Rationale

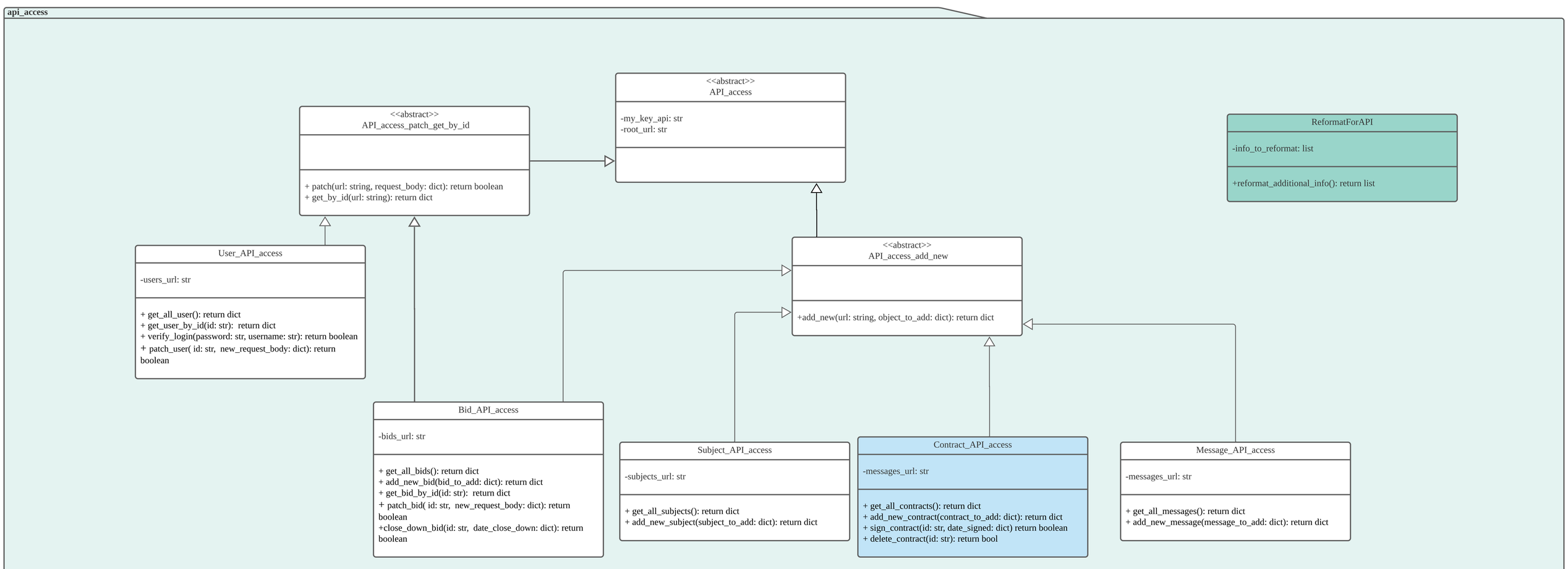
Tatiana Sutulova - 30806151

Elaf Abdullah Saleh Alhaddad - 31063977

Full system diagram

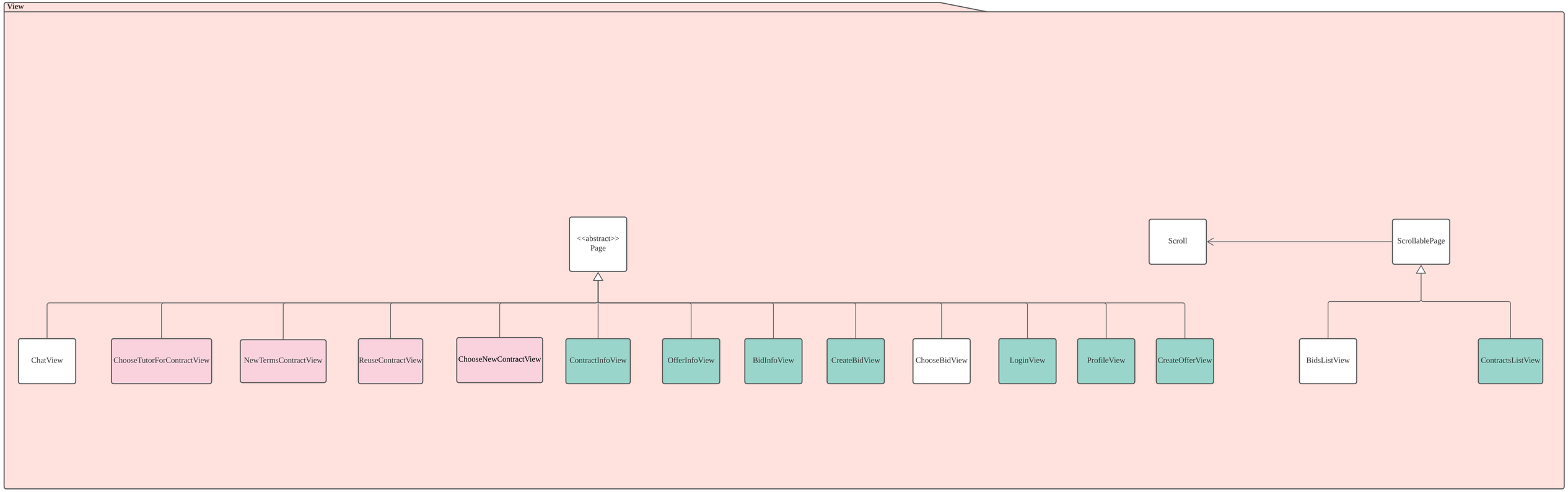
Color Code:
- **Green** - Inline code changes
- **Pink** - New class
- **Blue** - New methods added in class (& inline code changes)

