

# DEVELOPMENT OF INFORMATION SYSTEMS


## **Lecture 10**

### **Domain Specific Language**

# Review of the last lecture


**What is needed and what is essential for software development?**

**The goal is the product** (software) and its quality, measured by multiple factors.

- People and their cooperation
  - Plans, rules, processes, management
  - Documentation
  - Techniques and technologies
  - Long time
- 

# Review of the last lecture

## Program life cycle

1. Conception
  2. Requirements  
gathering/exploration/modeling
  3. Design
  4. Coding and debugging
  5. Testing
  6. Release
  7. Maintenance/software evolution
  8. Retirement
- 
- A series of three parallel white diagonal lines are located in the bottom right corner of the slide, extending from the middle of the right edge towards the bottom left.

# Review of the last lecture

## Software development models

### 1. Non iterative methods

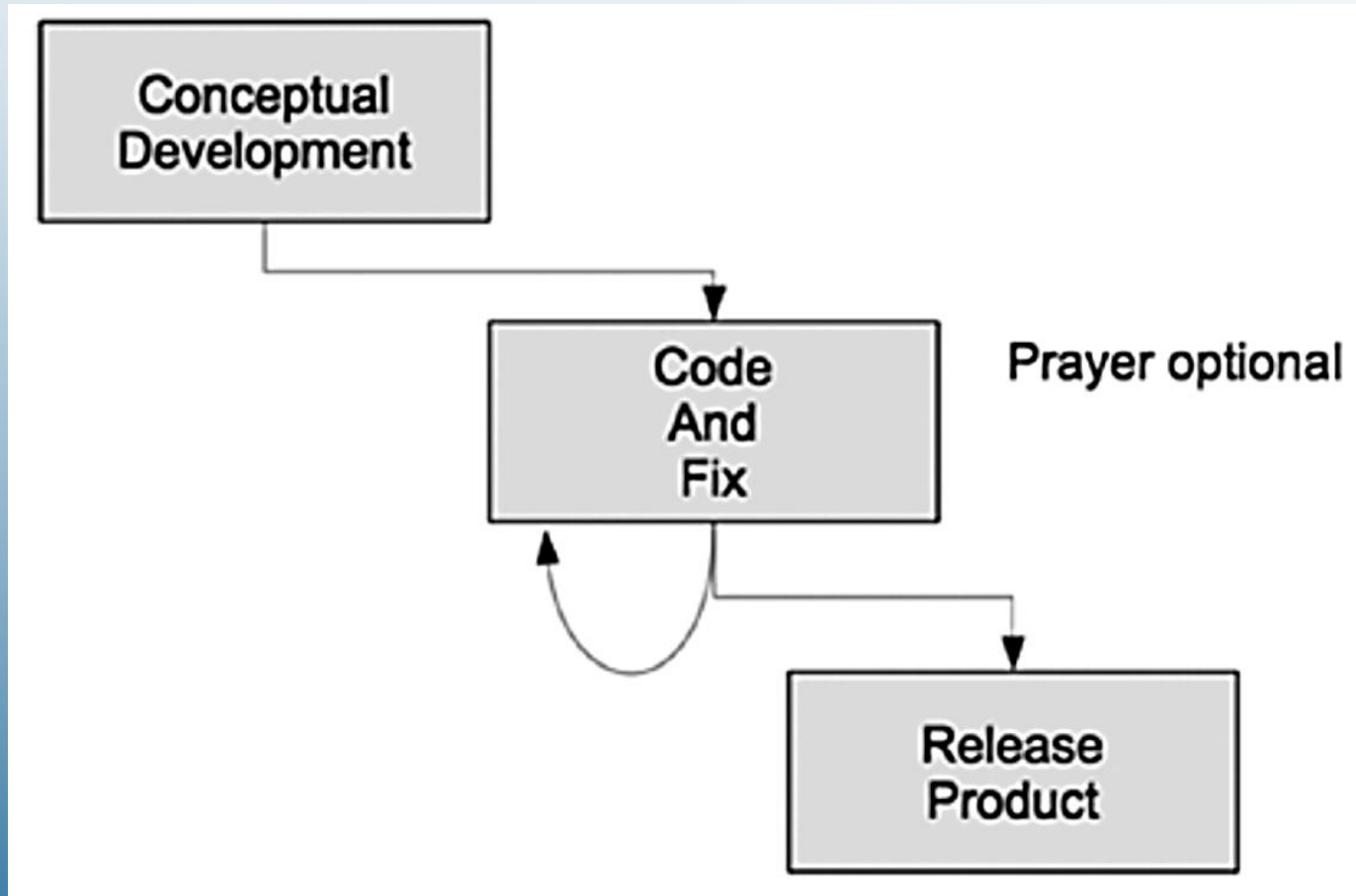
- Code and Fix
- Waterfall

### 2. Iterative methods

- Iterative and incremental
- Unified process
- Spiral model
- Microsoft Solution Framework
- *Agile methods:*
  - Agile development
  - Extreme programming
  - SCRUM
  - Test driven development

