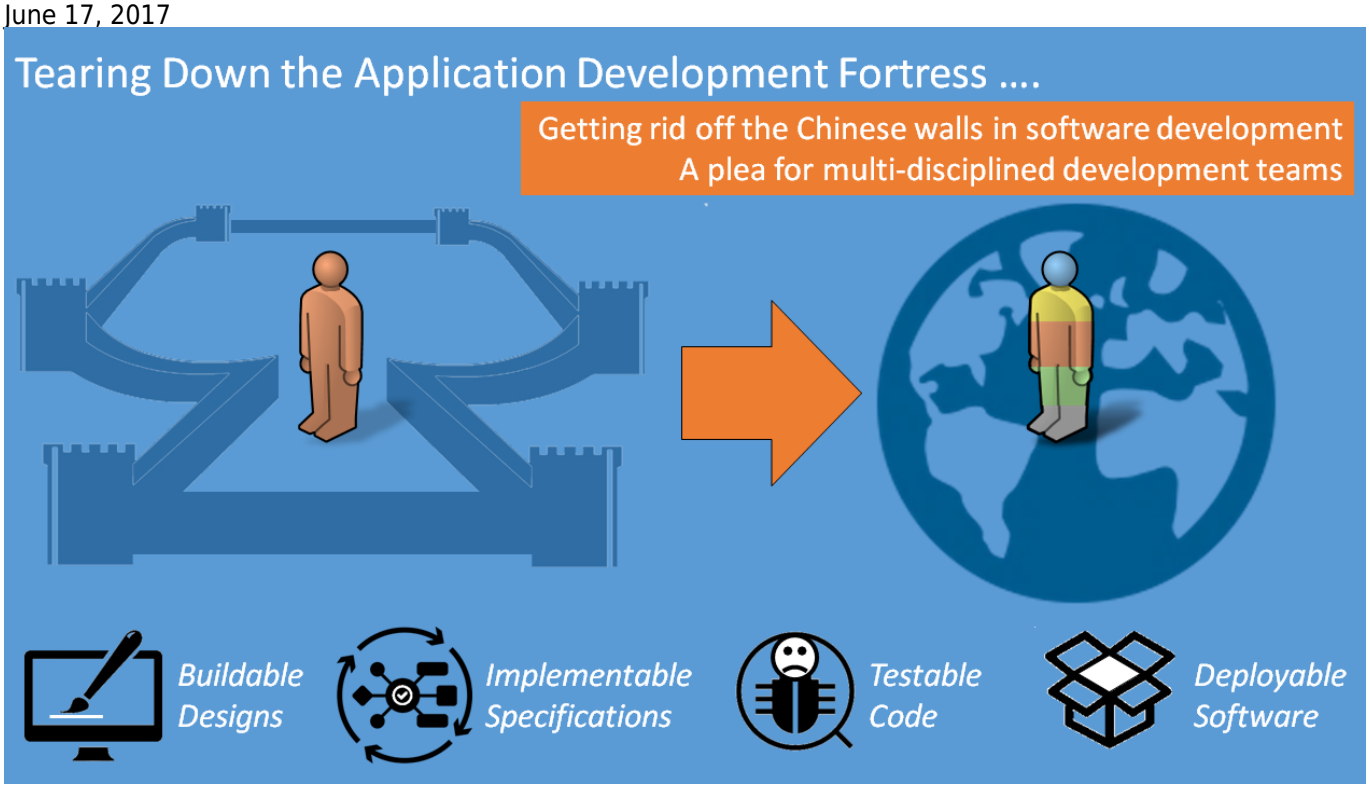
Tearing Down the Application Development Fortress

