

Indecisive Generality

Klaus Marquardt
Email: pattern@kmarquardt.de

Copyright © 2005 by Klaus Marquardt. Permission granted for EuroPLoP 2005.

When a project fails to clarify important issues, to complete the analysis, or to make clear statements where the system is meant to be extensible and where it needs to be stable, architects tend to answer the indecisiveness in a technical way: general and complex solutions.

This diagnosis pattern is about the balance between making decisions and introducing variability and configurability. It shows why unmanaged complexity is a major risk to the project's success, how it creeps into a project and how to detect it. Furthermore it gives techniques how to enforce decisions and limit the generality.

A note to the workshop participants

Primarily I seek feedback on the completeness of the symptoms, the correctness of the causes, and the applicability and completeness of the therapies.

Introduction

“Wer alles reinläßt, ist nicht ganz dicht!”¹

This was one of my first lessons in project management my company taught to new project leaders. After you won the contract, keep it manageable and leave out whatever you can. Only if you are seriously forced to, accept slips in the defined project scope – and have the customer pay for it. Those were the days of custom projects and a world full of hosts, terminals, clients and servers.

Then came the idea of reuse and the promise of the objects. After the objects turned to be misunderstood or inappropriate, the granule became larger and called component or plug-in. Those were the days of unified projects and a world full of frameworks, components, and services.

Then came the years of extreme programming and agile development. The customer became an undoubted sovereign, his decisions were the law – as long as he would pay for it. Those were the days of hope in humans and technology, and of carelessness in domain responsibility.

¹ (a German play with words) who tolerates pouring in, will drown.

The clients were not happy with agility. They were looking for competence that would guide them and see their problem. When a project fails to clarify important issues, to complete the analysis, or to make clear statements where the system is meant to be extensible and where stable, architects tended to answer the indecisiveness in a technical way: general and complex solutions. Independent of the chosen development method, complexity can stall a project.

Similar to the world of house building, customers expected to be told what they want. The choices they would make shall not be able to break the project, but make their house a comfortable home. Knowledgeable project managers and architects shall never allow choices that would cause trouble, that is their ultimate responsibility. Clients feel comfortable and can communicate with experts knowledgeable in their domain, and take for granted that these experts bring expertise in the solution domain.

The decisions customers make are on the domain side and bring the project's inherent complexity. The solutions managers and architects choose bring an additional, imposed complexity. When this fails to support handling the inherent complexity, or management introduces organizational complexity, e.g. by establishing distributed teams, the probability of project failure increases. Architecture is about minimizing complexity.

Tom DeMarco claims that project management for adults would be called risk management. Transferring this to architecture, software system architecture for adults would be called complexity management.

About this Paper

This paper aims at helping architects and other project participants to find out what is going wrong in their project and why, and what they can do about it.

For this purpose, the problem is described in a form appropriate for a medical disease, as a diagnosis. Known resolutions or measures are introduced as therapies. Similar to the medical world, a complex problem might have more than one solution, and a solution might help to solve more than one particular problem. Diagnoses and therapies do not stand on their own but are cross-linked back and forth.

Following this medical metaphor brings some unique features. A wealth of vocabulary becomes accessible. The metaphor also shows the limitations we face: none of the presented solutions might actually cure a particular system. Some therapies are only effective when applied preventively, others are merely palliative or might at best lead to a remission.






In this paper, diagnosis names are written in UNDERLINED SMALL CAPITALS and therapy names in SMALL CAPITALS. Names used but not listed herein are marked (↗) and can be found in the references. Both diagnoses and

therapies follow their own pattern formats including sections that contribute to the medical metaphor.

- The description of a diagnosis starts with a small summary and a picture. Symptoms and examination are discussed and concluded by a checklist. A description of possible pathogens and the etiology closes the diagnosis.

