A Technical Report on Petal Desktop Pet System

Sicheng Hu [1] **Keyue Xing** [1] **Zizhong Yang** [1]

1. Background and Objectives

In the context of widespread computer usage, digital fatigue and productivity bottlenecks stemming from prolonged screen time have become a prevalent challenge. **Petal** emerges as a novel solution to this issue. It is an innovative desktop application developed with *PyQt5*, whose core mission is to **seamlessly merge emotional engagement with productivity tools**, creating a digital companion that both alleviates stress and enhances efficiency.

The unique value of Petal is realized through its four key components:

1. **Emotional Interaction:**
   The application features anthropomorphic pet characters—*Doggy*, *Kitty*, and *Petal*—brought to life with sophisticated animations. They serve not merely as desktop ornaments but as points of emotional connection, effectively mitigating mental fatigue through periodic interaction.

2. **Scientific Efficiency:**
   It integrates evidence-based time management techniques, such as the Pomodoro Timer and Focus modes, to help users optimize their work rhythms and improve concentration, thereby ingeniously blending entertainment with high-efficiency work.

3. **High Customizability:**
   Leveraging its modular architecture, users can effortlessly personalize the pet's visual appearance and behavioral patterns through simple configuration parameters, catering to individual preferences.

4. **Cutting-Edge Technical Practice:**
   Petal itself serves as an outstanding technical learning platform, offering developers a valuable reference implementation for advanced programming paradigms, including multithreading and event-driven GUI design patterns.

Grounded in these features, Petal precisely serves distinct user segments. It offers **professionals** moments of relief from demanding workloads; provides **students and researchers** with tools to enhance focus; allows **customization enthusiasts** to enjoy the creativity of personalization; and stands as an excellent case study for **developers** mastering modern GUI development.

2. System Function

2.1. The Management Interface

`run_Petaler.py` provides a feature-rich interface with centralized management system through main window interface, sidebar navigation with multi-page switching capability, complete pet lifecycle management (`creation`, `modification`, `deletion`), and system-wide dark theme design with consistent visual hierarchy.

[1] Yuanpei College, Peking University, Class of 2024.

2.2. Petal Display and Animation System

2.2.1. Animation Playback Mechanism

The animation system is managed by `Animation_worker` thread class, featuring intelligent action selection based on probability distributions (`act_prob`), multi-frame sequence playback with configurable frame rates, composite action combinations like `["fall_asleep", "sleep"]`, and directional movement animations for bidirectional locomotion.

2.2.2. Animation Configuration System

Each pet's behavior is defined through dual configuration files: `pet_conf.json` for core parameters (`dimensions`, `scaling`, `refresh rate`, `action probabilities`) and `act_conf.json` for detailed action definitions (`image sequences`, `movement parameters`, `playback counts`).

2.3. Pet State Management System

2.3.1. State Value Types

The system tracks three critical metrics: HP (Health Points) represented by red progress bar showing vitality, EM (Emotion/Energy) indicated by yellow progress bar for mood status, and FC (Focus Capacity) blue progress bar activated during focus mode.

2.3.2. State Update Mechanism

Managed by `Scheduler_worker` timed tasks with periodic state reduction via `change_hp()` and `change_em()` methods. Configurable update intervals controlled by `hp_interval` and `em_interval` parameters, with real-time UI synchronization through signal-slot mechanisms.

2.4. User Interaction Features

2.4.1. Mouse Dragging Interaction

Complex interaction logic handled by `Interaction_worker` with drag detection initiated through `mousePressEvent`, dynamic following implemented in `mouseMoveEvent`, physics simulation triggered by `mouseReleaseEvent`, and velocity calculation based on mouse movement history analysis.

