CompositeFinancialModel Detailed Explanation

1 Class Definition

```python
class CompositeFinancialModel(FinancialModelBase):
    def __init__(self, financial_models: Dict[type, FinancialModelBase]):
        self.financial_models = financial_models

    def damage_to_loss(self, asset: Asset, impact: np.ndarray, currency: str):
        return self.financial_models[type(asset)].damage_to_loss(
            asset, impact, currency
        )

    def disruption_to_loss(
        self, asset: Asset, impact: np.ndarray, year: int, currency: str
    ):
        return self.financial_models[type(asset)].disruption_to_loss(
            asset, impact, year, currency
        )
```

2 Purpose

The `CompositeFinancialModel` is designed to provide a flexible way to apply different financial models to different types of assets within a single, unified interface. This is particularly useful in scenarios where various asset classes require distinct financial modeling approaches.

3 Key Features

1. **Type-based Model Selection**: Uses the type of the asset to determine which specific financial model to apply.

2. **Polymorphic Behavior**: Maintains the `FinancialModelBase` interface while allowing for diverse underlying implementations.

3. **Extensibility**: Easily accommodates new asset types and corresponding financial models without modifying existing code.

4. **Encapsulation**: Hides the complexity of multiple models from the user of the `CompositeFinancialModel`.

4 Detailed Breakdown

4.1 Constructor

```python
def __init__(self, financial_models: Dict[type, FinancialModelBase]):
    self.financial_models = financial_models
```

- **Parameter**: `financial_models` is a dictionary where:
  - Keys are Python types (presumably subclasses of `Asset`)
  - Values are instances of classes derived from `FinancialModelBase`
- This structure allows for a flexible mapping of asset types to specific financial models.

4.2 Method: damage_to_loss

```python
def damage_to_loss(self, asset: Asset, impact: np.ndarray, currency: str):
    return self.financial_models[type(asset)].damage_to_loss(
        asset, impact, currency
    )
```

- Uses `type(asset)` to select the appropriate financial model from the dictionary.
- Delegates the actual calculation to the selected model's `damage_to_loss` method.
- Maintains the same interface as `FinancialModelBase`, ensuring compatibility.

4.3 Method: disruption_to_loss

```python
def disruption_to_loss(
    self, asset: Asset, impact: np.ndarray, year: int, currency: str
):
    return self.financial_models[type(asset)].disruption_to_loss(
        asset, impact, year, currency
    )
```

- Similar to `damage_to_loss`, but for disruption calculations.
- Again, delegates to the type-specific model's method.

5 Usage Example

```python
class RealEstateModel(FinancialModelBase):
    # Specific implementation for real estate assets

class InfrastructureModel(FinancialModelBase):
    # Specific implementation for infrastructure assets

composite_model = CompositeFinancialModel({
    RealEstateAsset: RealEstateModel(),
    InfrastructureAsset: InfrastructureModel()
})

# Usage
real_estate = RealEstateAsset(...)
infrastructure = InfrastructureAsset(...)

loss_re = composite_model.damage_to_loss(real_estate, [0.1, 0.2], "
    USD")
loss_infra = composite_model.damage_to_loss(infrastructure, [0.05,
    0.15], "USD")
```

6 Advantages

1. **Modularity**: Each asset type can have its own specialized financial model.
2. **Single Interface**: Users interact with a single `CompositeFinancialModel`, simplifying the API.
3. **Easy Maintenance**: New asset types and models can be added without changing existing code.
4. **Separation of Concerns**: Each individual financial model can focus on its specific asset type.

7 Considerations and Potential Improvements

1. **Error Handling**: Add checks for missing asset types in the dictionary.
2. **Default Model**: Consider providing a default model for unrecognized asset types.
3. **Dynamic Registration**: Implement methods to add or remove models at runtime.
4. **Validation**: Add checks to ensure all provided models adhere to the `FinancialModelBase` interface.
5. **Documentation**: Include type hints and docstrings for better IDE support and user guidance.