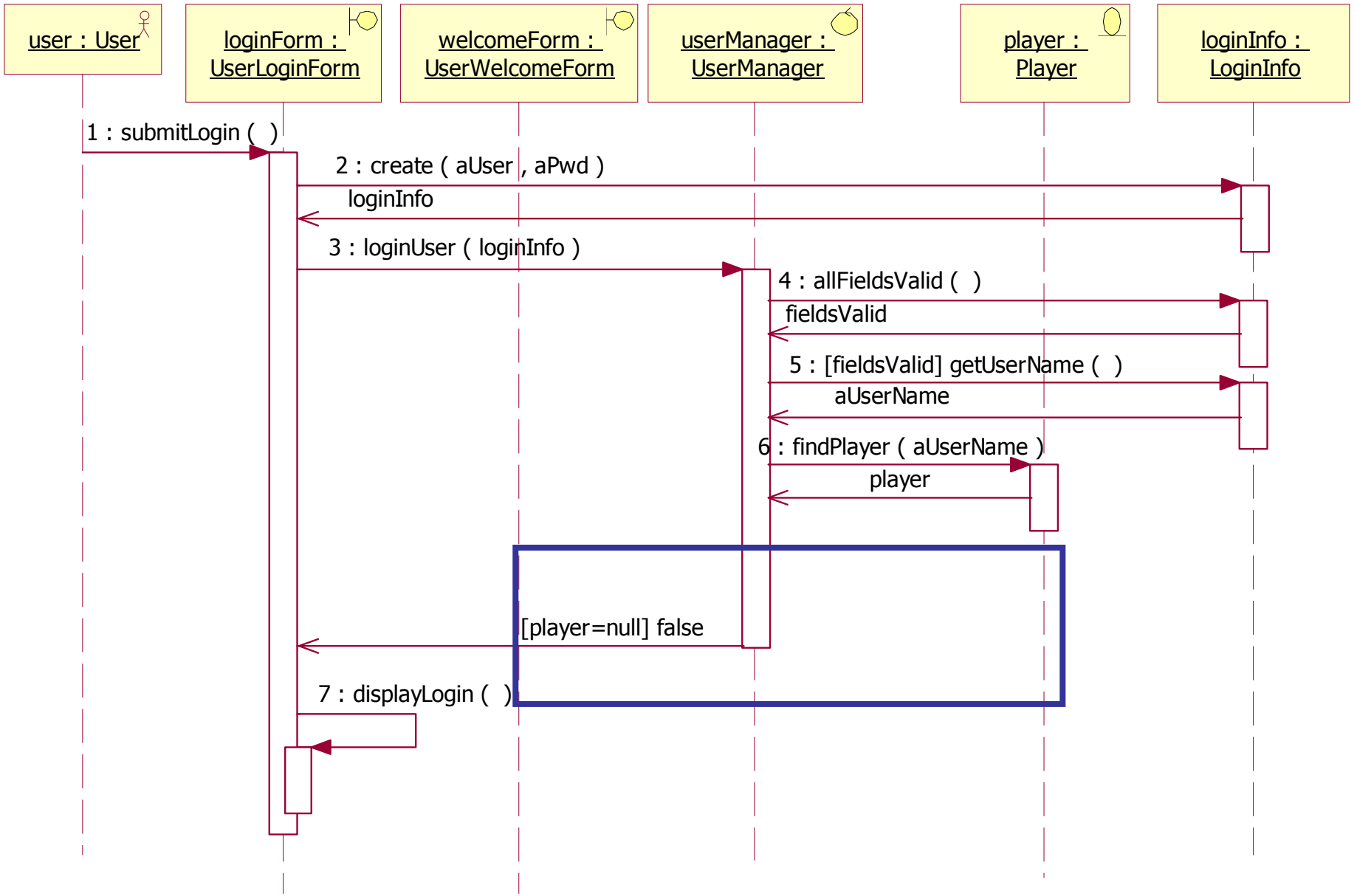


Alternate Flows: Login User

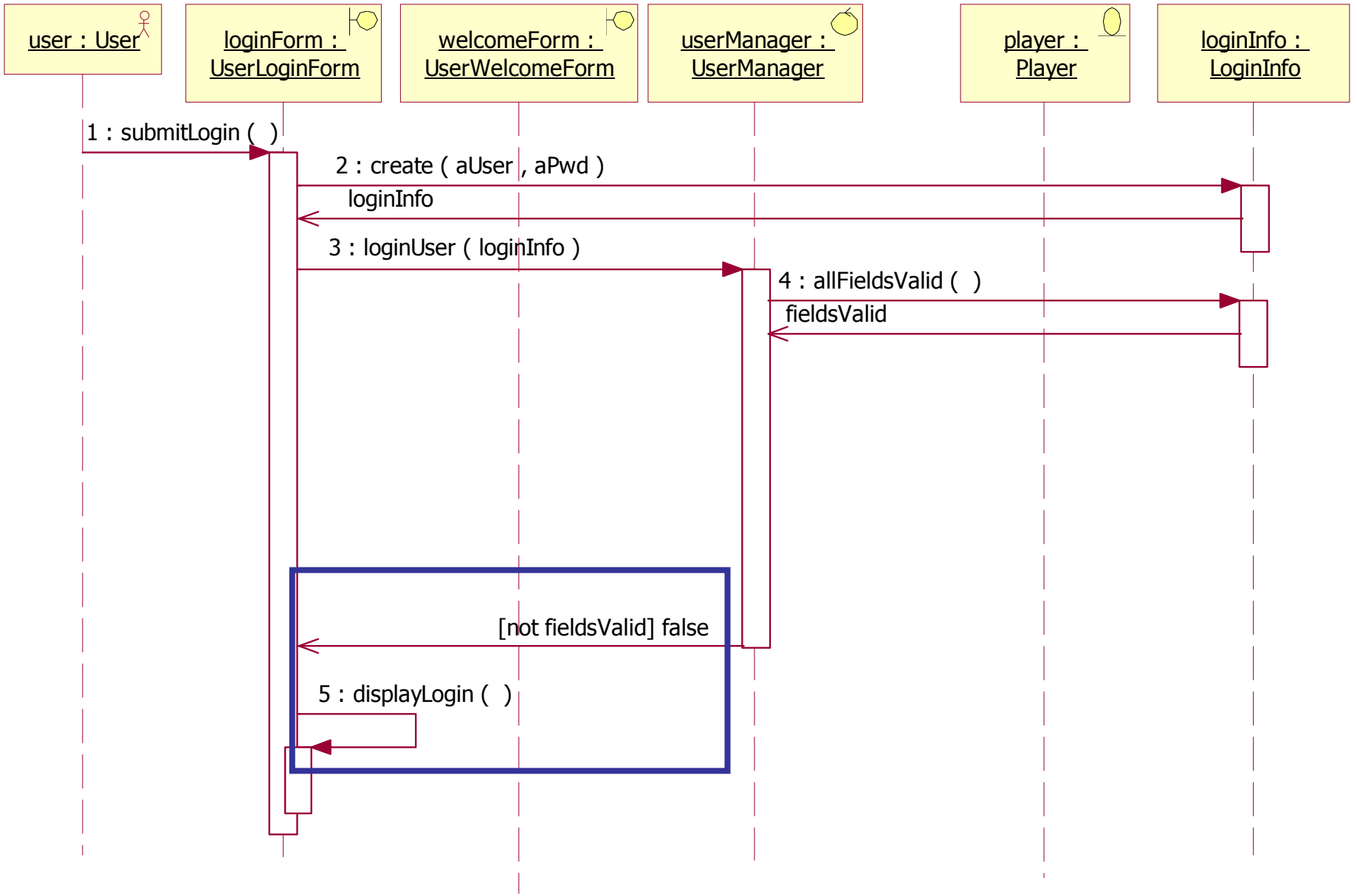
Login User – Incorrect Password



Login User – Non-Existing User



Login User – Invalid Login Info



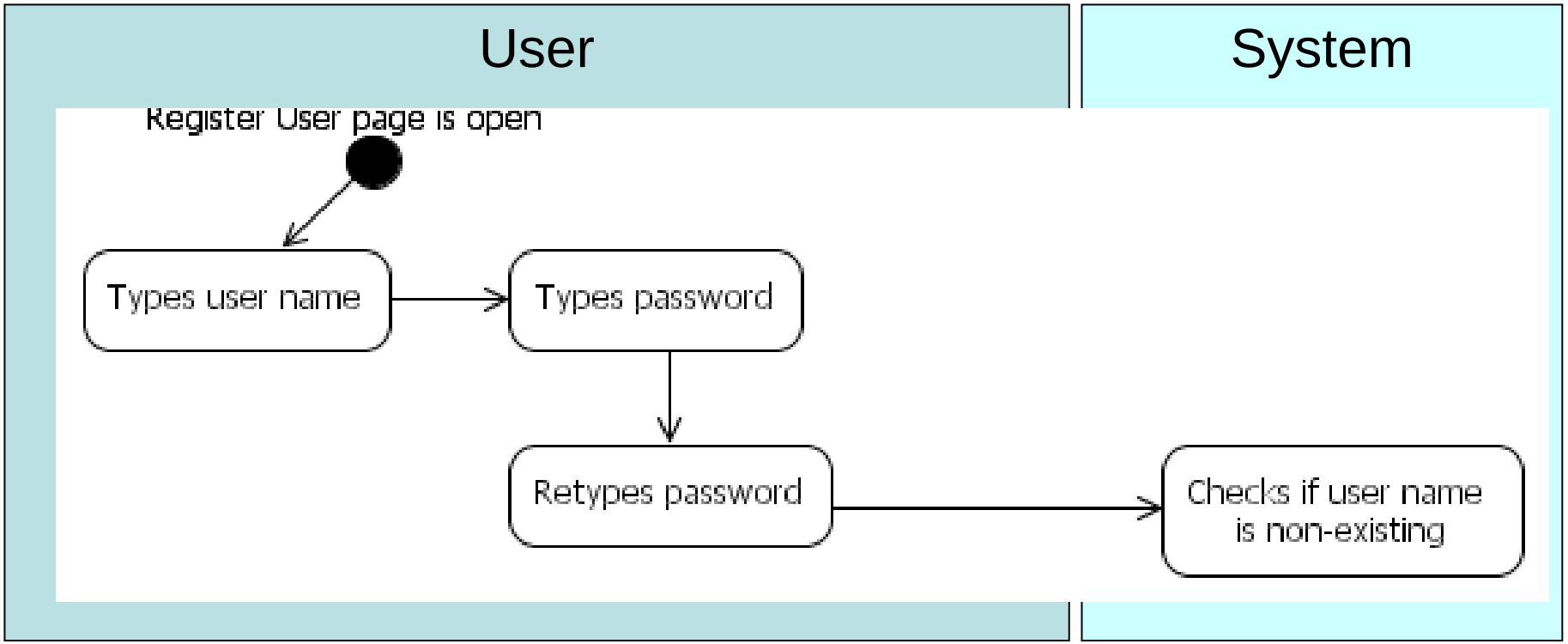
Login User: Implemention in Pseudo Code

```
class LoginForm {
UserWelcomeForm welcomeForm;
void submitLogin() {
    LoginInfo loginInfo =
            LoginInfo.create( … );
    UserManager userManager =
            new UserManager();
    if ( userManager.loginUser(loginInfo))
            welcomeForm.displayWelcome();
    else
            this.displayLogin();
}}
```

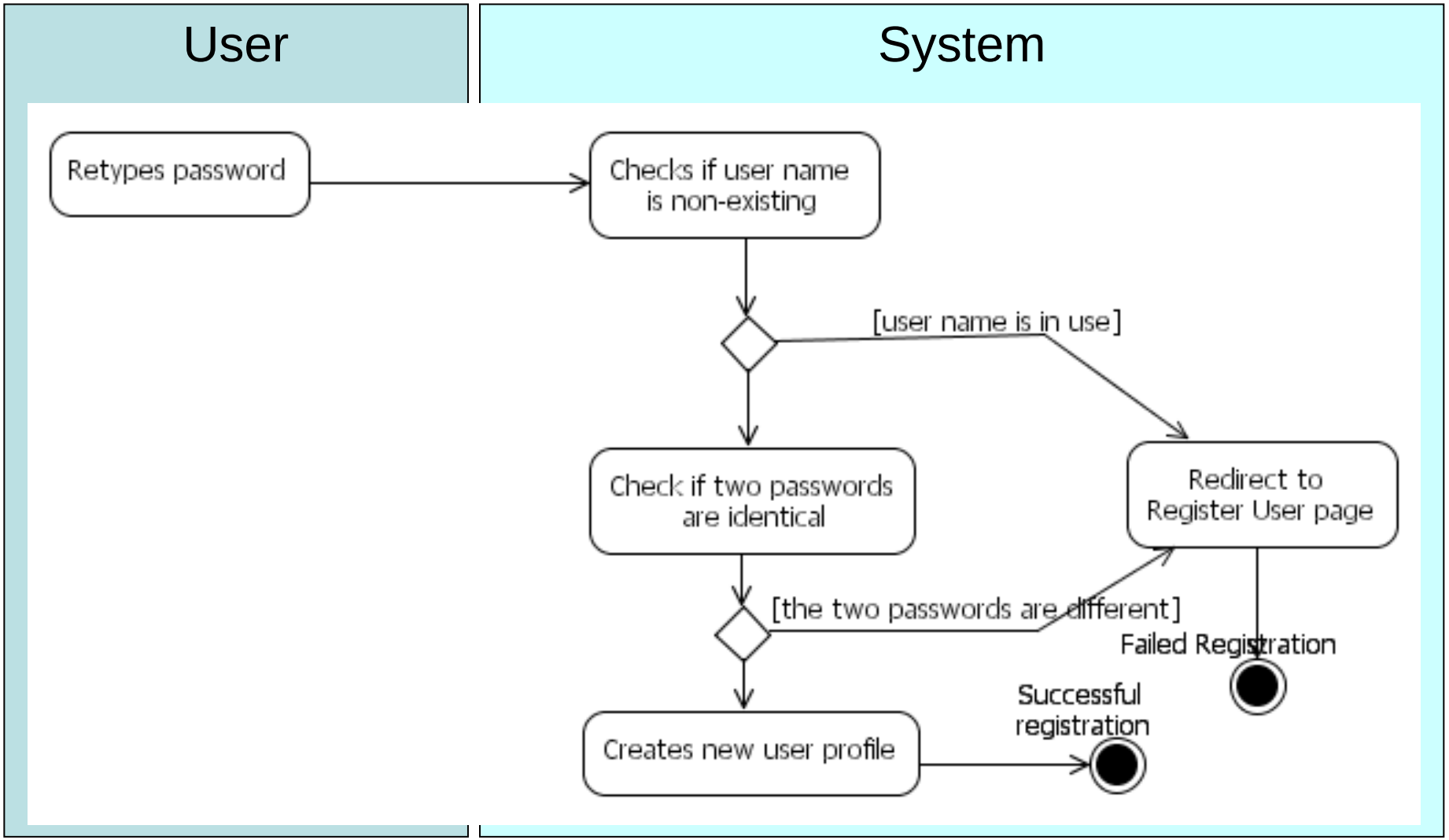
```
class UserManager {
bool loginUser(LoginInfo loginInfo) {
if (loginInfo.allFieldsValid()) {
    aUserName =
    loginInfo.getUserName();
    player = Player.findPlayer(loginInfo);
    if (player <> null) {
            pwd1 = loginInfo.getPassword();
            pwd2 = player.getPassword();
            if (pwd1 = pwd2) return true;
            else return false;
    } else return false;
} else return false;
}}
```