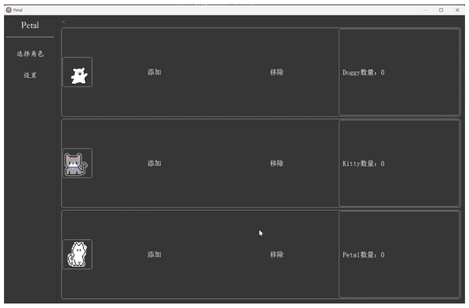
2.4.2. Physics Simulation System

Realistic physical behaviors are demonstrated in the following code:

```
# Calculating drag velocity from historical
    positions
self.settings.dragspeedx = (
    (self.settings.mouseposx1 - self.settings.
        mouseposx5) /
    self.settings.fixdragspeedx
)
'''...'''
# Gravity simulation
self.dragspeedy += self.pet_conf.gravity
# Friction damping
if speed > self.drag_speed_threshold:
    self.dragspeedx *= (1 - self.drag_base_friction)
    self.dragspeedy *= (1 - self.drag_base_friction)
```

2.4.3. Context Menu System

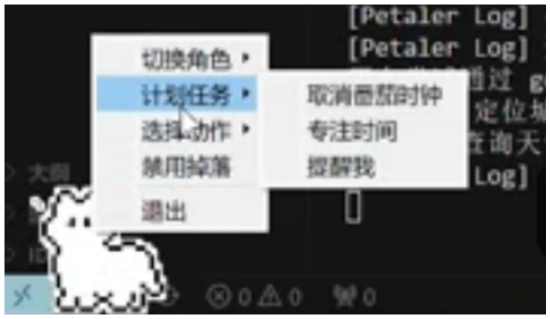
Right-click menu provides quick access to program termination, pet type switching, settings configuration, and feature shortcuts.



2.5. Task Scheduling and Productivity Tools

2.5.1. Pomodoro Timer (Tomato Class)

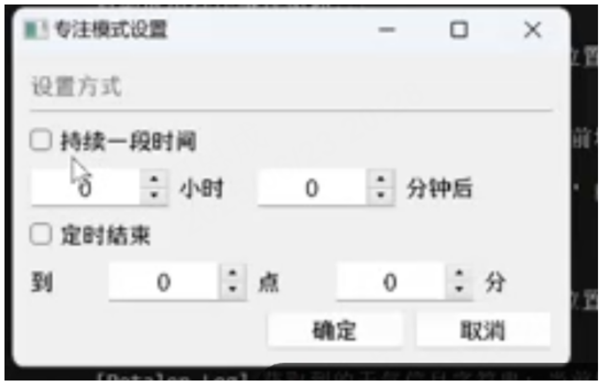
Advanced time management features including multi-cycle support with configurable session numbers, standard 25+5 minute cycles, conflict prevention with focus mode, status notifications for all session phases, and flexible cancellation at any time.



[Petaler Log]
[Petaler Log]

2.5.2. Focus Mode (Focus Class)

Two implementation modes available: duration-based with configurable hours/minutes and scheduled mode with specific end time. Exclusive visual effects during focus periods and completion alerts ensure effective time management.



专注模式设置

设置方式

☐ 持续一段时间

0 小时 0 分钟后

☐ 定时结束

到 0 点 0 分

确定 取消

2.5.3. Reminder System (Remindme Class)

Versatile reminder capabilities with one-time (relative/absolute timing) and recurring reminders (daily schedules or interval-based). Text management through editable right-side panel with auto-saving to `remindme.txt` file.

2.5.4. Time-based Greeting System

Context-aware greeting features with smart morning/afternoon/evening greetings, weather integration via `python_weather` library, and automatic geolocation using `geocoder` library.

2.6. Multi-pet Support System

2.6.1. Pet Type Management

Flexible pet management through central definition in `data/pets.json` configuration file, independent resource directories and configuration files per pet type, and runtime dynamic pet switching capability.

2.6.2. Resource Organization Structure

The resource directory follows hierarchical structure:

```
res/role/{pet_name}/
├── img.png
├── pet_conf.json
├── act_conf.json
└── action/
    ├── stand_0.png
    ├── leftwalk_0.png
    └── ...
```

2.7. System Tray Function

System tray integration enables minimized operation, quick-access context menu from tray icon, and status indication through tray icon visualization.