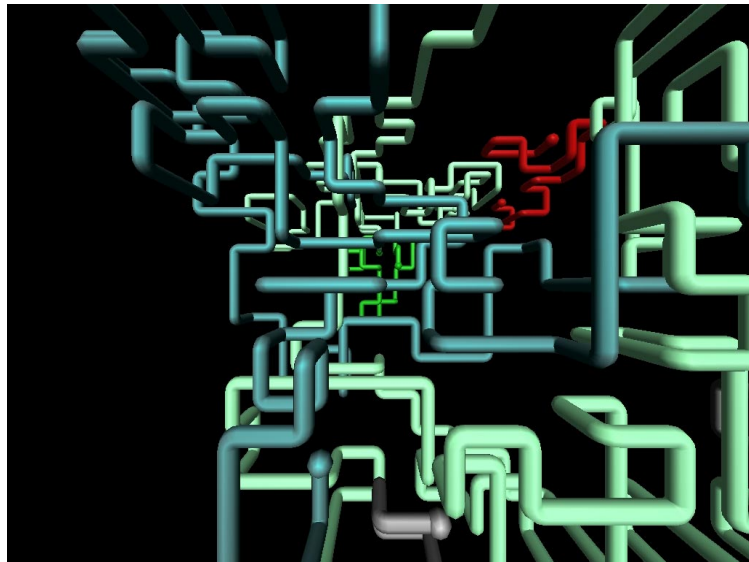
Each diagnosis comes with a brief explanation of applicable therapies. This includes possible therapy combinations and the kind of effect: curative, palliative or preventive. Where available, treatment schemes are described that combine several therapies. These are suggested starting points for a successful treatment of the actual situation.

- Therapies are measures, processes or other medications applicable to one or several diagnoses. Their description includes problem, forces, solution, implementation hints and an example or project report. Their initial context is kept rather broad. For each applicable diagnosis, applicability and particular consequences are evaluated.

In addition to the common pattern elements, therapeutic measures contain additional sections containing the medical information. These are introduced by symbols and show the mechanisms of a therapy and how it works (), the involved roles and related costs (), counter indications, side and overdose effects (), and cross effects when combined with other therapies (). For the diseases it can be applied to, usage sections are added ().

Diagnosis: Indecisive Generality

The inability to make consistent decisions increases the configurability of the software in directions that are not user relevant, the maintenance and installation costs, and the overall complexity of the architecture.



The innovative software of the fictitious project shall be developed for a domain in which automatization is not yet common. Thus, a huge market share could be reached if the software would fit for an international market.

The difficulties began when specifying the dialog to enter demographic data. How would that be done in Korea, for example, what data would be considered important? The products domain is highly regulated by national laws, and no expert for the Asian market was on the team. To avoid delays in development, a different approach was taken. The data model was extended to the maximum amount of fields imaginable, and the view became configurable.

Finally for each item in each dialog and for each locale a database entry existed that referenced the data item to present, and the dialogs were so generic that they read this table and configured the display dynamically. To function properly, more information was needed. How were the individual data items to be displayed? So the data model became extended to also contain reflective information about the type system.

The development effort became significant, but the achieved flexibility was considered beneficial and worth the effort. Unfortunately performance degraded as no indexes or predefined queries could be established due to the

high configurability. The team took some informed guesses on common properties, so at least in the US market the performance should be fine.

The testers were thinking thoroughly, and decided that they should test with all possible configurations. Sure enough they complained about the complexity of the system and demanded a change to reduce testing effort. The agreement was that the configuration data would be fixed for each local market, and would be released by a to be established expert team prior to the product launch.

So the configuration became a versioned item and had its own release cycle, while the software itself could remain stable. The idea to let the end user configure the product was abandoned. Furthermore, lawyers were contacted that would specify which data needs to be visible where. The team developed a small specification language based on Word macros, to extract the mandatory fields from the document and automatically create the configuration for that locale.

The tooling cost a fortune, but it was considered well spent as it helped to reduce the complexity and cope with the amount of configurability.

In a retrospective, the team detected the loop that the project had taken. Complexity was increased to avoid project delay and simplify the external interfaces. However, the self-imposed complexity leads to increased effort and delay, and needed costly removal afterwards.

