design-types.net

Dimensions

There are four dimensions:

simple vs. powerful (green)
abstract vs. concrete (blue)
pragmatic vs. idealistic (red)
robust vs. technologic (yellow)

Each dimension represents two contrary but related perspectives on design and each argument card provides a distinct aspect relevant from this perspective.

simple stands for simple solutions, no magic, nothing sophisticated but easy to read and maintain.
powerful stands for foresighted solutions, generic and flexible.
abstract stands for having the big picture in mind and keeping the bird's eye view.
concrete stands for knowing the details, being able to breathe code likea fish can breathe water.
pragmatic stands for creating value with a very customer-focused perspective.
idealistic stands for focusing on quality and professionalism, for avoiding dirty hacks and 80 percent solutions.
robust stands for stability and reduction of risks.
technologic stands for the potential new technology offers.

design-types.net

Card-based Discussions

Use Case

Use this when discussions about software design are not productive because:

- some participants have difficulties to express their thoughts
- the discussion itself lacks sound argumentation or
- a single developer dominates the discussion.

Preparation

1) Start with reading the cards carefully and get familiar with the arguments.
2) Start with the basic set (❯❯❯) and build a deck of not more than 20 cards.
3) Each developer participating in the discussion should have an own deck.

Basic Rules

1) When you play a card, explain how this argument is applied in the concrete situation.
2) You may only play one card at a time. Then it's your colleague's turn. Either use one card from your deck or the card your colleague has just played.
3) You can also play a question or action card but also only one at a time.

design-types.net

Games

Use Case and Preparation

Use the quiz and the learning game to get familiar with the Design Cards and with arguments and principles of software design. Use a single card set and read the cards before playing. Start with the basic set (❯❯❯) and add the other cards once you are familiar with it.

More Game Ideas

There are alternative rules and more ideas for games online: design-types.net/cardgames

Quiz

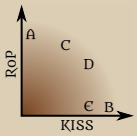
Rules for 2+ Players

1) Shuffle the cards.
2) Take turns. When it's your turn, one of the other players draws a card and reads the title to you.
3) If you can correctly explain the card based on the title, you get a point.
4) If your answer is not correct the other players get a chance.
5) In any case read the card aloud. Then it's the next player's turn—even if this person already got a point by explaning your card.
6) The game ends after 5 rounds (adjust based on the time you want to spend).
7) The player with the most points wins.

design-types.net

Conflicting Principles

Each principle describes a certain aspect of the problem. KISS for example tells you, that a solution is better when it is simpler. RoP on the other hand says that a more generic solution is better than a specific one. This is a typical example of two con-flicting principles. Both principles are valid but there is no totally generic solution that is also maximally simple.



A: generic but complex
B: simple but specific
C, D: typical trade-offs
E: bad solution

If there are two competing solutions, there are two quite different scenarios for the discussion:

a) One of the solutions is strictly better than the other, so in the current example it's simpler *and* more generic.

b) Both solutions are Pareto optimal, i.e. one of the solutions is simpler, the other more generic. Then you have to make a trade-off.

Keep sure that you find out which kind of situation you have.

Design Cards

Design Cards are a means to improve discussions among software developers.

Use them for:

- Code reviews
- Pair programming
- Architectural discussions
- Justifying decisions
- Learning design aspects

Use arguments from different dimensions to ensure that you don't miss important aspects. Use question cards to point out relevant questions and use action cards to make progress if a discussion gets stuck.

Detailed instructions:
design-types.net/cards

Imprint

Matthias Wittum & Christian Rehn GbR
Goethestr. 23
76474 Au am Rhein / Germany
email@design-types.net

design-types.net

Learning Game

Rules for 4+ Players

1) Split up into two teams who will play against each other.
2) Remove action and question cards and shuffle the rest.
3) Take turns. When it’s your turn, draw a card, read it quietly and try to explain the card without mentioning the words in the title or synonyms/antonyms thereof.
   TIP: Read the card carefully and use the examples given.
4) While you explain, your team member(s) have 90 sec. to guess the title. If they manage to do so, your team gets a point.
5) If your team members couldn’t guess correctly, the opposing team gets one single guess to get a point for themselves.
6) In any case read the card aloud. Then it’s the turn of the other team.
7) Every player should get the chance to explain and to guess. So also take turns within the teams.
8) The game ends after each team has read 5 cards (adjust based on the time you want to spend).
9) The team with the most points wins.

design-types.net

Card-based Discussions

Advanced Rules

Moderator: It can be helpful to have a kind of moderator. This person should get the orange cards (questions and actions).

