Paul Surridge

Software Principles

Personal principles of software design and development

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

Core Purpose

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

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

Core Aims

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

1.	Specification	Program meets the required functional specification.
2.	UI	UI meets the required user experience.
3.	Reliability/Robustness	UI and program is reliable and robust throughout all permutations of use.
4.	Maintenance/Extensibility	UI and program allows for maintenance, modification and extension.

Core Principles

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

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

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

- All good software stems from maximising the above x2 principles.
- All bad software stems from disregarding the above x2 principles.

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, development and implementation within both the UI and source code.
- The traits are applicable to all high-level and low-level aspects within both the UI and source code, 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 aims of all good software.
- The traits should form the central core of consideration during all design and implementation decisions.
- 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
	Intuitive	Readable	Elegant
	Conventional	Efficient	Neat
	Satisfaction	Clarity (in design and structure)	(Appropriate) Restriction
	Familiar	Clarity (in role)	(Appropriate) Flexibility
	Easy Mental Map	Naming Conventions	Confidence
	Ergonomic	Consistent	Reassurance
	Efficient	Conscientious	Motivation
	Smooth	Discipline	
	Polished	Safe	
	Balanced	Minimal Risk	
	Symmetry	Modular	
	Transparent/Invisible Technology	Cohesive	
	Closed Paths	Loose Coupling	
	'Non-Technical'		
	'Home Button'		
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

Techniques/Methodologies

Below are some of the general techniques and methodologies in order to:

- Promote the desired traits.
- Minimise the undesired traits.
- Applicable to all aspects of UI and source code from high-level design decision making through to low-level implementation.

Simplicity

- Simplicity is the core and 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 functionality should be achieved through the simplest means possible.
- Software should not be about delivering complicated programs, but making complicated programs simple.
- Do not deploy technology for technology's sake.
- Do not implement techniques for implementation's sake.

Organisation

- Effective organisation should be a central aim and consideration within all design and implementation decisions.
- Effective organisation should be instilled throughout all aspects of a project.
- Effective organisation is defined broadly as implementing complex functionality through:
    - Clear operational structure.
    - Clear delineation and segregation of functionality, roles and responsibilities.
    - Structure that achieves its functional goals through the utmost simplicity and elegance.
    - Structure that is maintainable and extensible.
    - Structure that is inherently robust and naturally prevents potential faults from occurring.

Naming Convention

- Naming convention is critical to all aspects of design and implementation.
- Naming convention 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, expressive and 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 be to promote:

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

Consistency

- Consistency should be a central aim and consideration within all design and implementation decisions.
- 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

    - Class/Abstract Class/Interface/Enum internal structure.

- Consistency should extend out and match all external aspects where appropriate, for example:

    - Database schema
    - API

- Consistency should exist within the ordering of all naturally occurring groups/sets of field instantiation and method declaration/structure.
- Consistency in naming convention should include:

    - Consistent 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 consistent 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 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 can be improved, they should remain adhered to and be maintained 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 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
Prevention of faults	Confidence/reassurance	Consistent expectation
Structural integrity	Productivity	

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
Prevention of faults	Confidence/reassurance	Consistent expectation
Structural integrity	Productivity	Consistent design
Comprehension	Mental mapping	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 become 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.

The Next Person

- The next person should be a central consideration within all design and implementation decisions.
- 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.

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 and leave the code inline where appropriate.

Low Level

- Low-level source code should always aim to promote:
    - Clarity and readability.
    - Benefit the next person.

- Low-level source code should carefully consider:

Structure/Arrangement:	Appropriate and clear structure to aid clarity, readability and comprehension.
Text Alignment:	Appropriate alignment of text to illustrate units of functionality.
Spacing:	Appropriate line spacing of text to illustrate units of functionality.

- Low-level source code should:

    - Not bunch up source code un-necessarily.
    - Not provide inadequate space around appropriate natural groupings of statements, expressions and functionality.

One Location

- Aspects of a similar nature should be arranged and organised to be physically located in one place, this includes:

    - 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 benefits:

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.
Organisation:	Simpler and more effective organisation.
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.

Solid Building Blocks

- Reliability and structural integrity should be at the core and built into all software building blocks from the smallest to largest.
- Reliability and structural integrity, as in the real world, is dependent upon the individual reliability of its constituent components.
- Reliability and structural integrity are accumulative, all functional components should provide their own inherent stability to the overall structure and executional flow of the program.

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 Number: 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 functionality being delivered.

- Permutation restriction emphasises:

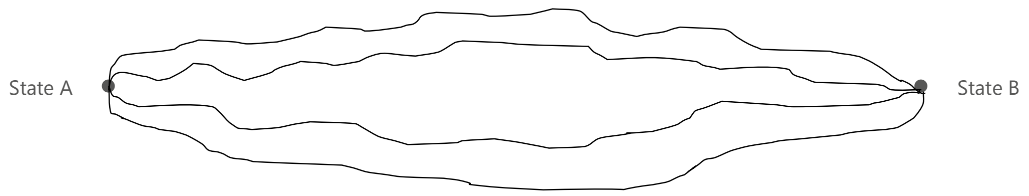
    - Permutations which only provide the necessary functionality.
    - Permutations which keep complexity to a minimum.
    - Permutations which may reasonably be expected to provide the necessary foundation for future features.