3. System Architecture and Module Design

3.1. Overall Architecture Design

The **Petal** project adopts a layered modular architecture consisting of three primary layers:

```
Petal
└── Application Layer
```

```
    Main entry points
    Main window management
    User interface components
  Business Logic Layer
    Animation control
    Interaction handling
    Task scheduling
  Data Layer
    Configuration management
    State management
    Resource management
```

3.2. Main Program Entry Design

```python
class AppManager:
    def __init__(self):
        self.app = QApplication(sys.argv)
        self.main_window = None
        self.init_platform_style()  # Set Fusion dark theme
        self.setup_logging()        # Configure logging system
```

The core responsibilities involve managing the application's entire lifecycle, from initialization to termination. This process includes several key setup procedures: initializing the global logging system to ensure robust error tracking, implementing the main application window using a singleton pattern to guarantee a single, consistent instance, and configuring the global visual style, with a specific focus on implementing a dark theme for the user interface.

3.3. Core Classes and Module Design

3.3.1. PetWidget Class - Main Window Controller

```python
def __init__(self, parent=None, curr_pet_name='', pets=()):
    # 1. Safety check
    if not curr_pet_name and not pets:
        raise ValueError("Must provide pet name or pet list")

    # 2. Base property initialization
    self.pets = pets
    self.settings = Settings()
    self.curr_pet_name = curr_pet_name

    # 3. UI component initialization
    self._init_ui()      # Create progress bars, labels, etc.
    self._init_widget()  # Set window properties

    # 4. Configuration loading
    self.init_conf(initial_pet_name_to_load)

    # 5. Background thread startup
    self.runAnimation()
    self.runInteraction()
    self.runScheduler()
```

The component's operation is driven by its core methods. The process begins with `init_conf()`, which loads necessary configurations and resources. Once initialized, the component actively listens for user input via the `mousePressEvent()`, `mouseMoveEvent()`, and `mouseReleaseEvent()` methods. The primary functionalities are exposed through specific task activation methods like `show_tomato()`, `show_focus()`, and `show_remind()`. Any resulting visual changes are rendered by the `set_img()` method, which serves as the central mechanism for image updates.

3.3.2. Animation_worker Class - Animation Processor

```python
def run(self):
    while not self.is_killed:
        self.random_act()  # Random action selection
        if self._check_pause_kill():
            break
        time.sleep(self.pet_conf.refresh)


def random_act(self):
    prob_num = random.uniform(0, 1)
    act_index =  sum(
        int(prob_num > self.pet_conf.act_prob[i])
                for i in  range( len(self.
                    pet_conf.act_prob)))
    acts = self.pet_conf.random_act[act_index]
    self._run_acts(acts)
```

The component relies on a signal system to drive its animations. The `sig_setimg_anim` signal is used to request updates to the visual content, while `sig_move_anim` handles requests for position changes. Both of these visual updates are ultimately applied when the `sig_repaint_anim` signal triggers a final repaint of the component, ensuring a smooth transition.

3.3.3. Interaction_worker Class - Interaction Handler

```
def mousedrag(self, act_name):
    """Handle mouse drag animation"""
    while self.settings.draging:
        self.sig_setimg_inter.emit()
        time.sleep(self.pet_conf.interact_speed /
            1000)

def drop(self):
    """Handle drop animation after release"""
    while not self.settings.onfloor:
        # Gravity and velocity calculation
        self.settings.dragspeedy += self.pet_conf.
            gravity
        # Position update
        self.sig_move_inter.emit(self.settings.
            dragspeedx, self.settings.dragspeedy)
```

3.3.4. Scheduler_worker Class - Task Scheduler

```
def __init__(self, pet_conf, parent=None, settings=
    None):
    self.scheduler = QtScheduler()
    self.scheduler.start()
```

```
def add_tomato(self, n_tomato=None):
    """Add Pomodoro task"""
    self.scheduler.add_job(
        func=self.run_tomato,
        trigger=date.DateTrigger(run_date=start_time
            ),
        args=[task_text],
         id=f'tomato_{start_time}',
        replace_existing=True
    )
```

The system's scheduling is powered by APScheduler, which is responsible for a variety of time-based tasks. It manages core functionalities, including the periodic reduction of state variables (`change_hp()`, `change_em()`), Pomodoro timing with notifications, and focus mode tracking. In addition, it supports user-centric features like custom reminders and automated, time-based greetings.