All-in: Think both for ten minutes and lay down your top three arguments at the same time. Then explain and start the discussion.

Other Use Cases

Base agreement: Decide together on a particular dimension, aspect, or card that is especially important for your project. Put it on a special place at the table (as reminder) so everybody is aware of this focus during the card based discussion.

Focus: When you realize that you often neglect a certain aspect, tape the respective card on your screen.

There are more rules and sugestions online:

design-types.net/cards

design-types.net

KISS: Keep It Simple Stupid

KISS

»Simple means readable, maintainable, and less error-prone. Overengineering is harmful.«

Complex code contains more bugs and it has to be maintained (maybe even by other people). To others, it may seem obscure which can lead to frustration and bad code quality. Striving for simplicity means to avoid having large modules (methods/classes/...), many modules (methods/classes/...), as well as inheritance, low-level optimization, complex algorithms, fancy (language) features, configurability, etc.

↑CF, ↓RoP, ↓NFR, ↓LC

design-types.net

YAGNI: You Ain't Gonna Need It

YAGNI

»It's currently not necessary, and we even have to maintain it!«

Code needs to be maintained. The more you have, the more complexity there will be. Adding features and capabilities that are not used (yet), wastes time twice: When you write the code and when you change or just read it. This becomes even more painful when you finally try to remove this dead code. So avoid runtime-configuration, premature optimization, and features that are only there "for the sake of completeness". If they are needed, add them later.

↑CF, ↓PSPG, ↓TP, ⇅FP

design-types.net

EUHM: Easy to Use and Hard to Misuse

EUHM

»It shouldn't require much discipline or special knowledge to use or extend that module.«

Some day there will be a new colleague who hasn't read the docs. Some day it will be Friday evening right before the deadline. No matter how disciplined or smart you are, some day somebody will cut corners. So better have the obvious way of usage be the correct one. Have the compiler or the unit tests fail in case of errors and keep sure that changing a module does not require much understanding.