- Permutation restriction avoids:

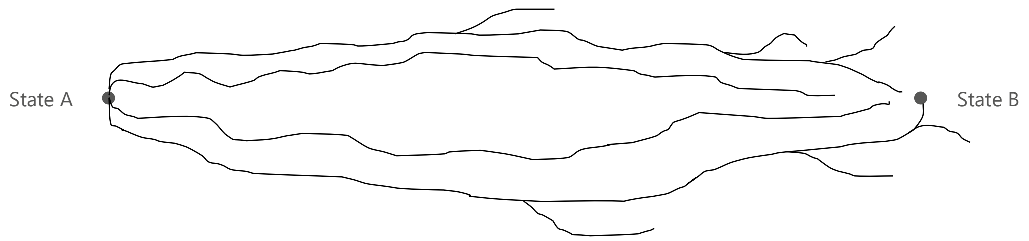
    - UI: Presenting the user with unnecessary or excessive permutations to navigate within the UI.
    - Source Code: Presenting the programmer with unnecessary or excessive permutations to navigate within a functional unit or component.

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 highways or trunk roads 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.

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.

Human Error

- 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 reassuring and a lot less anxiety inducing to know that a fault or potential fault cannot occur if it is impossible for that fault to occur.'

- A simple human error elimination analogy:
    - A fuel station has two underground storage tanks one for petrol and the other for diesel.
    - A fuel tanker driver is prevented from putting the wrong fuel in the wrong tank simply due to the unique shape of the hose connector.
    - This simple, cheap and effective design solution categorically rules out any potential fault by making it impossible to occur.
    - This conceptual approach should be applied throughout all levels and aspects of UI and source code.

Right Tool for the Right Job

- Time should be spent researching and ensuring that the right tool/implementation is being used for a particular functional role, due to:

Time Save:	Save significant time compared to developing own implementation and potentially reinventing the wheel.
Reliability:	Previous engineers likely spent significant time developing, testing and refining that tool.

- This principle is of course just as applicable within software development as in the real world.

Custom v Framework

- Discretion and careful consideration should be deployed at all times upon whether to use:

Framework:	Components from a framework (or pre-existing components from other projects/code base).
Custom:	Components created and custom built.

- Overuse of:

Framework Components:	May lead to clunky, verbose, awkward, brittle and an ill-fitting combination of components.
Custom Components:	May lead to longer development time, unnecessary work, less reliability, less maintainability and less portability.

- Optimum approach ideally being a considered, balanced combination of both framework and custom components, where:

Framework:	Framework components are used to provide the bedrock, interfacing, 'heavy lifting' and necessary boiler plate code.
Custom:	Custom components are created or refined from pre-existing components to provide specific unique functionality.

- If a program was a stone drywall:

    - Framework components should be the large stones which provide:

        - The broad organisational structure and stability.
        - Which may remain unaltered or slightly reshaped and refined as required.

    - Custom components should be the smaller stones:

        - That may sit in between the larger stones.
        - Provide the necessary minor but still critical roles in the overall functionality and stability.

Clarity in Brief

- Clarity in brief and ensuring that it is comprehensive, clear, understood and agreed by all parties is critical for:

Structural Organisation:	The initial design is accurate and appropriate for the specification.
Direction:	The initial design goes off in the appropriate direction.

- The relatively minimal time spent ensuring absolute clarity in the brief provides an ever-increasing amplified pay-out in productivity throughout the project.
- The relatively minimal time spent can save many orders of magnitude in time and complication throughout the life of the project.

Productivity/Speed

- Productivity and speed are primarily but not exclusively derived from:
  - Design Decisions:

    Consistently good design decisions throughout all levels lead to efficient implementation and not having to backtrack.
  - Anticipation:

    Consistently good foresight in predicting potential pitfalls or alterations by incorporating subtle design techniques to allow for changes to be applied with relative ease and minimal disruption.
  - Testing:

    Consistent testing of individual building blocks as they are produced, ensuring that all components are robust reducing the chance of later discovering an inherent structural flaw.
  - Reference Material:

    Efficient, rapid and accurate recall of reference material, working examples and being able to find the most appropriate solution with speed.

Reference Material

- Reference material should be:
  - Comprehensive
  - Accurate
  - Organised
  - Efficient
- Reference material should enable rapid search and retrieval of:

• Research:	Research notes or summaries of a particular topics or subjects.
• Sample code:	Sample code extracts from previous working implementations.
• Reminder:	Reminder of subtle considerations, advantages, disadvantages and pitfalls of the particular component or technique.

- Reference material should ideally be:

• OEM:	Sourced close to the original equipment manufacturers or managers of a technology, avoiding 2nd, 3rd hand replication.
• Personal:	Personalised, refined and tailored for own personal preference, comprehension and consumption.

- Caution should be applied in avoiding an over reliance of using blocks of working code from other projects, for it can quickly result in a program of little coherence or comprehension of its operation.

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 that 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:

Central:	Header is in a single central location at top of file and not spread throughout in multiple locations.
Self-contained:	Header is self-contained and a dedicated location.
Maintain:	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.

A Piece of Paper

- A piece of paper and the ability to draw something out is still one of the most effective means of constructing a design, algorithm or identifying a problem.

UI:	Draw out initial layout and pages.
Source Code:	Draw out initial design and architecture.
Algorithm:	Draw out the memory, components and actions that constitute the algorithm.

- A piece of paper provides:

Overview:	Ability to step back and observe the design, algorithm or problem from a clear perspective.
Visualisation:	Ability to better mentally map and comprehend any design, algorithm or problem.
Speed:	Ability to rapidly establish a design or identify design flaws. There is still nothing more ergonomic than pen and paper.

- If in doubt draw it out.

Outside - In

- Design should be derived from:

  - High level towards the low level.
  - Outside – in.
  - Full comprehension of the broad requirements followed by the internals.

- Design should not be derived 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

Only Change What Needs to Change

- During maintenance, works should only ever change the absolute minimum in order to implement and achieve the required functionality.

Is it needed?

- All design decisions should only include functionality that has a genuine need or reasonably likely to be a genuine need.