Example: Register User

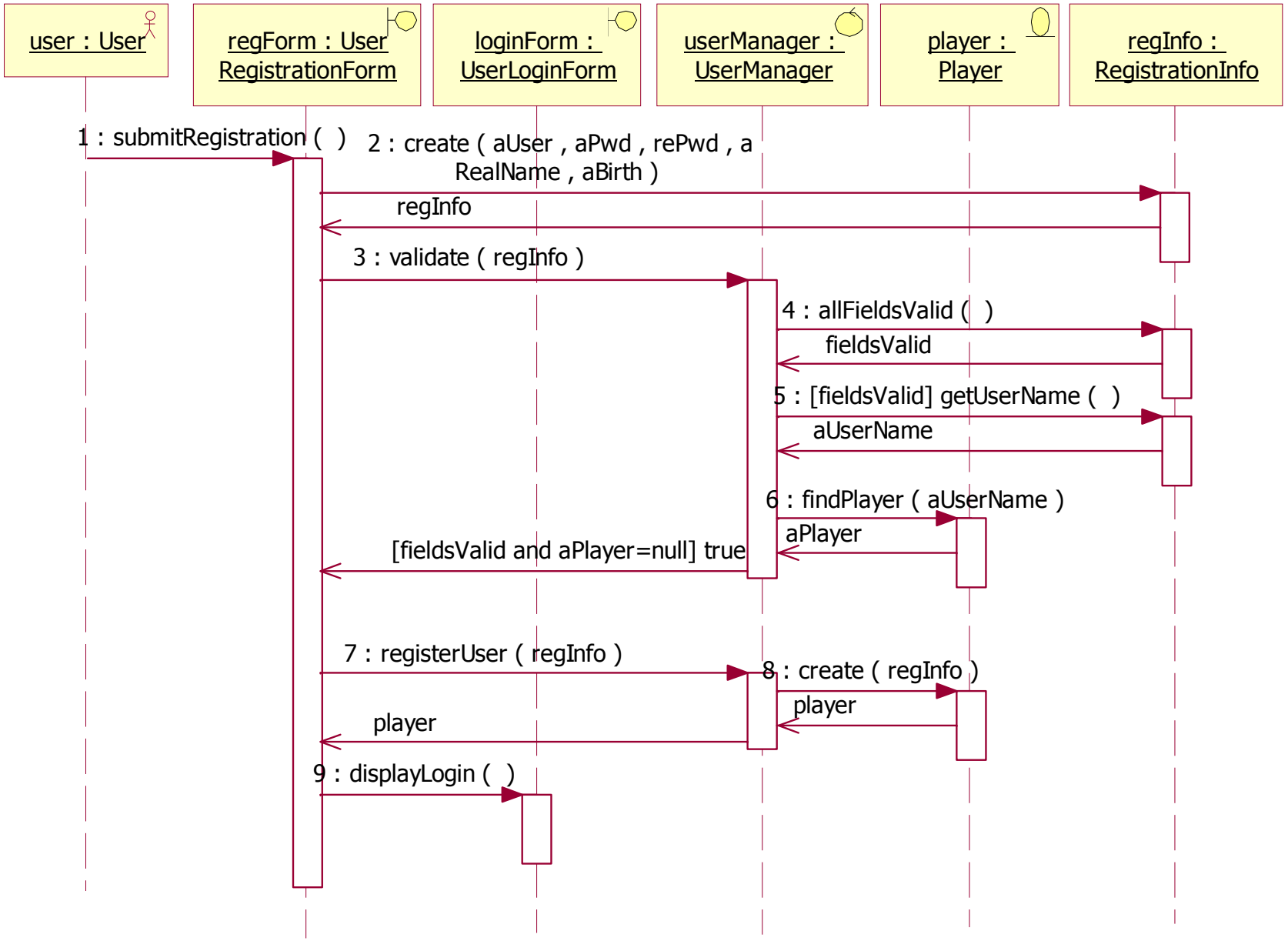
Scenario: Register User I.



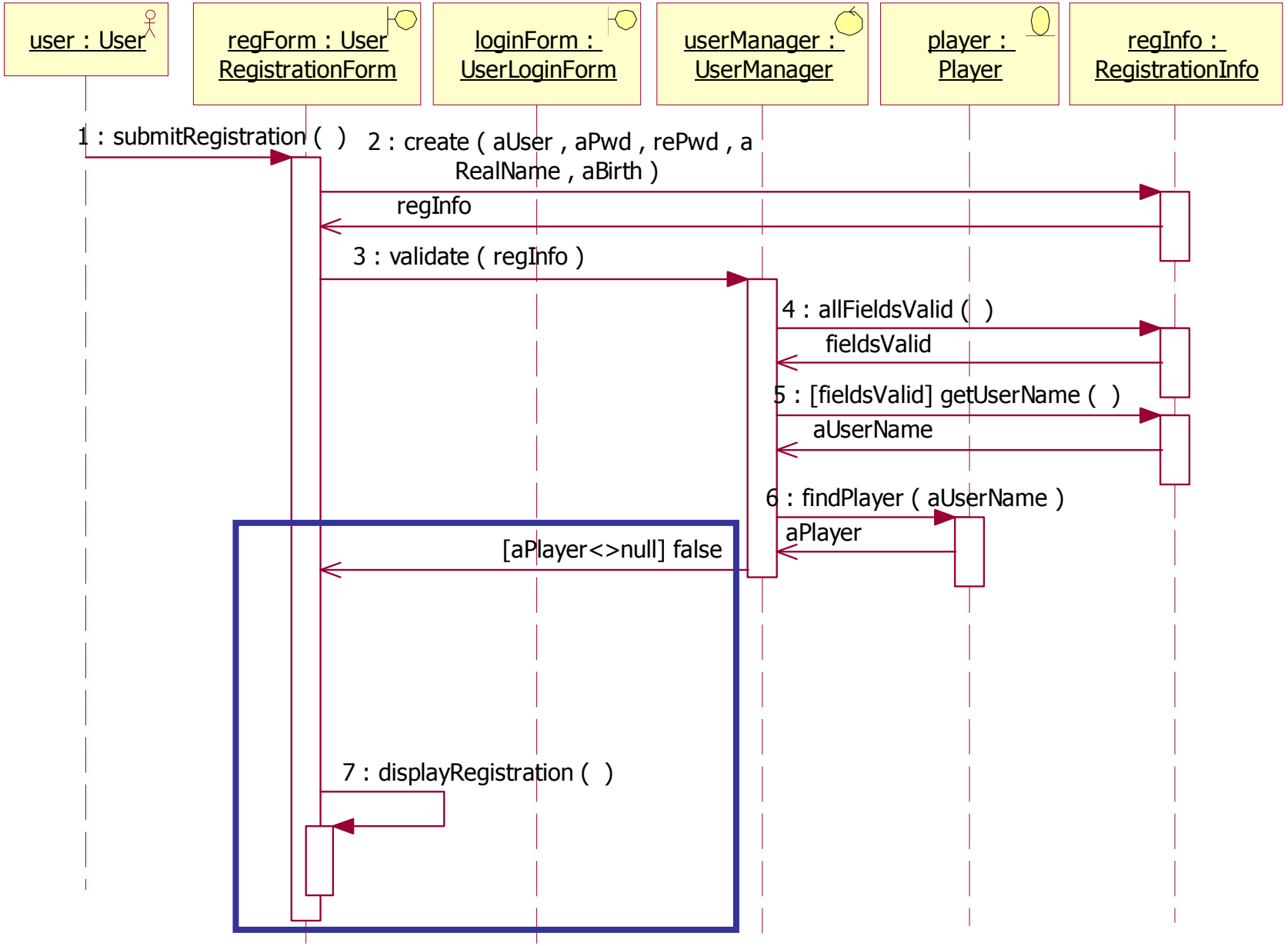
Scenario: Register User II.