# Review of the last lecture

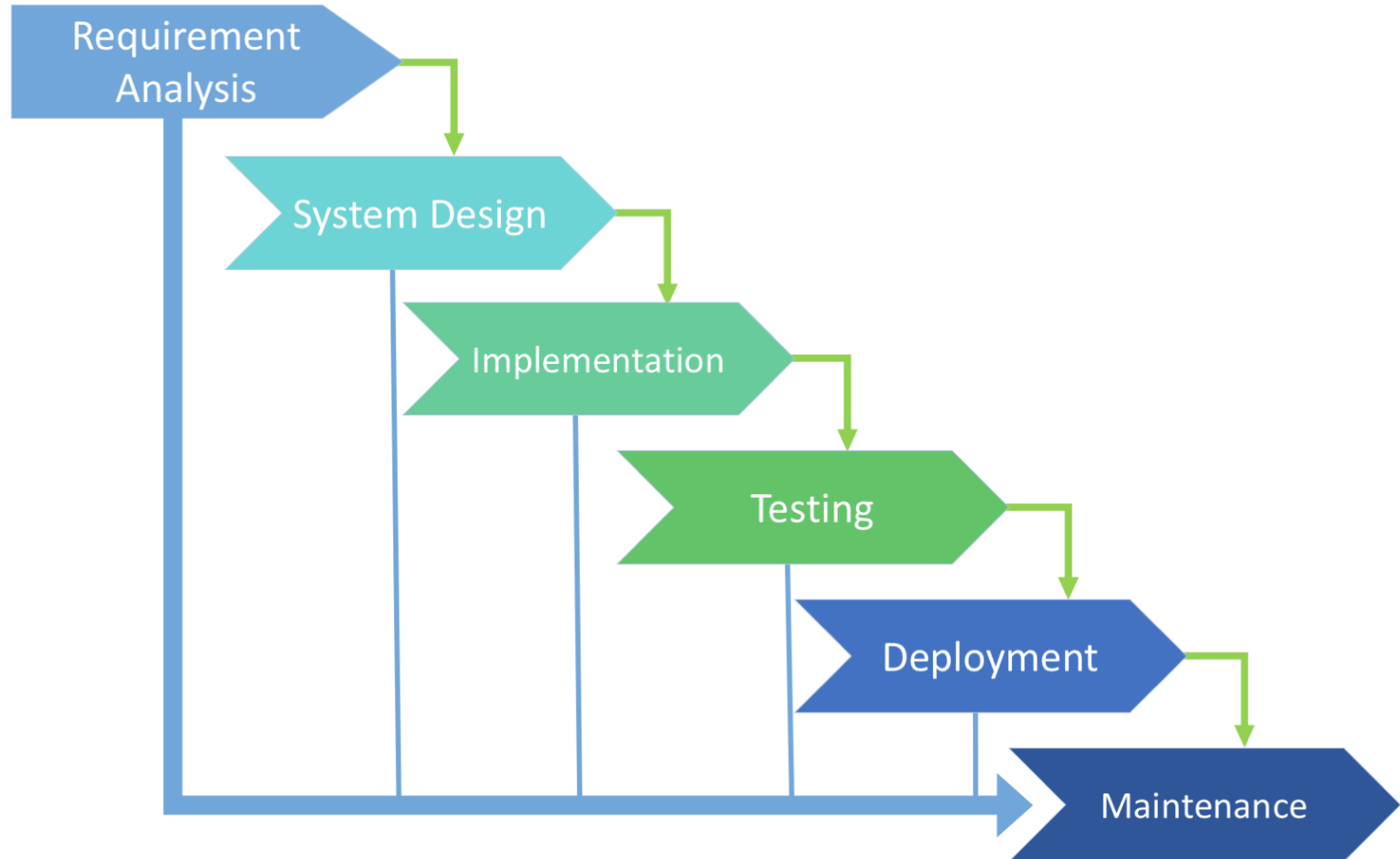
## Code and Fix



# Review of the last lecture

## Waterfall

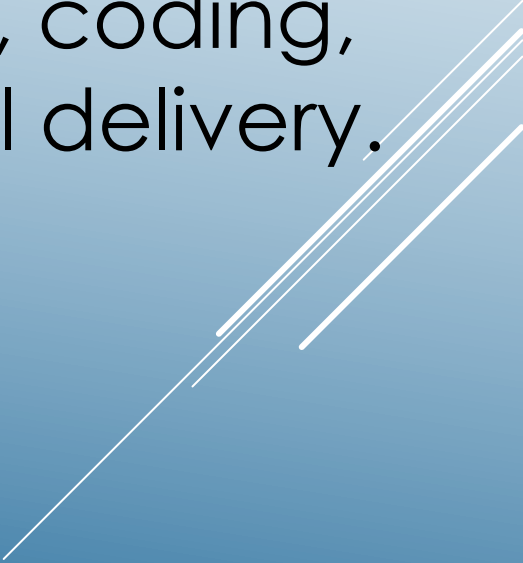
All phases are non overlapped.



# Review of the last lecture

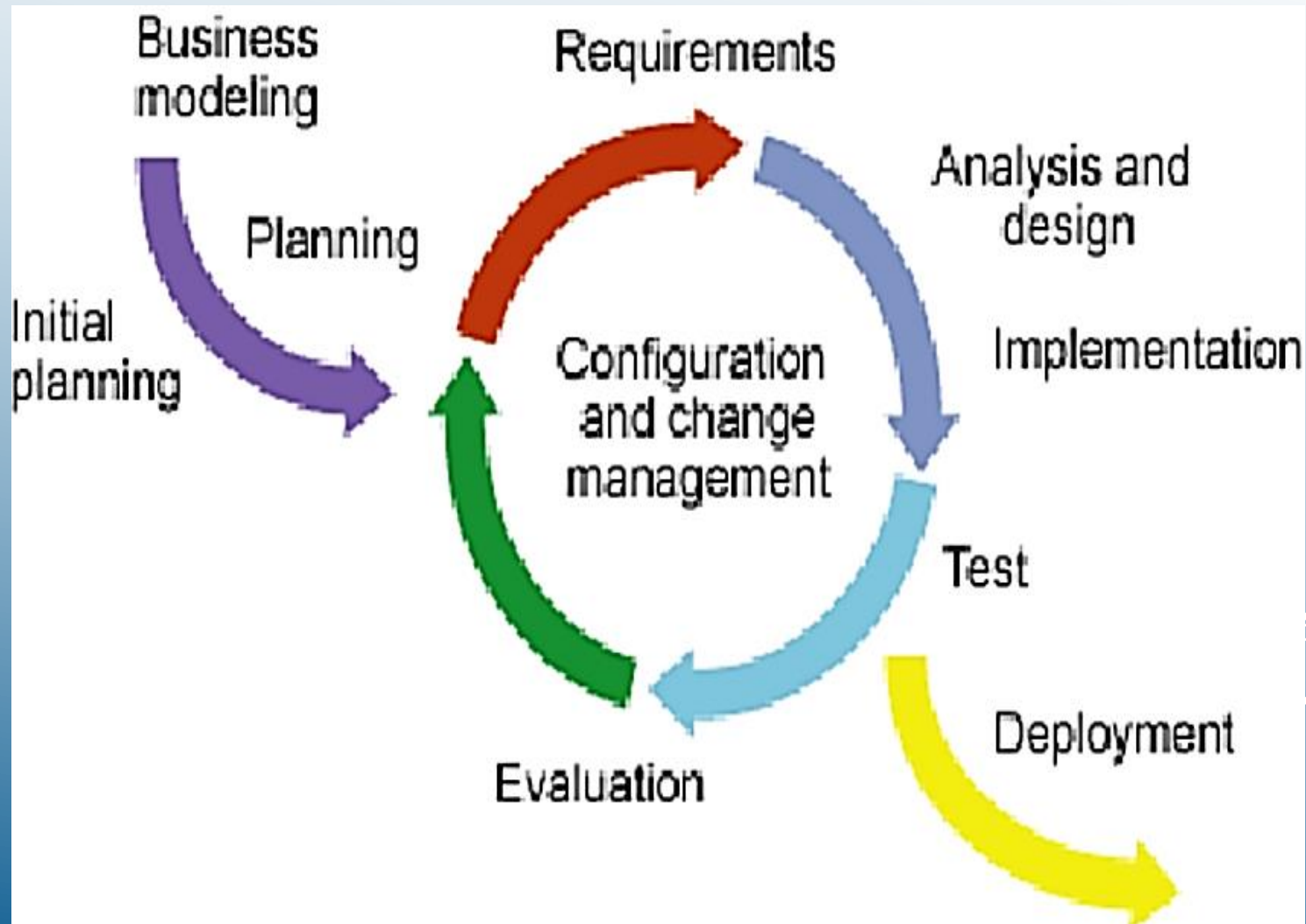
## Iterative Models

Iterate and deliver incrementally, treating each iteration as a mini-project, including complete requirements, design, coding, integration, testing, and internal delivery.

Three parallel white diagonal lines are located in the bottom right corner of the slide, extending from the right edge towards the bottom left.