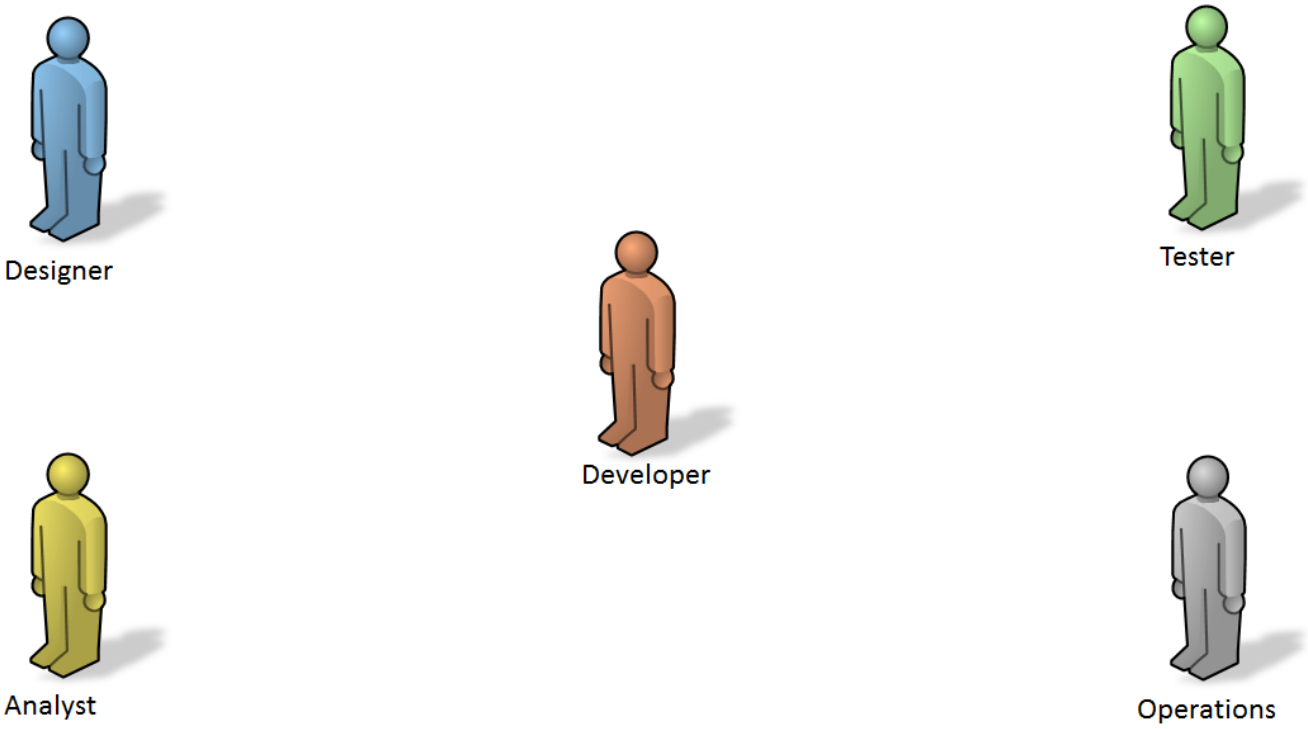
June 17, 2017



A plea for multi-disciplined development teams

Introduction

Creating applications is a multi-skill effort that comes with a set of challenges. For an average sized application there are 5 skills involved:



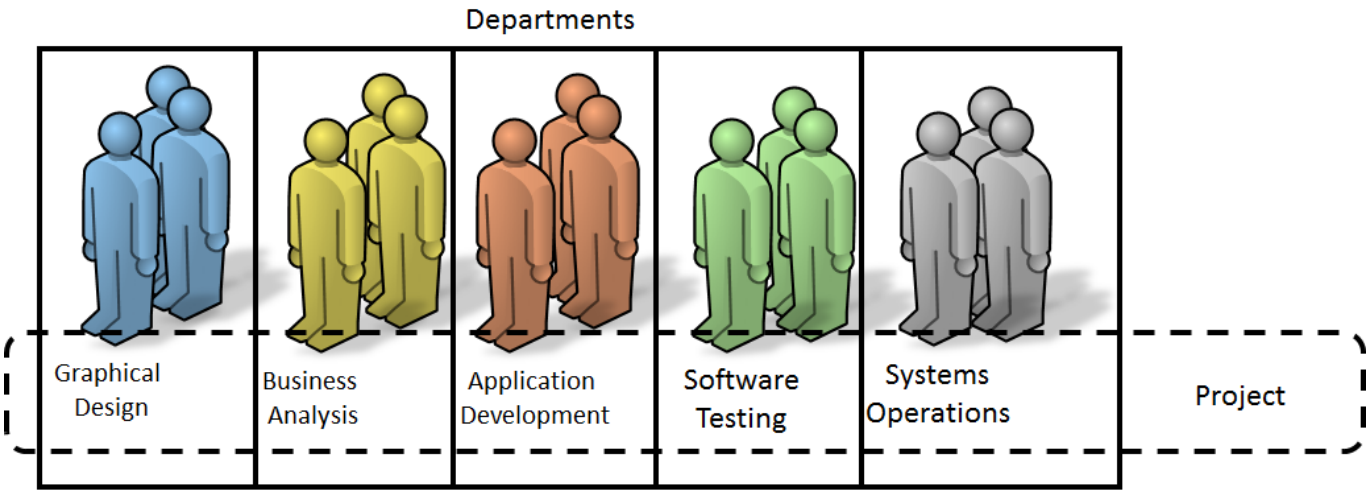
- Graphical design
- Business analysis
- Application development
- Software testing
- System operations

… not even taken into account **other stakeholders** like end-users, DBA's, enterprise architects, security officers, project and program managers, change management, user experience design …