↑ML, ↑PSU, ↑UP, ⇅KISS

design-types.net

RoE: Rule of Explicitness

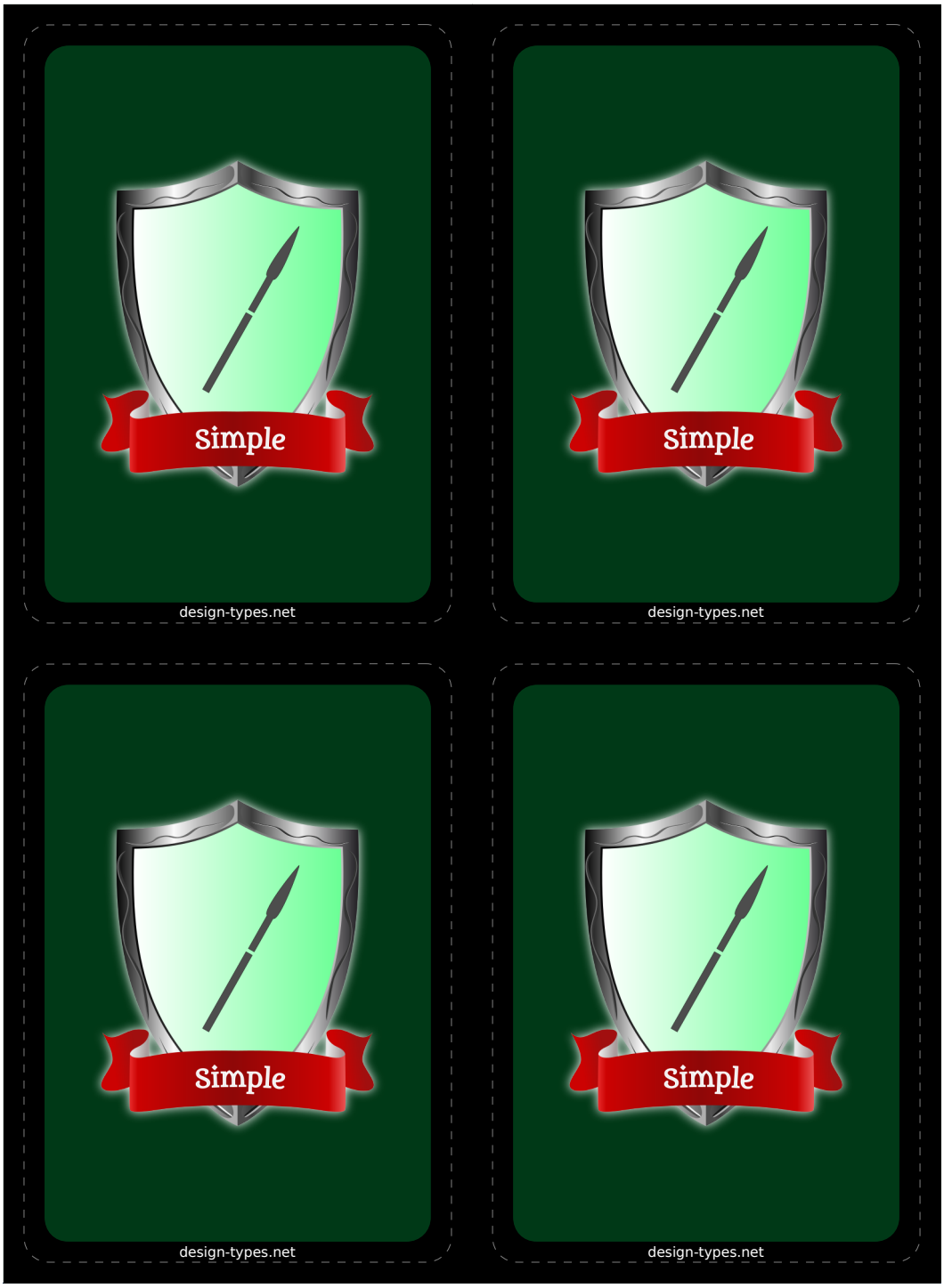
RoE

»Explicit solutions are less error-prone and easier to understand and debug.«

Implicit solutions require the developer to have a deeper understanding of the module, as it is necessary to "read between the lines". Explicit solutions are less error-prone and easier to maintain. So better avoid configurability, unnecessary abstractions and indirection (events, listeners, observers, etc.).

↑KISS, ↓RoP, ↓LC, ⇅FP

design-types.net


Simple
design-types.net
Simple
design-types.net
Simple
design-types.net
Simple
design-types.net

RoP: Rule of Power

RoP

»Foresighted, generic solutions are reusable and future requirements will be addressed, too.«

A powerful solution is better than a less potent one. Foresighted solutions reduce the necessity of refactoring and are more stable over time. Generic solutions often need less code and additionally offer extensibility by design. So better use abstractions, indirection, GoF patterns, polymorphism, etc.

↑FP, ↑DRY, ↓YAGNI, ↓CF

design-types.net

FP: Flexibility Principle

FP

»We have to make sure that we can change that later on.«

While it is often not necessary to implement a fully generic solution, in many cases it is important to be flexible. Even if a generic solution isn't implemented right away, it must still be possible to do so. E.g. if you don't want to implement runtime-configurability, at least have a constant ready to be made configurable. Make sure that the solution does not spoil or hinder future changes or enhancements.

↑RoP, ↑LC, ↑ML, ↓ICC

design-types.net

NFR: Non-Functional Requirements

NFR

»We have to think about NFRs now. Adding these qualities later will be very hard.«

Software needs to be efficient, scalable, secure, usable, maintainable, testable, resilient, reliable, compliant with (data privacy) regulations, etc. These qualities have a huge impact on the architecture. You might need to choose certain technologies for performance, use microservices for scalability, or provide redundant subsystems for reliability. Thinking about this later results in waste and additional cost/effort.