# Review of the last lecture

## Iterative and incremental development





# Review of the last lecture

# UP: Unified Process

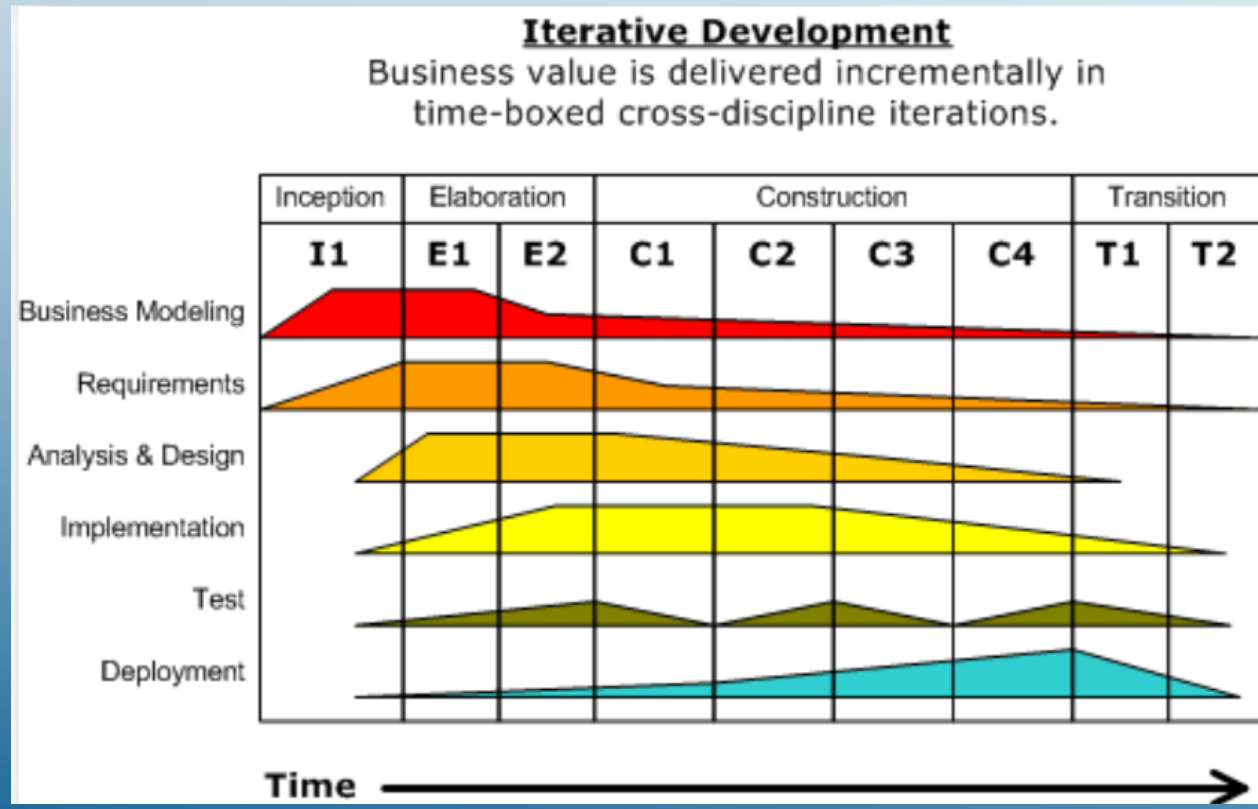
# Iterative and incremental development.

- Use-case-driven,
- Architecture-centric,
- Risk-focused.

## Phases:

- Inception,
- Elaboration,
- Construction,
- Transition

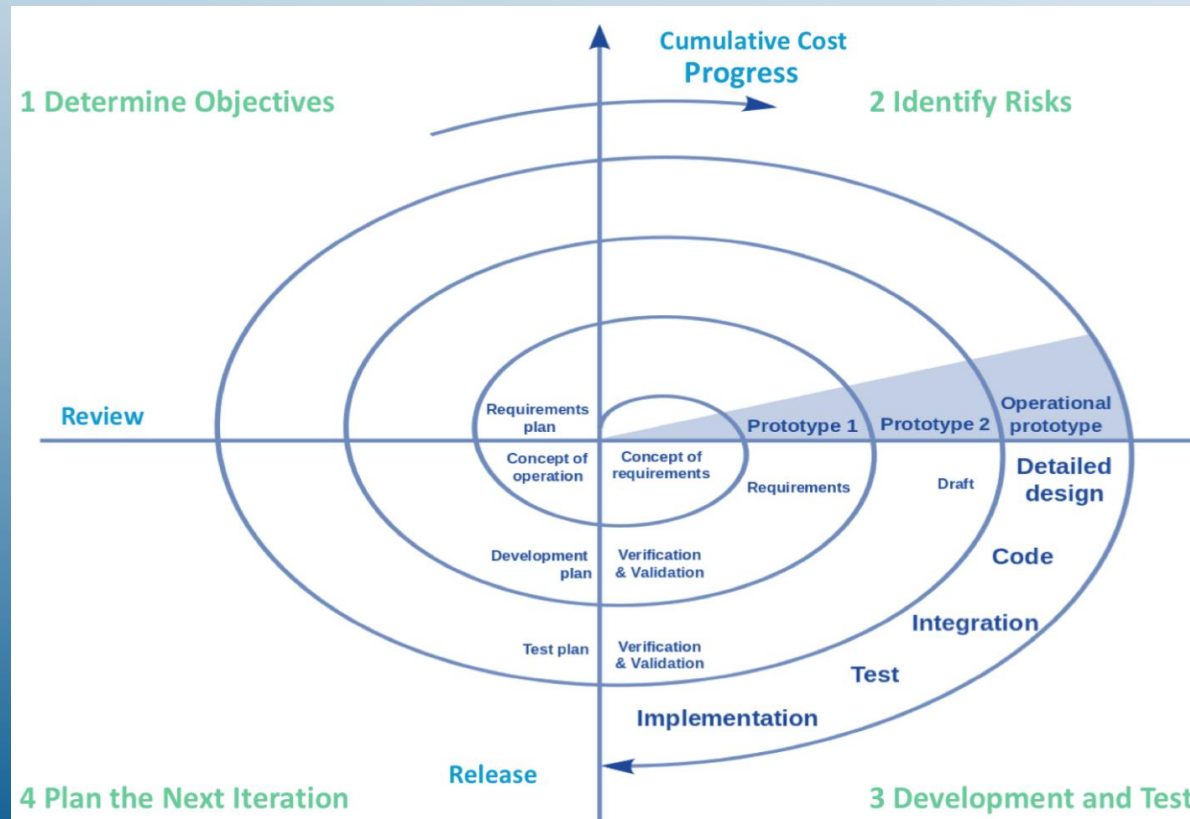
# UML, documents.