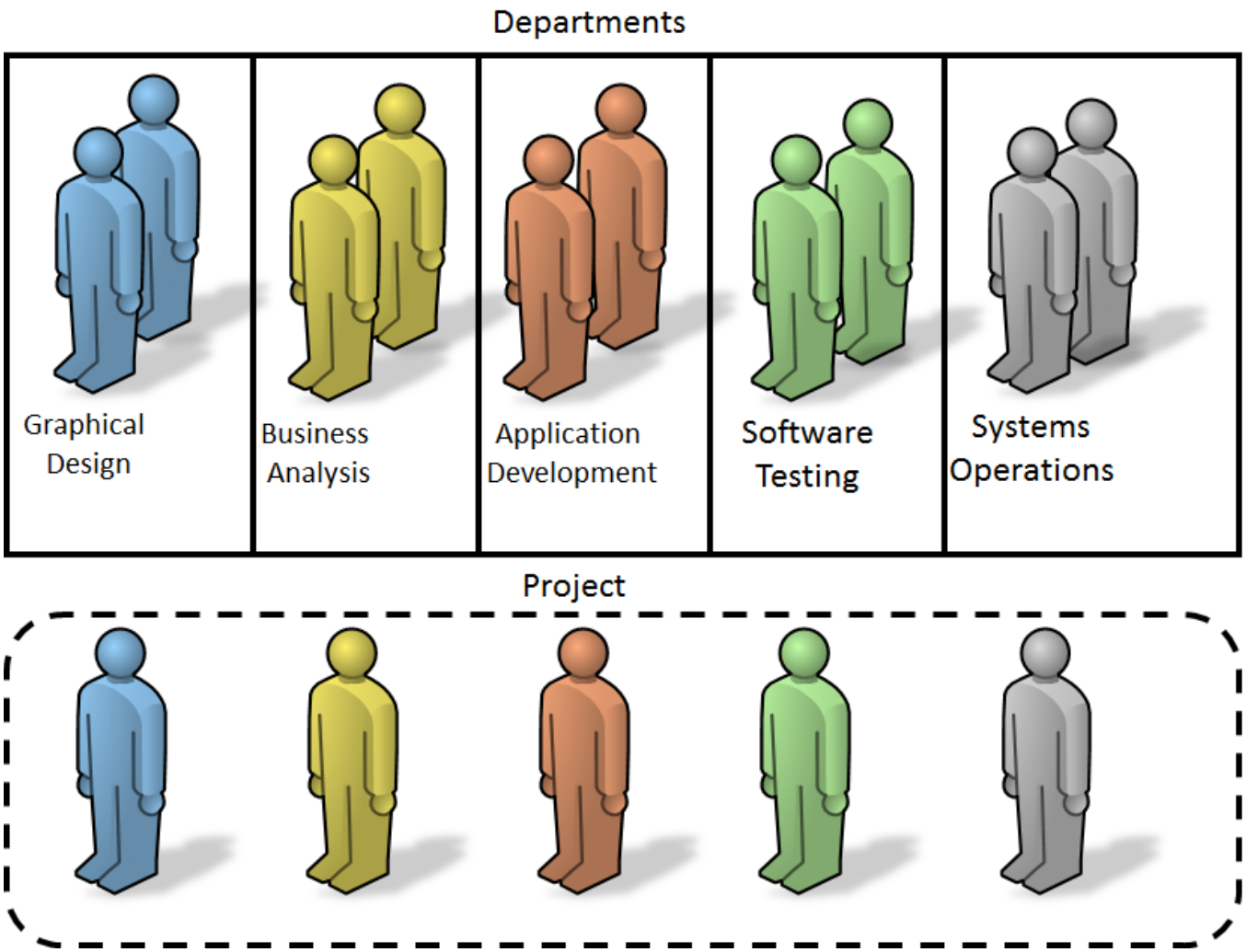
Different Collaboration Models

In bigger organizations we still see these skills organized in separate **departments that are offering services to the software development project**. Depending on how agile the organization works, we see collaboration happen on a scale **from isolated teams to mixed teams**:

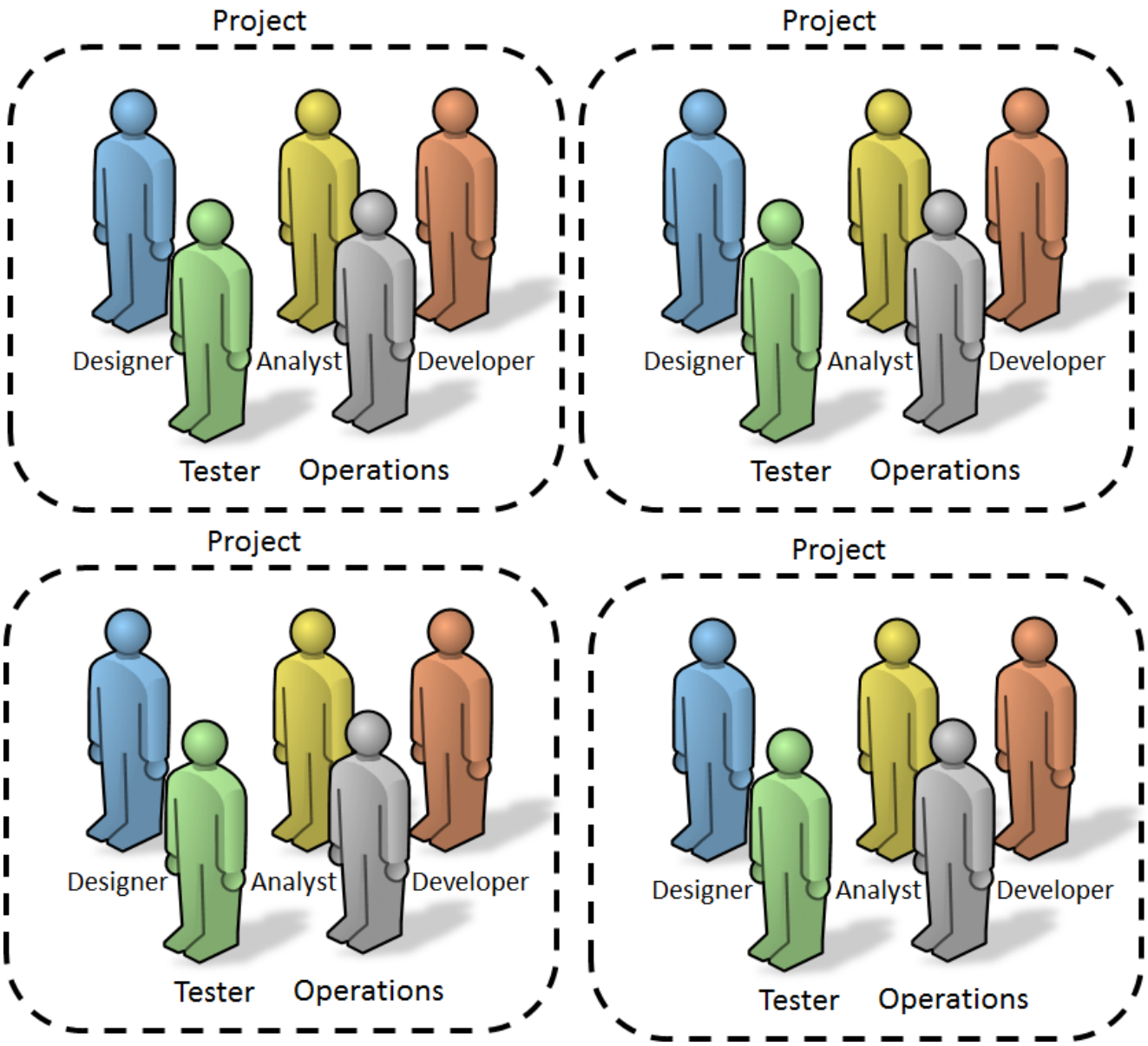
- **Isolated teams**: the project work is split based on required skills and subsets of the work is assigned to specific departments



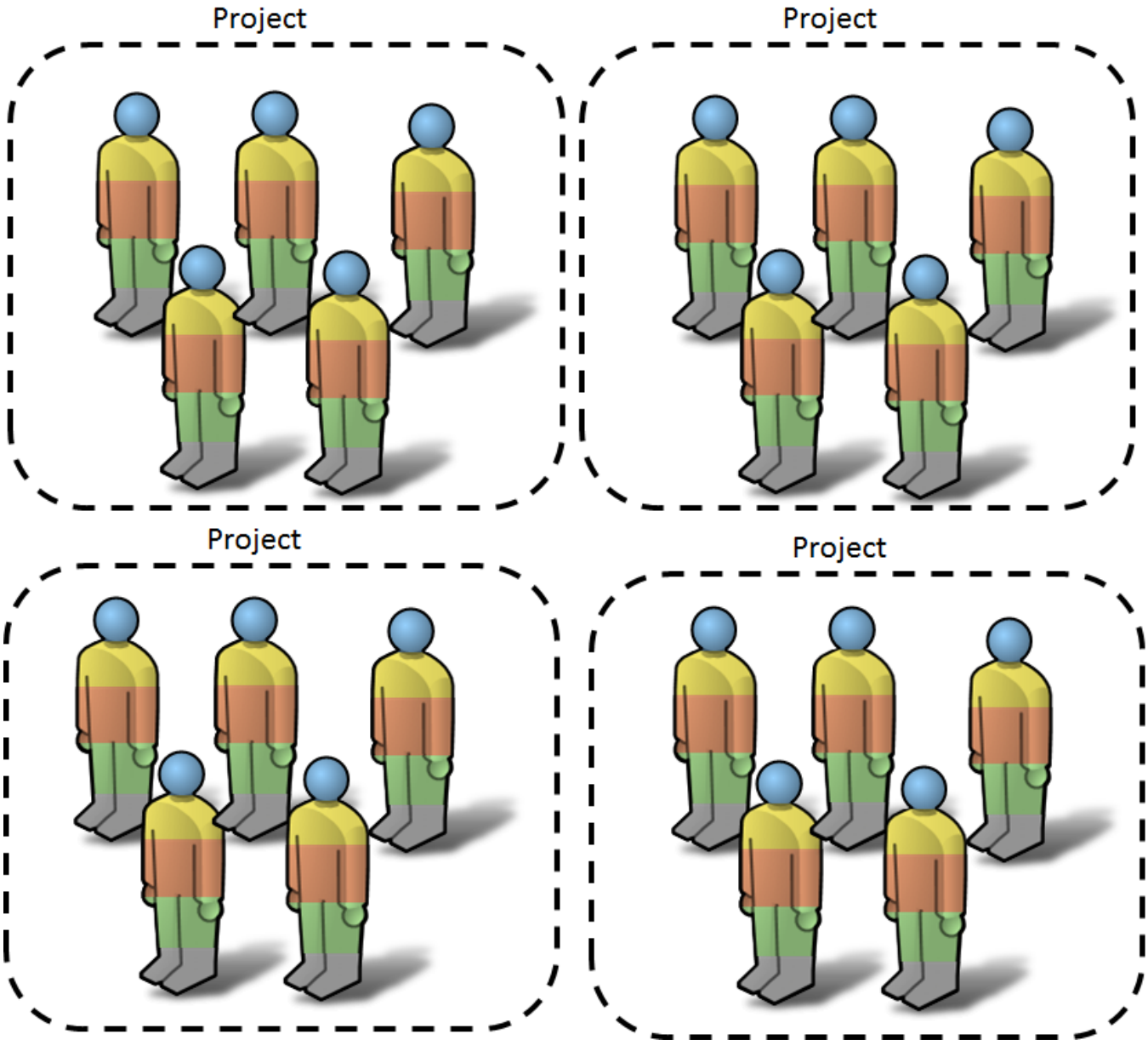
- **Detached team members**: members are part of a department and for the duration of the project they are detached to the project team based on their skills.