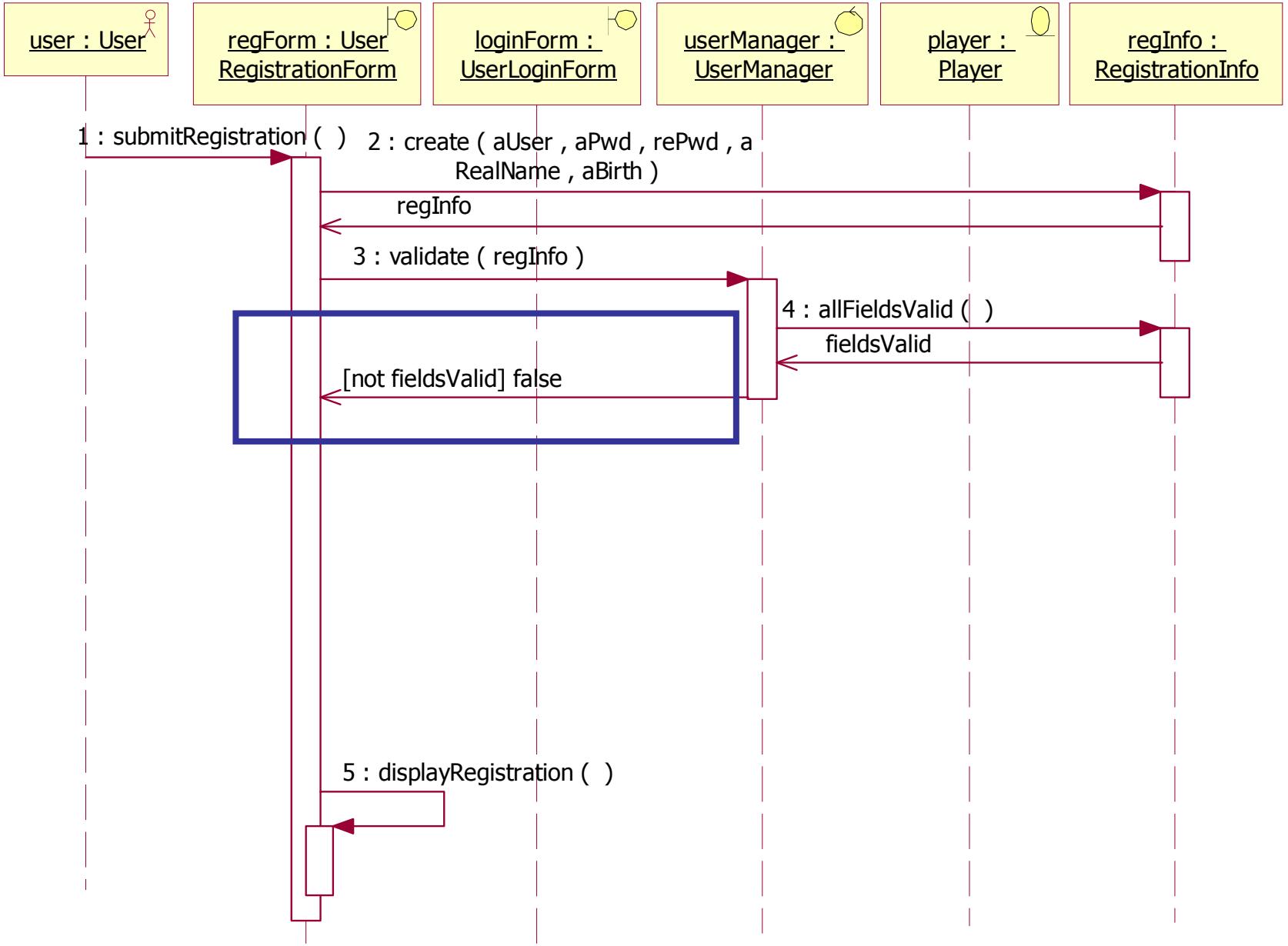
Register User – Basic Flow



Register User – Existing User

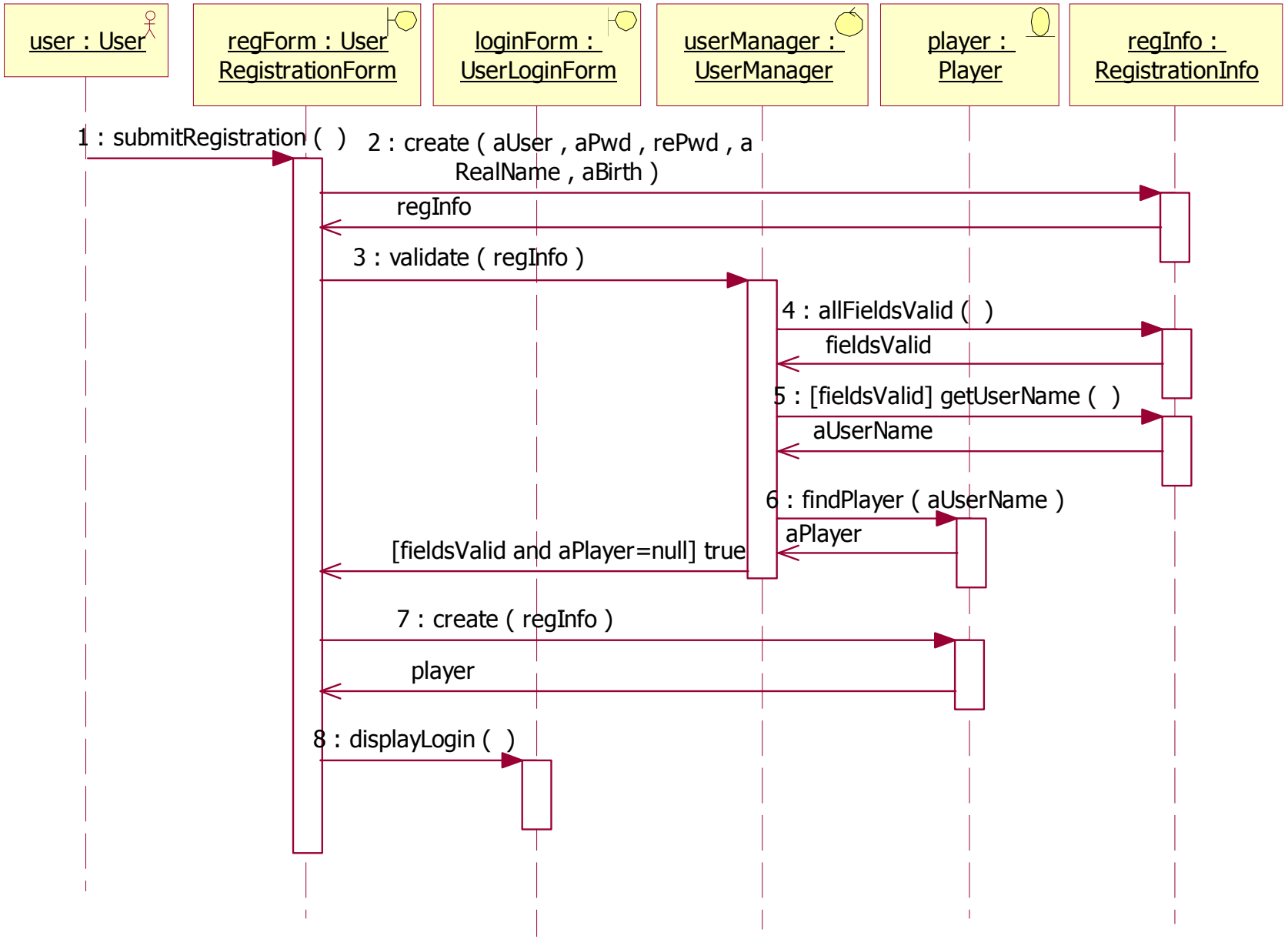


Register User – Invalid Fields

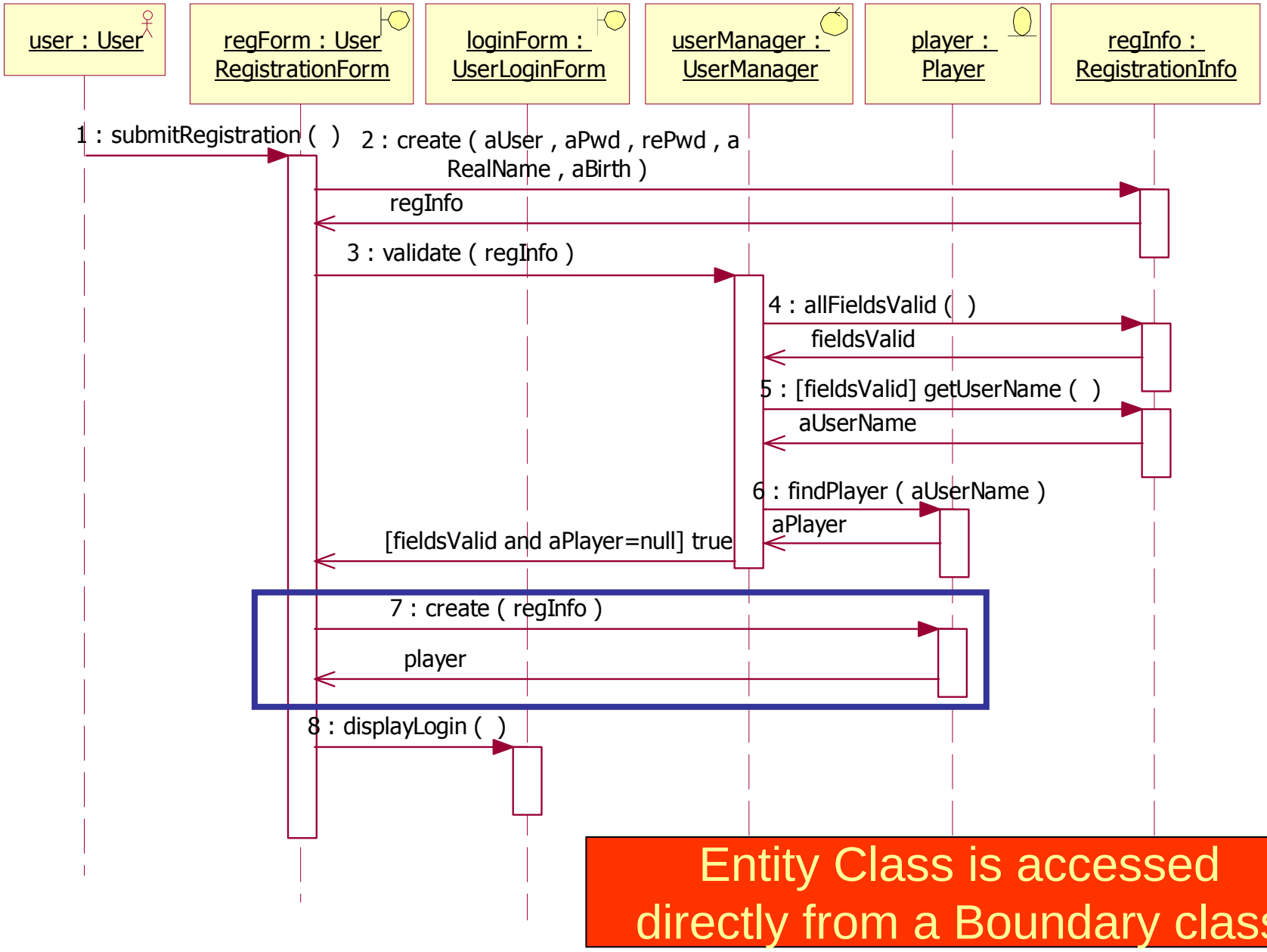


What is wrong?
(Antipatterns)