# Review of the last lecture

## Spiral Model (risk-focused)

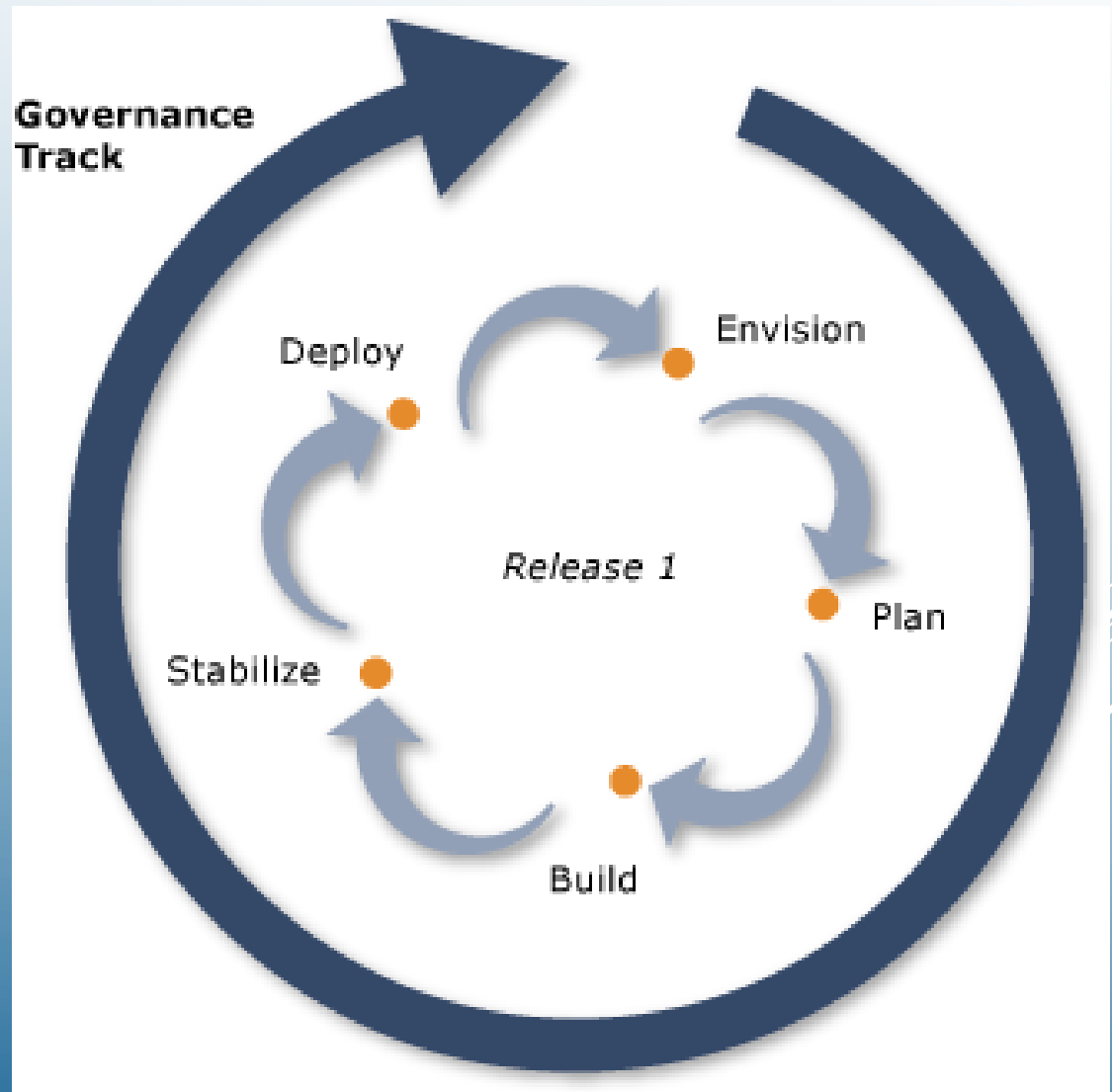
Combines architecture and prototyping by stages  
Focused on the risk analysis.



# Review of the last lecture

## Microsoft Solution Framework

- Principles
- Models,
- Disciplines,
- Concepts,
- Guidelines



# Review of the last lecture

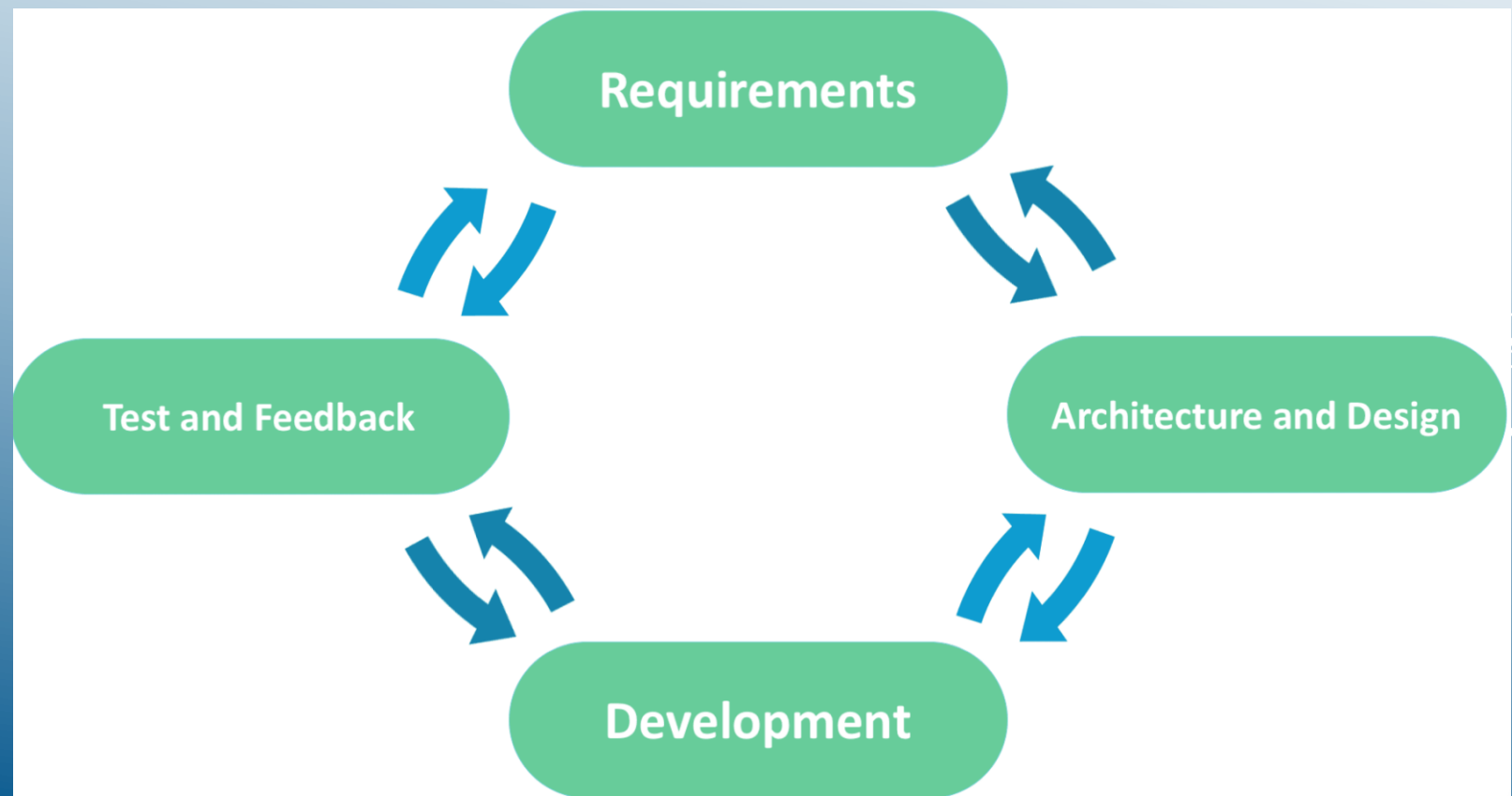
## Agile development

- Interactions over processes and tools.
- Working software over comprehensive documentation.
- Customer collaboration over contract negotiation.
- Responding to change over following a plan.

# Review of the last lecture

## Agile model of development

After every development iteration, the customer is able to see the result and understand if he is satisfied with it or he is not.