API



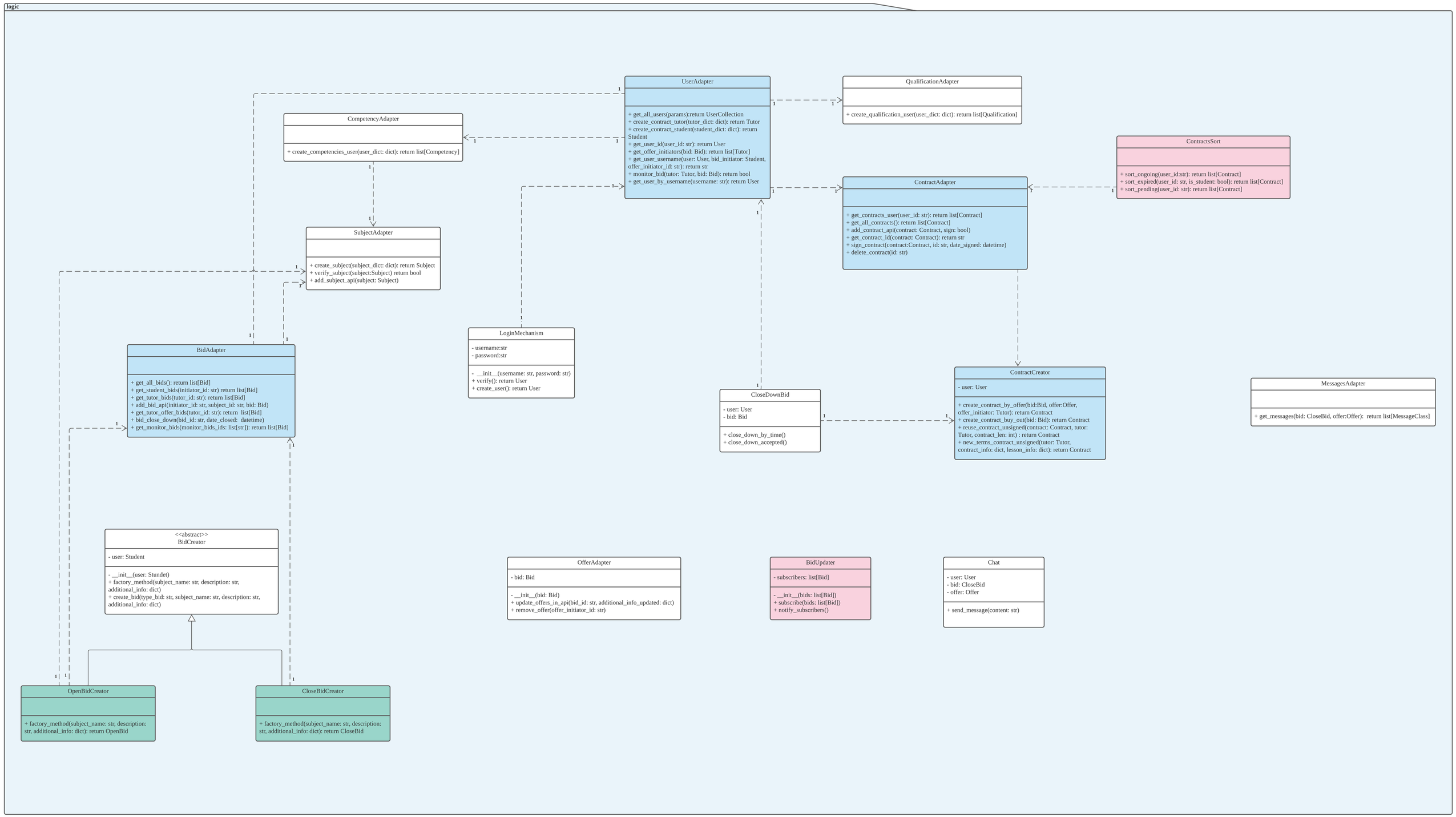
Color Code:
- **Green** - Inline code changes
- **Pink** - New class
- **Blue** - New methods added in class (& inline code changes)