- **Mixed role teams**: only development teams exists, team members have a specific skill and are assigned a role in the development team, a development teams get projects assigned to work on.



- **Mixed team**: the development team gets assigned a project, team members decide how the activities of project are organized and distribute the work across the members according to experience.



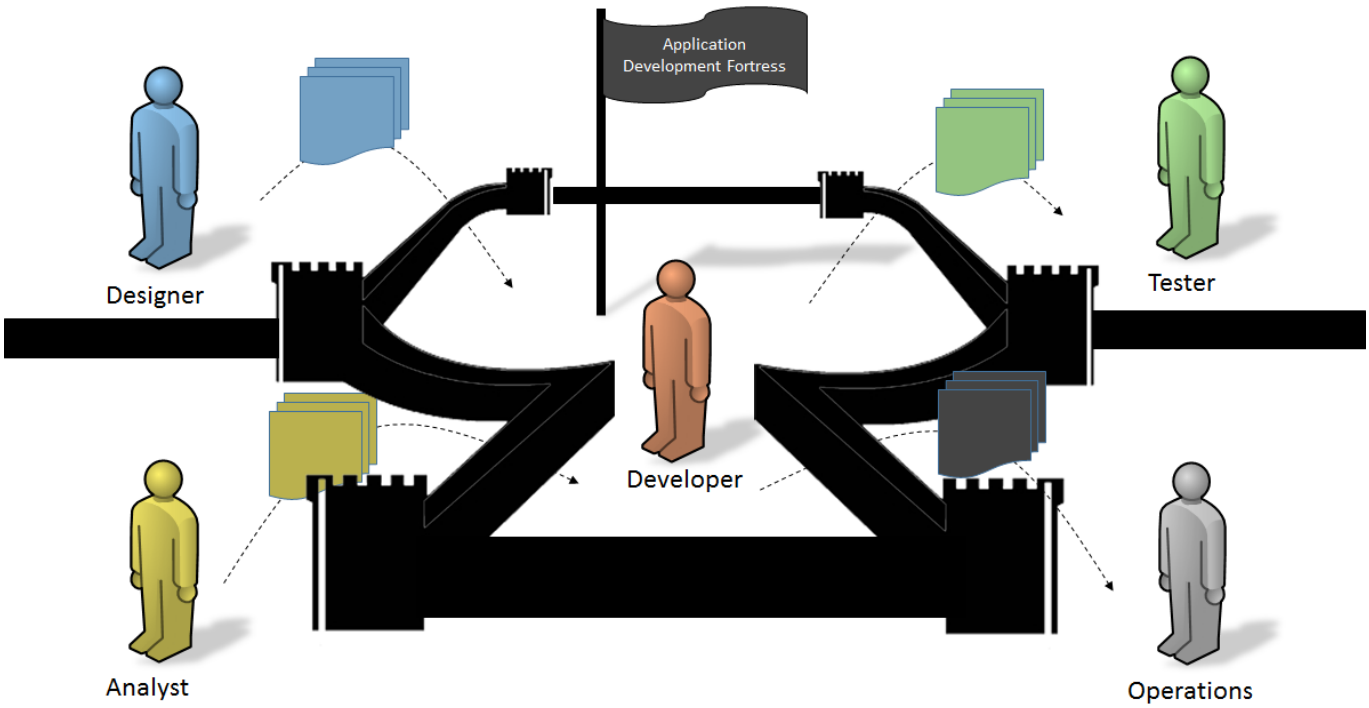
If you are familiar with **agile methodologies**, only the last two are considered to be aligned with agile. Agile calls these **multi-skilled teams** i.e. the team has all skills for the project and **multi-skilled team members** i.e. the member are experienced and can contributed on more than one skill. The ultimate goal is to have multi-skilled team members.