What is wrong? Register User

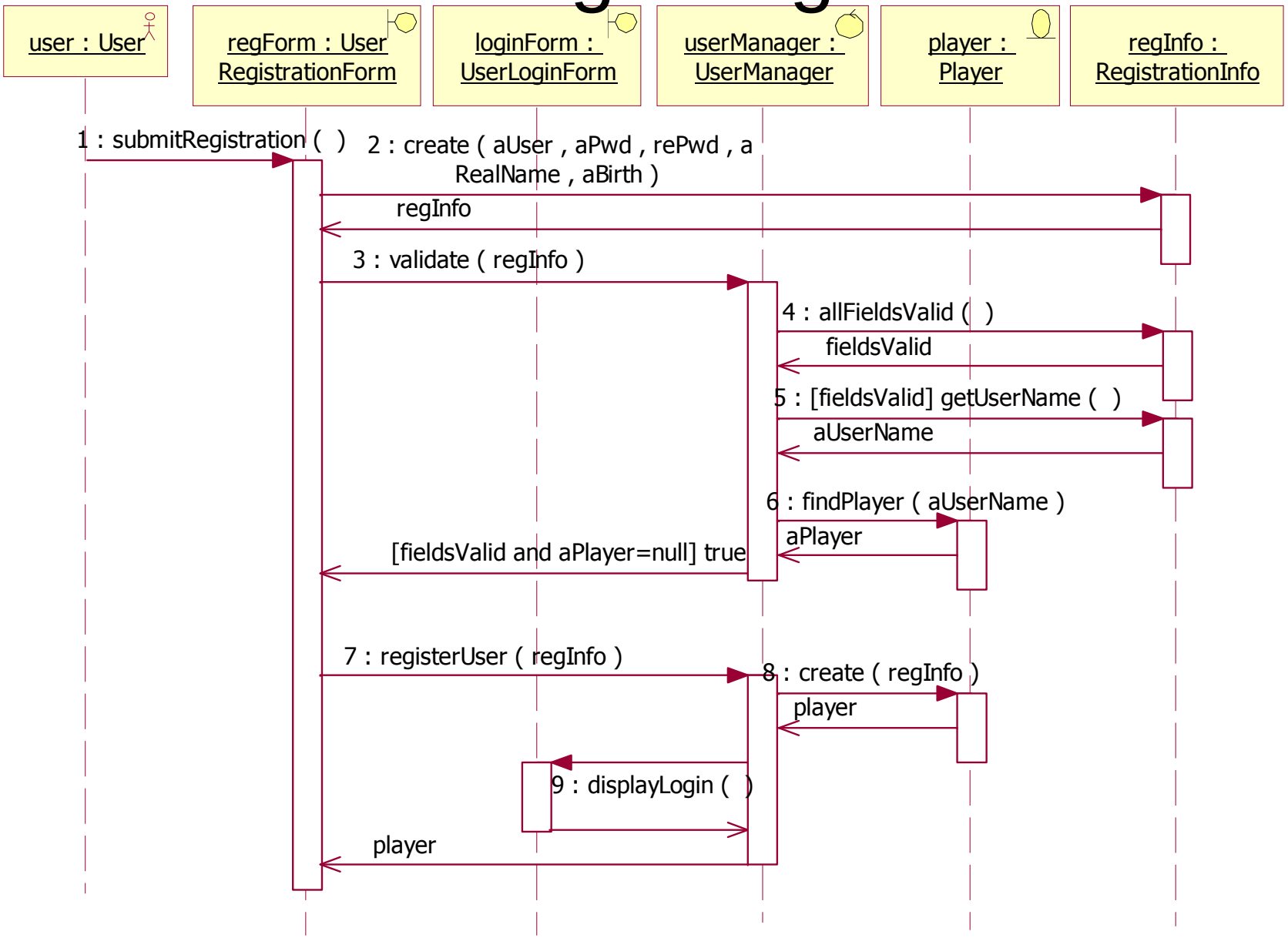


What is wrong? Register User

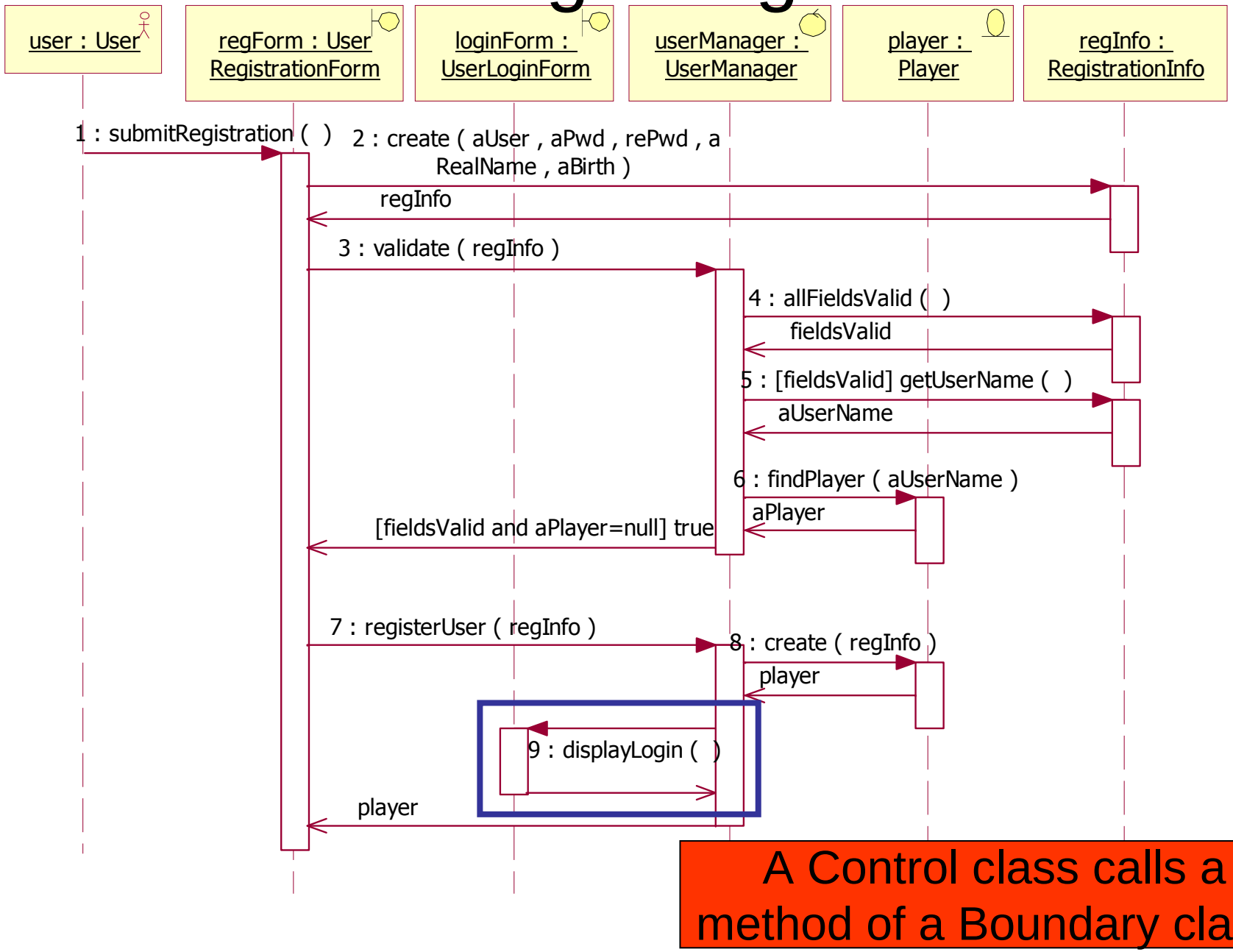


Entity Class is accessed directly from a Boundary class!

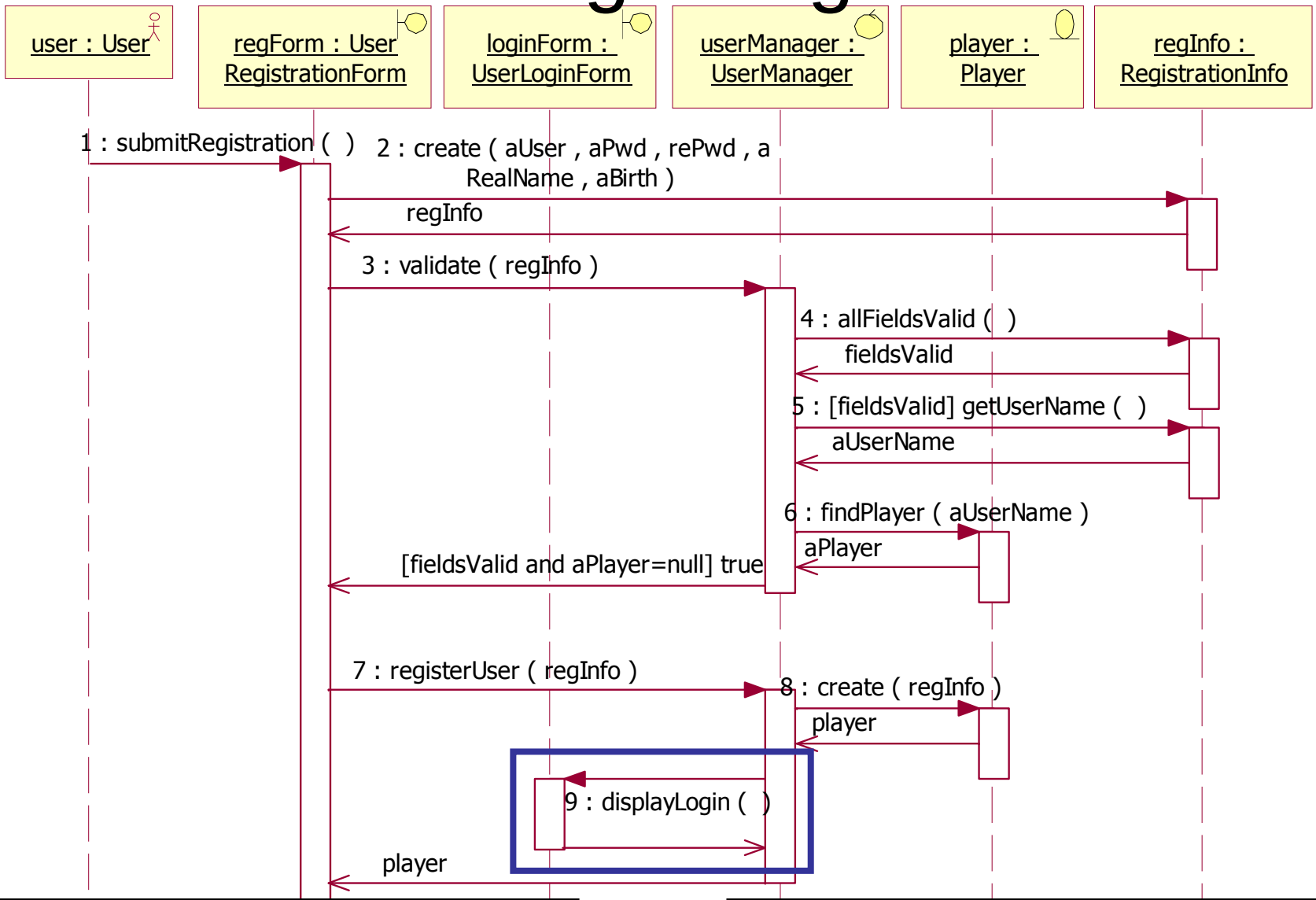
What is wrong? Register User



What is wrong? Register User



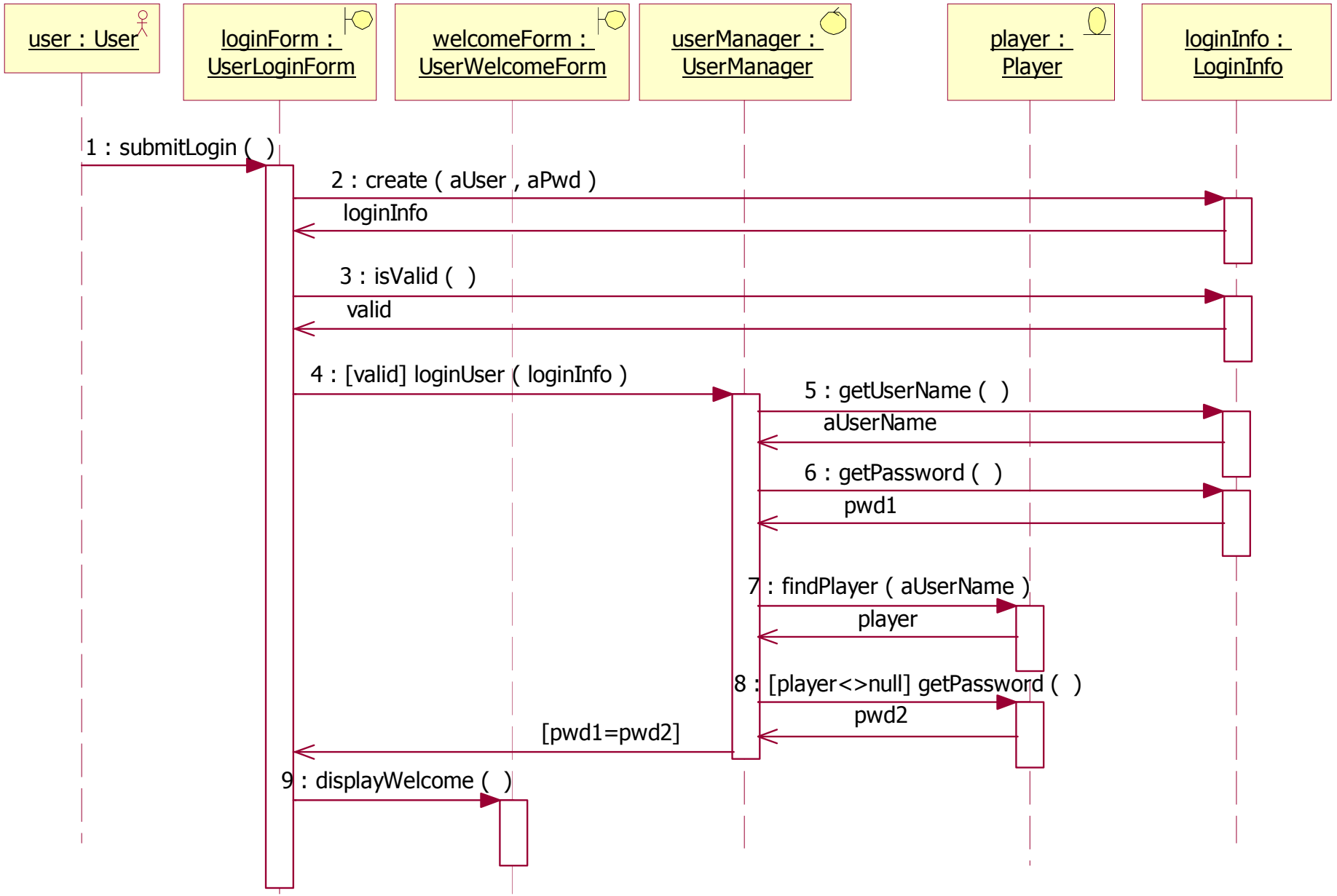
What is wrong? Register User



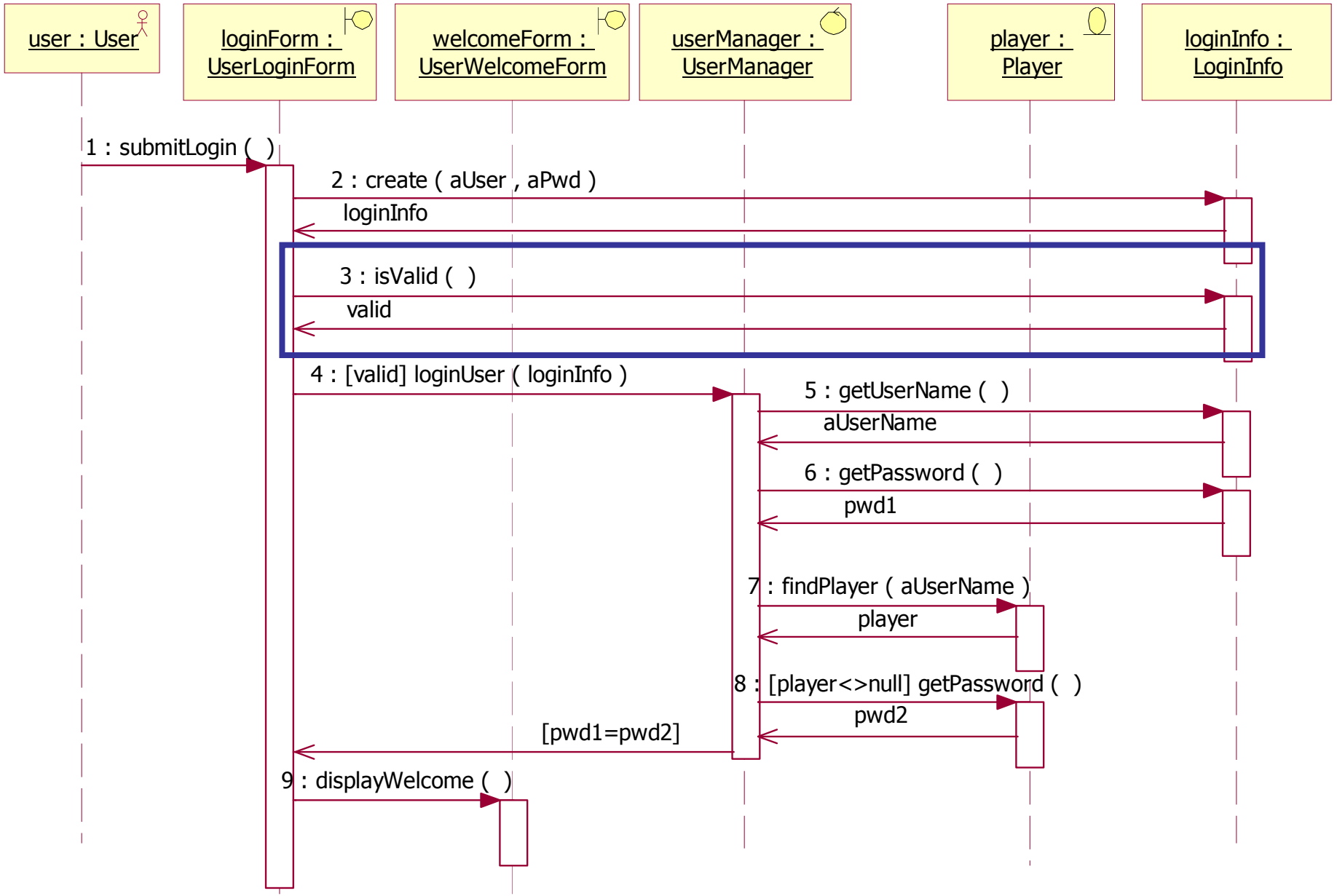
Acceptable only for exception handling (see later)

A Control class calls a method of a Boundary class!