“In a joint venture, all new products should follow a common look and feel. To establish a global user interface style guide, a joint team of domain experts worked for several months and came to conclusion. When the first products were developed, it occurred that while the style guide had described the “look”, it had left room for interpretation in the “feel”, the behavior of the GUI elements and dialogs. The architect chose to resolve these shortcomings by adding configurability to the library of GUI elements, so that all individual products’ wishes could be covered.”

Symptoms:

- You are planning a new product, or a project with uncommon scope
- Your specification team is insecure in the domain, and there is no knowledgeable customer
- Discussions about open specification issues are mostly considered lengthy, and eventually blocking the progress
- Technical people answer open issues by allowing configurable solutions

-
- Variability in behavior is handled in configuration files and data tables, not in code
 - The number of configurable items and configuration files grows
 - The configuration contains not just static data, but is used to configure dynamic aspects of the system behavior
 - The specification is created by a committee from different parties
 - The stakeholders come from different countries and cultures
 - Coding and configuring are placed in distinct responsibilities, no person in the project sees the entire picture
 - The configuration data grows complex, so that both its syntax and contents require extensive documentation
 - Potential misunderstandings are not resolved in design and code, but with external processes that defer detection and initiative to late project phases
 - Integration work and detection of problems is shifted outside of the own responsibility
 - Test and integration are scheduled for the end of the project
 - When problems become visible, major stakeholders demand stability and are resilient to course corrections
 - Seasoned testers reject the proposed solution because of the amount of system testing needed to cover the possible scenarios
 - The system performance degrades due to the amount of variability that needs to be checked at runtime

The root cause is a combination of two aspects that come together. Fear of making mistakes is the first aspect, stemming from insecurity in a new domain. Compensation is the second aspect. The fear is compensated by a bunch of solutions in a well-known but unrelated domain. The education of software engineers introduces configurability as a healthy concept, so the developers are not reluctant to use it to compensate for indecisiveness.

Finally, the solution attempt does not match the actual problem of the project. Both are on different levels of understanding. A non-technical problem cannot be solved by a purely technical solution.²

² This is the basic insight behind the proverb “computers help us solve those problems we wouldn’t have without them”

Differential Diagnosis

Before you go and remove all configurability from your projects software – there are situations that would indicate otherwise, typically related to reuse of software in different situations. When can configurability be considered a Good Thing?

- In standard software that is well understood and can be adapted to fulfill the merely slightly differing needs of various users.
- In frameworks and applications that apply for a number of different domains. These domains share common thoughts, infrastructure and abstractions, but require differences in detail.
- In applications that are used in different markets. Not all users worldwide follow the same flow of work, or have the same legal constraints.

Therapy Overview

Insecurity and fear can be addressed by learning or by ensuring that the development processes prevents error prone steps. A RIGID PROCESS MODEL is a great help when followed consequently. It can be implemented choosing Agile Development or a Waterfall Process Model. To both, the courage to temporarily establish WORKING HYPOTHESES is supporting, when combined with the courage to abandon these and refactor when proven wrong. Attention to maintain a ↗ CONSISTENT LEVEL OF ABSTRACTION throughout the architectural decisions can make a conflict and upcoming problem visible early.

On the tactical side are therapies that help to cope with the negative effects of strategic mismatches. A COMPLEXITY BUDGET for decisions that increase the variability and configurability of the solution is a powerful tool to create and increase attention on the specification side. As a last resort, a CONSISTENT CONFIGURATION MECHANISM soothes some of the pain that developers and testers experience in their daily work.

Rigid Process Model

Also known as: Specification before Implementation

Consider a project where the implementation team is insecure and not knowledgeable in the application domain.

You need a way to start the project or task. You do not want to spent effort for technical solutions that might be incorrect due to unclear or changing specification.

You know that the tasks' completion requires domain expertise, ... but that expertise is not available to your team.

You could start using technical means to compensate for deferred decisions, ... but these require effort, increase overall project complexity and cost, and are likely to require rework.