↑ML, ↓YAGNI, ↓KISS, ⇅FP

design-types.net

ECV: Encapsulate the Concept that Varies

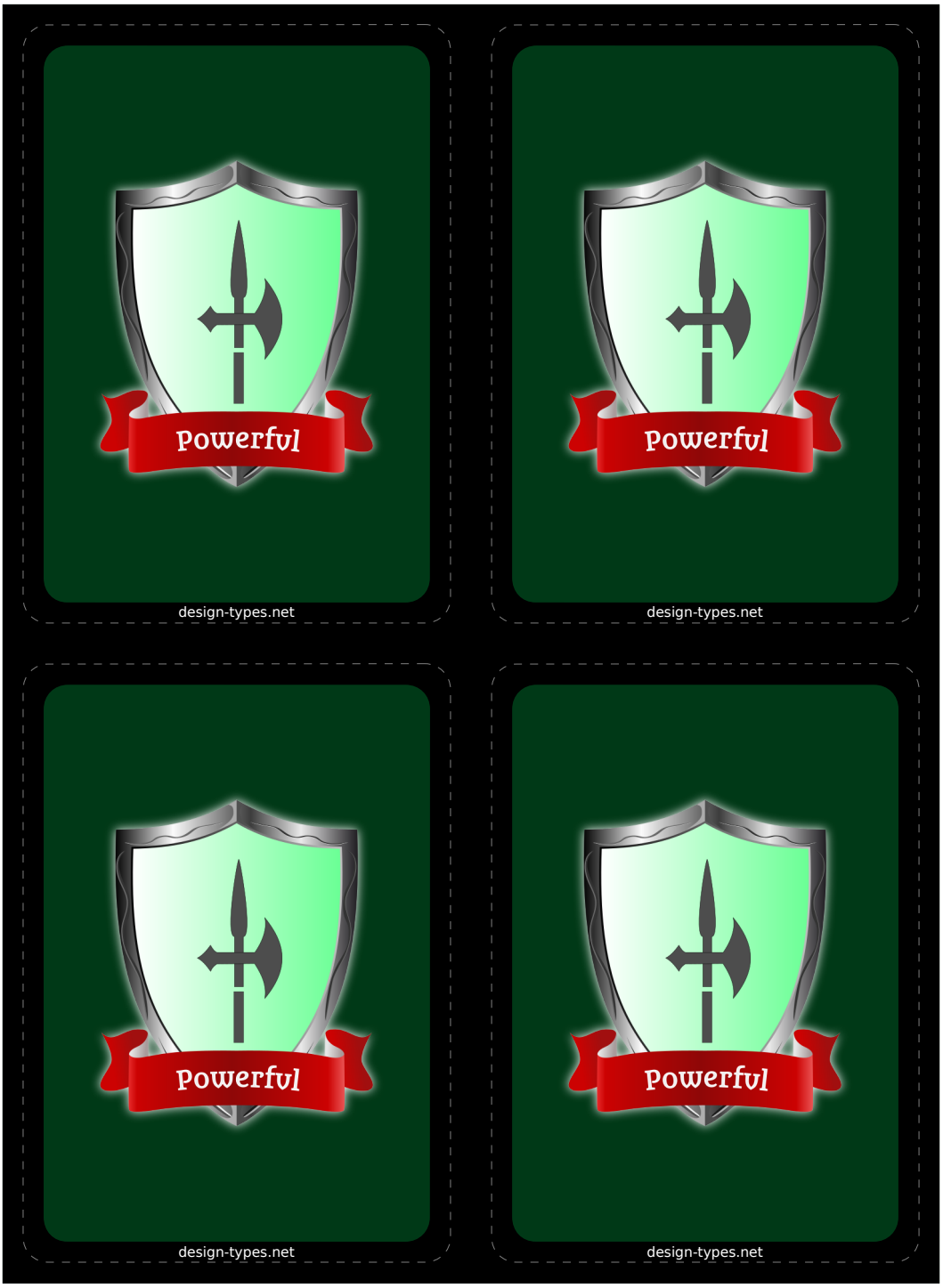
ECV

»Changing parts of the software should get their own module or abstraction.«

If you have to change your software, you'd like those changes to be isolated, so you don't have to change half your system. So put the changing parts into separate modules. Isolate changing APIs via gateway classes, data access technology using DAOs, encapsulate algorithms using the strategy pattern, etc. Conversely, don't use abstractions for those parts that won't change.

↑RoP, ↑IH/E, ↓YAGNI, ↓RoE

design-types.net


Powerful
design-types.net
Powerful
design-types.net
Powerful
design-types.net
Powerful
design-types.net

LC: Low Coupling

LC

»Tight coupling creates ripple-effects and makes the code less maintainable.«

If you decouple, you don't need to know internal details about other parts of the system. Furthermore, it makes you independent of changes in those other parts and it even enables reuse. So better reduce the number of dependencies and assumptions about other modules, use narrow interfaces, additional layers, indirection, dependency injection, observers, messaging, etc.

↑FP, ↑ML, ↓KISS, ⇅SRP

design-types.net

SRP: Single Responsibility Principle

SRP

»One module should do one thing only.«

If there is more than one reason to change a certain module (method/class/artifact/...), i.e. the module has more than one responsibility, then code becomes fragile. Changing one responsibility may result in involuntary changes to the other. Furthermore, changing the module is more difficult and takes more time. And even when you don't change the module at all, understanding it is more complex. So better separate concerns into separate modules.

