Git Engine Design

Detailed design documentation for the Git Engine component

Overview

The Git Engine is the heart of Solo Git, responsible for all Git operations, repository management, and workpad lifecycle. It provides a clean abstraction over Git while maintaining full compatibility with standard Git workflows.

Design Goals

1. **Simplicity**: Hide Git complexity from users
2. **Safety**: Prevent destructive operations
3. **Speed**: Optimize for fast operations
4. **Compatibility**: Remain compatible with standard Git

Architecture

Class Structure

```
GitEngine
├── Repository Management
│   ├── init_from_zip(zip_buffer, name) → repo_id
│   ├── init_from_git(git_url, name) → repo_id
│   ├── get_repo(repo_id) → Repository
│   └── list_repos() → List[Repository]
│
├── Workpad Management
│   ├── create_workpad(repo_id, title) → pad_id
│   ├── get_workpad(pad_id) → Workpad
│   ├── list_workpads(repo_id) → List[Workpad]
│   └── delete_workpad(pad_id) → void
│
├── Checkpoint System
│   ├── create_checkpoint(pad_id, message) → checkpoint_id
│   ├── restore_checkpoint(pad_id, checkpoint_id) → void
│   └── list_checkpoints(pad_id) → List[Checkpoint]
│
├── Merge Operations
│   ├── promote_workpad(pad_id) → commit_hash
│   ├── can_promote(pad_id) → bool
│   └── revert_last_commit(repo_id) → void
│
└── Query Operations
    ├── get_diff(pad_id, base='trunk') → diff_string
    ├── get_repo_map(repo_id) → FileTree
    └── get_history(repo_id, limit) → List[Commit]
```

Core Data Structures

Repository

```
@dataclass
class Repository:
    id: str                        # repo_a1b2c3d4
    name: str                      # Human-readable name
    path: Path                     # /data/repos/repo_a1b2c3d4
    trunk_branch: str              # main
    created_at: datetime
    workpad_count: int
```

Workpad

```
@dataclass
class Workpad:
    id: str                        # pad_x9y8z7w6
    repo_id: str                   # repo_a1b2c3d4
    title: str                     # add-login-feature
    branch_name: str               # pads/add-login-feature-20251016-1423
    created_at: datetime
    checkpoints: List[str]         # [t1, t2, t3]
    last_activity: datetime
```

Checkpoint

```
@dataclass
class Checkpoint:
    id: str                        # t1, t2, t3
    pad_id: str
    tag_name: str                  # pads/add-login-feature-20251016-1423@t1
    commit_hash: str
    message: str
    created_at: datetime
```

Implementation Details

Repository Initialization from Zip

```python
def init_from_zip(self, zip_buffer: bytes, name: str) -> str:
    """Initialize repository from zip file."""
    # 1. Generate unique ID
    repo_id = f"repo_{uuid4().hex[:8]}"
    repo_path = self.repos_path / repo_id

    # 2. Extract zip
    repo_path.mkdir(parents=True)
    with ZipFile(BytesIO(zip_buffer)) as zf:
        zf.extractall(repo_path)

    # 3. Initialize Git
    git = Git(repo_path)
    git.init()
    git.add('.')
    git.commit('-m', 'Initial commit from zip')
    git.branch('-M', 'main')

    # 4. Store metadata
    repo = Repository(
        id=repo_id,
        name=name,
        path=repo_path,
        trunk_branch='main',
        created_at=datetime.now(),
        workpad_count=0
    )
    self.repo_db[repo_id] = repo
    self._save_metadata()

    return repo_id
```

Repository Initialization from Git

```python
def init_from_git(self, git_url: str, name: str) -> str:
    """Initialize repository from Git URL."""
    # 1. Generate unique ID
    repo_id = f"repo_{uuid4().hex[:8]}"
    repo_path = self.repos_path / repo_id

    # 2. Clone repository
    repo_path.mkdir(parents=True)
    git = Git()
    git.clone(git_url, str(repo_path))

    # 3. Detect default branch
    git = Git(repo_path)
    trunk_branch = git.symbolic_ref('--short', 'HEAD').strip()

    # 4. Store metadata
    repo = Repository(
        id=repo_id,
        name=name or Path(git_url).stem,
        path=repo_path,
        trunk_branch=trunk_branch,
        created_at=datetime.now(),
        workpad_count=0
    )
    self.repo_db[repo_id] = repo
    self._save_metadata()

    return repo_id
```