It is clear that with multiple teams involved, decent guidelines on how to collaborate are required, to ensure everyone collectively delivers the application. In organizations where there is a strict distinction between roles, I often see issues arise as every team works on its own. But even in mixed teams **we must value and align the opinions of different roles in the development project**.

The Chinese Wall

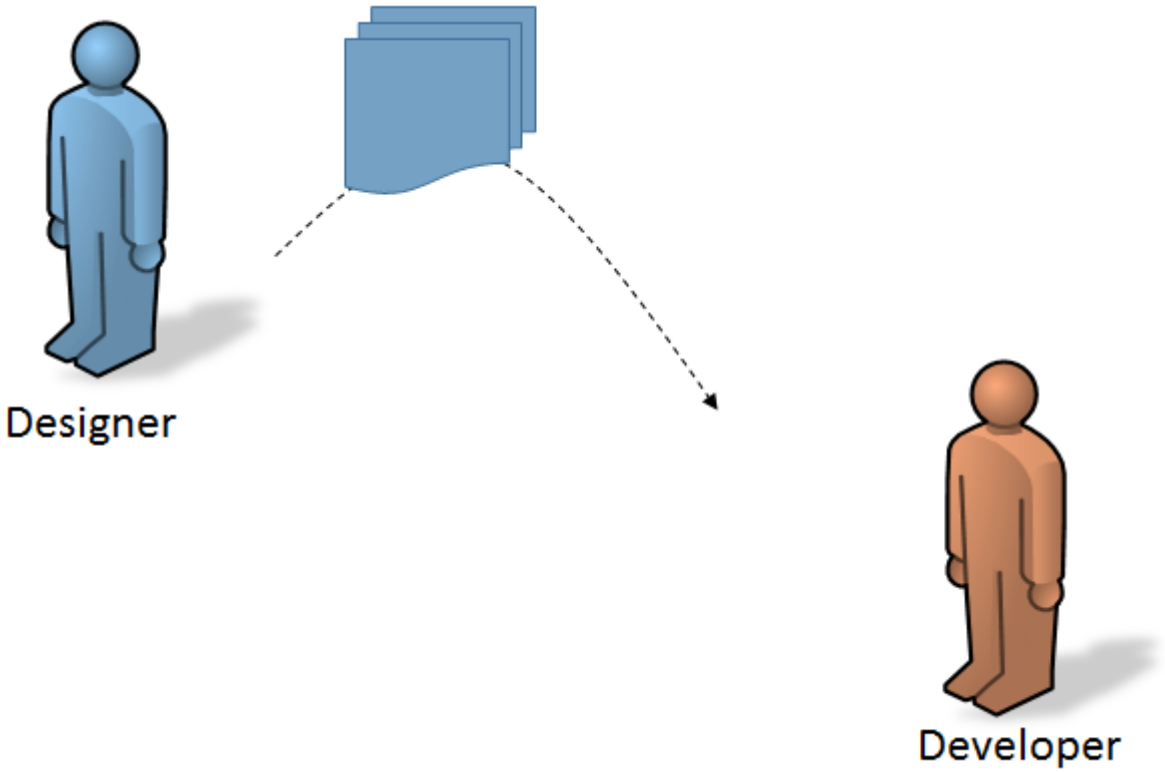
Working as a software engineer myself, I tend to look at things from a developers viewpoint. You could say I'm in my **application development fortress**. If collaboration is not well organized you see **a Chinese-wall phenomena: people stick to their role and are not looking at the bigger picture**. Specifications and information are thrown over the wall and when the activities are finished the resulting artefacts are returned i.e.

thrown back over the wall.



In order to reduce the Chinese-wall effect and brake down the fortress **we need to look at the interactions and exchanges between the different roles in the team.**

The Designer – Developer Interaction



The interaction between designers and developers should focus on **Buildable Designs**. Often frustrations arise from a designer – developer ping-pong:

- **This is not user friendly and looks horrible**
- **These controls are too complex to build**

Buildable designs means that a designer must leave his fortress and get involved in the more technical aspects of an UI. **Moving away from pure graphical design into the world of user experience design**. You can even measure this evolution by looking at the proportion of pure graphical tools in the toolset of your designer.

Two important aspects drive Buildable Designs:

- **Design decomposability:**

Can the design be split functionally and conceptually? Conceptually there are the **content** (data), the **mark-up instructions** and the **templates**. Functionally there are out of the box controls linked to the platform we are developing for and custom written controls. Can designs be realized by **combining and reusing existing controls** into widgets **instead of custom creating something from scratch**.