↑PSU, ↑IOSP, ⇅LC, ⇅KISS

design-types.net

ADP: Acyclic Dependencies Principle

ADP

»Cyclic dependencies create rigid structures.«

Cyclic dependencies result in all sorts of nasty consequences: tight couplings, deadlocks, infinite recursions, ripple effects, bad maintainability, etc. The larger the cycle, the worse the consequences will get and the harder they are to understand and to break apart. So avoid them by using dependency inversion, publish-subscribe mechanisms or just by assigning responsibilities to modules hierarchically.

↑LC, ↑ML, ↓RoE, ↓ICC

design-types.net

IOSP: Integration Operation Segregation Principle

IOSP

»A module should either contain business logic or integrate other modules but not both.«

Either a module (method/class/...) is an operation, i.e. it contains business logic and/or API calls or it is an integration, i.e. it calls other modules. That means operations should never call other modules and integrations should have no business logic and no API calls. Operations are easy to read, test, and reuse. And integrations are very simple, too. This ensures that modules are small and systems well-structured.

↑LC, ↑SRP, ⇅KISS, ⇅PSU

design-types.net


Abstract
design-types.net
Abstract
design-types.net
Abstract
design-types.net
Abstract
design-types.net

DRY: Don't Repeat Yourself

DRY

»Duplication makes changing the code cumbersome and leads to bugs.«

Having a functionality more than once means to update or bug-fix it at every occurrence which is more error-prone and more effort. Refactorings like method or class extraction may help as well as inheritance, higher-order functions, polymorphism, and some design patterns.

↑RoP, ↑PoQ, ↓KISS, ↓PSU

design-types.net

IH/E: Information Hiding/ Encapsulation

IH/E

»Only what is hidden, can be changed without risk.«

There are 3 levels of IH/E: 1) Having a capsule means, that you have methods for accessing the data of the module. 2) Making the capsule opaque means that you can only access the data through the methods (i.e. all fields are private). 3) Making the capsule impenetrable means that you avoid returning references to mutable internal data structures. Either you make them immutable or you create copies in getter/setter methods.

↑MP, ↑LC, ↑FP, ⇅ KISS

design-types.net

PSU: Principle of Separate Understandability

PSU

»You shouldn't need to know other parts for understanding this one.«

Each module (method/class/artifact/service/...) should be understandable on its own. Understanding becomes a lot more difficult if you cannot apply divide and conquer. Furthermore, if something is not separately understandable, this typically means either that a part of the functionality does not belong here or the module has the wrong abstraction.

↑LC, ↑MP, ↑ML, ⇅TdA/IE

design-types.net

TdA/IE: Tell don't Ask/ Information Expert

TdA/IE

»Functionality should be where the data is.«

Instead of asking a module for data, processing it, and putting it back afterwards, better just delegate. This reduces complexity in those modules which are already large (and may even become god classes). So avoid getters and setters in favor of methods containing domain logic. In other words: Logic should be implemented in that module that already has the necessary data, that is the information expert.

↑IH/E, ↓PSU, ⇅SRP, ⇅LC

design-types.net


Concrete
design-types.net
Concrete
design-types.net
Concrete
design-types.net
Concrete
design-types.net

CF: Customer Focus

CF

»This is not what the customer pays us for!«

If something is not requested, there has to be a very good reason to do it. Anything in addition costs additional time (also for removing or maintaining it). It creates the additional risk of more bugs and makes you responsible for it. Continuously remember what was requested e.g. by looking into the requirements or asking the customer.

↑EaO, ↑YAGNI, ↓PoQ, ⇅FP

design-types.net

ICC: In the Concrete Case

ICC

»Your arguments are valid but in the concrete case the advantages won't be important.«

Many arguments hold true in general but when we look at the decision to be made, the effects they describe are sometimes negligible. Yes, low coupling is important, uniformity is helpful, and flexibility is desirable. But these aspects are sometimes crucial and sometimes irrelevant. So better focus on arguments that are relevant in the concrete case instead of insisting on aspects just to satisfy idealistic pettiness.

↑CF, ↑YAGNI, ↓PoQ, ↓PSPG

design-types.net

€aO: Early and Often

€aO

»Going online soon means to get value and feedback soon.«

Business success is often built on being faster than competitors. Building minimum viable products and 80%-solutions facilitate a faster time to market. Moreover the best feedback for improvement comes after a release and is rarely designed up front. So avoid perfectionism, release early and often, and accept a certain amount of technical debt.