View



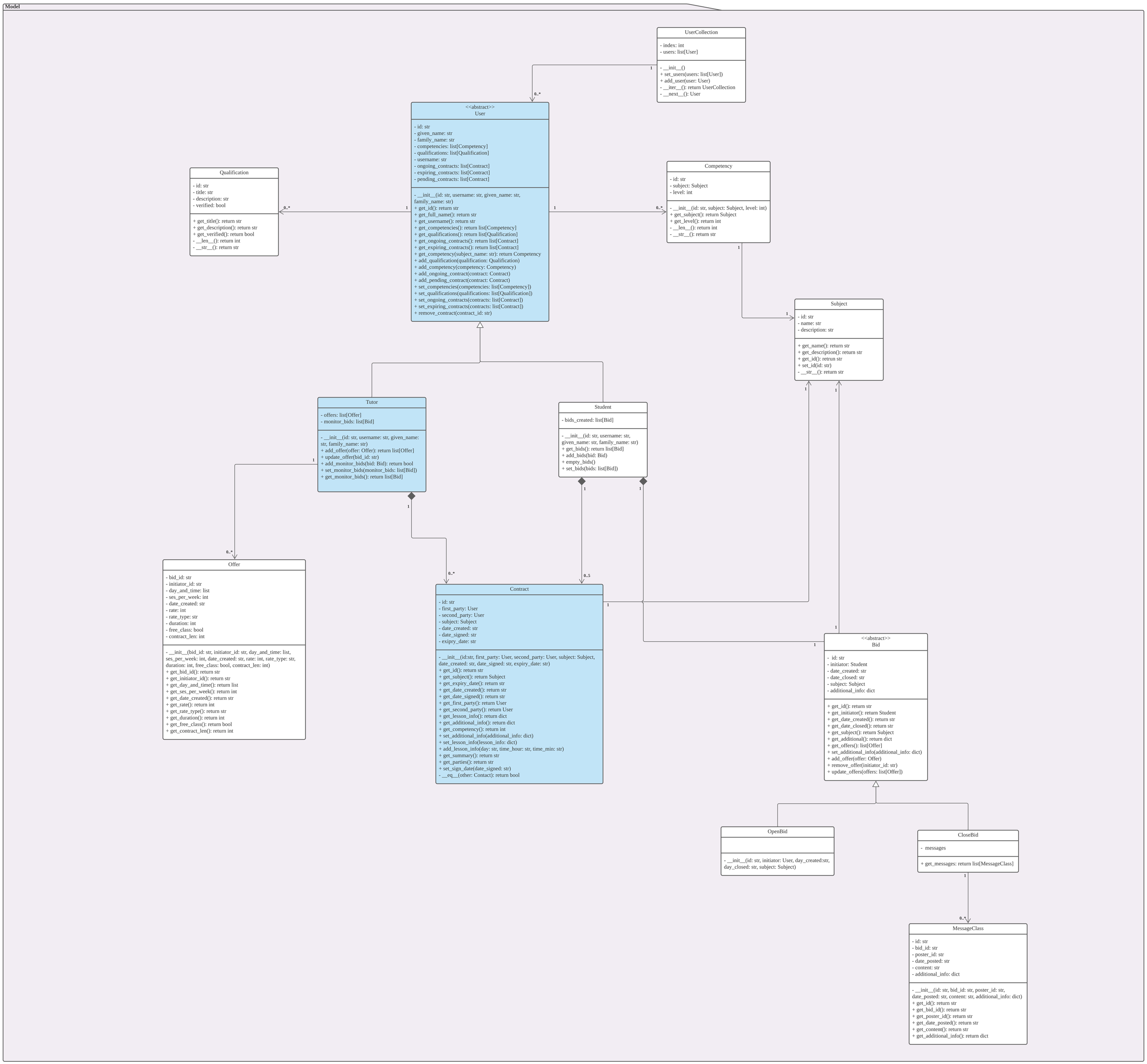
Logic

Color Code:
- **Green** - Inline code changes
- **Pink** - New class
- **Blue** - New methods added in class (& inline code changes)



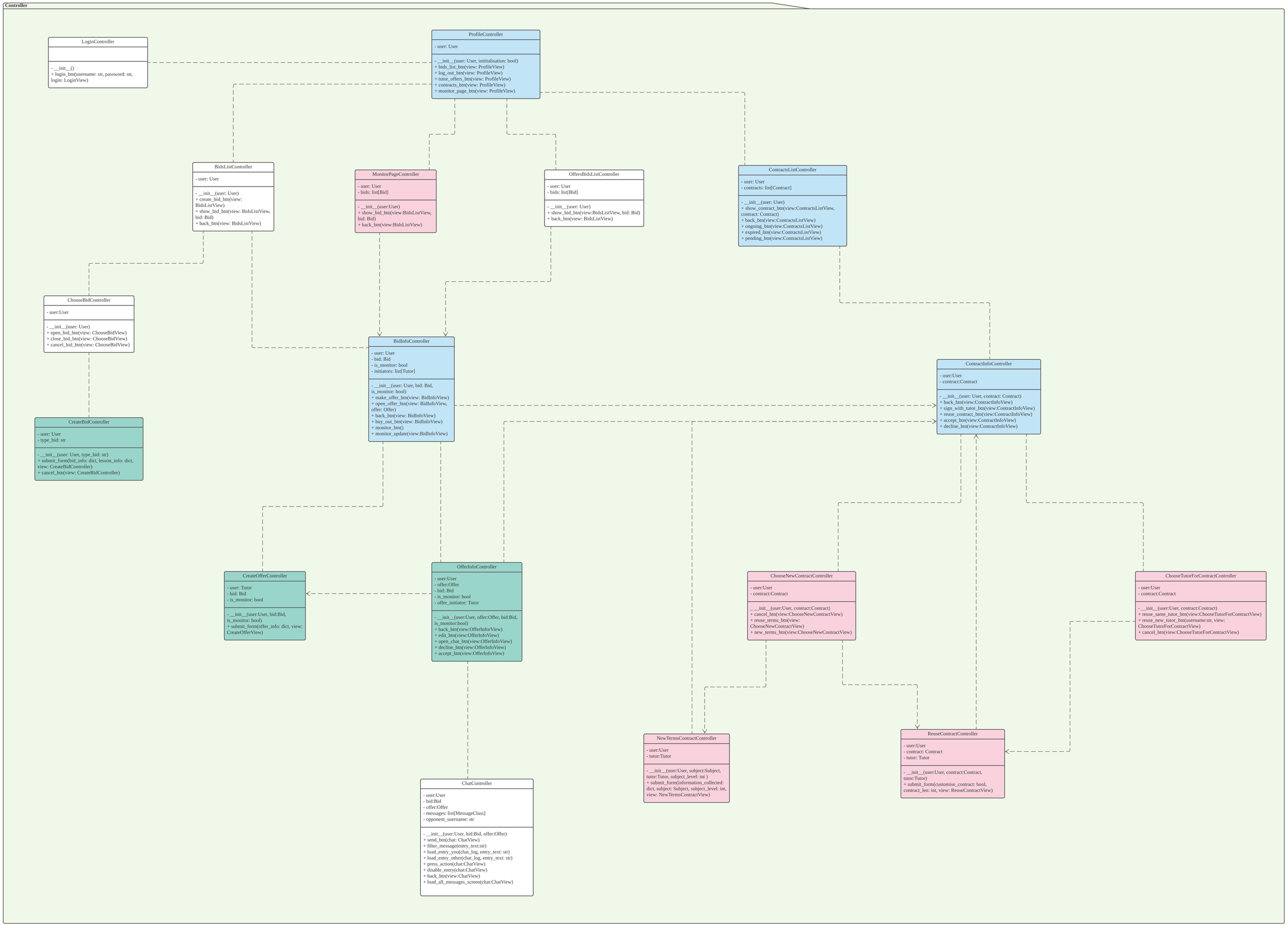
Model

Color Code:
- **Green** - Inline code changes
- **Pink** - New class
- **Blue** - New methods added in class (& inline code changes)

