Survey Automation Tool - Modular Refactoring Implementation Plan

🎯 Phase-by-Phase Modularization Strategy

Phase 1: Core Infrastructure (30 minutes)

Create the foundational structure and extract core browser/session management.

Step 1.1: Create Project Structure

```bash
# Create the new modular structure
mkdir survey_automation
cd survey_automation

# Create all directories
mkdir -p {config,core,handlers,utils,models,data}
touch {config,core,handlers,utils,models}/__init__.py

# Copy your data file
cp ../enhanced_myopinions_knowledge_base.json data/
```

Step 1.2: Extract Core Components

Priority: Browser management and session handling (most complex, foundational)

Create: `core/browser_manager.py`

- Extract persistent session creation
- Extract stealth browser setup
- Extract manual navigation phase
- Keep all browser configuration logic

Create: `core/survey_detector.py`

- Extract tab detection logic
- Extract confidence scoring
- Extract survey state validation
- Keep all domain detection logic

Create: `core/navigation_controller.py`

- Extract next button finding
- Extract page navigation
- Extract consent/agreement handling
- Keep all navigation logic

Phase 2: Handler System (45 minutes)

Create the handler framework and extract all question handlers.

Step 2.1: Create Base Handler Framework

Create: `handlers/base_handler.py`

python

```python
from abc import ABC, abstractmethod
from typing import Optional
import random
import time

class BaseQuestionHandler(ABC):
  def __init__(self, page, knowledge_base, intervention_manager):
    self.page = page
    self.knowledge_base = knowledge_base
    self.intervention_manager = intervention_manager

  @abstractmethod
  def can_handle(self, page_content: str) -> float:
    """Return confidence score (0.0-1.0) for handling this question"""
    pass

  @abstractmethod
  def handle(self) -> bool:
    """Process the question and return success status"""
    pass

  def human_like_delay(self, min_ms=1500, max_ms=4000):
    """Human-like delays with variation"""
    delay = random.randint(min_ms, max_ms) / 1000
    time.sleep(delay)

  def get_user_profile(self):
    """Get user profile from knowledge base"""
    return self.knowledge_base.get("user_profile", {})

  def log_success(self, action_description: str):
    """Log successful action"""
    print(f"✅ {action_description}")

  def request_intervention(self, reason: str):
    """Request manual intervention"""
    return self.intervention_manager.request_manual_intervention(
      self.__class__.__name__.replace('Handler', '').lower(),
      reason,
      self.page.inner_text('body')
    )
```

Step 2.2: Extract Individual Handlers

Create focused handler files for each question type:

Create: `handlers/demographics_handler.py` **Create:** `handlers/brand_familiarity_handler.py` **Create:** `handlers/rating_matrix_handler.py` **Create:** `handlers/multi_select_handler.py` **Create:** `handlers/recency_activities_handler.py` **Create:** `handlers/trust_rating_handler.py` **Create:** `handlers/research_required_handler.py` **Create:** `handlers/unknown_handler.py`

Each handler inherits from `BaseQuestionHandler` and implements:

- `can_handle()` - confidence scoring
- `handle()` - actual question processing logic

Phase 3: Utility Services (30 minutes)

Extract utility functions into focused service modules.

Step 3.1: Create Utility Services

Create: `utils/knowledge_base.py`

- Extract KB loading/saving
- Extract user profile access
- Extract pattern matching

Create: `utils/intervention_manager.py`

- Extract manual intervention system
- Extract logging and reporting
- Extract question/answer capture (PART 2 features)

Create: `utils/research_engine.py`

- Extract Google search integration
- Extract result processing
- Extract caching logic

Create: `utils/reporting.py`

- Extract report generation
- Extract analytics

- Extract improvement suggestions

Step 3.2: Create Models

Create: `models/question_types.py`

- Extract question type detection
- Extract handler selection logic
- Extract pattern matching

Create: `models/survey_stats.py`

- Extract statistics tracking
- Extract progress monitoring
- Extract performance metrics

Phase 4: Main Entry Point (15 minutes)

Create clean entry point and wire everything together.

Step 4.1: Create Main Application

Create: `main.py`

python

```python
#!/usr/bin/env python3
"""
MyOpinions Survey Automation Tool v2.4.0
Modular architecture with enhanced session management.
"""

from core.browser_manager import BrowserManager
from core.survey_detector import SurveyDetector
from core.navigation_controller import NavigationController
from utils.knowledge_base import KnowledgeBase
from utils.intervention_manager import InterventionManager
from utils.reporting import ReportGenerator
from models.question_types import QuestionTypeDetector
from models.survey_stats import SurveyStats
from handlers.handler_factory import HandlerFactory

class SurveyAutomationTool:
  def __init__(self):
    # Initialize core components
    self.knowledge_base = KnowledgeBase()
    self.intervention_manager = InterventionManager()
    self.browser_manager = BrowserManager()
    self.survey_detector = SurveyDetector()
    self.navigation_controller = NavigationController()

    # Initialize analysis components
    self.question_detector = QuestionTypeDetector(self.knowledge_base)
    self.handler_factory = HandlerFactory(
      self.knowledge_base,
      self.intervention_manager
    )
    self.survey_stats = SurveyStats()
    self.report_generator = ReportGenerator()

  def run_persistent_session(self):
    """Main persistent session workflow"""
    # Implementation from your current method
    pass

  def run_legacy_method(self, start_from_dashboard=False):
    """Legacy workflow for compatibility"""
    # Implementation from your current method
    pass
```

```python
def main():
    print("🚀 MyOpinions Survey Automation Tool v2.4.0")
    print("Modular Architecture with Enhanced Session Management")

    tool = SurveyAutomationTool()

    choice = input("Choose method (1=Persistent, 2=Legacy Dashboard, 3=Legacy URL): ")

    if choice == "1":
        tool.run_persistent_session()
    elif choice == "2":
        tool.run_legacy_method(start_from_dashboard=True)
    elif choice == "3":
        tool.run_legacy_method(start_from_dashboard=False)

if __name__ == "__main__":
    main()
```