↑FRD, ↑TP, ⇅PoQ, ⇅IR

design-types.net

VFT: Use Familiar Technology

VFT

»Using well-known technology results in faster outcome and fewer time-consuming bugs.«

Well known technologies are easier to handle because you can focus on the job and you know all the pitfalls. If you use unfamiliar technology, you most likely won't do that well at first. This results in even more bugs and worse design. So better use those technologies that all (current and future) developers are most familiar with.

↑UP, ↑IR, ↓TP, ⇅ML

design-types.net


Pragmatic
design-types.net
Pragmatic
design-types.net
Pragmatic
design-types.net
Pragmatic
design-types.net

PoQ: Principle of Quality

PoQ

»Bad quality kills us in the long run!«

It may be faster now, but we need to be fast tomorrow, too. Bad quality frustrates maintainers, makes fixing bugs harder and leads to huge efforts for changes. This often starts by being careless once. Don't let a vicious circle begin. Use metrics, adhere to the architecture, have a high test coverage, apply code reviews and refactor continuously. Don't be lazy.

↑LC, ↑ML, ↓CF, ⇅EaO

design-types.net

UP: Uniformity Principle

UP

»Solve similar problems in the same way.«

Following UP reduces the number of different solutions. There are fewer concepts to learn, fewer problems to solve and fewer kinds of defects that can occur. So have a consistent structure, a consistent naming scheme and use the same mechanisms and libraries everywhere. Prefer using the same approaches and not just similar ones as subtle differences lead to bugs.

↑ML, ↑RoS, ↓ICC, ⇅KISS

design-types.net

MP: Model Principle

MP

»Program close to the problem domain.«

Software should model and mirror the concepts and actions of the real world. So avoid everything that works "accidentally". If it works accidentally, it breaks accidentally. So be precise with semantics. If you need to delete an order in a data migration routine, call `deleteOrder` and not `cancelOrder`—even if that currently does the same. `cancelOrder` might get enhanced such that it creates a reverse order which wouldn't be correct for data migration anymore.

↑ML, ↑TdA/IE, ⇅KISS, ⇅ADP

design-types.net

PSPG: A Penny Saved Is a Penny Got

PSPG

»It might not be a big advantage, but it's not a big cost either.«

Making little improvements a habit sums up to a big advantage. This is the reason behind the boy scout rule ("Leave the campground cleaner than you've found it"). You don't have to clean the whole forest, but if everyone leaves the campground just a little cleaner, we will have a clean forest in the end. So if it's not a big deal, update libraries, improve documentation, and refactor the modules you are currently touching anyway.

↑PoQ, ↑EaO, ↑FRD, ↓CF

design-types.net


Idealistic
design-types.net
Idealistic
design-types.net
Idealistic
design-types.net
Idealistic
design-types.net

ML: Murphy's Law

ML

»Avoid possibilities for something to go wrong or to get misused.«

If there is a possibility for something to be used in the wrong way (like supplying parameters in the wrong order), it will eventually happen. So better avoid possible future bugs by using defensive programming, immutability, a common naming scheme, avoiding duplication and complexity.

↑FF, ↑EUHM, ↓ICC, ⇅KISS

design-types.net

IR: Instability Risk

IR

»Bleeding edge often leads to blood and pain.«

New technology often comes with teething problems. Using too unstable software, beta versions of libraries, or anything that hasn't stood the test of time is risky. There may be unknown bugs, nasty little quirks and compatibility issues no one has heard of, yet. This also means that if you encounter these problems, you will be one of the first to face them.

↑RoS, ↑UFT, ↓TP, ↓FRD

design-types.net

FF: Fail fast

FF

»Program defensively or you'll have a hard time debugging.«

If you don't check your inputs, cascading failures can occur. This results in security problems and error messages which are hard to decipher because they are not thrown at the position of the actual fault. This may even lead to situations where teams have to investigate failures which are not theirs. So log and throw an error as soon as you realize a problem. The earlier the better, so throwing a compile-time error is preferable to run-time checks.

↑ML, ↑EUHM, ↓KISS, ⇅NFR

design-types.net

RoS: Rule of Standardization

RoS

»Adhering to standards makes systems easier to understand and reduces bugs.«

Sticking to standards reduces complexity. If you are familiar with the standard, understanding systems that adhere to it will be much easier. Also, standards ensure a certain degree of interoperability and maturity. So use standard technologies, standard architectures, standard coding styles, standard formatting, standardized checklists, etc. If there are no formal standards, create your own in-house standard.