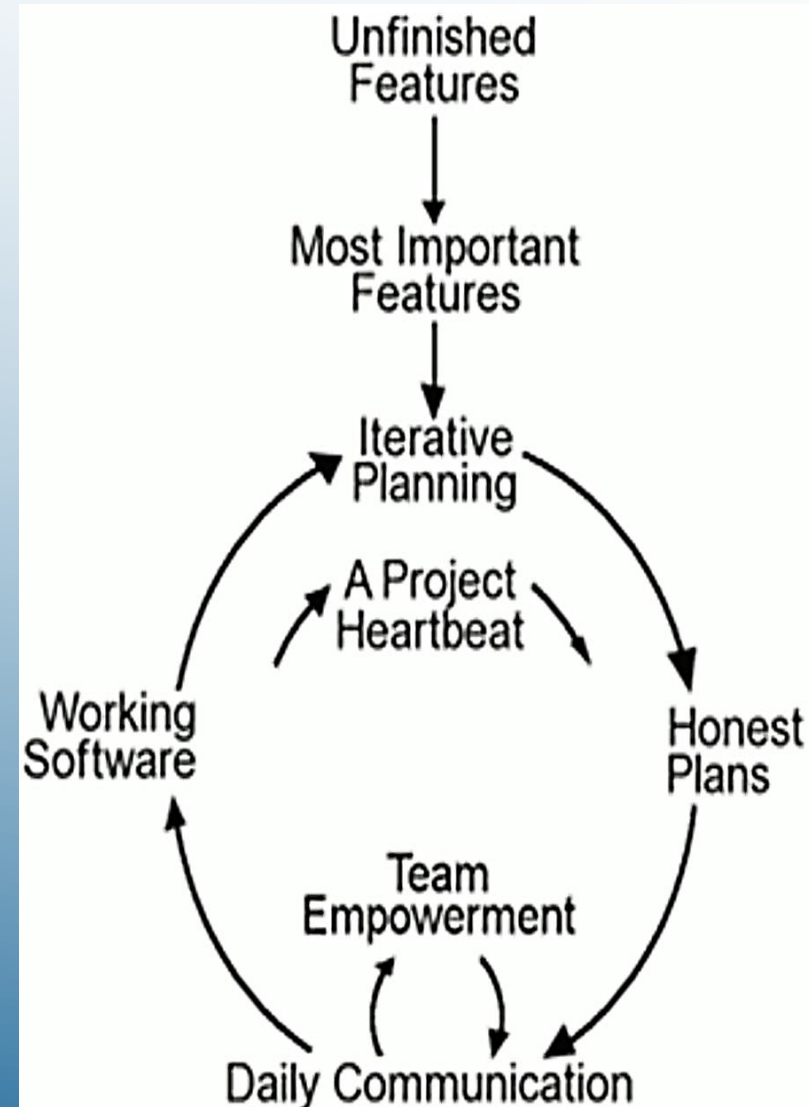


# Review of the last lecture

## Extreme Programming (XP)

It stresses customer satisfaction. Instead of delivering everything you could possibly want on some date far in the future this process delivers the software you need as you need it.

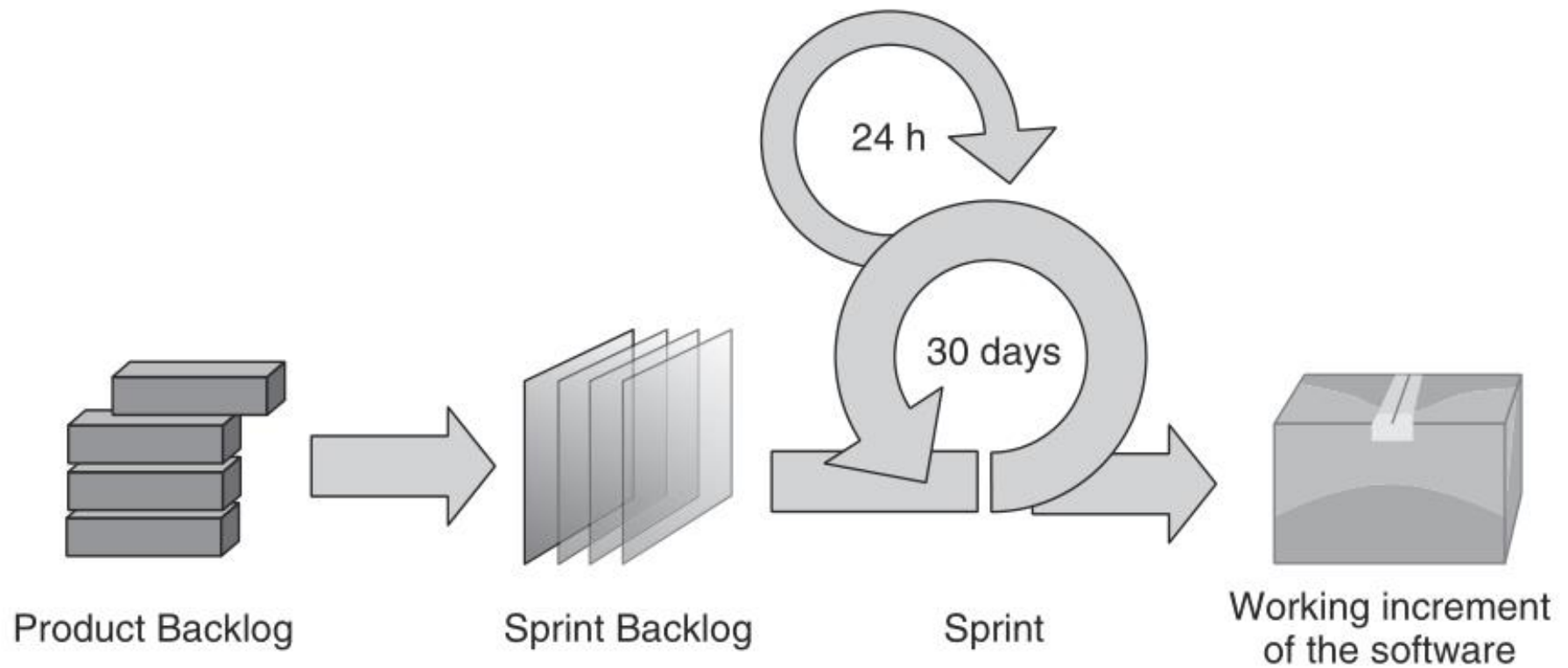
Is possible changing customer requirements at any stage of development.