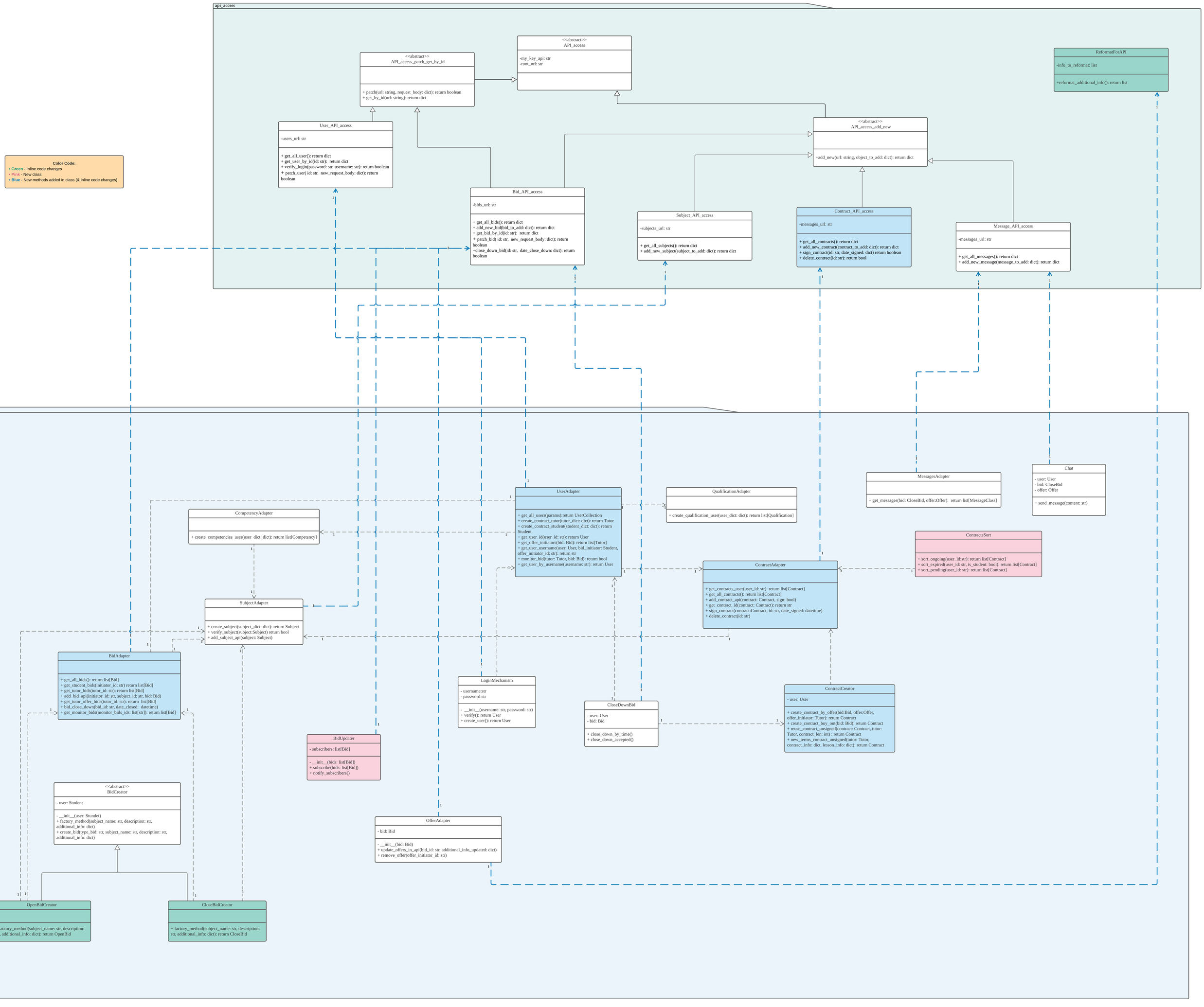


Controller

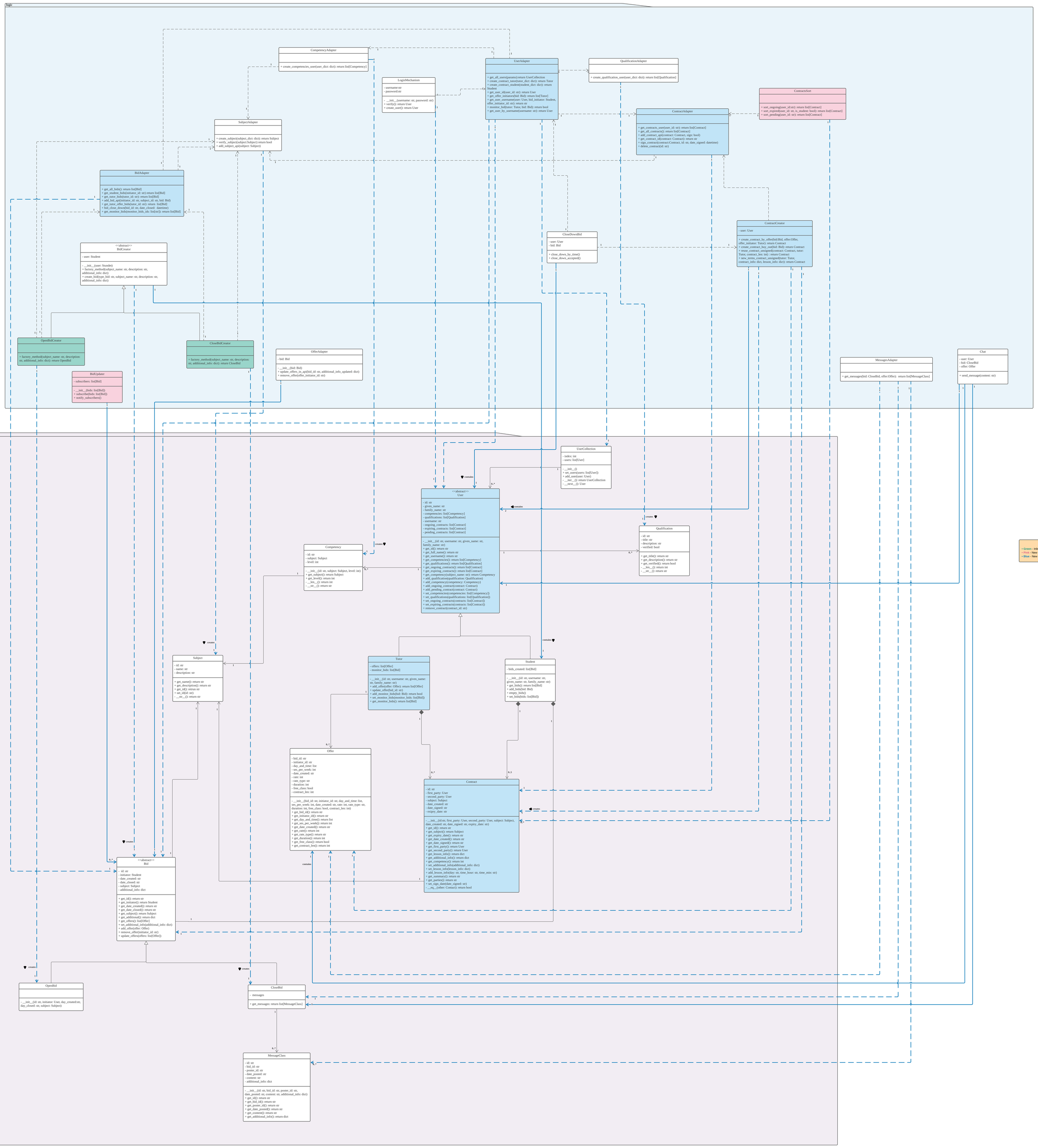
Color Code:
- **Green** - Inline code changes
- **Pink** - New class
- **Blue** - New methods added in class (& inline code changes)