↑DRW, ↑NFR, ⇅KISS, ⇅TP

design-types.net


Robust
design-types.net
Robust
design-types.net
Robust
design-types.net
Robust
design-types.net

TP: Technological Progress

TP

»Progress must not be ignored in a competitive environment.«

New technology is not only motivating but also comes with benefits like more features, more performance, better maintainability, and fixed bugs. Furthermore, old technology won't be supported for much longer and new people don't know the old stuff anymore. Continuously challenge existing solutions by evaluating alternatives.

↑FRD, ↓IR, ↓UFT, ⇅RoS

design-types.net

FRD: Frequency Reduces Difficulty

FRD

»If it hurts, do it more often!«

Typically, it's easier and less effort to go small steps continuously than to wait until there is a huge gap to bridge. The pain will be bigger the more you postpone it—break the cycle and update to new versions, refactor regularly, merge and release early and often. Doing something more often, leads to more practice and fewer mistakes.

↑ML, ↑EaO, ↓IR, ↓ICC

design-types.net

DRW: Don't Reinvent the Wheel

DRW

»Focus on real challenges instead of old ones.«

If something has already been solved, it's probably solved in a better way than we will manage to do in the time we have. No one would ever reimplement a cache or a search algorithm except it is one's core competency. So focus on the challenges of your core business and use standards, libraries, and frameworks. They are the core business of those people who create and maintain them. They've solved many problems that we haven't even thought of, yet.

↑EbE, ↓LC, ↓ICC, ⇅RoS

design-types.net

EbE: Experience by Experiments

EbE

»We'll never know if we don't try!«

Discussing advantages and disadvantages theoretically can be helpful but at a certain point you will never know which variant is better if you don't try. So if you have a standard solution to a problem, try the other one. Carefully but regularly try out new frameworks and libraries, new coding guidelines, architectural/design patterns etc. in real-world projects. Failed experiments will be refactored and successful experiments will stay and become the new standard.

↑TP, ↓IR, ↓CF, ⇅FP

design-types.net


Technologic
design-types.net
Technologic
design-types.net
Technologic
design-types.net
Technologic
design-types.net

Focus

?

»Are we still on the right track or have we lost focus?«

Discussions are weird sometimes. You start with the clear aim to decide whether solution A or B is better and end up with a discussion on something completely different. You easily get lost in side issues or you heavily argue on something unimportant (“bike-shedding”). So clarify the main topic of your discussion and keep the focus on it.

design-types.net

Third Solution

?

»Are we discussing all relevant possibilities?«

Sometimes we argue heavily if solution A or B is better when in fact there is a third solution C that may be preferable. So regularly ask yourself (and your colleagues) whether there is a solution that needs to be discussed, too.

design-types.net

The Right Time

?

»What will happen if we don’t decide right now?«

A design decision should be taken as late in the project as possible. But it’s likewise harmful to take it too late. In order to find out if a decision really needs to be made now, think about what will happen, if the decision is deferred.

design-types.net

Stakeholders

?

»Do we have the needs of all stakeholders in mind?«

Some decisions have influence on many stakeholders—some of which are often forgotten. Also, think about QA, Ops, etc.

design-types.net


?
Question
design-types.net
?
Question
design-types.net
?
Question
design-types.net
?
Question
design-types.net

Consequences

?

»What will happen if we make the wrong decision?«

Think about possible impacts, chances of occurrence, and possibilities to revert. If the consequences are not bad at all, then it might be better to shorten the discussion. If the consequences are severe, there should be some means of mitigation in place. In any case think about the consequences of a decision.

design-types.net

Mutual Understanding

?

»Do we really understand each other's points of view?«

Sometimes a discussion gets stuck because of misunderstandings or misinterpretations. Commonly that's because everyone is busy explaining their own point of view without trying to understand the other. If that's the case, it is necessary to realize that. Otherwise, there will be no progress in the discussion.

design-types.net

From Scratch

?

»What would be the "real solution" if we'd start from scratch?«

Often the solutions we come up with are tied to the current state of the software. Our thinking is restricted such that we do not consider certain possibilities. In such cases it is helpful to neglect the circumstances of the current system for a moment—think outside the box. Even if the greenfield solution you then come up with is not directly applicable, it's often a starting point for an alternative.

design-types.net

Best Solution for the User

?