# Review of the last lecture

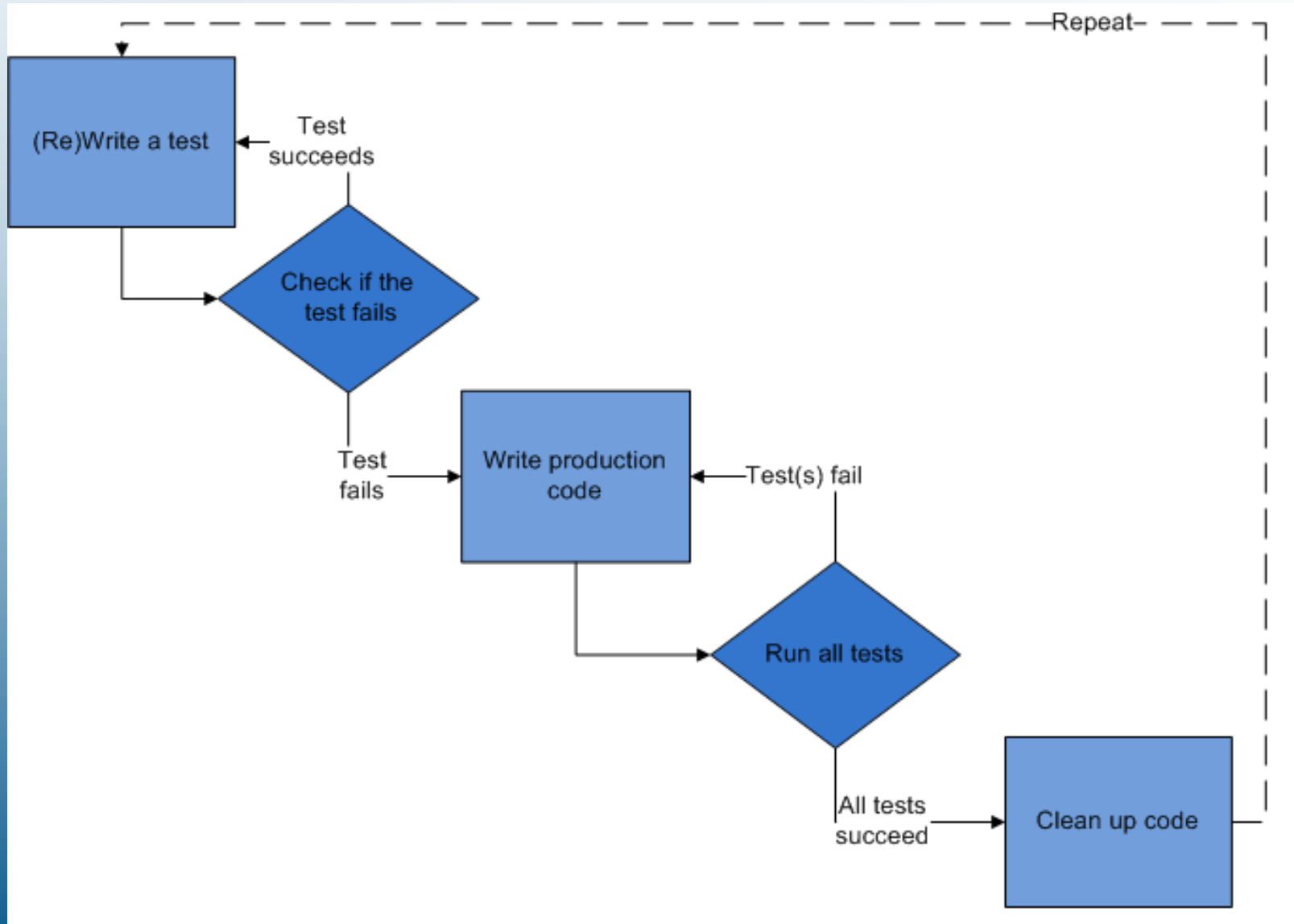
## SCRUM model

The Scrum model is a framework for planning and conducting software projects based on the principles of Agile development



# Review of the last lecture

## Test Driven Development





# Domain Specific Languages

Several thin, white, parallel diagonal lines are positioned in the bottom right corner of the slide, extending from the right edge towards the center.

# Brief Introduction to the Terminology

**Programming language** refer to general-purpose languages (GPLs) such as Java, C++, Lisp or Haskell.

**Model or program or code** can be written in a GPL or in a DSL.

**Target platform** is what your DSL program has to run on in the end and is assumed to be something we can not change during the DSL development process.

**The execution engine** bridges the gap between the DSL and the platform. It may be an interpreter or a compiler.

**Compiler** takes the DSL program and transforms it into an artifact (often GPL source code) that can run directly on the target platform.

# Brief Introduction to the Terminology

A language, domain-specific or not, consist of the following main ingredients:

- **Concrete syntax** defines the notation with which users can express programs. It may be textual, graphical, tabular or a mix of these.
- **Abstract syntax** is a data structure that can hold the semantically relevant information expressed by a program. It is typically a tree or a graph.
- **Static semantics** of a language are the set of constraints and/or type system rules to which programs have to conform, in addition to being structurally correct.
- **Execution semantics** refers to the meaning of a program once it is executed. It is realized using the *execution engine*.

# Brief Introduction to the Terminology

Sometimes it is useful to distinguish between technical DSLs and application domain DSLs.

Technical DSLs to be used by programmers and application domain DSLs to be used by non-programmers.

This can have significant consequences for the design of the DSL. There is often a confusion around meta-ness (as in metamodel) and abstraction.

The meta model of a model (or program) is a model that defines (the abstract syntax of) a language used to describe a model. For example, the meta model of UML is a model that defines all those language concepts that make up the UML, such as classifier, association or property.

# From General Purpose Languages to DSLs

**General Purpose Programming Languages (GPLs)** are a means for programmers to instruct computers. All of them are Turing complete, which means that they can be used to implement anything that is computable with a Turing machine.

## Why we need more than one GPL?

The real reason why many different languages are used for what they are used for is that the features they offer are optimized for the tasks that are relevant in the respective domains.

So, if we want to "program" for even more specialized environments, it is obvious that even more specialized languages are useful.

# Domain Specific Languages

Domain-specific language is a computer programming language of limited expressiveness focused on a particular domain.

**There are four key elements to this definition:**

1. **Computer programming language:** A DSL is used by humans to instruct a computer to do something.
2. **Language nature:** A DSL is a programming language, and as such should have a sense of fluency where the expressiveness comes not just from individual expressions but also from the way they can be composed together.
3. **Limited expressiveness:** A DSL supports a bare minimum of features needed to support its domain.
4. **Domain focus:** A limited language is only useful if it has a clear focus on a small domain. The domain focus is what makes a limited language worthwhile.

# Domain Specific Languages

In general a DSL construct a shared language between the domain people and programmers.

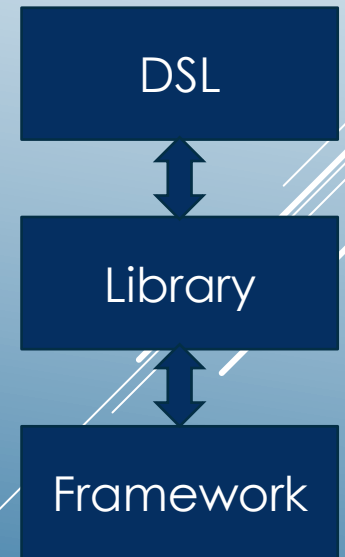
## **DSLs are popular for several reasons:**

- Improving productivity for developers.
- Improving communication with domain experts.

Informal:

DSL is a language facade over a library or framework

He has the ability to express himself in a manner appropriate to the domain for which he is the language is intended for.





# Domain Specific Languages - examples

- **SQL** as an expression for communication with the database.
- **HTML** as a declarative language used to describe how what a web page should look like.
- **A programming language** "library" language designed to work with specific tasks belonging to a single domain.
  - E.g. **DateTime** and support for date and time tasks.
- **UML diagram** of a State machine. A good abstraction for specifying how something works.
- **Regular expression** description how to select text
  - `input =~ /\d{3}-\d{3}-\d{4}/`
- **Configuration file, XAML** – based on technology XML, JSON



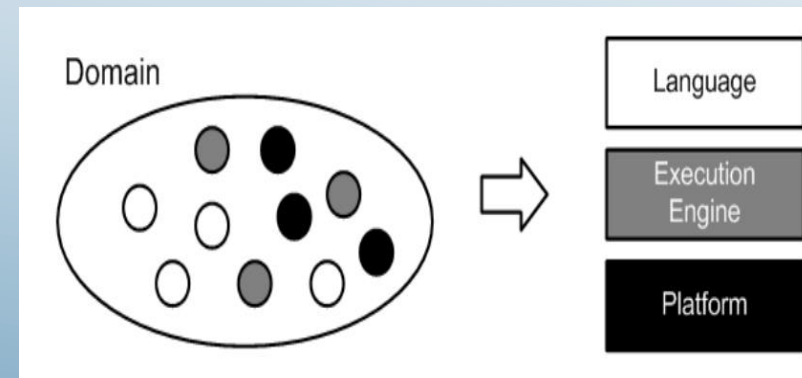
# Categories of Domain Specific Languages

- **External DSL** - Language separate from the main language of the application it works with. Usually, has a custom syntax, but using another language's syntax is also common (e.g. XML config files, SQL, regular expressions).
- **Internal DSL** - A particular way of using a general-purpose language. A script in an internal DSL is valid code in its general-purpose language, but only uses a subset of the language's features in a particular style to handle one small aspect of the overall system. (Ruby, Lisp)
- **Language workbench** - A specialized IDE for defining and building DSLs. Is used not just to determine the structure of a DSL but also as a custom editing environment for people to write DSL scripts.