Logic and API



Logic and Model



Logic and Controller

Color Code:
- **Green** - Inline code changes
- **Pink** - New class
- **Blue** - New methods added in class (& inline code changes)

logic

UserAdapter
+ get_all_users(params):return UserCollection
+ create_contract_tutor(tutor_dict: dict): return Tutor
+ create_contract_student(student_dict: dict): return Student
+ get_user_id(user_id: str): return User
+ get_offer_initiators(bid: Bid): return list[Tutor]
+ get_user_username(user: User, bid_initiator: Student, offer_initiator_id: str): return str
+ monitor_bid(tutor: Tutor, bid: Bid): return bool
+ get_user_by_username(username: str): return User

QualificationAdapter
+ create_qualification_user(user_dict: dict): return list[Qualification]

CompetencyAdapter
+ create_competencies_user(user_dict: dict): return list[Competency]

SubjectAdapter
+ create_subject(subject_dict: dict): return Subject
+ verify_subject(subject:Subject) return bool
+ add_subject_api(subject: Subject)

ContractAdapter
+ get_contracts_user(user_id: str): return list[Contract]
+ get_all_contracts(): return list[Contract]
+ add_contract_api(contract: Contract, sign: bool)
+ get_contract_id(contract: Contract): return str
+ sign_contract(contract:Contract, id: str, date_signed: datetime)
+ delete_contract(id: str)

ContractsSort
+ sort_ongoing(user_id:str): return list[Contract]
+ sort_expired(user_id: str, is_student: bool): return list[Contract]
+ sort_pending(user_id: str): return list[Contract]

BidAdapter
+ get_all_bids(): return list[Bid]
+ get_student_bids(initiator_id: str) return list[Bid]
+ get_tutor_bids(tutor_id: str): return list[Bid]
+ add_bid_api(initiator_id: str, subject_id: str, bid: Bid)
+ get_tutor_offer_bids(tutor_id: str): return list[Bid]
+ bid_close_down(bid_id: str, date_closed: datetime)
+ get_monitor_bids(monitor_bids_ids: list[str]): return list[Bid]

LoginMechanism
- username:str
- password:str
- __init__(username: str, password: str)
+ verify(): return User
+ create_user(): return User

CloseDownBid
- user: User
- bid: Bid
+ close_down_by_time()
+ close_down_accepted()

ContractCreator
- user: User
+ create_contract_by_offer(bid:Bid, offer:Offer, offer_initiator: Tutor): return Contract
+ create_contract_buy_out(bid: Bid): return Contract
+ reuse_contract_unsigned(contract: Contract, tutor: Tutor, contract_len: int) : return Contract
+ new_terms_contract_unsigned(tutor: Tutor, contract_info: dict, lesson_info: dict): return Contract

MessagesAdapter
+ get_messages(bid: CloseBid, offer:Offer): return list[MessageClass]

<<abstract>> BidCreator
- user: Student
- __init__(user: Stundet)
+ factory_method(subject_name: str, description: str, additional_info: dict)
+ create_bid(type_bid: str, subject_name: str, description: str, additional_info: dict)

OpenBidCreator
+ factory_method(subject_name: str, description: str, additional_info: dict): return OpenBid

CloseBidCreator
+ factory_method(subject_name: str, description: str, additional_info: dict): return CloseBid

Chat
- user: User
- bid: CloseBid
- offer: Offer
+ send_message(content: str)

OfferAdapter
- bid: Bid
- __init__(bid: Bid)
+ update_offers_in_api(bid_id: str, additional_info_updated: dict)
+ remove_offer(offer_initiator_id: str)

BidUpdater
- subscribers: list[Bid]
- __init__(bids: list[Bid])
+ subscribe(bids: list[Bid])
+ notify_subscribers()

Controller

LoginController
- __init__()
+ login_btn(username: str, password: str, login: LoginView)

ProfileController
- user: User
- __init__(user: User, inititalisation: bool)
+ bids_list_btn(view: ProfileView)
+ log_out_btn(view: ProfileView)
+ tutor_offers_btn(view: ProfileView)
+ contracts_btn(view: ProfileView)
+ monitor_page_btn(view: ProfileView)

BidsListController
- user: User
- __init__(user: User)
+ create_bid_btn(view: BidsListView)
+ show_bid_btn(view: BidsListView, bid: Bid)
+ back_btn(view: BidsListView)

MonitorPageController
- user: User
- bids: list[Bid]
- __init__(user:User)
+ show_bid_btn(view:BidsListView, bid: Bid)
+ back_btn(view:BidsListView)

OffersBidsListController
- user: User
- bids: list[Bid]
- __init__(user: User)
+ show_bid_btn(view:BidsListView, bid: Bid)
+ back_btn(view: BidsListView)

ContractsListController
- user: User
- contracts: list[Contract]
- __init__(user: User)
+ show_contract_btn(view:ContractsListView, contract: Contract)
+ back_btn(view:ContractsListView)
+ ongoing_btn(view:ContractsListView)
+ expired_btn(view:ContractsListView)
+ pending_btn(view:ContractsListView)

ChooseBidController
- user:User
- __init__(user: User)
+ open_bid_btn(view: ChooseBidView)
+ close_bid_btn(view: ChooseBidView)
+ cancel_bid_btn(view: ChooseBidView)

CreateBidController
- user: User
- type_bid: str
- __init__(user: User, type_bid: str)
+ submit_form(bid_info: dict, lesson_info: dict, view: CreateBidController)
+ cancel_btn(view: CreateBidController)

BidInfoController
- user: User
- bid: Bid
- is_monitor: bool
- initiators: list[Tutor]
- __init__(user: User, bid: Bid, is_monitor: bool)
+ make_offer_btn(view: BidInfoView)
+ open_offer_btn(view: BidInfoView, offer: Offer)
+ back_btn(view: BidInfoView)
+ buy_out_btn(view: BidInfoView)
+ monitor_btn()
+ monitor_update(view:BidInfoView)

ContractInfoController
- user:User
- contract:Contract
- __init__(user: User, contract: Contract)
+ back_btn(view:ContractInfoView)
+ sign_with_tutor_btn(view:ContractInfoView)
+ reuse_contract_btn(view:ContractInfoView)
+ accept_btn(view:ContractInfoView)
+ decline_btn(view:ContractInfoView)