3.4. Configuration and Data Management

3.4.1. PetConfig Class - Configuration Management

```
@classmethod
def init_config(cls, pet_name:  str, pic_dict:
    dict):
    config_instance = cls()

    # 1. Load base config (pet_conf.json)
    with  open(pet_conf_path, 'r', encoding='UTF-8'
        ) as f:
        conf_params = json.load(f)

    # 2. Load action config (act_conf.json)
    with  open(act_conf_path, 'r', encoding='UTF-8'
        ) as f:
        act_conf = json.load(f)

    # 3. Create Act instances
    act_dict = {
        act_name: Act.init_act(act_params, pic_dict,
             scale, pet_name)
        for act_name, act_params in act_conf.items()
    }
    return config_instance
```

The component's behavior is highly configurable through several distinct parameter groups. These parameters cover the foundational visual and physical properties, such as dimensions, scaling, and refresh rate, alongside physics attributes like gravity and movement speeds. In addition, the system allows for detailed customization of its actions, from mapping specific behaviors to user inputs (e.g., `up`, `down`, `drag`) to defining the properties of random events, including their probability and display names. Finally, core mechanics are governed by state interval parameters, which set the timing for HP and EM reduction.

3.4.2. Act Class - Action Definition

```
class Act:
    def __init__(self, images, act_num=1, need_move=
        False, direction=None, frame_move=10.0,
        frame_refresh=0.04):
        self.images = images          # QImage
            sequence
        self.act_num = act_num        # Repeat count
        self.need_move = need_move    # Movement
            requirement
        self.direction = direction    # Movement
            direction
        self.frame_move = frame_move    # Distance
            per frame
        self.frame_refresh = frame_refresh  # Frame
            interval
```

3.4.3. Settings Class - Global State Management

```
class Settings:
    def __init__(self):
        # Image state
        self.current_img = QImage()
        self.previous_img = QImage()

        # Physics state
        self.onfloor = 1       # Ground detection
        self.draging = 0       # Dragging status
        self.set_fall = 1      # Drop permission

        # Drag parameters
        self.dragspeedx = 0.0
        self.dragspeedy = 0.0
        self.drag_base_friction = 0.1

        # Mouse history (velocity calculation)
        self.mouseposx1_5 = 0
        self.mouseposy1_5 = 0
```

3.5. Auxiliary Window System

3.5.1. extra_windows.py Module

The application includes a suite of specialized window classes, each tailored for a distinct task: the Tomato window for `Pomodoro timing`, the `Focus` window for managing focus sessions, and the `Remindme` window for configuring reminders. Despite their different functionalities, these windows are built on a shared foundation that ensures a consistent and robust user experience. Key common features include automatic font scaling, responsive layouts that adapt to different sizes, a unified visual style, and a signal-driven mechanism for data communication.

3.5.2. Window Interaction Pattern

```
class Tomato(QWidget):
    close_tomato = pyqtSignal()
    confirm_tomato = pyqtSignal( int)

    def confirm(self):
        n_tomato = self.n_tomato.value()
        self.confirm_tomato.emit(n_tomato)
        self.close()
```

3.6. Main Window Management System

The user interface is structured around several key components. A central `SideBar` handles navigation, controlling a `QStackedWidget` that serves as the main container for different views. These include a dedicated page for pet management, offering functionalities like adding, removing, and counting pets, and a separate page for system configuration.

```
def add_pet(self, pet_type):
    # Create new pet instance
    pet_instance = petal.create_pet_widget('data/
        pets.json', pet_type)
    self.pet_instances.append(pet_instance)
    self.pet_counts[pet_type] += 1
    self.update_controls(pet_type)

def remove_pet(self, pet_type):
    # Safely remove pet instance
    for inst in  reversed(self.pet_instances):
        if inst.curr_pet_name == pet_type:
            inst.close()
            inst.deleteLater()
            self.pet_instances.remove(inst)
            break
```

4. Key Technical Implementation Details

4.1. Multithreading Architecture Design

4.1.1. Inter-thread Communication Mechanism

```
# Connecting signals to slots in main thread
self.workers['Animation'].sig_setimg_anim.connect(
    self.set_img)
self.workers['Animation'].sig_move_anim.connect(self
    ._move_customized)
self.workers['Interaction'].sig_move_inter.connect(
    self._move_customized)
```