- **Intent orientation:**

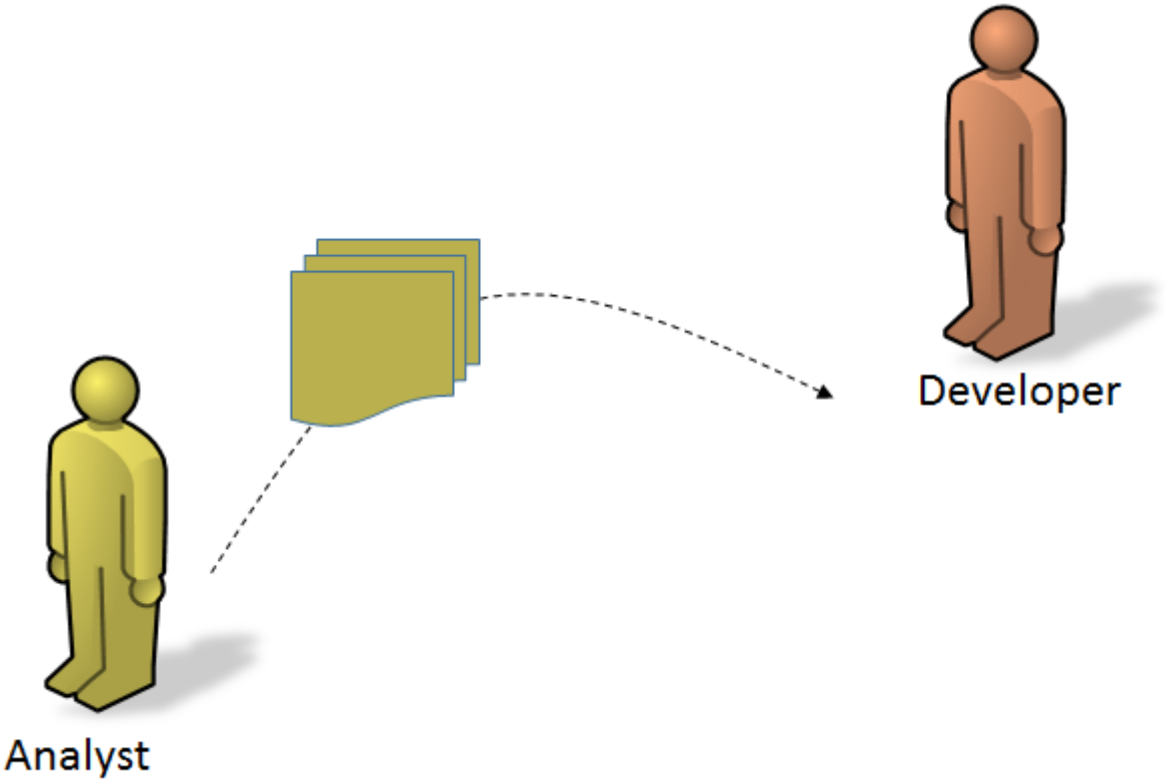
Have we designed the interface based on how data is stored and organized technically; or based on the user's activities and what he wants to achieve? **Focus on the intent of the user** helps to think about a logical task organization before we are even considering how this graphically will be represented. E.g. The use of **wireframes** will force you to think about activities instead of the size and color of buttons.

This means for designers that they must understand the concept of **templates** (e.g. CSS for web applications, XAML for desktop applications). Developers must learn to think as an end-user and understand that **data representation** might differ substantially from how data is stored in order to support that user's activity.

Tools:

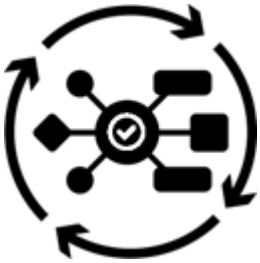
- Use **prototyping** and mocking tools instead of drawing tools.
- Use **collaboration tools** where designers can edit the mark-up and developers code the templates and data retrieval

The Analyst – Developer interaction



The interaction between analysts and developers should focus on **Implementable Specifications**. Often frustrations arise from a analyst – developer ping-pong:

- **The functional requirement was ignored**
- **This can’t be implemented like that**



Implementable specifications means the analyst leaves his fortress and gets involved in the practical aspects of the business logic. Developers must learn to think in terms of **business processes** and learn to express the **limitations of certain technologies**. Developers and analysts need to inform each other if any deviation is required compared to the original specifications. Sometimes analysts go overboard a create almost technical specification with pseudo-code and database queries. You can measure this by calculating the proportion of pseudo-code in a functional design.

Three important aspects drive Implementable Specifications:

- **Flow robustness:**

Make sure that the **unhappy flow** gets modelled as well. Analysis will describe process activities with pre- and post-conditions. If these conditions are not met, the specification should contain information on how to handle that situation. The unhappy flow can be considered an expected situation but we also must make business logic

robust by adding **logic for unexpected situations**. In order to function in a stable way, a system needs to move from one stable state to the next. Developers should never make assumptions and ask for clarification if anything was omitted (logic must be exhaustive).

- **Conflict awareness:**

Business logic often makes **abstraction of work that is being done in concurrency**. Multiple users working at the same time or multiple system requests managed at the same time. In the functional to technical specification translation, developers should help analyst **to identify these conflicts** and **reorganize code to reduce the conflicts in time and space**. This can be done by reordering activities and actions. During analysis side effects of conflicts should be modelled for. E.g. systems integration can be made fault tolerant at the expense that the communicating parties must organize idempotency.