Gathering domain expertise is expensive or time consuming, ... but delaying the project or task initiation may impose a business risk.

Therefore, come as you are. Start with the knowledge you have and gather more during the project's course. Insist on understanding the problem and its specification prior to design and implementation. Code only what is specified and agreed.

Establish a change procedure so that you known when to rethink the design, and make this step and its consequences visible, such as additional effort and delay.

The important insight is that it does not matter much which process model you prefer. Both waterfallish and agile development methods include elements where the implementation starts only when the problem is really understood. However, they differ on the problem size and granularity.

The key effects come from selecting a consistent process model and consequently following it. Do not take slips in the process that would compromise the understanding phase.

The selected RIGID PROCESS MODEL should include the option for trials. The project can gather expertise from the end users. Trials help avoid uninformed decisions, and must be concluded prior to specification.

Complexity is the limiting factor that prevents large projects from scaling with respect to resources, time, budget, and quality. Each increase in the technical, i.e. solution imposed complexity limits the amount of problem inherent complexity that the project can handle. Make sure that the client makes a conscious decision which kind of complexity to increase. If technical complexity is desired, treat it as a domain requirement and budget it including the lifetime costs.



The main mechanism is that a RIGID PROCESS MODEL does not allow for mental slips. Both the implementor and the specifying client are forced to agree on scope and cost, and do not start compromising based on assumptions.



The entire team including management and customer needs to subscribe to a RIGID PROCESS MODEL. If your company is not familiar to that culture, the initial learning curve will be steep. The costs then include a learning team for a few weeks, and a coach for several months.



For small projects, the effort of newly introducing a RIGID PROCESS MODEL may outweigh its advantages. If your specifying customer is not willing to join, a RIGID PROCESS MODEL is counter indicated, and you should consider backing out from the project.



For courage to make decisions while avoiding arrogance, combine with WORKING HYPOTHESIS. To limit the technical complexity, consider maintaining a COMPLEXITY BUDGET.



INDECISIVE GENERALITY: The need and ability to make decisions helps to avoid the trap of over-configurability. Make an item configurable only when the specification (like use case, requirement, user story or application test) clearly indicates so.

- ☺ All stakeholders know what they should do when.
- ☺ A defined process model brings its own security, thus it helps to overcome fear from domain insecurity.
- ☺ All technical compensation mechanism are justified by the customer.
- ☺ Procedures need to be tailored, and this tailoring can limit the intended effects.
- ☹ In practice, especially rigid process models are compromised. They do not change people, and they do not react on changing perceptions.

Working Hypothesis

Consider a project where all team members are inexperienced or insecure.

You need domain related decisions to proceed with development. There is no sufficient specification or domain expertise.

You lack expertise, Getting the necessary expertise is expensive or time consuming,	... but you are not stupid. ... but your schedule is tight, and there is little opportunity to compensate for an early slip.
A faithful start requires courage and limited risk,	... but courage does not come by itself, and arrogance is no replacement for courage.

Therefore, identify the unmade decision that blocks your development most, and make a preliminary decision.

Do not allow a key decision not to be made. However, when you rely on guessing and your sharp intellectual skills instead of true expertise, make these circumstances to decision making visible. Call the decisions **WORKING HYPOTHESES** and allow yourself and others to revise them as soon as the missing expertise becomes available.

Start with the knowledge you have and gather more during the project's course. Make decisions based on your current knowledge, and rethink them when you learned otherwise. Collect feedback to gain courage.

Document your decisions, their alternatives and their motivation, as in ↗**DOCUMENTED ALTERNATIVES AND MOTIVATION**. Include a marker for **WORKING HYPOTHESES** and indicate which answer to a question has not been available. As soon as the final answer contradicts your assumption, revise and actively initiate all changes.

Agile development explicitly considers the learning curve the project team experiences. A number of agile development practices support feedback, learning, simplicity, and change. For **WORKING HYPOTHESES**, the practices to Travel Light and to Refactor are most relevant.

