Software Principles

Personal principles of software design and development

Please find a collection of some the core principles, aims, traits and techniques that I attempt to instil in all software design, development and implementation regardless of size or scope.

Core Purpose

The core purpose of all software is to ultimately create and provide value:

- Increased productivity
- Increased convenience
- Increased ergonomics
- Provide a service
- Provide insight
- Solve a problem

Core Objectives

All software regardless of size or scope has x4 core objectives:

1.	Specification	Software meets the required functional specification.
2.	UI	UI meets the required user experience.
3.	Reliable/Robust	Software is reliable and robust throughout all permutations of use.
4.	Maintainable/Extensible	Software allows for maintenance, modification and extension.

These are the foremost basic/minimum objectives that all software must meet while having consideration for the following traits listed below.

Core Principles

All software design decisions regardless of size or scope should aim to continually work towards improving and refining the following x2 core principles:

1.	Simplicity	How simple the required functionality is implemented.
2.	Organisation	How well the implementation of the required functionality is organised.

All software should aim to be a combination of highly effective organisation and elegant simplicity.

Traits

Below is a summary of some of the desired and undesired traits of UI and source code design:

- The traits are applicable to all areas of design and implementation.
- The traits are applicable to all levels of design from high to low-level, they are independent of scale and scope.
- The traits naturally combine, compound, contribute, promote, re-enforce, build-in and inherently work towards achieving the x2 core principles listed above.
- The traits should form the core considerations behind all design and implementation decisions.
- The traits are nebulous and can be applicable across both UI and source code.
- The traits are guidelines not rules, such that appropriate discretion should be applied at all times.

	UI	**Source Code**	
Desired Traits:	Simple	Simple	Symmetry
	Organised	Organised	Clean
	Maintainable	Maintainable	Elegant
	Extensible	Extensible	Neat
	Intuitive	Readable	(Appropriate) Restriction
	Conventional	Efficient	(Appropriate) Flexibility
	Satisfaction	Clarity (in design and structure)	Confidence
	Familiar	Clarity (in role)	
	Easy Mental Map	Naming Conventions	
	Ergonomic	Consistent	
	Efficient	Conscientious	
	Smooth	Discipline	
	Polished	Safe	
	Balanced	Minimal Risk	
	Symmetry	Modular	
	Transparent/Invisible Technology	Cohesive	
	Closed Paths	Loose Coupling	
	‘Non-Technical’	Reassurance	
	‘Home Button’	Motivation	
Undesired Traits:	Frustration	Unnecessary layers of complexity	Verbosity/Ambiguity
	Unresponsiveness	Unnecessary layers of processing or handling	Misleading comments/code
	Impedance	Unnecessary dependencies	Redundant comments/code
	Hinderance	Unnecessary generalisation	Rigidity
	Clunkiness	Unnecessary feature anticipation	Brittleness
	Awkwardness	Premature optimisation	Clunky
		Over engineering	Anxiety

Abstract Qualities of a Good Programmer

Below are some of the general abstract qualities that should ideally be present within all good software developers:

- Learn/Understand: An ability to learn/understand systematically, thoroughly, deeply and consistently.
- Reference Material: An ability to organise their reference material so that retrieval is quick, easy, efficient and accurate.
- Assess: An ability to assess, understand an entity created, written or designed by other people.
- Update: An ability to update an existing entity with minimal/no disruption.
- UI/Source Code: An ability to apply all of the above desired traits and minimise all of the above undesired traits.
- Organise: An ability to assess and implement highly effective organisation of any context, environment or system.
- Simplicity: An ability to apply simplicity in all design and implementation at all levels.
- Teamwork: An ability to work, co-ordinate and communicate with others, ensuring that work is smooth and synchronous.
- Fault Find: An ability to fault find in any context or environment, an unglamorous and underappreciated but highly crucial skill.
- Discipline: An ability to maintain discipline at all times ensuring adherence to standards and procedures, no laziness or corner cutting.
- Commitment: An ability to have the drive and commitment to ensure that works are completed as required no matter how arduous.
- Inventiveness: An ability to find permutations and avenues of usefulness for users (and preferably for users who are non-technically minded).

