```
import inspect
import json
import logging
import os
from datetime import datetime
from typing import Any, Dict, Set
from uuid import UUID

logger = logging.getLogger(__name__)


class SafeLoaderUtils:
    """
    Utility class for safely loading and saving object states while automatically
    detecting and preserving class instances and complex objects.
    """

    @staticmethod
    def is_class_instance(obj: Any) -> bool:
        """
        Detect if an object is a class instance (excluding built-in types).

        Args:
            obj: Object to check

        Returns:
```

```
        bool: True if object is a class instance
    """
    if obj is None:
        return False

    # Get the type of the object
    obj_type = type(obj)

    # Check if it's a class instance but not a built-in type
    return (
        hasattr(obj, "__dict__")
        and not isinstance(obj, type)
        and obj_type.__module__ != "builtins"
    )

@staticmethod
def is_safe_type(value: Any) -> bool:
    """
    Check if a value is of a safe, serializable type.

    Args:
        value: Value to check

    Returns:
        bool: True if the value is safe to serialize
    """
```

```
# Basic safe types
safe_types = (
    type(None),
    bool,
    int,
    float,
    str,
    datetime,
    UUID,
)

if isinstance(value, safe_types):
    return True

# Check containers
if isinstance(value, (list, tuple)):
    return all(
        SafeLoaderUtils.is_safe_type(item) for item in value
    )

if isinstance(value, dict):
    return all(
        isinstance(k, str) and SafeLoaderUtils.is_safe_type(v)
        for k, v in value.items()
    )
```

```
    # Check for common serializable types
    try:
        json.dumps(value)
        return True
    except (TypeError, OverflowError, ValueError):
        return False

@staticmethod
def get_class_attributes(obj: Any) -> Set[str]:
    """
    Get all attributes of a class, including inherited ones.

    Args:
        obj: Object to inspect

    Returns:
        Set[str]: Set of attribute names
    """
    attributes = set()

    # Get all attributes from class and parent classes
    for cls in inspect.getmro(type(obj)):
        attributes.update(cls.__dict__.keys())

    # Add instance attributes
    attributes.update(obj.__dict__.keys())
```

```
    return attributes

@staticmethod
def create_state_dict(obj: Any) -> Dict[str, Any]:
    """
    Create a dictionary of safe values from an object's state.

    Args:
        obj: Object to create state dict from

    Returns:
        Dict[str, Any]: Dictionary of safe values
    """
    state_dict = {}

    for attr_name in SafeLoaderUtils.get_class_attributes(obj):
        # Skip private attributes
        if attr_name.startswith("_"):
            continue

        try:
            value = getattr(obj, attr_name, None)
            if SafeLoaderUtils.is_safe_type(value):
                state_dict[attr_name] = value
        except Exception as e:
```

```
            logger.debug(f"Skipped attribute {attr_name}: {e}")

    return state_dict

@staticmethod
def preserve_instances(obj: Any) -> Dict[str, Any]:
    """
    Automatically detect and preserve all class instances in an object.

    Args:
        obj: Object to preserve instances from

    Returns:
        Dict[str, Any]: Dictionary of preserved instances
    """
    preserved = {}

    for attr_name in SafeLoaderUtils.get_class_attributes(obj):
        if attr_name.startswith("_"):
            continue

        try:
            value = getattr(obj, attr_name, None)
            if SafeLoaderUtils.is_class_instance(value):
                preserved[attr_name] = value
        except Exception as e:
```

```
                logger.debug(f"Could not preserve {attr_name}: {e}")

        return preserved


class SafeStateManager:
    """
    Manages saving and loading object states while automatically handling
    class instances and complex objects.
    """

    @staticmethod
    def save_state(obj: Any, file_path: str) -> None:
        """
        Save an object's state to a file, automatically handling complex objects.

        Args:
            obj: Object to save state from
            file_path: Path to save state to
        """
        try:
            # Create state dict with only safe values
            state_dict = SafeLoaderUtils.create_state_dict(obj)

            # Ensure directory exists
            os.makedirs(os.path.dirname(file_path), exist_ok=True)
```

```
        # Save to file
        with open(file_path, "w") as f:
            json.dump(state_dict, f, indent=4, default=str)

        logger.info(f"Successfully saved state to: {file_path}")

    except Exception as e:
        logger.error(f"Error saving state: {e}")
        raise

@staticmethod
def load_state(obj: Any, file_path: str) -> None:
    """
    Load state into an object while preserving class instances.

    Args:
        obj: Object to load state into
        file_path: Path to load state from
    """
    try:
        # Verify file exists
        if not os.path.exists(file_path):
            raise FileNotFoundError(
                f"State file not found: {file_path}"
            )
```

```
    # Preserve existing instances
    preserved = SafeLoaderUtils.preserve_instances(obj)

    # Load state
    with open(file_path, "r") as f:
        state_dict = json.load(f)

    # Set safe values
    for key, value in state_dict.items():
        if (
            not key.startswith("_")
            and key not in preserved
            and SafeLoaderUtils.is_safe_type(value)
        ):
            setattr(obj, key, value)

    # Restore preserved instances
    for key, value in preserved.items():
        setattr(obj, key, value)

    logger.info(
        f"Successfully loaded state from: {file_path}"
    )

except Exception as e:
```

```
            logger.error(f"Error loading state: {e}")
            raise



# # Example decorator for easy integration
# def safe_state_methods(cls: Type) -> Type:
#     """
#     Class decorator to add safe state loading/saving methods to a class.

#     Args:
#         cls: Class to decorate

#     Returns:
#         Type: Decorated class
#     """
#     def save(self, file_path: str) -> None:
#         SafeStateManager.save_state(self, file_path)

#     def load(self, file_path: str) -> None:
#         SafeStateManager.load_state(self, file_path)

#     cls.save = save
#     cls.load = load
#     return cls
```