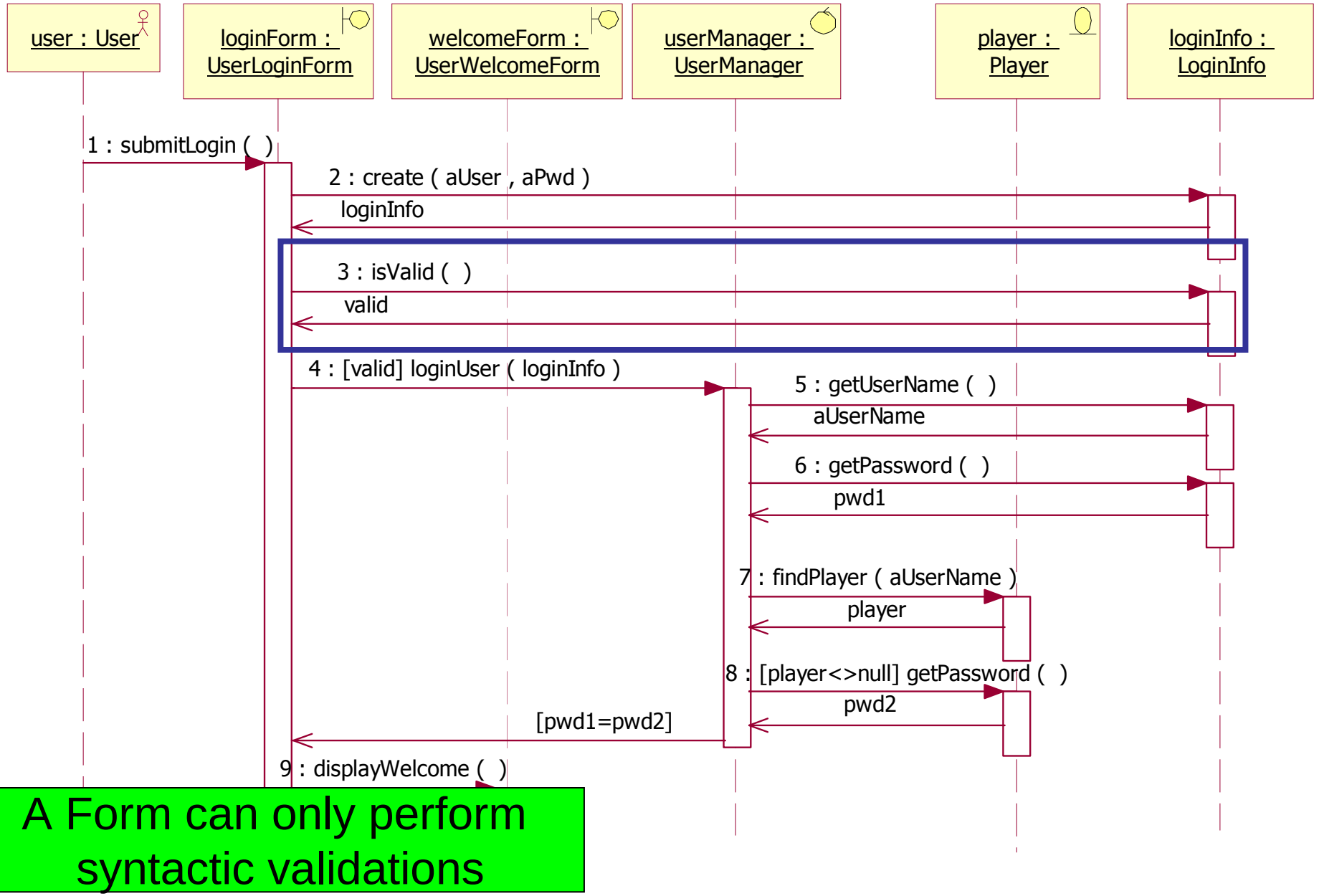
Is this wrong? Login User



Is this wrong? Login User

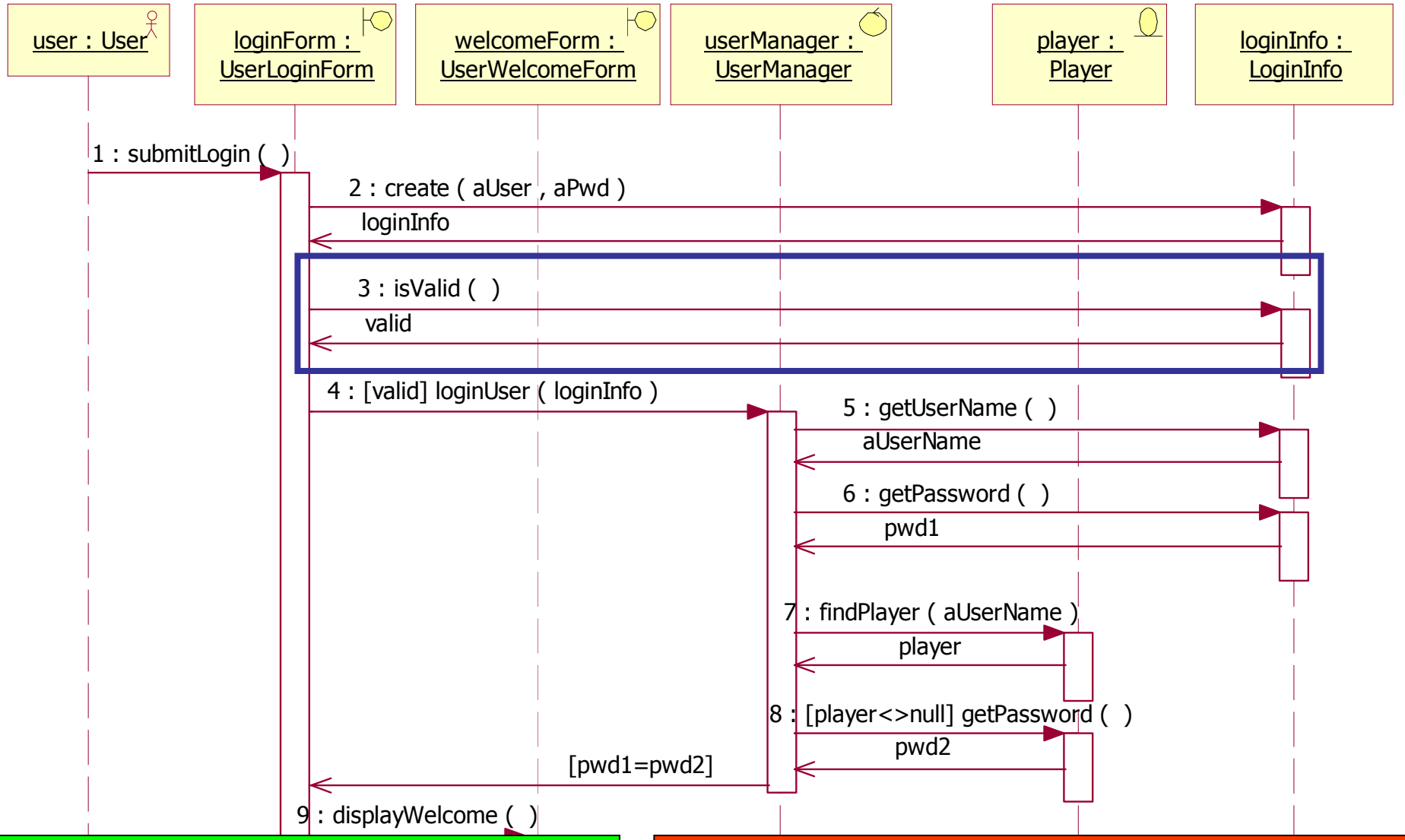


Is this wrong? Login User



A Form can only perform syntactic validations

Is this wrong? Login User

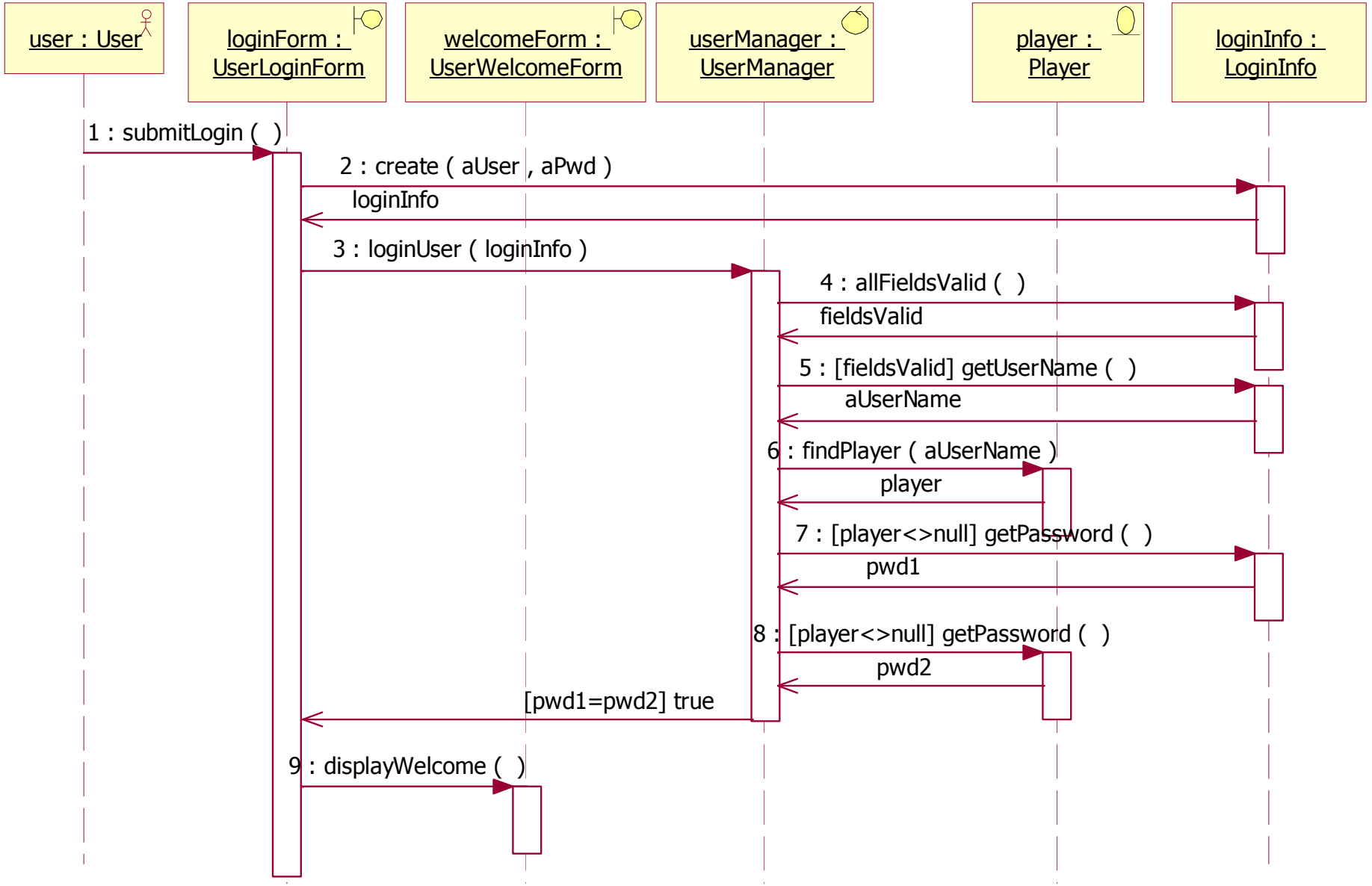


A Form can only perform syntactic validations

Semantic validation should only be accessible from a Control class!