The above qualities are applicable to all areas of software development including any technology, language, system, architecture, code base, program or source code.

Techniques/Methodologies

Below is a selection of some of the general approaches, techniques and methodologies which:

- Desired Traits: Promote desired traits.
- Undesired Traits: Minimise undesired traits.
- All Levels: Applicable to all levels of UI and source code, from high-level design decisions through to low-level implementation.

Simplicity

- Simplicity is the most important principle of all software.
- Simplicity is at the heart of all successful software.
- Simplicity should be the central aim and consideration within all design and implementation decisions.
- Software is about trying to achieve the necessary functionality through the simplest means possible.
- Software is not about delivering complicated programs, but making complicated programs simple.
- Software is not about deploying technology for technology's sake but only when there is a justified reason to do so.

Organisation

- Effective organisation is a subtle but crucial principle of all successful software.

- Effective organisation is a structural state achieved through ongoing considered design decisions which naturally lead to an implementation which is inherently:

    - Simple
    - Elegant
    - Reliable
    - Manageable
    - Flexible
    - Extensible

- Effective organisation naturally leads to many desired traits, the main reason why a particular piece of software may work extremely well is because it is organised extremely well, good engineering is good organisation.

Naming Convention

- Naming convention is critical to all aspects at all levels of design and implementation and should take into consideration:

    - Context: Appropriate to its surrounding context.
    - Extent of use: Appropriate to its extent of use within the program.
    - Scope: Appropriate to its scope within the program.
    - Confliction: Appropriate to not conflict but adheres with any other pre-existing naming convention.

- If an identifier is highly local, nested, not likely to conflict/confuse with others and restricted to a single block/method, it is appropriate to be:

    - Minimal length
    - Concise
    - Single letters

- If the identifier is global/static then it is appropriate to be:

    - Full length.
    - Highly descriptive and expressive to accurately reflect its role.
    - Critical that its role and purpose are clearly defined and maintained throughout the program.

- Identifiers of all kinds whether within source code or an external entity e.g. filename, should be as explicit and clear as reasonably possible, only contain specific references and fully utilise any wider relevant context. Under no circumstances contain vague or ambiguous elements for example:

    Instruction_Manual_New.pdf
    Instruction_Manual_Latest.pdf

- Naming convention should promote:

Readability	Confidence/reassurance	Fault prevention	Reduced brittleness/rigidity
Comprehension	Consistent behaviour	Self-documentation	Reduced confusion
Clarity (in role/design)	Consistent expectation	Reduced ambiguity	Reduced generalisation

Consistency

- Consistency should be applied to all aspects, including:
    - Structure/Organisation
    - Form
    - Paths of Execution Flow
    - Styling
    - Formatting
    - Implementation Techniques
- Consistency should be maintained throughout as defined by the originator of that particular consistency, for example an originator may be:
    - Specification documents
    - UI
    - Existing source code
    - Database schema
    - API
- Consistency should persist within the ordering of all naturally occurring groups/sets of field instantiation and method declaration.
- Consistency in naming convention should include:
    - Form and structure of adjectives, verbs and nouns within identifiers:

findSum()	iNumberOfItems
findProduct()	iNumberOfElements
findQuotient()	

    - Avoid loss of discipline or any deviation from any pre-existing naming conventions:

addNumbers()	numObjects
multiple()	Objects
divideValues()	nObjects

    - Avoid allowing consistency to become a preoccupation until it is reasonably expected that any established consistency will not change at a later date.