For the implementation of an agile development approach, refer to the literature [*Beck99* etc]. Agile development is possible even in medium to large projects [*Eckstein05*]. Consider asking a Coach or ↗**MENTOR** for help.



The main mechanism of WORKING HYPOTHESES is to remove the fear of making decisions and thus to avoid the need for ill-minded compensation mechanism. The implied simplicity is an additional barrier that prevents technical overkill solutions.



When the project participants are reluctant to use or accept WORKING HYPOTHESES, both project leader and architect need to support this visibly. The effort is negative, as a hypothesis based process can unblock a project and overcome obstacles such as analysis paralysis.



Lack of possibilities for feedback is a counter indication to WORKING HYPOTHESES. When you cannot verify or falsify your assumption early, each hypothesis introduced a risk to the project success.



Combine WORKING HYPOTHESES with an agile development method that includes early feedback loops.



INDECISIVE GENERALITY: WORKING HYPOTHESES should be applied in all situations where insecurity cannot be resolved quickly.

- ☺ The fear of making mistakes is reduced, as potential mistakes are replaced by assumptions and become socially accepted.
- ☹ Each hypothesis proven wrong requires rework, potentially late in the project.

“Framework development was new to the company and to most team members. Fortunately, the project leader encouraged us to use assumptions as a basis for development, called ‘working hypotheses’, and did not wait for the various clients to specify their requirements. This approach unleashed all team members. At its inception we were somewhat reluctant, but it proved that the future users of the frameworks were unable to state what they needed or wished.

“For various reasons, the framework project was cancelled after two years. However, most of the hypotheses the team introduced were proven correct as the software was considered valuable and became the basis of a new product.”

Complexity Budget

Consider a project where all team members are inexperienced or insecure.

Indecisiveness in the requirements is answered by the request for additional configuration, adding complexity to the technical solution that goes beyond the complexity inherent to the application domain.

Maintaining measures and budgets costs effort and is boring,

... but not having data to detect deviations and decide on actions increases the project's risks.

Most problems in computing can be solved by adding one more level of indirection,

... but indirection means complexity, and uncontrolled growth of complexity is a major project risk.

Therefore, budget the amount of variability and configurability in the same manner as you budget resource consumption. Allow the customer to select a small number of configurable items. Whenever the demand for further configuration arises, the necessary budget needs to be freed by removing configurability in another area of the application.

A budget requires both a metric (currency) and a responsible budget owner. The budget owner is the project leader, the expert to determine the metrics is the architect. There is little knowledge in metrics now except for the traded and trained guts feeling – seen during estimation as “this is a complex task” indicating a high probability of large effort and risk.

A complexity metric should employ

- code and design complexity metrics [*McCabe76*]
- inhomogeneity, inconsistency, and coupling metrics [*Martin95*]
- procedures metrics
- organizational metrics
- variability metrics
- indirection and implicitness metrics

In case the necessary complexity outgrows the initially assumed budget, adapt the budget to the real project needs. This change of project size and scope needs to be reflected in the overall plan and estimate.



The main mechanism is to create awareness and induce reflection. If the amount of available complexity is limited, decisions are necessary to reduce it.



COMPLEXITY BUDGET involves all leaders' roles in the project: customer, project leader, and architect. The effort is in two areas: establish an agreed complexity metrics and budget, and keeping track of that budget.



COMPLEXITY BUDGET is counter indicated if you have no idea how to measure it. This is a common problem to software development.



A COMPLEXITY BUDGET is most easily tracked if you have a RIGID PROCESS MODEL in place.



INDECISIVE GENERALITY: Apply COMPLEXITY BUDGET early and often throughout the project. Make it an established part of your process such as requirements and risk management.

- ☺ Awareness of complexity prevents an unobserved increase of complexity from indecisiveness.
 - ☺ Dealing with complexity leaves the technical scope and is visible to all stakeholders.
 - ☹ No known metric covers all aspects of complexity that are relevant to software projects.
-

Budgets and their maintenance are common practice in systems with limitations, such as memory or CPU time consumption [NobleWeir].