Implementing Interactions (Preview)

Login User – Basic Flow



Login User: Implemention in (Pseudo) Java

```
class LoginForm {
UserWelcomeForm welcomeForm;
void submitLogin() {
    LoginInfo loginInfo =
          LoginInfo.create( … );
    UserManager userManager =
          new UserManager();
    if (userManager.loginUser(loginInfo))
          welcomeForm.displayWelcome();
    else
          this.displayLogin();
}}
```

```
class UserManager {
bool loginUser(LoginInfo loginInfo) {
if (loginInfo.allFieldsValid()) {
    aUserName =
    loginInfo.getUserName();
    player = Player.findPlayer(loginInfo);
    if (player != null) {
          pwd1 = loginInfo.getPassword();
          pwd2 = player.getPassword();
          if (pwd1 == pwd2) return true;
          else return false;
    } else return false;
} else return false;
}}
```

Login User: Implemention in (Pseudo) Java

```
class LoginForm {
UserWelcomeForm welcomeForm;
void submitLogin() {
    LoginInfo loginInfo =
        LoginInfo.create( … );
    UserManager userManager =
        new UserManager();
    if (userManager.loginUser(loginInfo))
        welcomeForm.displayWelcome();
    else
        this.displayLogin();
}}
```

```
class UserManager {
bool loginUser(LoginInfo loginInfo) {
if (loginInfo.allFieldsValid()) {
    aUserName =
    loginInfo.getUserName();
    player = Player.findPlayer(loginInfo);
    if (player != null) {
        pwd1 = loginInfo.getPassword();
        pwd2 = player.getPassword();
        if (pwd1 == pwd2) return true;
        else return false;
    } else return false;
} else return false;
}}
```

Problem:
No information is returned what went wrong if login fails

Login User: Implemention with Exceptions

```
class LoginForm {
UserWelcomeForm welcomeForm;
void submitLogin() {
    LoginInfo loginInfo =
          LoginInfo.create( … );
    UserManager userManager =
          new UserManager();
    try {
      userManager.loginUser(loginInfo);
      welcomeForm.displayWelcome();
    } catch (LoginException e) {
      this.displayLogin(e.getMessage());
    }
}}
```

```
class UserManager {
bool loginUser(LoginInfo loginInfo)
    throws LoginException {
if (loginInfo.allFieldsValid()) {
    aUserName =
    loginInfo.getUserName();
    player = Player.findPlayer(loginInfo);
    if (player != null) {
          pwd1 = loginInfo.getPassword();
          pwd2 = player.getPassword();
          if (pwd1 == pwd2) return true;
          else throw new LoginException("
          Invalid password");
    } else throw new LoginException("
          Non-existing user name");
} else throw new LoginException("
          Invalid field");
}}
```

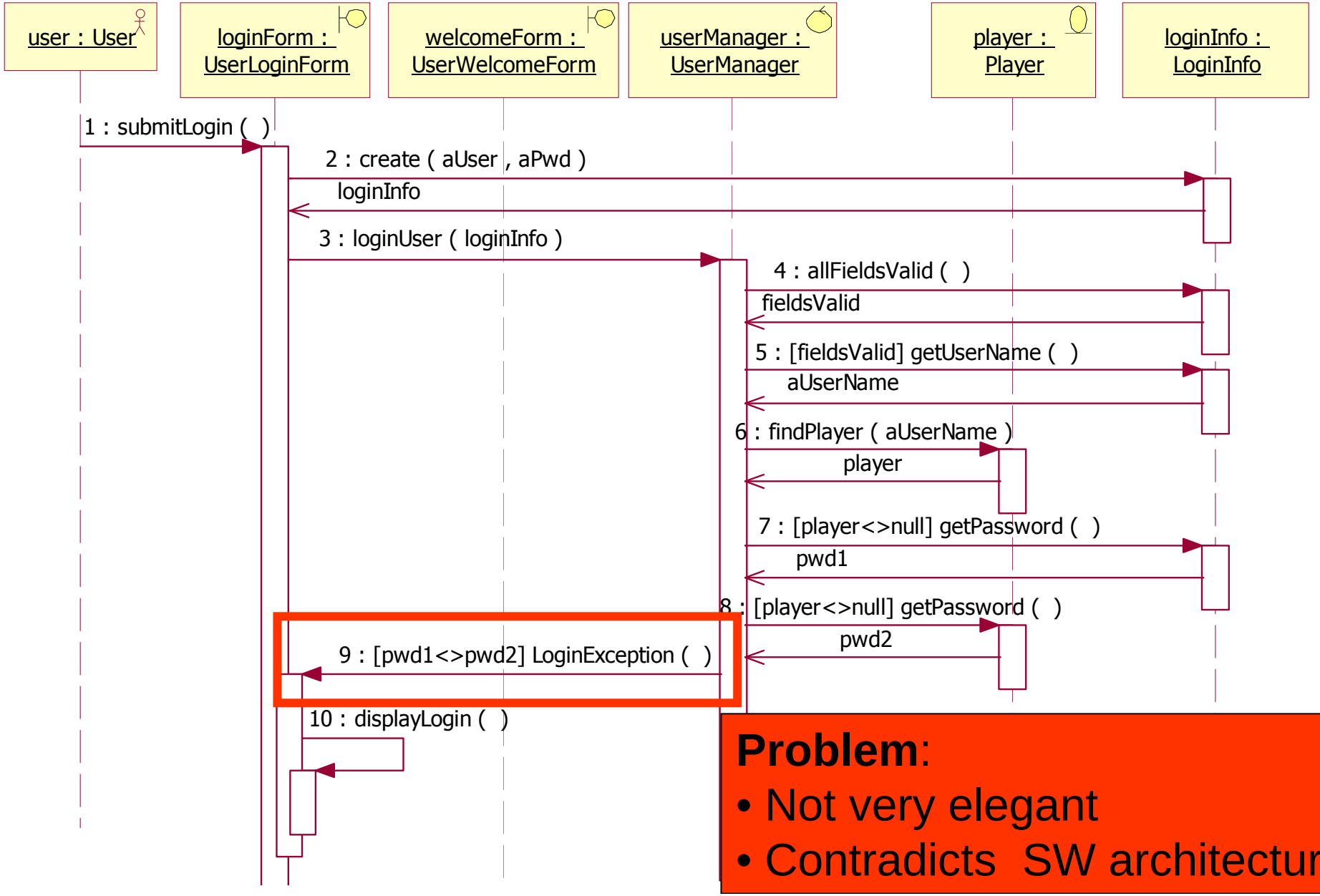
Login User: Implemention with Exceptions

```
class LoginForm {
UserWelcomeForm welcomeForm;
void submitLogin() {
    LoginInfo loginInfo =
          LoginInfo.create( … );
    UserManager userManager =
          new UserManager();
    try {
      userManager.loginUser(loginInfo);
      welcomeForm.displayWelcome();
    } catch (LoginException e) {
      this.displayLogin(e.getMessage());
    }
}}
```

Question:
How to model exceptions in sequence diagrams?

```
class UserManager {
bool loginUser(LoginInfo loginInfo)
    throws LoginException {
if (loginInfo.allFieldsValid()) {
    aUserName =
    loginInfo.getUserName();
    player = Player.findPlayer(loginInfo);
    if (player != null) {
          pwd1 = loginInfo.getPassword();
          pwd2 = player.getPassword();
          if (pwd1 == pwd2) return true;
          else throw new LoginException("
          Invalid password");
    } else throw new LoginException("
          Non-existing user name");
} else throw new LoginException("
          Invalid field");
}}
```

Login User with Exception



UML 2.0 Sequence Diagrams

Problems of Sequence Diagrams

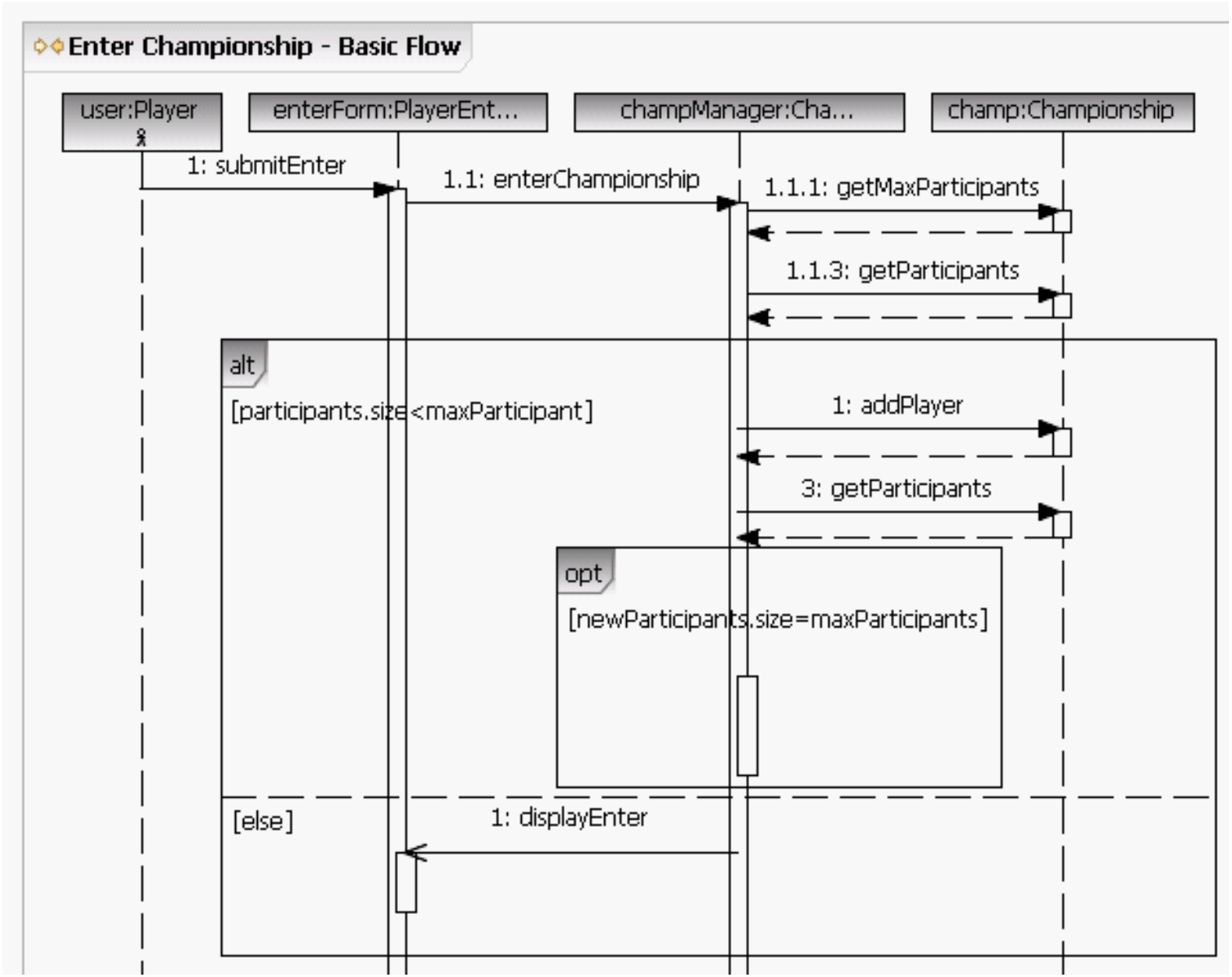
Problems of Sequence Diagrams

- Problems: How to define
  - potential behavior?
  - mandatory behavior?
  - negative (forbidden) behavior?
  - iterations?
  - refinement?
  - incremental development?
- → **UML 2.0 Sequence Diagrams**

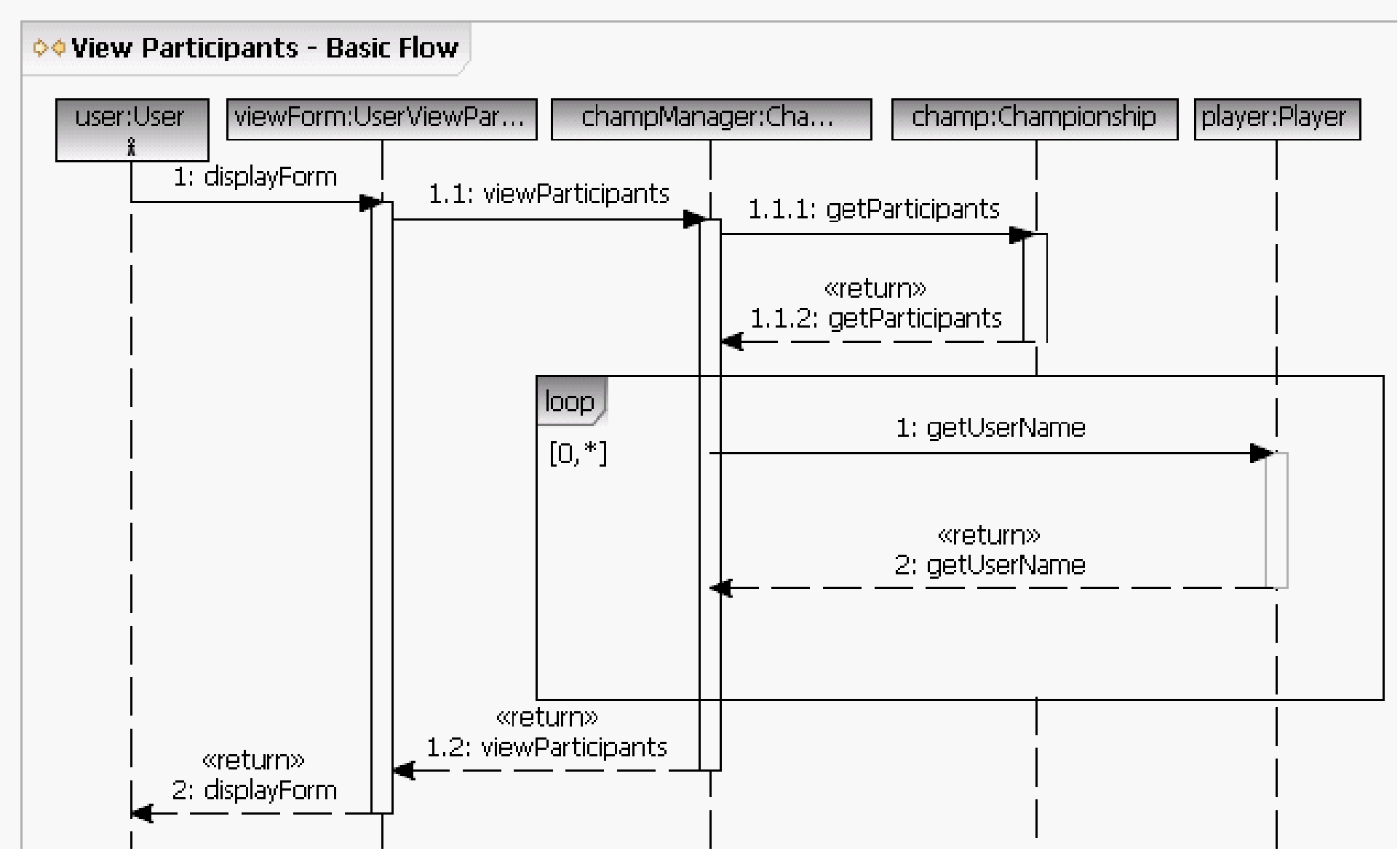
Basic Notions

- Event:
  - Sending a message
  - Receiving a message
- The sender and the receiver are independent in general, but
  - **Send before receive**: The sending of an event should precede the receiving of the same message
  - **Swimlane is ordered**:
    Ordering of messages on a swimlane: if a message is higher than another then it precedes the other
- Trace: a sequence of events respecting the precedence relation (a run of the system)