- Consistency should ideally be introduced within:
    - Greenfield
    - Refactoring
    - (Brownfield should follow any pre-existing consistencies/conventions and only introduce new consistencies/conventions which will not conflict).
- Consistency that is implemented within pre-existing or brownfield projects should:
    - Follow and maintain any pre-existing consistencies, forms, formatting or styling.
    - Even if those pre-existing consistencies are inferior or could be improved, they should still be adhered to throughout.
    - Avoid introducing any new consistencies which may conflict with any pre-existing consistencies.
- Consistency and the establishment of consistency should be pursued as early as possible but only once:
    - All wider context is fully and accurately comprehended.
    - Appropriate and optimum in accordance with high level design goals.
    - It can reasonably be expected that there will be no further change at a later date at greater cost.
- Consistency improves productivity by:
    - Increasing speed of development.
    - Reduces further design decisions in other areas of the program.
    - Reduces the need for continual referral to other parts of the program.
    - Gradually builds up and re-enforces confidence in the programmer.
- Consistency promotes:

Readability	Reliability	Consistent behaviour	Structural integrity
Productivity	Confidence/reassurance	Consistent expectation	Prevention of faults

Complementation, Grouping and Symmetry

- Complementation, grouping and symmetry involves:
    - Finding natural groupings to establish convention and consistency.
    - Finding natural complementation's to existing structures to establish symmetry and functional harmony.
- Complementation, grouping and symmetry is applicable to all aspects including:
    - Structure/Organisation
    - Form
    - Paths of Executional Flow
    - Styling
    - Formatting
    - Implementation Techniques
- Complementation, grouping and symmetry should be continually sort, found, identified and refined.
- Complementation, grouping and symmetry is typically more appropriate to greenfield works or refactoring, brownfield should adhere to any pre-existing forms.
- Complementation, grouping and symmetry promotes:

Readability	Reliability	Consistent behaviour	Comprehension
Productivity	Confidence/reassurance	Consistent expectation	Mental mapping
Structural integrity	Prevention of faults	Consistent design	Neatness/cleanliness

Task/Role

- A component's role of any size or granularity should be:
    - Clear, unambiguous and specific.
    - One job, one task, one responsibility as clearly described by its name.
- A component's role should not be:
    - Unnecessarily generalised.
    - Unnecessarily encompass multiple unrelated tasks.
    - Effectively a Swiss army knife within the operation of the program.
- A component's generalisation should be:
    - Only when absolutely necessary.
    - Only span the range of functionality that is appropriate and necessary to achieve the necessary wider functional requirement.

- Implemented with strict restriction of inputs to minimise potential error and improve testing.
- Implemented via generics or similar technology in order to maintain type safety.

- A component's role is generally dependent on its level within the program:

  Higher Level: Managerial Role

  - Component contents form a more human readable script of actions.
  - Component 'pulls the levers' of the lower components.

  Lower-Level: Worker Role

  - Component is more specific and dedicated to specific tasks.
  - Component performs, contains and hides low level actions using techniques which do not 'need to be known' by higher level components.

Methods/Functional Units

- Methods and functional units should ideally be:

  - Single Task: Perform a single specific or common task.
  - Highly Descriptive: Accurate, expressive and descriptive name/identifier.
  - Cohesive/Independent: Inputs provide all that is needed to compute the output with minimal or no dependency on external aspects.
  - Test: Easy to categorically test all permutations of use.
  - Building Blocks: Strong, solid, unambiguous and confident building blocks within the construction of the program.

- Methods and functional units should avoid:

  - Multi Task: Performing multiple tasks of varying scope, degree or character.
  - Vague: Obscure or vague naming convention making it difficult to discern its role or task.
  - Dependent: Inputs do not provide all that is necessary to compute its output with heavy dependency on external aspects.

- Methods and functional units should be created when:

  - Duplication: Functionality or code is duplicated and should be placed in a single location within its own function.
  - Multiple Use: Functionality or code is called multiple times and should be placed in a single location within its own function.

- Methods and functional units incur overhead therefore discretion should be applied to avoid excessive/unnecessary method creation/invoking whereby leaving the code inline where appropriate.