While a complexity budget seems a good idea to me, I have not seen it implemented in significant depth, i.e. beyond code and design. I'd be more than interested in learning about known uses and your experience. Especially I would like to see metrics on complexity beyond code and design.

Consistent Configuration Mechanism

Consider a project where a large number of items are subject to installation or run time configuration.

Your application needs to be configurable in many dimensions.

For each variation point, some mechanism is more adequate than others,

... but users as programmers are likely to become confused when they have to search many places.

One single point of configuration is easy to administrate,

... but the required variability would abuse that mechanism and leave its intended scope.

Therefore, chose a consistent policy and stick to it. Limit the effort that users in various roles need to understand and actually configure the system.

Different mechanisms are available with different strengths and weaknesses. Pick at most two to support your needs, and discourage the use of all others.

The main mechanisms to provide options to change the behavior from outside an application are listed below. Typically, one of them is not sufficient to cover all your needs, but three are not necessary.

Configuration file: is most powerful when presence of single data affects the program behavior, and scales up to several dozen entries.

Database entries: custom tables can serve as persistent configuration file that is accessible via common infrastructure services. Difficult to access without special tooling.

Registry: similar to a configuration file, but with infrastructure facilities such as persistence, basic type support, installation support, and change notification. Is only available on few operating systems.

XML file: scales for large amount of data and supports a type safe storage and retrieval. Does not scale to configure variations in behavior or clauses in the data itself.

Code: if the behavior variations are significant, and your data description starts growing into a self made interpreter, rather use a real programming language [Fowler04]. Mostly pluggable components (PLUG-INS) are brought into the application at defined places [Marq99].

Scripts: can cover the middle ground where configuration data does not allow for the behavior you need, but a fully fledged and possibly compiled executable component would be overkill. Scripts have a tendency to appear easy to change, tempting the uninitiated user and administrator.



The main mechanism is to limit the secondary complexity, that relates to the handling of configurability.



A CONSISTENT CONFIGURATION MECHANISM requires buy in from all developers, and continuous attention from the architect. Over the project lifetime, it will save costs; however, possible early or not so early corrections require refactoring effort.



There are no known counter indications or overdose effects. Note that the dosage needs to be constant; a little helps nothing, and abandoning the therapy will invalidate all previous efforts.



A CONSISTENT CONFIGURATION MECHANISM is best implemented using patterns and best practices related to the chosen mechanism(s). COMPLEXITY BUDGET can help you to evaluate the adequateness of your choice of configuration mechanisms.



INDECISIVE GENERALITY: Apply CONSISTENT CONFIGURATION MECHANISM early in the project, and continue to the end.

- ☺ The complexity of the chosen solution is limited.
- ☹ The complexity limitation is of second order.
- ☹ The core of the problem is not addressed.
- ☹ When complexity is the key risk in your project, effort spent here would better pay off elsewhere.

Acknowledgements

Thanks to the EuroPLoP 2005 shepherd of this paper, Wolfgang Zuser, for his thoughtful remarks. Björn Schneider commented on symptoms and gave hints to refine the therapies.

References

- CONSISTENT LEVEL OF ABSTRACTION to be published
- Martin95* Robert Martin: Designing Object-Oriented C++ Applications Using the Booch Method. Prentice-Hall 1995
- McCabe76* Thomas McCabe: A Complexity Measure. In: IEEE Transactions on Software Engineering, Dec. 1976
- Beck99* Kent Beck: Extreme Programming
- Eckstein05* Jutta Eckstein: Agile Software Development in the Large: Diving into the Deep. Dorset House 2005
- Fowler04* on code versus configuration files, online at <http://www.martinfowler.com/articles/injection.html#CodeOrConfigurationFiles>
- Marq99* Klaus Marquardt: Patterns for Plug-Ins. In: Proceedings of EuroPLoP 1999
- NobleWeir* Small Memory Patterns. AWL
- Wikipedia* <http://en.wikipedia.org/wiki/>