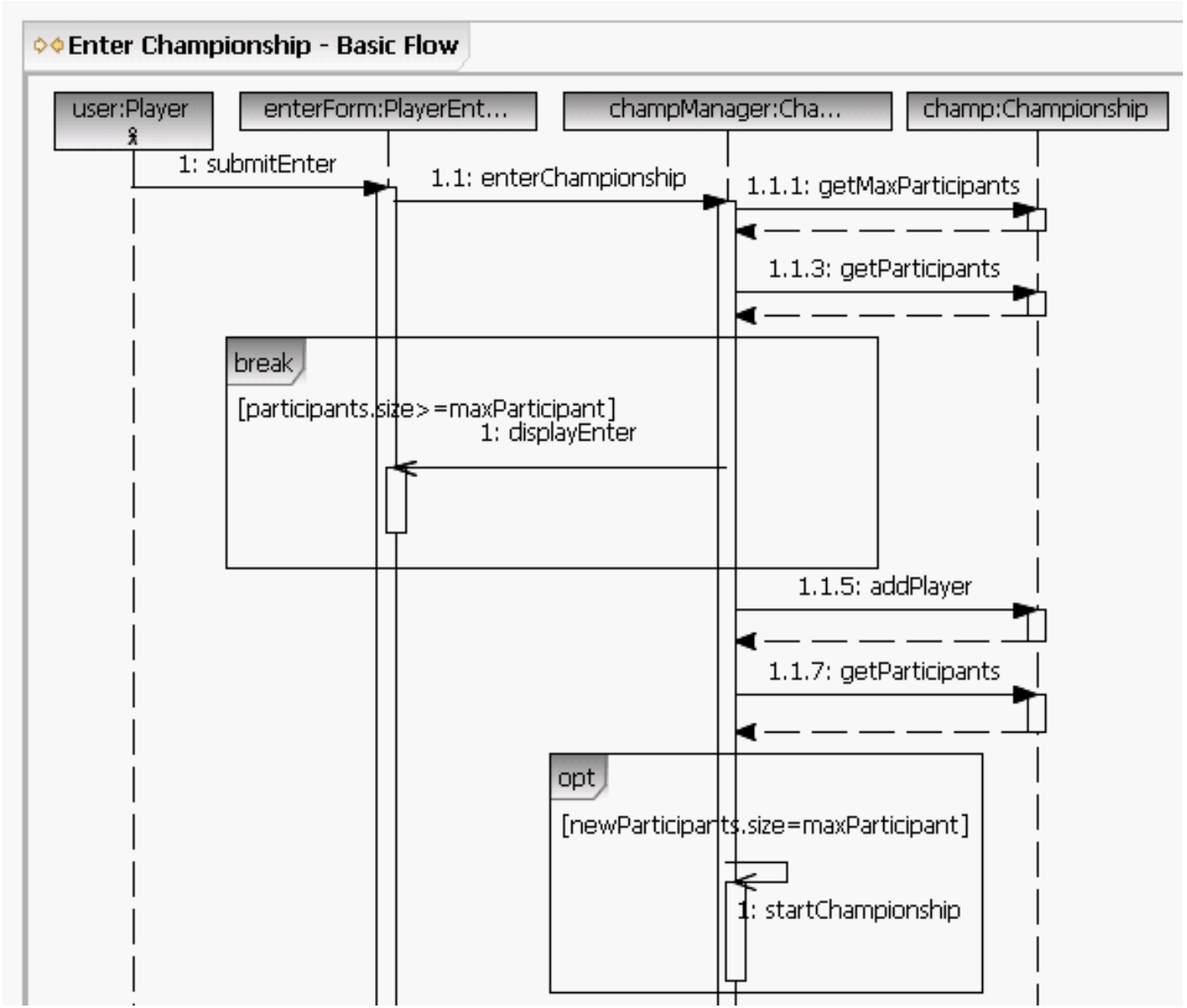
Alternative and Optional Fragment



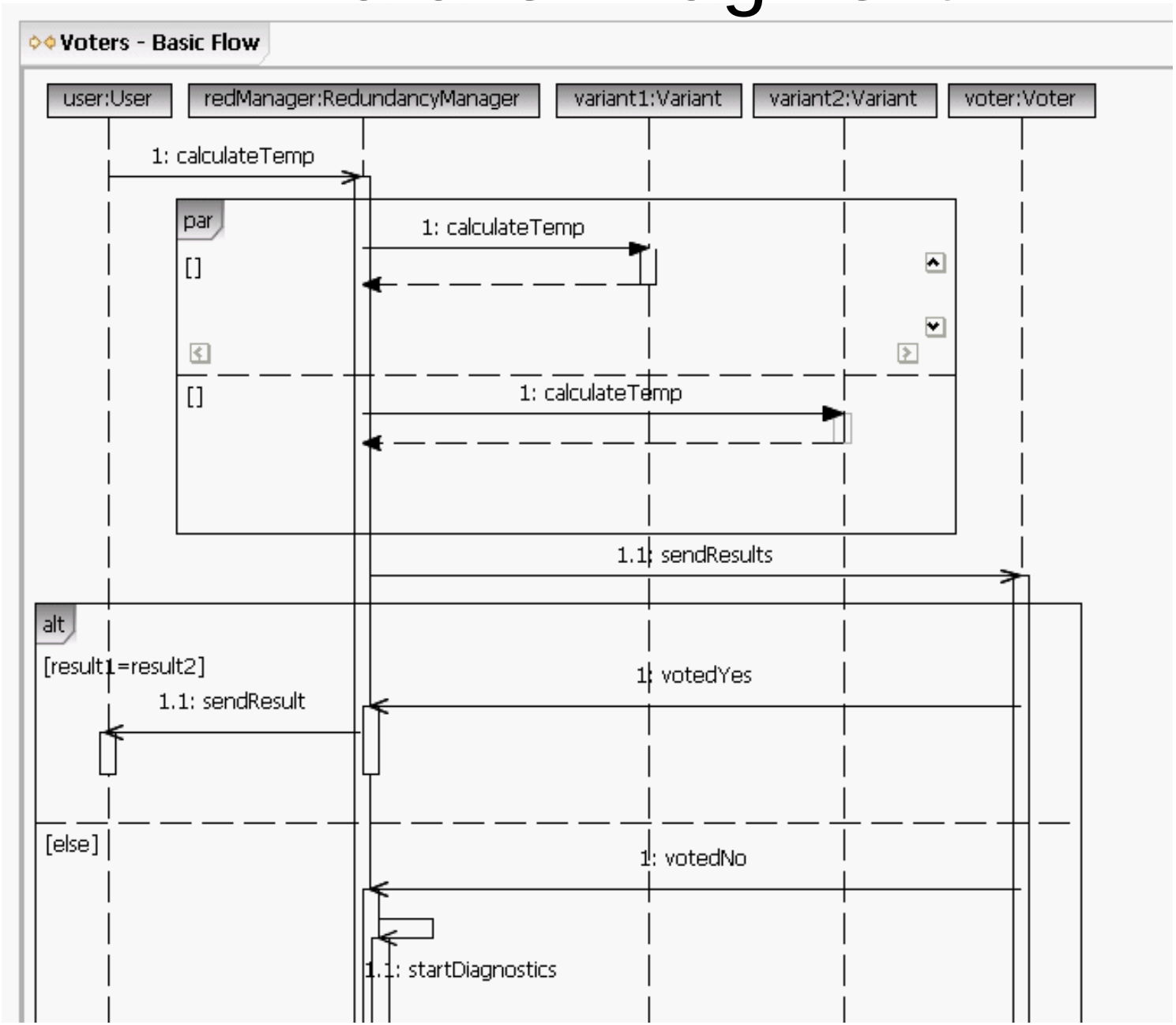
Loop (List Participants)



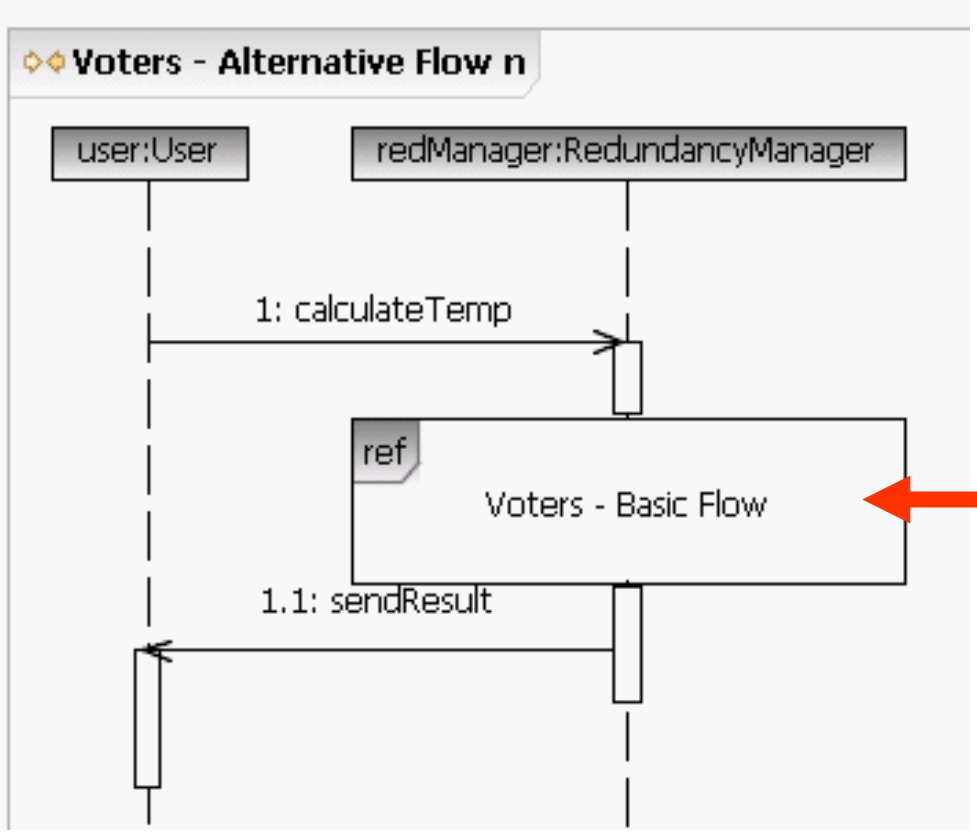
Break Fragment



Parallel Fragment



Interaction Occurrence

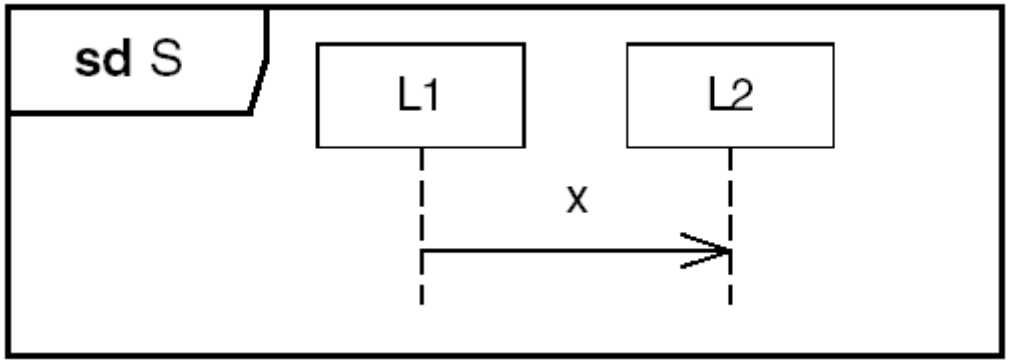


- Goal: reuse of existing sequence diagrams
- Two types:
  - Between messages
  - Called by messages (gates)
- Parameters of interactions
  - parameter passing

Other Interactions

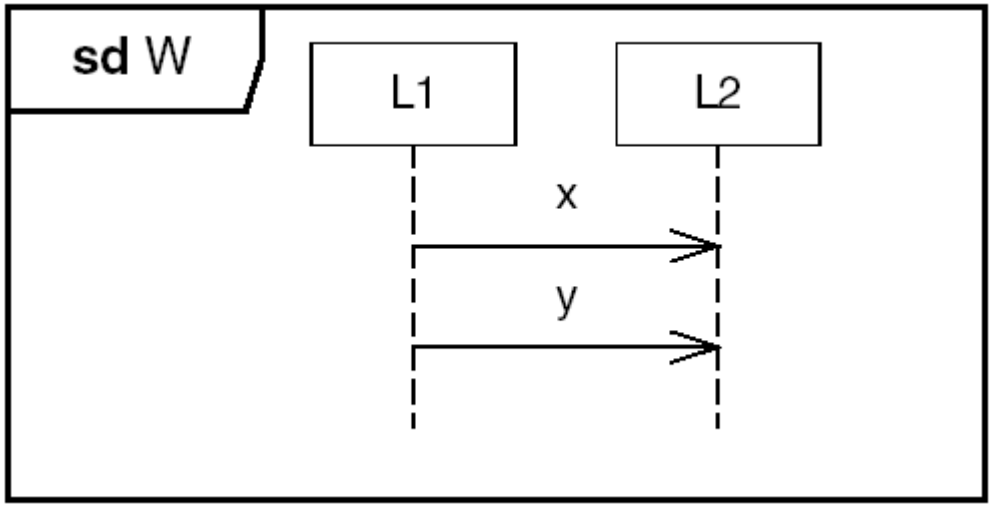
- Assert:
  - The selected interaction must occur in the way indicated
- Neg:
  - The interaction is invalid; it must not occur
- Region:
  - Critical region, no other messages can interleave
- Weak sequencing
  - Specifies the normal weak sequencing rules are in force in the fragment.
- Strict sequencing
  - Specifies that the messages in the interaction fragment are fully ordered
  - Only a single execution trace is consistent with the fragment.