CreateOfferController
- user: Tutor
- bid: Bid
- is_monitor: bool
- __init__(user:User, bid:Bid, is_monitor: bool)
+ submit_form(offer_info: dict, view: CreateOfferView)

OfferInfoController
- user:User
- offer:Offer
- bid: Bid
- is_monitor: bool
- offer_initiator: Tutor
- __init__(user:User, offer:Offer, bid:Bid, is_monitor:bool)
+ back_btn(view:OfferInforView)
+ edit_btn(view:OfferInfoView)
+ open_chat_btn(view:OfferInfoView)
+ decline_btn(view:OfferInfoView)
+ accept_btn(view:OfferInfoView)

ChooseNewContractController
- user:User
- contract:Contract
- __init__(user: User, contract:Contract)
+ cancel_btn(view:ChooseNewContractView)
+ reuse_terms_btn(view: ChooseNewContractView)
+ new_terms_btn(view:ChooseNewContractView)

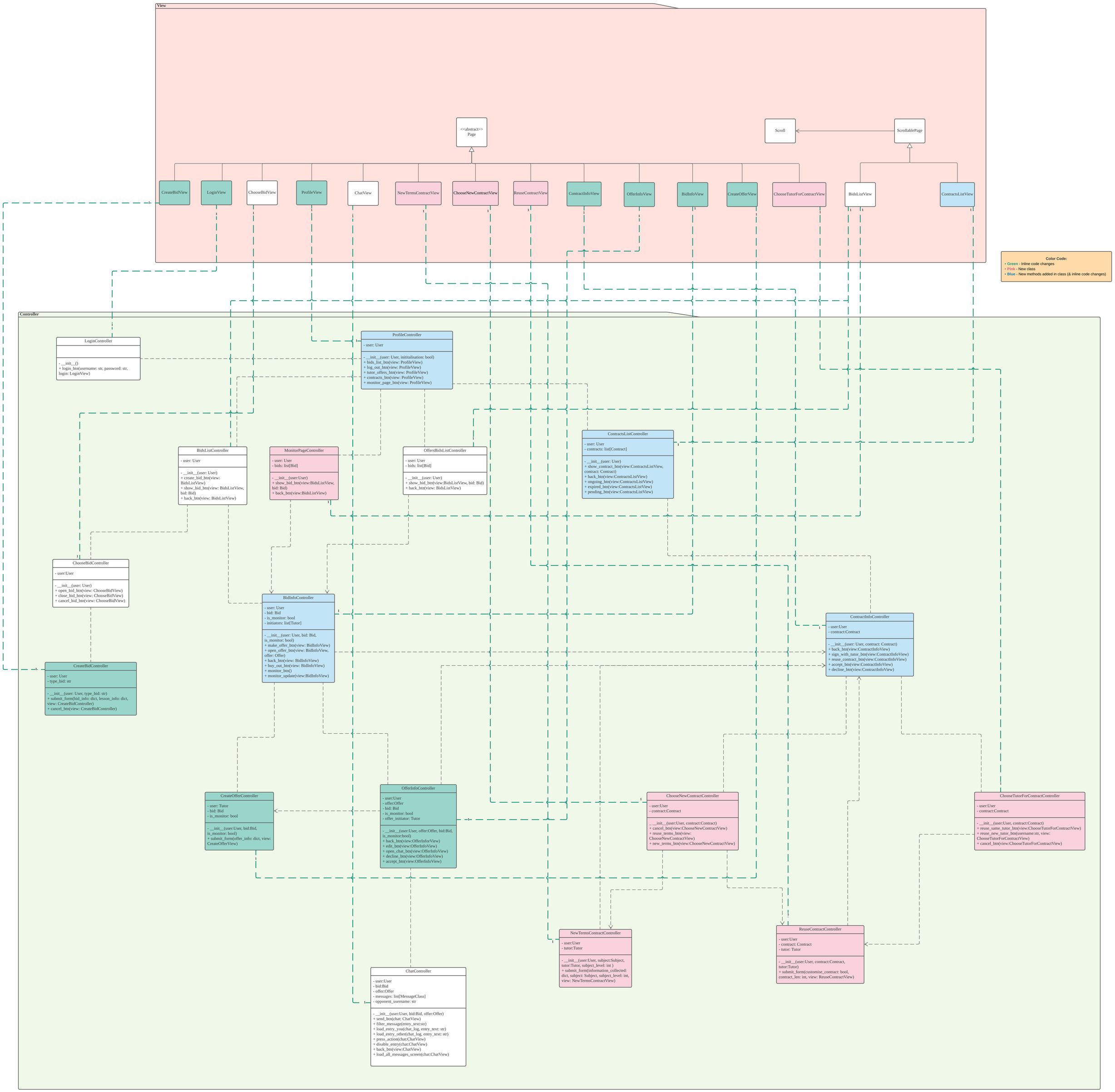
ChooseTutorForContractController
- user:User
- contract:Contract
- __init__(user:User, contract:Contract)
+ reuse_same_tutor_btn(view:ChooseTutorForContractView)
+ reuse_new_tutor_btn(username:str, view: ChooseTutorForContractView)
+ cancel_btn(view:ChooseTutorForContractView)

NewTermsContractController
- user:User
- tutor:Tutor
- __init__(user:User, subject:Subject, tutor:Tutor, subject_level: int)
+ submit_form(information_collected: dict, subject: Subject, subject_level: int, view: NewTermsContractView)