# Executing the Language

Engineering a DSL (or any language) is not just about syntax, it also has to be "brought to life" – DSL programs have to be executed somehow.

- Some concerns are different for each program in the domain (white circles). The DSL provides tailored abstractions to express this variability concisely.
- Some concerns are the same for each program in the domain (black circles). These typically end up in the platform.
- Some concerns can be derived by fixed rules from the program written in the DSL (gray circles). While these concerns are not identical in each program in the domain, they are always the same for a given DSL program structure.



# Differences between GPLs and DSLs

- DSLs sacrifice some of the flexibility to express any program in favor of productivity and conciseness of relevant programs in a particular domain. But beyond that, how are DSLs different from GPLs, and what do they have in common?

	GPLs	DSLs
<b>Domain</b>	large and complex	smaller and well-defined
<b>Language size</b>	large	small
<b>Turing completeness</b>	always	often not
<b>User-defined abstractions</b>	sophisticated	limited
<b>Execution</b>	via intermediate GPL	native
<b>Lifespan</b>	years to decades	months to years (driven by context)
<b>Designed by</b>	guru or committee	a few engineers and domain experts
<b>User community</b>	large, anonymous and widespread	small, accessible and local
<b>Evolution</b>	slow, often standardized	fast-paced
<b>Deprecation/incompatible changes</b>	almost impossible	feasible

# Modeling and Model-Driven Development

- **Descriptive model** represents an existing system. It abstracts away some aspects and emphasizes others. It is usually used for discussion, communication and analysis.
- **Prescriptive model** is one that can be used to (automatically) construct the target system. It must be much more rigorous, formal, complete and consistent.

Defining and using DSLs is a flavor of MDSD: we create formal, tool-processable representations of specific aspects of software systems.

We then use interpretation or code generation to transform those representations into executable code expressed in GPL and the associated XML/HTML/whatever files.

# Difference between programming and (prescriptive) modeling

	Modeling	Programming
Define your own notation/language	Easy	Sometimes possible to some extent
Syntactically integrate several langs	Possible, depends on tool	Hard
Graphical notations	Possible, depends on tool	Usually only visualizations
Customize generator/compiler	Easy	Sometimes possible based on open compilers
Navigate/query	Easy	Sometimes possible, depends on IDE and APIs
View Support	Typical	Almost Never
Constraints	Easy	Sometimes possible with Findbugs etc.
Sophisticated mature IDE	Sometimes, effort-dependent	Standard
Debugger	Rarely	Almost always
Versioning, diff/merge	Depends on syntax and tools	Standard

# Modular Languages

Language size simply refers to the number of language concepts in that language.

Language scope describes the area of applicability for the language, i.e. the size of the domain. The same domain can be covered with big and small languages.

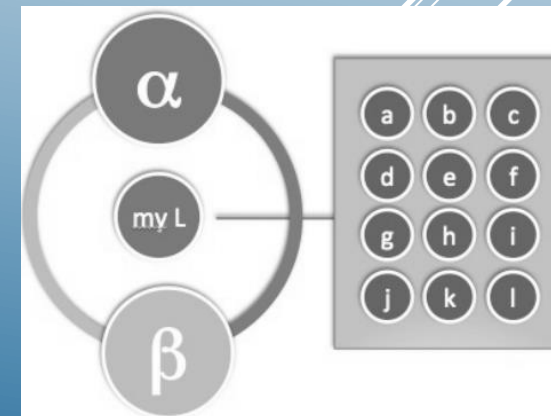
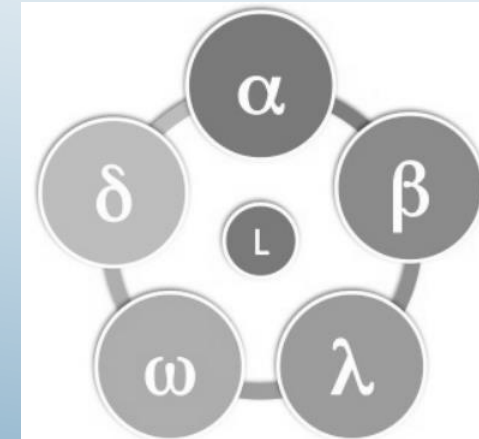
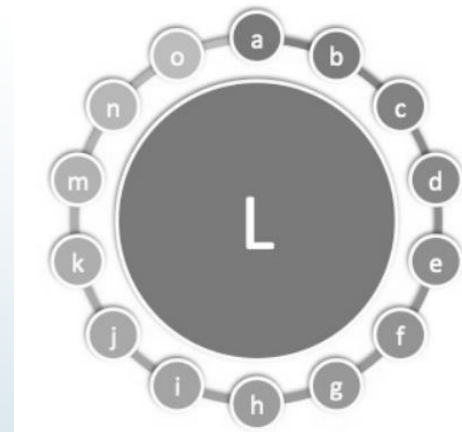
A **big language** makes use of linguistic abstraction, whereas a **small language** allows the user to define their own in-language abstractions.

A **modular language** is made up of a minimal language core, plus a library of language modules that can be imported for use in a given program.



# Domain Specific Languages

- **Big languages** have a relatively large set of very specific language concepts. Proponents of these languages say that they are easy to learn, since "There's a keyword for everything". (COBOL, ABAP)
- **Small languages** have few, but very powerful language concepts that are highly orthogonal, and hence, composable. Users can define their own abstractions. (Lisp, Smalltalk)
- **Modular language** is made up of a minimal language core, plus a library of language modules that can be imported for use in a given program.



# Benefits of using DSLs

- **Productivity** - presumably the amount of DSL code you have to write is much less than what you'd have to write if you used the target platform directly. You can replace a lot of GPL code with a few lines of DSL code.
- **Quality** - using DSLs can increase the quality of the created product: fewer bugs, better architectural conformance, increased maintainability. This is the result of the removal of (unnecessary) degrees of freedom for programmers, the avoidance of duplication of code. This is also known as correct-by-construction: the language only allows the construction of correct programs.



# Benefits of using DSLs

- **Validation and Verification** - since DSLs capture their respective concern in a way that is not cluttered with implementation details, DSL programs are more semantically rich than GPL programs. Analyses are much easier to implement, and error messages can use more meaningful wording, since they can use domain concepts
- **Data Longevity** - models are independent of specific implementation techniques. They are expressed at a level of abstraction that is meaningful to the domain – this is why we can analyze and generate based on these models. Models can be transformed into other representations.

# Benefits of using DSLs

- **A Thinking and Communication Tool** – it is a way of expressing domain concerns in a language that is closely aligned with the domain, your thinking becomes clearer, because the code is not cluttered with implementation details. This also makes team communication simpler.
- **Domain Expert Involvement** - DSLs whose domain, abstractions and notations are closely aligned with how domain experts (i.e. non-programmers) express themselves, allow for very good integration between developers and domain experts: domain experts can easily read, and often write program code, since it is not cluttered with implementation details irrelevant to them.