Workpad Creation

```python
def create_workpad(self, repo_id: str, title: str) -> str:
    """Create ephemeral workpad."""
    # 1. Get repository
    repo = self.repo_db.get(repo_id)
    if not repo:
        raise ValueError(f"Repository {repo_id} not found")

    # 2. Generate workpad ID and branch name
    pad_id = f"pad_{uuid4().hex[:8]}"
    timestamp = datetime.now().strftime("%Y%m%d-%H%M%S")
    title_slug = title.replace(' ', '-').lower()
    branch_name = f"pads/{title_slug}-{timestamp}"

    # 3. Create branch from trunk
    git = Git(repo.path)
    git.checkout(repo.trunk_branch)
    git.checkout('-b', branch_name)

    # 4. Store workpad metadata
    workpad = Workpad(
        id=pad_id,
        repo_id=repo_id,
        title=title,
        branch_name=branch_name,
        created_at=datetime.now(),
        checkpoints=[],
        last_activity=datetime.now()
    )
    self.workpad_db[pad_id] = workpad
    repo.workpad_count += 1
    self._save_metadata()

    return pad_id
```

Checkpoint Creation

```python
def create_checkpoint(self, pad_id: str, message: str = "") -> str:
    """Create checkpoint (autosave) in workpad."""
    # 1. Get workpad
    workpad = self.workpad_db.get(pad_id)
    if not workpad:
        raise ValueError(f"Workpad {pad_id} not found")

    repo = self.repo_db[workpad.repo_id]
    git = Git(repo.path)

    # 2. Ensure on correct branch
    git.checkout(workpad.branch_name)

    # 3. Commit if changes exist
    status = git.status('--porcelain')
    if status:
        git.add('.')
        checkpoint_num = len(workpad.checkpoints) + 1
        commit_msg = message or f"Checkpoint {checkpoint_num}"
        git.commit('-m', commit_msg)

    # 4. Create lightweight tag
    checkpoint_id = f"t{len(workpad.checkpoints) + 1}"
    tag_name = f"{workpad.branch_name}@{checkpoint_id}"
    git.tag(tag_name)

    # 5. Update metadata
    workpad.checkpoints.append(checkpoint_id)
    workpad.last_activity = datetime.now()
    self._save_metadata()

    return checkpoint_id
```

Workpad Promotion

```python
def promote_workpad(self, pad_id: str) -> str:
    """Promote workpad to trunk (fast-forward merge)."""
    # 1. Get workpad
    workpad = self.workpad_db.get(pad_id)
    if not workpad:
        raise ValueError(f"Workpad {pad_id} not found")

    repo = self.repo_db[workpad.repo_id]
    git = Git(repo.path)

    # 2. Check if can fast-forward
    if not self.can_promote(pad_id):
        raise ValueError(
            f"Cannot promote {pad_id}: not fast-forward-able. "
            "Trunk has diverged."
        )

    # 3. Checkout trunk
    git.checkout(repo.trunk_branch)

    # 4. Fast-forward merge
    git.merge(workpad.branch_name, '--ff-only')

    # 5. Get commit hash
    commit_hash = git.rev_parse('HEAD').strip()

    # 6. Delete workpad branch
    git.branch('-D', workpad.branch_name)

    # 7. Clean up metadata
    del self.workpad_db[pad_id]
    repo.workpad_count -= 1
    self._save_metadata()

    return commit_hash
```

Can Promote Check

```python
def can_promote(self, pad_id: str) -> bool:
    """Check if workpad can be promoted (fast-forward)."""
    workpad = self.workpad_db.get(pad_id)
    if not workpad:
        return False

    repo = self.repo_db[workpad.repo_id]
    git = Git(repo.path)

    # Get merge base
    merge_base = git.merge_base(
        repo.trunk_branch,
        workpad.branch_name
    ).strip()

    # Get trunk HEAD
    trunk_head = git.rev_parse(repo.trunk_branch).strip()

    # Can fast-forward if merge base == trunk HEAD
    return merge_base == trunk_head
```

Rollback Last Commit

```python
def revert_last_commit(self, repo_id: str) -> None:
    """Revert last commit on trunk (for Jenkins rollback)."""
    repo = self.repo_db.get(repo_id)
    if not repo:
        raise ValueError(f"Repository {repo_id} not found")

    git = Git(repo.path)
    git.checkout(repo.trunk_branch)

    # Hard reset to previous commit
    git.reset('--hard', 'HEAD~1')

    logger.warning(
        f"Rolled back last commit on {repo.trunk_branch} "
        f"for repo {repo_id}"
    )
```