Gateway Restriction

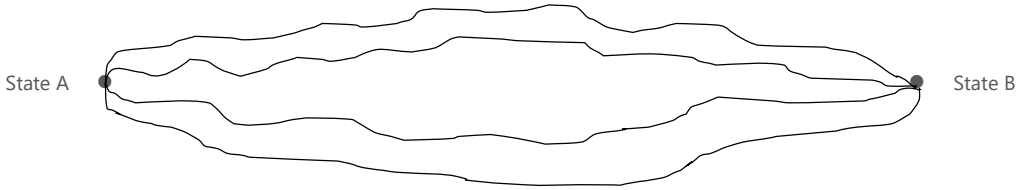
- Gateway restriction is the organisation of functionality in order to funnel all internal and external IO through as fewer gateways as possible in order to:
    - IO Control: Control IO flow to help guarantee reliability and prevention of faults.
    - Fault Prevention: Categorically prevent and rule out groups of potential faults.
    - Fault Find: Significantly aid fault finding and fault resolution.
- Gateway restriction should form part of the structural organisation via:
    - UI: Layout and functionality should be arranged to form natural gateways to minimise potential faults.
    - Source Code: Architecture and structure should be arranged to form natural gateways to minimise potential faults.
- Gateways should contain:
    - Minimal Access: Provide access which is only absolutely necessary.
    - Minimal Prevalence: Number of gateways kept to a minimum and avoid being unnecessarily and excessively scattered throughout a program.

Permutation Restriction

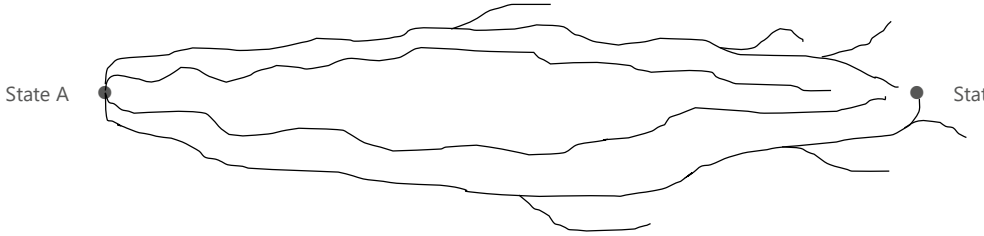
- Permutation restriction is the subtle organisation of functional units in order to avoid unnecessary and excessive permutations of use, in order to minimise:
    - Paths of Execution: Avoid unnecessary and excessive potential pathways of execution.
    - Fault Prevention: Categorically prevent and rule out whole groups of potential faults.
    - Fault Find: Significantly aid fault finding and fault resolution.
- Permutation restriction automatically rules out large categories of potential faults by:
    - Building in imperceivable restrictions which naturally rule out potential faults from ever occurring.
    - Guide the user/programmer without any degradation to the level of overall functionality being delivered.
- Permutation restriction emphasises:
    - Permutations which only provide the necessary functionality.
    - Permutations which keep complexity to a minimum.
    - Permutations which may reasonably be expected to provide the necessary foundation for future features.
- Permutation restriction avoids:
    - UI: Presenting the user with unnecessary or excessive permutations to navigate within the UI.
    - Source Code: Presenting the programmer with unnecessary or excessive permutations to navigate within a functional unit or component.

Closed Paths of Execution

- Paths of execution should form closed pathways whereby providing clear, unambiguous and definite paths between states.
- Paths of execution should:
    - Clearly indicate the natural/intended use and flow of an overall mechanism.
    - Consist of primary pathways within the program with minimal smaller/finer pathways.
    - Suitably connect clearly demarcated, compartmentalised and segregated states and functionality.



- Paths of execution should be arranged and constructed to naturally avoid:
    - Functional dead-ends, 'loose ends' or be ill-defined/incomplete.
    - Fanning out into a web of 'loose ends' which are not identified and captured by testing.
    - Appearing (from afar) closed, reliable and robust but containing a multitude of subtle permutations which can led to unexpected behaviour or faults:



- Closed paths can be naturally promoted through consideration of the following aspects:
    - UI Design: UI structural form, design and layout.
    - Simplicity: Simplicity in all levels/aspects of the UI/program naturally lead to more closed paths and fewer paths overall.
    - Restriction: Restriction and imperceivably guiding the user/programmer to funnel activity.