Create: `handlers/handler_factory.py`

python

```python
from typing import List, Optional
from .base_handler import BaseQuestionHandler
from .demographics_handler import DemographicsHandler
from .brand_familiarity_handler import BrandFamiliarityHandler
from .rating_matrix_handler import RatingMatrixHandler
from .multi_select_handler import MultiSelectHandler
from .recency_activities_handler import RecencyActivitiesHandler
from .trust_rating_handler import TrustRatingHandler
from .research_required_handler import ResearchRequiredHandler
from .unknown_handler import UnknownHandler

class HandlerFactory:
  def __init__(self, knowledge_base, intervention_manager):
    self.knowledge_base = knowledge_base
    self.intervention_manager = intervention_manager

    # Register all handlers
    self.handlers = [
      DemographicsHandler,
      BrandFamiliarityHandler,
      RatingMatrixHandler,
      MultiSelectHandler,
      RecencyActivitiesHandler,
      TrustRatingHandler,
      ResearchRequiredHandler,
      UnknownHandler  # Always last (fallback)
    ]

  def get_best_handler(self, page_content: str) -> BaseQuestionHandler:
    """Find the best handler for the current question"""
    best_handler = None
    best_confidence = 0.0

    for handler_class in self.handlers:
      handler = handler_class(
        None,  # Page will be set later
        self.knowledge_base,
        self.intervention_manager
      )
      confidence = handler.can_handle(page_content)

      if confidence > best_confidence:
        best_confidence = confidence
```

```
            best_handler = handler_class

        # Return instantiated handler
        return best_handler(
            None,  # Page will be set by caller
            self.knowledge_base,
            self.intervention_manager
        )
```

Phase 5: Migration and Testing (30 minutes)

Migrate functionality and validate everything works.

Step 5.1: Function-by-Function Migration

For each major component:

1. **Copy relevant methods** from `integrated_automation_system_v20_fixed.py`
2. **Update imports** to use new modular structure
3. **Adjust class initialization** to use dependency injection
4. **Test individual components** before integration

Step 5.2: Integration Testing

1. **Test browser management** - Ensure sessions work
2. **Test question detection** - Verify pattern matching
3. **Test handlers individually** - Each question type
4. **Test full workflow** - End-to-end automation
5. **Test both methods** - Persistent + Legacy modes

🚀 Implementation Benefits

Immediate Developer Experience Improvements:

- **Faster IDE loading** - No more 2100-line files
- **Better debugging** - Issues isolated to specific modules
- **Cleaner git diffs** - Changes focused on specific functionality
- **Enhanced autocomplete** - Better IntelliSense support

Maintenance Benefits:

- **Add new handlers easily** - Just create new file in handlers/
- **Update question detection** - Only touch models/question_types.py
- **Modify browser behavior** - Only touch core/browser_manager.py
- **Enhance reporting** - Only touch utils/reporting.py

Future Enhancement Benefits:

- **PARTS 2 & 3 implementation** - Clean separation makes feature addition easier
- **Testing** - Each module can be unit tested independently
- **Documentation** - Each module has focused responsibility
- **Collaboration** - Multiple developers can work on different modules

📋 Migration Checklist

Phase 1: Core Infrastructure

- [ ] Create project structure
- [ ] Extract `core/browser_manager.py`
- [ ] Extract `core/survey_detector.py`
- [ ] Extract `core/navigation_controller.py`
- [ ] Test browser sessions work

Phase 2: Handler System

- [ ] Create `handlers/base_handler.py`
- [ ] Extract 8 individual handlers
- [ ] Create `handlers/handler_factory.py`
- [ ] Test handler selection logic

Phase 3: Utility Services

- [ ] Extract `utils/knowledge_base.py`
- [ ] Extract `utils/intervention_manager.py`
- [ ] Extract `utils/research_engine.py`
- [ ] Extract `utils/reporting.py`
- [ ] Create `models/` modules

Phase 4: Main Entry Point

- [ ] Create `main.py`
- [ ] Wire all components together

- [ ] Test persistent session method
- [ ] Test legacy methods

Phase 5: Validation

- [ ] Run full survey automation
- [ ] Compare with original functionality
- [ ] Fix any integration issues
- [ ] Document new structure

⚡ Quick Start Commands

```bash
# 1. Create structure
mkdir survey_automation && cd survey_automation
mkdir -p {config,core,handlers,utils,models,data}
touch {config,core,handlers,utils,models}/__init__.py

# 2. Copy data
cp ../enhanced_myopinions_knowledge_base.json data/

# 3. Start with core extraction
# (Then follow phase-by-phase implementation)

# 4. Test the modular version
python main.py
```

🎯 Success Criteria

✅ All original functionality preserved ✅ 2100+ lines broken into 10-15 focused files ✅ Each file under 300 lines ✅ Clear separation of concerns ✅ Easy to add new question handlers ✅ Enhanced maintainability and testability

Ready to start the modularization? I recommend beginning with **Phase 1** to establish the core infrastructure, then proceeding systematically through each phase!