4.1.2. Thread Lifecycle Management

```
def stop_thread(self, module_name):
    if module_name in self.workers:
        self.workers[module_name].kill()
        self.threads[module_name].quit()
        self.threads[module_name].wait()
        del self.workers[module_name]
        del self.threads[module_name]
```

4.2. Animation System Implementation

4.2.1. Frame-based Animation Playback

```
def _run_act(self, act: Act):
    for _ in  range(act.act_num):
        for img in act.images:
            self.settings.current_img = img
            self.sig_setimg_anim.emit()
            time.sleep(act.frame_refresh)
            self._move(act)
```

4.2.2. Probability-driven Behavior Selection

```
{
    "random_act": [
        ["default"],
        ["left_walk", "right_walk", "default"],
        ["fall_asleep", "sleep"]
    ],
    "act_prob": [0.85, 0.1, 0.15]
}
```

The system implements a probability-weighted behavior selection mechanism where actions are chosen based on defined probability distributions. This enables natural-looking behavioral patterns while maintaining deterministic configuration control.

4.3. Configuration File System

4.3.1. Dynamic Image Loading

```
def _load_all_pic(pet_name:  str) ->  dict:
    """Load all image resources for specified pet"""
    pic_dict = {}
    role_path = f'res/role/{pet_name}'
    # Traverse image files in action directory
    action_path = os.path.join(role_path, 'action')
    for img_file in os.listdir(action_path):
        if img_file.endswith('.png'):
            img_path = os.path.join(action_path,
                img_file)
            img_name = os.path.splitext(img_file)[0]
            pic_dict[img_name] = QImage(img_path)

    return pic_dict
```

4.3.2. Configuration Validation and Error Handling

```
# Safety checks in Act.init_act
if not list_image_files:
    raise FileNotFoundError(f"Missing action images:
        {img_dir_pattern}_*.png")

try:
    original_image = pic_dict[image_key]
    except KeyError:
    raise KeyError(f"Image dictionary missing key '{
        image_key}'")
```

5. Project Summary and Reflection

5.1. Project Achievements Overview

This project successfully delivered a fully functional desktop pet system that merges a rich interactive experience with practical productivity tools. The pet's lifelike quality is driven by an advanced animation engine and a realistic physics model, while its utility is provided by a complete `Pomodoro`, `Focus`, and `Reminder` suite.

These features are built upon a robust and scalable technical architecture. A key strength is its modular, configuration-driven design, which not only enhances maintainability but also empowers non-developers to customize pet behaviors via JSON files. To ensure a smooth, non-blocking user experience, the system employs a three-thread concurrency model for animation, interaction, and scheduling, all underpinned by a comprehensive error handling and logging infrastructure for stability.

5.2. Technical Highlights Analysis

5.2.1. Animation System Innovations

```
# Cumulative probability implementation
act_index =  sum(
    int(prob_num > self.pet_conf.act_prob[i])
              for i in  range(
                  len(self.pet_conf.act_prob)))
```

This design creates more natural and randomized pet behaviors by implementing weighted random selection, avoiding monotonous fixed patterns.

Furthermore, each `Act` object combines image sequences with movement parameters, achieving perfect synchronization between visual animation and positional changes.

5.2.2. Physics Simulation Innovations

To create a realistic drag-and-drop experience, the system simulates a tangible sense of ”weight” and ”inertia”. This is achieved by calculating the pet's initial velocity from its recent mouse movement history, effectively capturing the momentum of the ”throw”. Once released, the pet's movement is governed by a physics model incorporating both gravity and friction, causing it to arc towards the bottom of the screen and gradually slow to a halt. Boundary collision detection confines this motion within the screen space, completing the simulation.

5.2.3. Task Scheduling Innovations

Task scheduling is managed by the APScheduler library, enabling flexible scheduling of one-time, recurring, and interval tasks. Its asynchronous execution model ensures a smooth, non-blocking user experience, while a built-in conflict detection mechanism gracefully handles overlapping tasks to maintain system stability.

5.3. Limitations and Shortcomings

5.3.1. Limited Multi-pet Interaction

The current implementation focuses on user-pet interaction, but does not yet support communication between the pets themselves. As a result, they act in isolation rather than as a cohesive group.