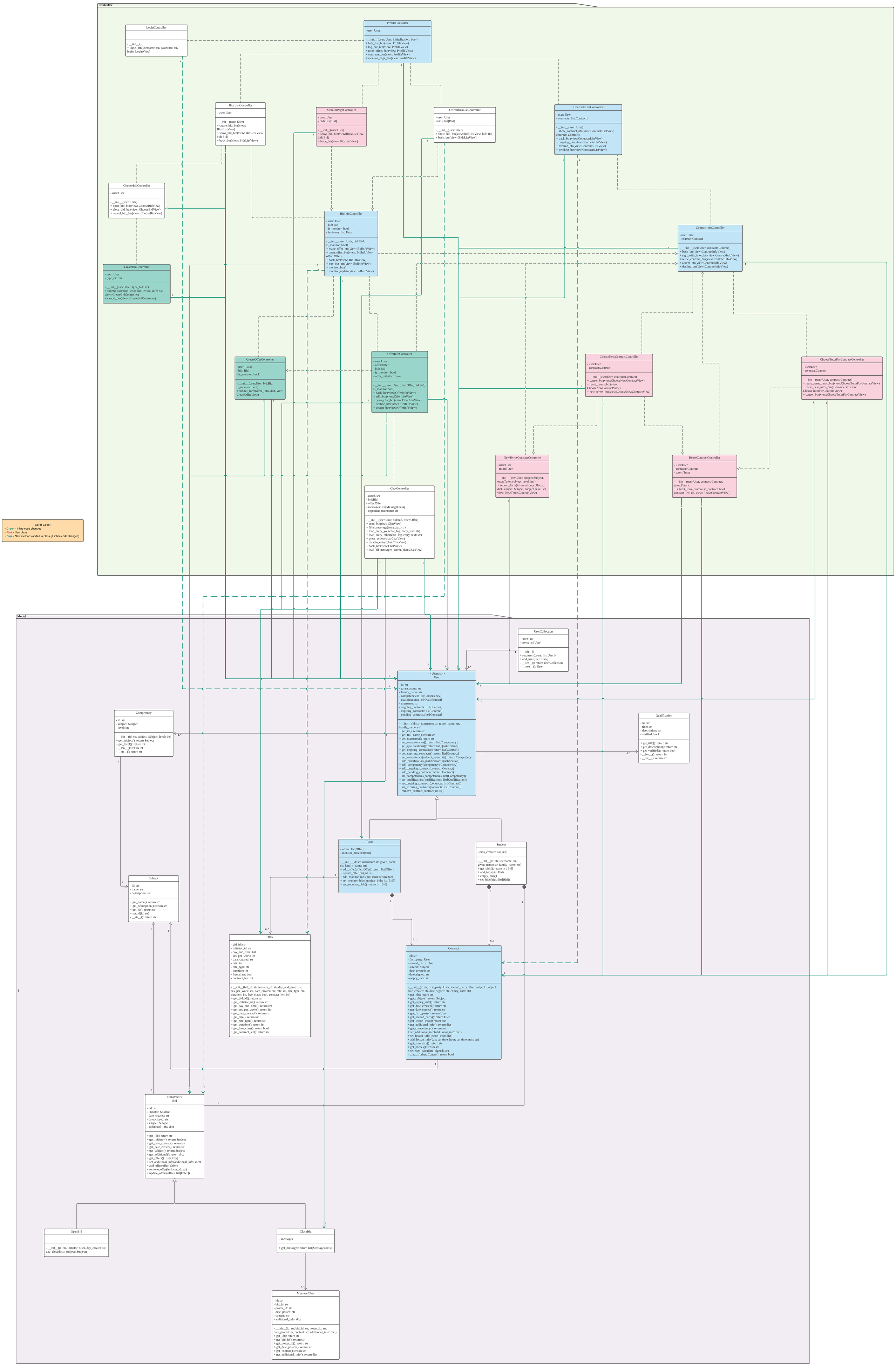
ReuseContractController
- user:User
- contract: Contract
- tutor: Tutor
- __init__(user:User, contract:Contract, tutor:Tutor)
+ submit_form(customise_contract: bool, contract_len: int, view: ReuseContractView)

ChatController
- user:User
- bid:Bid
- offer:Offer
- messages: list[MessageClass]
- opponent_username: str
- __init__(user:User, bid:Bid, offer:Offer)
+ send_btn(chat: ChatView)
+ filter_message(entry_text:str)
+ load_entry_you(chat_log, entry_text: str)
+ load_entry_other(chat_log, entry_text: str)
+ press_action(chat:ChatView)
+ disable_entry(chat:ChatView)
+ back_btn(view:ChatView)
+ load_all_messages_screen(chat:ChatView)