One Location

- Aspects of a similar nature should be arranged and organised to be physically located in one place, this may include:
    - Functionality
    - Characteristics
    - Role
    - State information
    - Interface/gateway
- Aspects of a similar nature should avoid being:
    - Scattered amongst various locations throughout a program.
    - Shared or duplicated amongst various variables throughout a program.
- Aspects of a similar nature should be physically located in one place in order to provide the following:
    - Lookup: Rapid lookup of those associated aspects.
    - Fault Find: Rapid assessment, diagnosis and resolution of any associated faults.
    - Fault Prevention: Reduction in the number of potential fault locations.
    - Mental Map: Easier to mentally map the overall structure, organisation and executional flow of the program.
    - Gateway: Implement funnelling of execution flow through minimal gateways.
    - Confidence: Knowing that all relevant items are located in one place.

Comments

- Comments should only be used when absolutely necessary. Source code should be:
    - Self-Commenting: Inherent design and naming convention should be accurate and effective in providing the necessary clarity of role.
- Comments should only be used when they are:
    - Fixed: Section of source code will likely remain fixed and unaltered but would be particularly helpful for the next persons clarification.
    - Header: Header at the top of a file which will likely remain fixed in providing an overview and explanation of its contents intended use.
- Comment Header at the top of a source code file is appropriate given:
    - Centralised: Header is in a single central location at top of file and not spread throughout the program in multiple locations.
    - Self-contained: Header is self-contained and in a dedicated location.
    - Maintainable: Header is easy to maintain.
    - Accurate: Header is less likely to go out of date without being noticed (rather than multiple comments spread throughout a file).

- Comments should not be:
    - Excessive: Excessive, unruly comments can become more of a hinderance than a benefit.
    - Unmaintained: Inaccurate, misleading or out of date comments cause clutter and are typically more detrimental than no comment at all.

Outside - In

- Design should be constructed and devised from:
    - High level towards the low level.
    - Outside – in.
    - Full comprehension of the broad requirements followed by the internals.
- Design should not be constructed and devised from:
    - Low level towards the high level.
    - Inside – out.
    - Avoid constructing any internal design and implementation without fully comprehending the outermost external requirements.
- Design that is derived from 'inside-out' can lead to:
    - Suboptimal architecture
    - Suboptimal implementation
    - Loss of focus
    - Loss of direction
    - Excessive complexity
    - Unreliability
    - Being painted into a corner

The Next Person

- The next person is the next person to use/read the code.
- Considerations for the next person should continually involve:

    'What would naturally make this easier to work with'
    'What would naturally be of benefit'
    'What would be the preferred method of implementation'
    'What conventions would the next person most likely be familiar with'

- Considerations for the next person reduces the potential of them having to request further clarity consuming both your time and theirs.

- Considerations for the next person are a valuable, productive, efficient and beneficial use of time due to:
    - Promotion of source code maintainability.
    - Reduction in any gradual degradation of source code quality.

Human Error

- Human error is a fact of life, it will always happen.
- Human error should be minimised through the inherent design, structure and layout of a UI/program, for example:
    - UI: Layout should have an inherent design which naturally reduces errors by making them impossible to occur.
    - Source Code: Program should have an inherent design which naturally reduces errors by making them impossible to occur.
- It is a lot more reassuring knowing that a potential fault cannot occur if it is impossible for that fault to occur.

Future Features

- Future features should only be included prior to their use or requirement when:
    - Likelihood: It is reasonably likely that they will genuinely become a future requirement.
    - Ease/Cost: It is reasonably easy and economical to include them now rather than at a later date.
    - Disruption: It is less disruptive to include them now rather than at a later date.
    - Risk: It is more reliable and safer to include them now rather than at a later date.
    - Stability/Security: It is not going to impinge upon the immediate stability or security of the current software.
- Future features should be designed to enable the ‘bolting on’ of additional functionality opposed to modifying current functionality.