5.3.2. Missing Audio Feedback

While the application provides strong visual cues for interaction, it currently lacks an audio component. Consequently, the user experience is less engaging and misses the satisfying feedback that sound can provide.

5.3.3. Performance Optimization

To ensure maximum visual fluidity, the animations are rendered at a fixed high refresh rate. However, this static approach can create a performance bottleneck on low-end devices, leading to a compromised and stuttering user experience.

5.4. Lessons Learned and Benefits

Through this project, our team achieved significant growth in both technical execution and development process maturity. Technically, we mastered advanced GUI development with *PyQt5*, complex multithreading, the design of extensible configuration systems, and best practices for organizing large-scale Python projects. This technical expertise was forged through direct experience, as we learned critical operational lessons: the importance of rigorous requirement analysis to prevent refactoring, the efficiency gains from test-driven development, and the necessity of disciplined version control and documentation. These process insights transformed our approach, enabling us to build more robust and maintainable software.

5.5. Future Development Directions

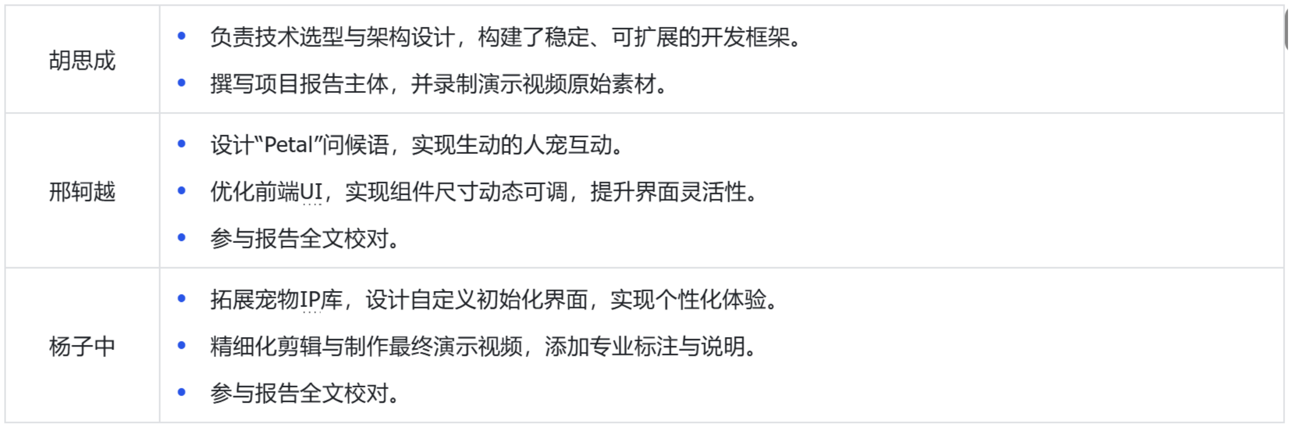
- Enhance AI Interaction:
  We plan to integrate machine learning models that enable the pet to develop adaptive behaviors based on user patterns. This will be complemented by an emotion recognition system to facilitate more dynamic, context-aware responses.

- Build a Connected Ecosystem:
  To move beyond a solo experience, we will try to implement a multi-user pet visiting system. This social feature will be supported by a robust cloud synchronization architecture to ensure all pet states are persistent and shared seamlessly across users and devices.

Code Availability

The source code for the **Petal** presented in this report is publicly available on GitHub. The repository can be accessed at: `https://github.com/Hsch22/Petaler`.

A. Team Contributions and Division of Labor



胡思成	• 负责技术选型与架构设计，构建了稳定、可扩展的开发框架。 • 撰写项目报告主体，并录制演示视频原始素材。
邢轲越	• 设计“Petal”问候语，实现生动的人宠互动。 • 优化前端UI，实现组件尺寸动态可调，提升界面灵活性。 • 参与报告全文校对。
杨子中	• 拓展宠物IP库，设计自定义初始化界面，实现个性化体验。 • 精细化剪辑与制作最终演示视频，添加专业标注与说明。 • 参与报告全文校对。