Controller and view



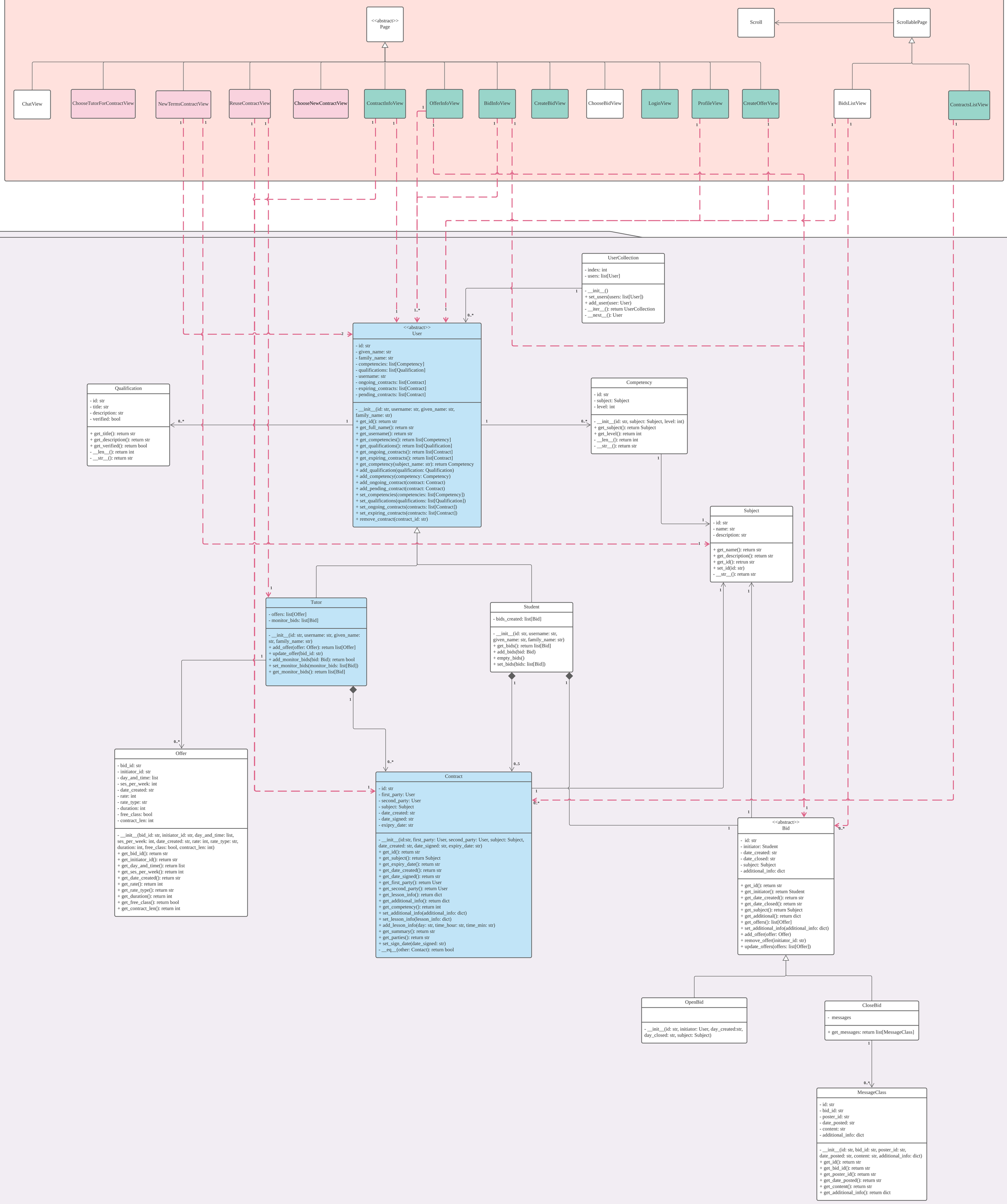
Controller and Model



View and Model

Color Code:
- **Green** - Inline code changes
- **Pink** - New class
- **Blue** - New methods added in class (& inline code changes)

View



Model

1. Architectural pattern

To begin with, the architectural pattern that was initially chosen to build our software was MVC architecture with one additional package API_access, which was responsible for accessing the API. Extending the system has proven that the choice of the architecture was made correctly, since the app was highly extensible due to ease of understanding of what each package is responsible for.

However, due to implementing new functionalities and requiring new classes, the model module was overloaded. This made the code difficult to navigate through and overcomplicated the class diagram. In order to solve this issue, the model package was separated into two packages: logic and model, where the model contains only the 'noun' classes, such as the Tutor, User, Qualification, etc. and the logic package includes all the mechanism, adapter and creator classes that are responsible for changing the state of the app.

The renewed model package does not depend on any other packages, whereas the logic package is dependent on the API_access package and on the model packages, since it communicates with the api and model classes.

However, using the MVC did not eliminate the cyclic dependencies between view and controller classes. This because the application is implemented using tkinter GUI and would require both the controller and view classes know about the implementation of each other to allow action such as clicking buttons and filling up forms possible.

2. Design Patterns

Besides the design patterns that were implemented previously, we added a new behavioural design pattern - observer pattern - to the system. It was useful for "each bid in the monitor list updates every N seconds" feature, since it allowed the tutor to monitor a bid and receive updates on new offers for that bid every N seconds without refreshing the page.

In order to do that we made each instance of the monitored OpenBid class a subscriber of the observer (BidUpdater class) and ensured that all the subscribed OpenBid instances are updated simultaneously. This BidUpdater class gets information from the API every N seconds and updates it's subscribers accordingly if there are any changes in their offers.

This design pattern was beneficial in a way that rather than modifying the code in existing classes we added a new observer class, which controls the existing OpenBid classes, ensuring it does not violate the OCP and does not break the program in any other way. Moreover, observer pattern ensured the loosely coupled design between the classes that interact, since the only new dependency that appeared in the system was the dependency of BidUpdater observer class on the OpenBid classes, which are being monitored by the tutor.

3. Design principles

3.1: Single responsibility principle (SRP)

Most of the previously added classes in the system were following the single responsibility principle (SRP), meaning that each class has only one responsibility and one reason to change. All the newly added classes were following this principle as well. For example, the ContractsSort class is only responsible for sorting the list of contracts into ongoing, expired and pending. As well as, BidUpdater class which is only responsible for updating it's subscribers.

Applying SRP principle increased the extensibility and maintainability of the system, since the system is highly cohesive and the responsibility of each class is obvious, making it clear which class has to be extended or modified in order to add new features

3.2: Open-Close Principle (OCP)

The other design principle that was implemented by both previously and newly created classes is open-close principle (OCP), which states that the system should be open for extension and closed for modification. That means that the system should be allowed to add new features and support new functionalities, however it should be done in a way that does not change the existing code.

When extending the system, the application of the OCP principle ensured that there were no issues faced; there was no big change to the existing class implementation required and there was no need to change the classes that were dependent on the classes that were extended in the iteration.

New features were implemented by adding new classes, adding new methods to existing classes and adding small lines in existing classes and methods. The changes did not cause any errors to appear in the existing system.

4. Package level principles

4.1 Stable dependencies principle (SDP)

In order to ensure that our model and api_access packages don't depend on other packages, we follow the stable dependency principle. Classes from outside the packages will depend on classes inside these packages but not vice versa. This makes our stable dependency matrix closer to 0 for these two packages making them stable.

Following the SDP principle improves the design quality of our system, since the less stable classes are dependent on the more stable classes. This ensures that the packages that are more likely to change will not lead to changes in other existing packages in the system.

4.2 Release Reuse Equivalence Principle (RREP)

The packages in our system groups classes with similar goals, for example; all classes under View package are responsible for UI and displaying information to the user. This allows our system to be as cohesive as possible with minimal coupling required between the packages.

Implementing release reuse equivalence principle allows our code to be easily extensible since it is easier to understand which packages that the new classes required to implement new functionalities should belong to or which package need to be modified.

5. Other refactoring applied

The implementation of our API_Access module was modified to have abstract classes that contain methods that are shared between several classes of the API_access package. For example, getting all objects from the api was required by most of the API_access classes and so we added a parent class that will be inherited by these classes.
In addition, we change the information passing from view classes that require submitting a form. For instance; CreateBidView passes information inputted by the user to create a bid to CreateBidController, submit_form method. This information

was passed as arguments to the method which in return compromised the quality of the code since the submit form method had more than 3 parameters passed to it. Instead we create a dictionary object in CreateBidView that will contain all the information and is passed to the submit form method. This allows the method to have less than 3 parameters.

These refactored codes allowed the improvement of the source code quality of our application and made it easier to understand and extend.

6. Conclusion

In conclusion, it is important to mention that the initial design was highly extensible and maintainable due to the application of the right design principles. When adding the new features to the system, we haven't faced the issue that caused the need to change a large part of the source code, since all the new functionalities were added through extending the existing code rather than modifying it. Furthermore, it was not troublesome to find where and which package/class the new classes/methods should be added to, since the responsibility of each component of our system is obvious.