»Do we really address the real user's needs?«

Not in every case the person who specifies what to do is identical to the user of the system. Wrong interpretations or misunderstandings may lead to unsuitable solutions that do not satisfy the real user's needs. Every now and then, you should ask yourself if you are still designing a system that really helps those who will eventually use it.

design-types.net


?
Question
design-types.net
?
Question
design-types.net
?
Question
design-types.net
?
Question
design-types.net

Mediator

»We cannot agree. Let's get some help!«

Sometimes a discussion gets stuck. In these cases it is often advisable to ask another colleague for an opinion or mediation. Usually a colleague who hasn't already participated in the discussion, adds a new, unbiased perspective.

design-types.net

Team Decision

»The decision is too important to take alone. Let's have the whole team decide!«

Important decisions which affect many people like architectural decisions, big refactorings, and external APIs should be taken by the whole team. First, this typically results in better decisions. Second, the team will be much more committed to the decision. And third this fosters knowledge transfer.

design-types.net

Divide and Conquer

»Actually we are mixing up two aspects or two decisions. Let's discuss them separately.«

Design decisions get complicated or stuck if there is actually more than one decision to make. The discussion shifts from one topic to the other and back again. This gets even worse if nobody realizes that there is actually more than one problem. Step back, find out which decisions or problems there are and discuss them separately.

design-types.net

Research

»Let's have a look if there is already a suitable solution.«

When making a decision, make sure that you know all relevant solutions. Many problems have already been solved. So before inventing an own algorithm, have a look at libraries and scientific papers. For certain design decisions have a look at standards and patterns. Also, consider researching code snippets for common programming issues. Maybe there is even commercial-off-the-shelf (COTS) or open-source software you can leverage.

design-types.net


Action
design-types.net
Action
design-types.net
Action
design-types.net
Action
design-types.net

Flip a Coin

!

»That's not worth the discussion!«

In some cases the difference between several solutions is negligible. Or both the solutions have their pros and cons without one being superior to the other. It is then better to just take the decision by flipping a coin than to waste time in a lengthy and pointless discussion.

design-types.net

Devil's Advocate

!

»There is no real discussion, and we risk missing a point. Let's appoint a devil's advocate.«

Sometimes you agree too fast on a solution—probably because you all have a similar way of thinking. In such a case you can appoint someone who has to argue against that solution. A similar problem occurs when none of you has a strong tendency towards any of the solutions. In such a case, for each solution appoint a representative who tries to argument for this and against the other solutions.

design-types.net

Product Owner Decides

!

»This has a significant impact on the business, so we have to talk with the product owner.«

Some technical decisions influence the product itself. Often there is an impact on cost and time and sometimes there are even legal issues. Trade-offs include hosting an application in the cloud (flexibility and time vs. privacy and cost), adding a caching layer (performance vs. complexity and cost), make-or-buy (time vs. flexibility and cost), etc. In those cases the decision is not merely a technical one. Involve the PO.

design-types.net

Client Decides

!

»The client who calls the API knows best how the ideal API should look like.«

APIs need to be intuitive to those who use it and sometimes it's hard to predict if that's the case. Some decisions have an impact on how a module can be used. Some use cases may get simpler and others may get harder and less intuitive. Better stop assuming you know what's best for the clients. Just ask and involve them in your decision.

design-types.net


Action
design-types.net
Action
design-types.net
Action
design-types.net
Action
design-types.net

Joker

»I don't have a card for this, but let me explain.«

That's OK. Don't feel restricted in the arguments you use as long as they are valid. Maybe you don't find the corresponding card fast enough, maybe the card is not in your deck or maybe there is no card for your argument. Never mind: If it's convincing, use it. Nevertheless, limit yourself to a single and graspable argument.

design-types.net

Deck Building

There are many cards and using them all at the same time can be quite unhandy. So you should build a deck.

- Start with the basic set (») and get familiar with it.
- Slowly add cards as you learn them and use those wich will be helpful in your environment. This may especially depend on your team and project.
- A good deck is tailored to the situation at hand. You will use other cards for code reviews than for architecture discussions.
- A good deck is no larger than 20 cards.

Card Symbols

Card Set:

basic

advanced

extended

custom

Linked Arguments:

↑Complementary (adds further aspects)
↓Contrary (probably favors another solution)
⇅Both

design-types.net

Design Types

Discussions can be quite exhausting, can't they? Have a look at our free Design Types questionnaire and learn more about yourself, your colleagues and how to make discussions even more productive.

→ *design-types.net*

The Authors

Matthias Wittum

Christian Rehn

design-types.net

design-types.net