Semantics of Interactions I.



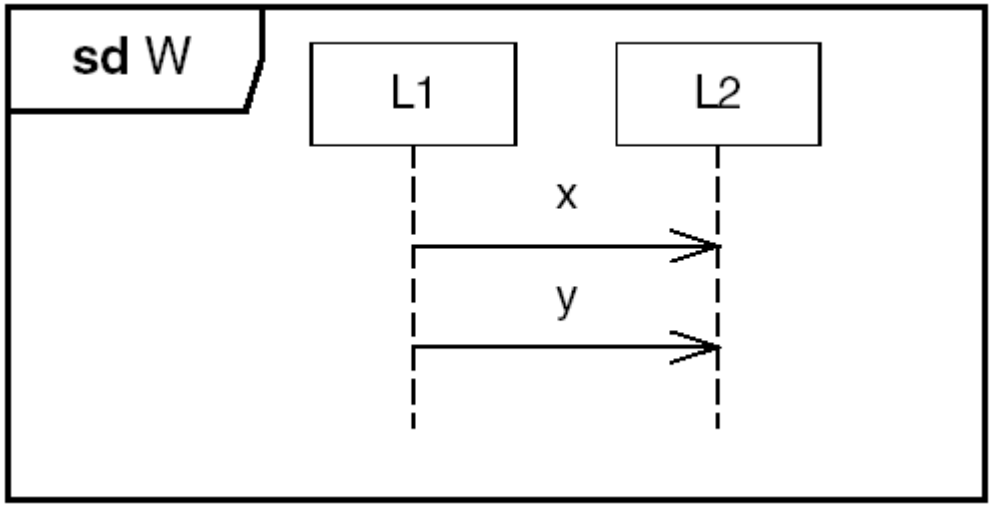
- 2 events
  - Sending x in L1 **!x**
  - Receiving x in L2 **?x**
- Trace set:
  - {**<!x,?x>**}

Semantics of Interactions II.



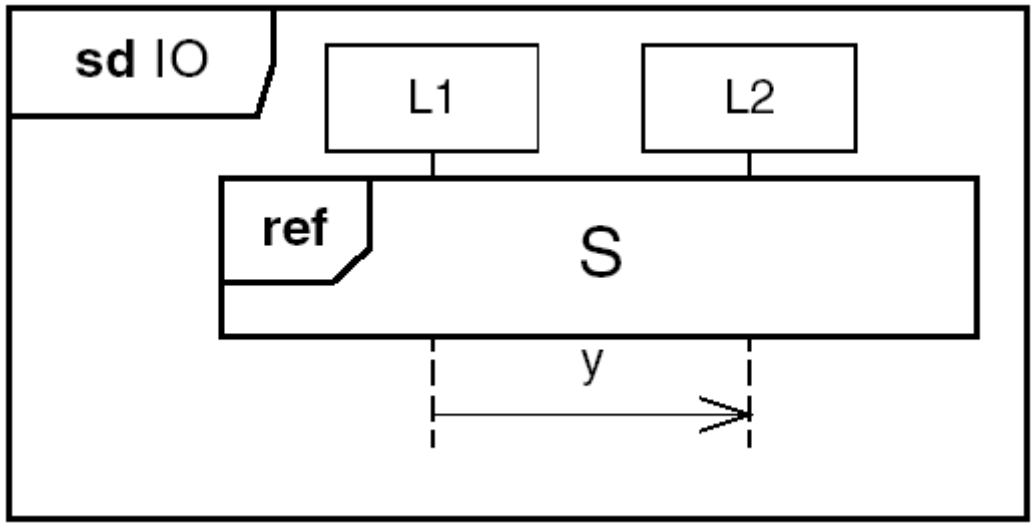
- Weak sequencing: **<!x,?x> seq <!y,?y>**
  - preserves the order within the operands: **!x,?x**
  - different swimlanes: events in arbitrary order
  - same swimlane:
    first the events of operand 1, then events of operand 2
- Trace set: {**<!x,?x,!y,?y>,<!x,!y,?x,?y>**}

Semantics of Interactions III.



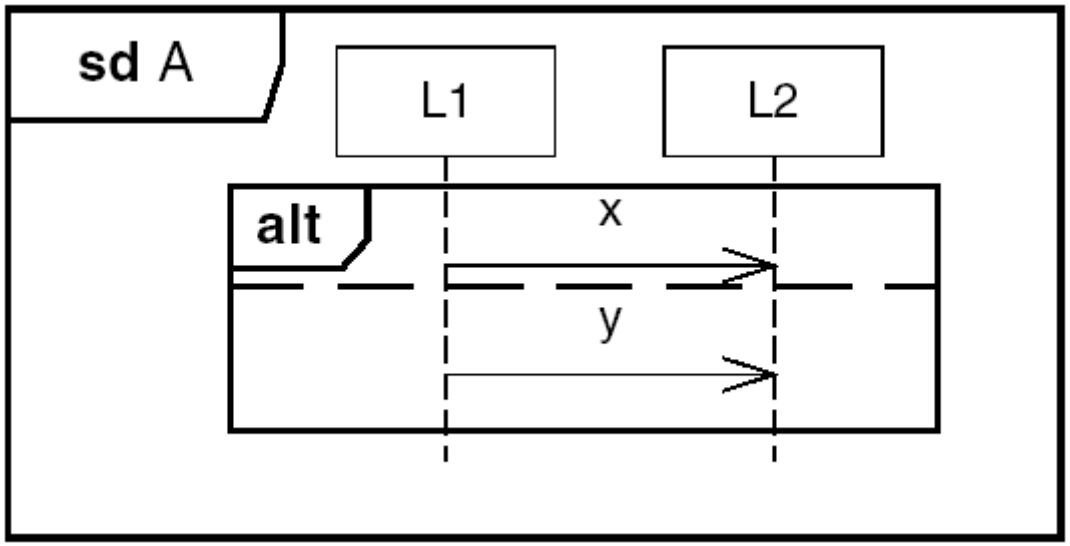
- Strict sequencing: **<!x,?x> strict <!y,?y>**
  - preserves the order within the operands: **!x,?x**
  - same swimlane:
    first the events of operand 1, then events of operand 2
  - different swimlanes: events are also ordered
- Trace set: {**<!x,?x,!y,?y>**}

Semantics of Interactions III.



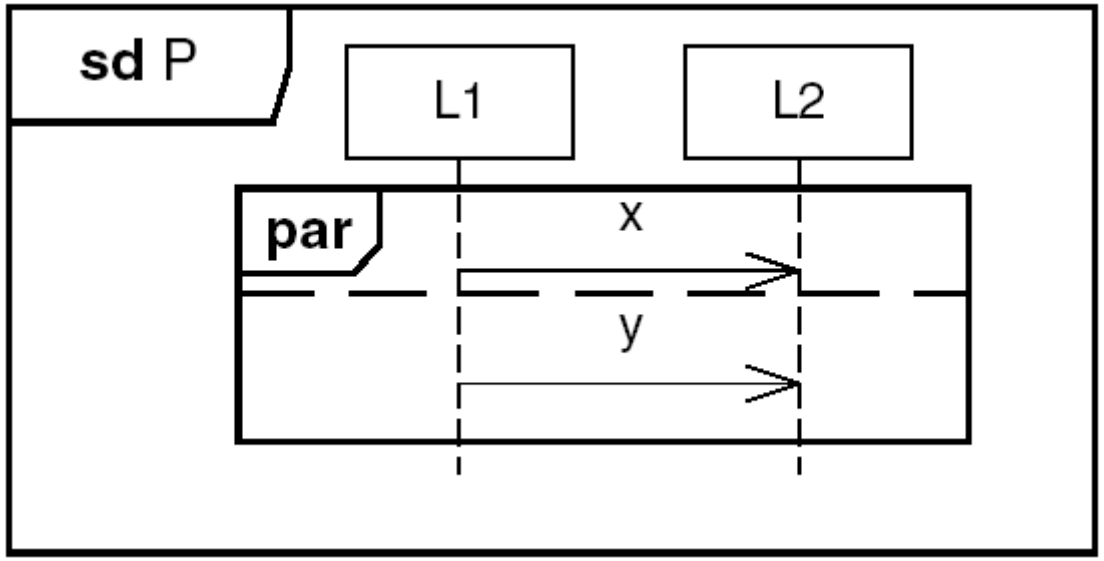
- Interaction occurrence: **S seq <!y,?y>**
  - shortcut: equivalent with the content of S
- Trace set:
  S seq <!y,?y> = <!x,?x> seq <!y,?y> =
  {**<!x,?x,!y,?y>,<!x,!y,?x,?y>**}

Semantics of Interactions IV.



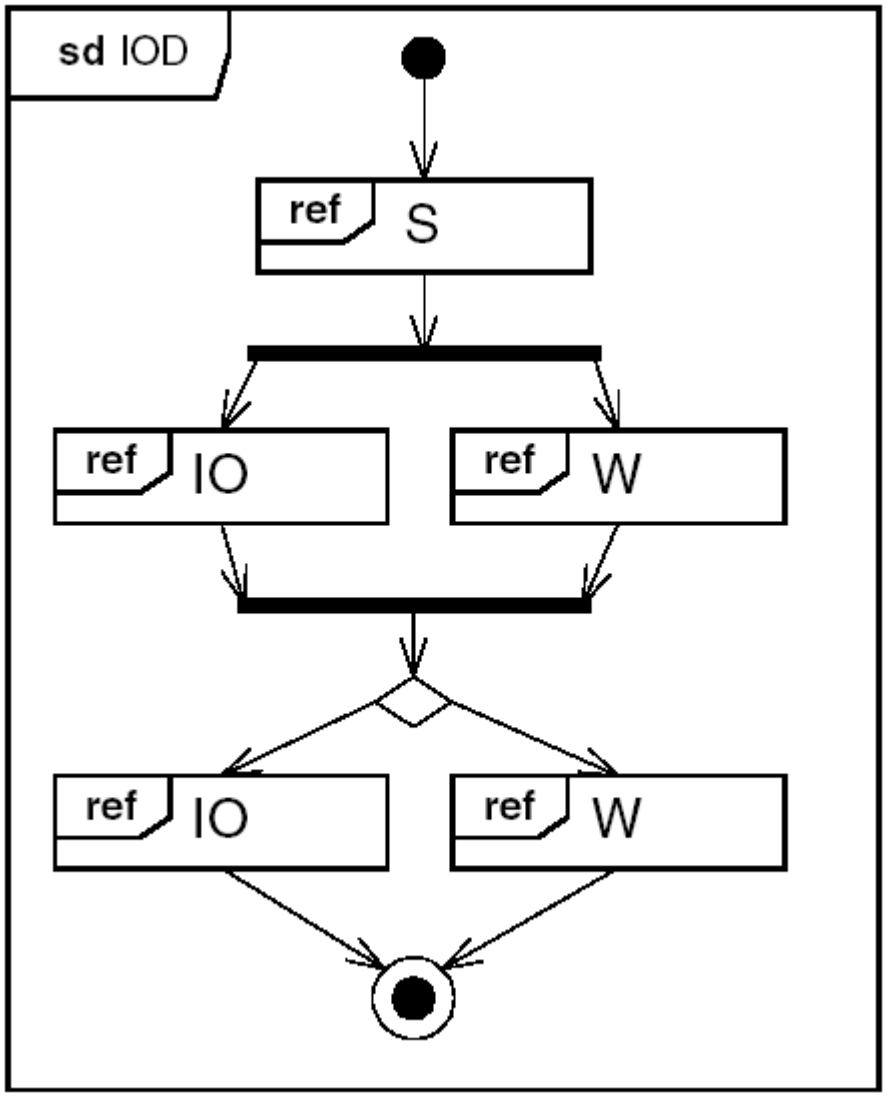
- Alternative fragments: **`P alt Q`**
  - Union of the behavior of P and Q
- Trace set:
  `<!x,?x>` **`alt`** **`<!y,?y>`** **`=`**
  {**`<!x,?x>,<!y,?y>`**}

Semantics of Interactions V.



- Parallel fragments: **`P par Q`**
  - an arbitrary interleaving of the behavior of P and Q observing from a global viewpoint
- Trace halmaz: **`<!x,?x> par <!y,?y> =`** {
  `<!x,?x,!y,?y>,<!x,!y,?x,?y>,<!x,!y,?y,?x>,`
  `<!y,?y,!x,?x>,<!y,!x,?y,?x>,<!y,!x,?x,?y>` }

Semantics of Interactions VI.



- Interaction overview diagram:
  IOD = S seq (IO par W) seq (IO alt W)

Next Lecture: OCL Constraints

- How to capture class invariants?
- How to capture pre- and postconditions of operations?
- How to capture guards?