# Benefits of using DSLs

- **Productive Tooling** – external DSLs can come with tools, i.e. IDEs that are aware of the language. This can result in a much improved user experience. Static analyses, code completion, visualizations, debuggers, simulators and all kinds of other niceties can be provided. These features improve the productivity of the users and also make it easier for new team members to become productive
- **No Overhead** - If you are generating source code from your DSL program (as opposed to interpreting it) you can use domain-specific abstractions without paying any runtime overhead, because the generator, just like a compiler, can remove the abstractions and generate efficient code. And it generates the same low overhead code, every time, automatically.

# Summarizing - why use Domain-Specific Languages?

- DSLs are very good at taking certain narrow parts of programming and making them easier to understand and therefore quicker to write, quicker to modify, and less likely to breed bugs.
- Since DSLs are smaller and easier to understand, they allow nonprogrammers to see the code that drives important parts of their business.
- Communication between programmers and their customers is the biggest bottleneck in software development, so any technique that can address it is worth its weight in single malts.

# Bottleneck or Problems of DSL

- **Language Cacophony**

- The concern that languages are hard to learn, so using many languages will be much more complicated than using a single one. Having to know multiple languages makes it harder to work on the system and to introduce new people to the project.

- **Cost of Building**

- A DSL may be a small incremental cost over its underlying library, but it's still a cost.
- There's still code to write, and above all to maintain.
- The maintenance of the DSL is an important factor.
- The cost of a DSL is the cost over the cost of building the model.

# Bottleneck or Problems of DSL

- **Ghetto Language**

- Company that's built a lot of its systems on an in-house language which is not used anywhere else. This makes it difficult for them to find new staff and to keep up with technological changes.
- There's always a danger for a DSL to accidentally evolve into a general-purpose language.

- **Blinkered Abstraction (Tunnel vision)**

- DSL It allows you to express the behavior of a domain much more easily than if you think in terms of lower-level constructs.
- Any abstraction, be it a DSL or a model, always carries with it a danger - that of putting blinkers on your thinking.



# Challenges - Domain Specific Languages

- **Effort of Building the DSLs**

- Before a DSL can be used, it has to be built. If the DSL has to be developed as part of a project, the effort of building it has to be factored into the overall cost-benefit analysis.
- There are three factors that make DSL creation cheaper:
  - deep knowledge about a domain
  - experience of the DSL developer
  - productivity of the tools.

This is why focussing on tools in the context of DSLs is important.

Several white lines of varying lengths and angles are drawn in the bottom right corner of the slide, creating a modern, abstract graphic element.

# Challenges - Domain Specific Languages

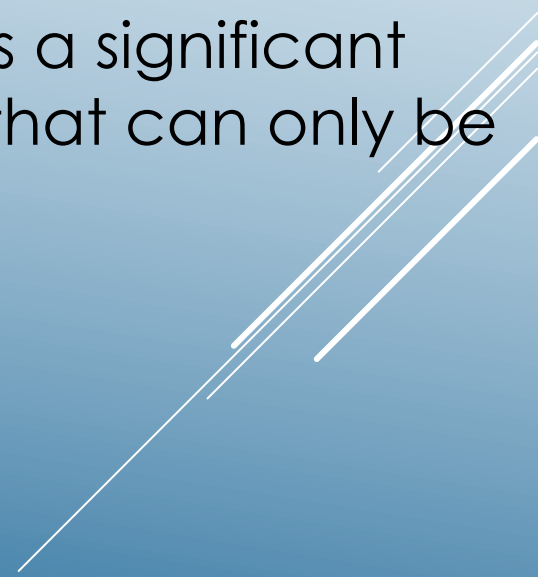
- **Language Engineering Skills**

- Building DSLs requires experience and skill. The whole language/compiler thing has a bad reputation that mainly stems from "ancient times" when tools like lex/yacc, ANTLR, C and Java were the only ingredients you could use for language engineering.
- Modern language workbenches have changed this situation radically, but of course there is still a learning curve.
- DSL Design, but it nevertheless requires a significant element of experience and practice that can only be build up over time.



# Challenges - Domain Specific Languages

- **Process Issues**

- Using DSLs usually leads to work split: some people build the languages, others use them.
  - It is important that you establish some kind of process for how language users interact with language developers and with domain experts.
  - DSL Design, but it nevertheless requires a significant element of experience and practice that can only be build up over time.
- 
- Several thin, white, parallel diagonal lines are drawn across the bottom right corner of the slide, extending from the middle of the right edge towards the bottom left.

# Challenges - Domain Specific Languages

- **Evolution and Maintenance**

- Just like any other asset you develop for use in multiple contexts, you have to plan ahead (people, cost, time, skills) for the maintenance phase.
- A language that is not actively maintained and evolved will become outdated over time and will become a liability.

- **DSL Hell**

- Once development of DSLs becomes technically easy, there is a danger that developers create new DSLs instead of searching for and learning existing DSLs. This may end up as a large set of half-baked DSLs, each covering related domains, possibly with overlap, but still incompatible.

# Application of Domain Specific Languages

- **Utility DSL**

- One use of DSLs is simply as utilities for developers. A developer, or a small team of developers, creates a small DSL that automates a specific, usually well bounded aspect of software development.
- Examples include the generation of array-based implementations for state machines, any number of interface/contract definitions from which various derived artifacts (classes, WSDL, factories) are generated, or tools that set up project and code skeletons for given frameworks.

Often, these DSL serve as a "nice front end" to an existing library or framework, or automates a particularly annoying or intricate aspect of software development in a given domain.

# Application of Domain Specific Languages

- **Architecture DSLs**

- A larger-scale use of DSLs is to use them to describe the architecture (components, interfaces, messages, dependencies, processes, shared resources) of a (larger) software system or platform. In contrast to using existing architecture modeling languages (such as UML), the abstractions in an architecture DSL can be tailored specifically to the abstractions relevant to the particular platform or system architecture.
- From the architecture models expressed in the DSL, code skeletons are generated into which manually written application code is inserted.

# Application of Domain Specific Languages

- **Full Technical DSLs**

- For some domains, DSLs can be created that don't just embody the architectural structure of the systems, but their complete application logic as well, so that 100% of the code can be generated.
- Examples include DSLs for some types of Web application, DSLs for mobile phone apps, as well as DSLs for developing state-based or dataflow-based embedded systems.

- **Application Domain DSLs**

- DSLs describe the core business logic of an application system independent of its technical implementation. These DSLs are intended to be used by domain experts, usually non-programmers

# Domain Specific Languages

Recommended reading

Fowler, M. Domain-specific languages. Pearson Education, 2010.

Voelter, M., et al. DSL Engineering - Designing, Implementing and Using Domain-Specific Languages. [dslbook.org](http://dslbook.org), 2013

Several thin, white, parallel diagonal lines are positioned in the bottom right corner of the slide, extending from the right edge towards the bottom.

# Semestral project -Artifact 7

## Presentation your IS

Present your developed IS, which consists of:

- three implemented use cases
- two different graphical user interfaces
- two different data repositories
- complete documentation (one pdf A1-A6)

Presentation will be at the last exercise on 15.12.2021



# Exercise tasks

- Present your current state of semestral work code.
- Continue the implementation the semester project.

# Lecture checking questions

- What is domain-specific language? Why and when is it good to use it? Give examples.
- What is the purpose of using a domain-specific language and what are the characteristics should it have?
- What is the difference between external and internal domain-specific language? Give examples.
- What problems can we encounter when designing domain-specific domain-specific language? Give examples.