Get Diff

```python
def get_diff(self, pad_id: str, base: str = 'trunk') -> str:
    """Get diff between workpad and trunk."""
    workpad = self.workpad_db.get(pad_id)
    if not workpad:
        raise ValueError(f"Workpad {pad_id} not found")

    repo = self.repo_db[workpad.repo_id]
    git = Git(repo.path)

    # Get diff
    base_ref = repo.trunk_branch if base == 'trunk' else base
    diff = git.diff(base_ref, workpad.branch_name)

    return diff
```

Get Repository Map

```python
def get_repo_map(self, repo_id: str) -> dict:
    """Get file tree of repository."""
    repo = self.repo_db.get(repo_id)
    if not repo:
        raise ValueError(f"Repository {repo_id} not found")

    git = Git(repo.path)
    git.checkout(repo.trunk_branch)

    # Walk directory tree
    file_tree = self._walk_directory(repo.path)
    return file_tree

def _walk_directory(self, path: Path, max_depth: int = 5) -> dict:
    """Recursively walk directory and build tree."""
    if path.name.startswith('.') and path.name != '.gitignore':
        return None

    if path.is_file():
        return {
            'name': path.name,
            'type': 'file',
            'size': path.stat().st_size,
            'path': str(path)
        }

    if path.is_dir():
        children = []
        for child in sorted(path.iterdir()):
            child_node = self._walk_directory(child, max_depth - 1)
            if child_node:
                children.append(child_node)

        return {
            'name': path.name,
            'type': 'directory',
            'children': children,
            'path': str(path)
        }
```

Git Commands Reference

Used Commands

Operation	Git Command	Notes
Init	`git init`	Initialize new repo
Add	`git add .`	Stage all changes
Commit	`git commit -m "msg"`	Create commit
Branch	`git branch -M main`	Rename branch
Checkout	`git checkout -b name`	Create and switch
Merge	`git merge --ff-only`	Fast-forward only
Delete Branch	`git branch -D name`	Force delete
Tag	`git tag name`	Lightweight tag
Reset	`git reset --hard HEAD~1`	Hard reset
Diff	`git diff base..head`	Show differences
Merge Base	`git merge-base base head`	Find common ancestor
Rev Parse	`git rev-parse HEAD`	Get commit hash

Prohibited Commands

- ❌ `git push --force` - Never force push
- ❌ `git rebase` - Conflicts with fast-forward model
- ❌ `git cherry-pick` - Use merges instead
- ❌ `git stash` - Not needed with workpads

Error Handling

Common Errors

Error	Cause	Resolution
`RepositoryNotFound`	Invalid repo_id	Validate before operation
`WorkpadNotFound`	Invalid pad_id	Check workpad exists
`CannotPromote`	Not fast-forward	Trunk has diverged, manual merge needed
`CheckoutFailed`	Branch doesn’t exist	Verify branch name
`MergeConflict`	Conflicting changes	Shouldn’t happen with FF-only

Performance Considerations

Optimization Strategies

1. **Lazy Loading**: Only load metadata when needed
2. **Caching**: Cache frequently accessed repo data
3. **Batch Operations**: Group Git operations when possible
4. **Shallow Clones**: Use `--depth 1` for large repos (future)

Benchmarks (Target)

Operation	Time	Notes
Init from zip	< 10s	For typical app (~100 files)
Init from Git	< 30s	Depends on repo size
Create workpad	< 1s	Just branch creation
Apply patch	< 2s	Small patches
Promote	< 1s	Fast-forward merge

Future Enhancements

Phase 2+

- **Shallow Clones**: Faster init for large repos
- **Sparse Checkout**: Only checkout needed files

- **LFS Support**: Handle large binary files
- **Submodule Support**: Manage submodules
- **Remote Sync**: Push/pull to remote repos

Phase 3+

- **Conflict Resolution**: AI-assisted merge conflict resolution
- **Branch Policies**: Configurable branch protection
- **Hooks**: Pre-commit, post-merge hooks
- **Audit Trail**: Enhanced logging and tracking

Related Documents

- Test Orchestrator Design (./test-orchestrator.md)
- Phase 1 Overview (../phases/phase-1-overview.md)
- CLI Reference (../guides/cli-reference.md)

Last Updated: October 16, 2025