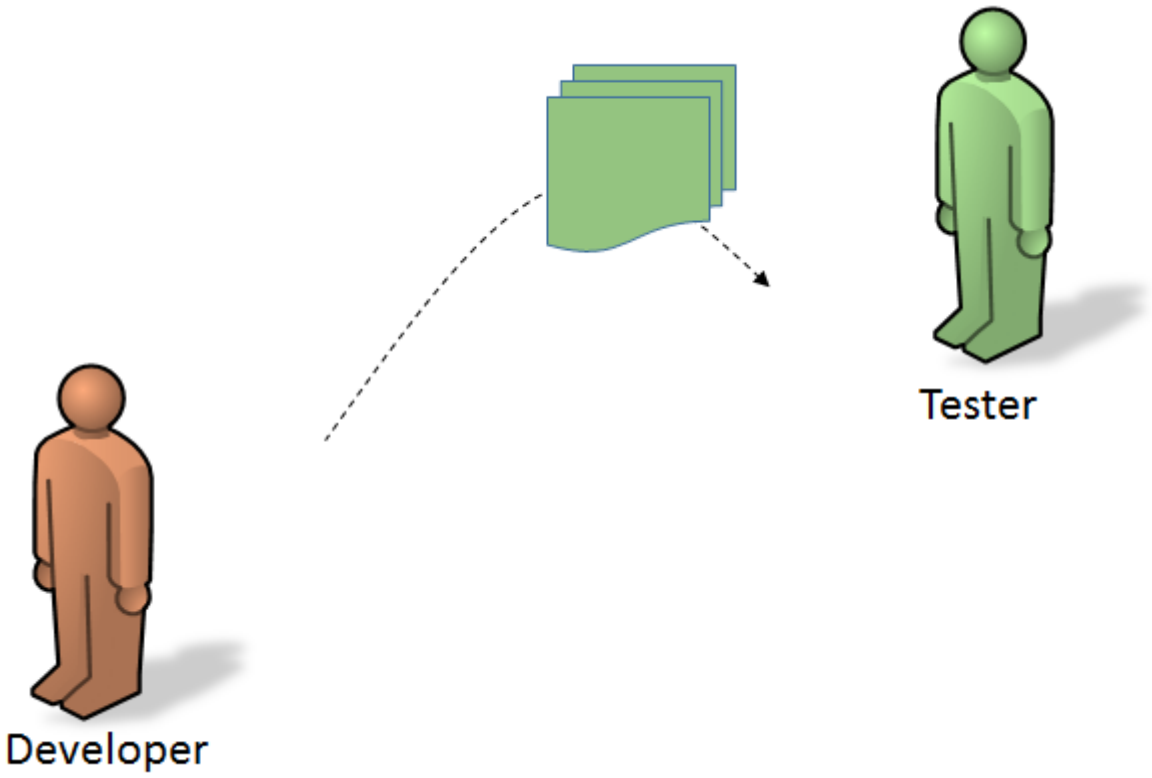
- **Realistic Units of Work:**

Distributed processing, global delivery and high-availability will allow for **performant scalable solutions** if during analysis we have taken into account the side-effects of **such a technologies**. In general we should understand that **there is no magic: vast sets of data accessed real time by a huge number of concurrent users in different locations whilst kept synchronized without delay … will not work.** Typically one or more degrees of freedom will be required. The **CAP theorem** describes this very well for distributed data processes. The unit of work handled by the business processes should be aligned with the functional requirements.

Tools:

- Use tools that **link functional specifications** created by analysts **and technical specifications** created by developers. Requirements from these specifications should be traced all the way down to the code level.
- In the past we have seen a lot of attempts of tooling that work in two directions: down from functional analysis over technical design to the code and up again from code to functional analysis. It has been proven hard for non -trivial changes to reflect code changes back into specification changes. Nowadays tooling should **focus on trackability** (from high-level requirement to the code) **and impact analysis** (from code changes to affected high-level requirements) but modification should be applied in one direction.

The Tester – Developer interaction



The interaction between testers and developers should focus on **Testable Code**. Often frustrations arise from a tester – developer ping-pong:

- **This problem cannot be reproduced**
- **This code cannot be tested**

Testable code means the developer must leave his fortress and get involved in the **user acceptance process**. Moving away from code bashing towards **functionality assurance** for the end user. How can we prove and show the code is working without relying on testers to execute manually tests over and over. You can measure this by the proportion code that has automated tests linked to it.

Two important aspects drive Testable Code:

- **Independent components:**

Through **loose coupling** a developer not only enhances maintainability but facilitates testing at the same time. When layers of an application are independent they allow for **fine grained testing** E.g. the presentation layer can be tested without touching the business logic or data logic; the business logic can be tested without having to use the presentation layer … Loose coupling is organized through **Inversion of Control** (IoC) or **Dependency Injection (DI).** Developers use this in unit testing. Test teams can benefit from it in integration

or user acceptance testing by **mocking or stubbing** components.

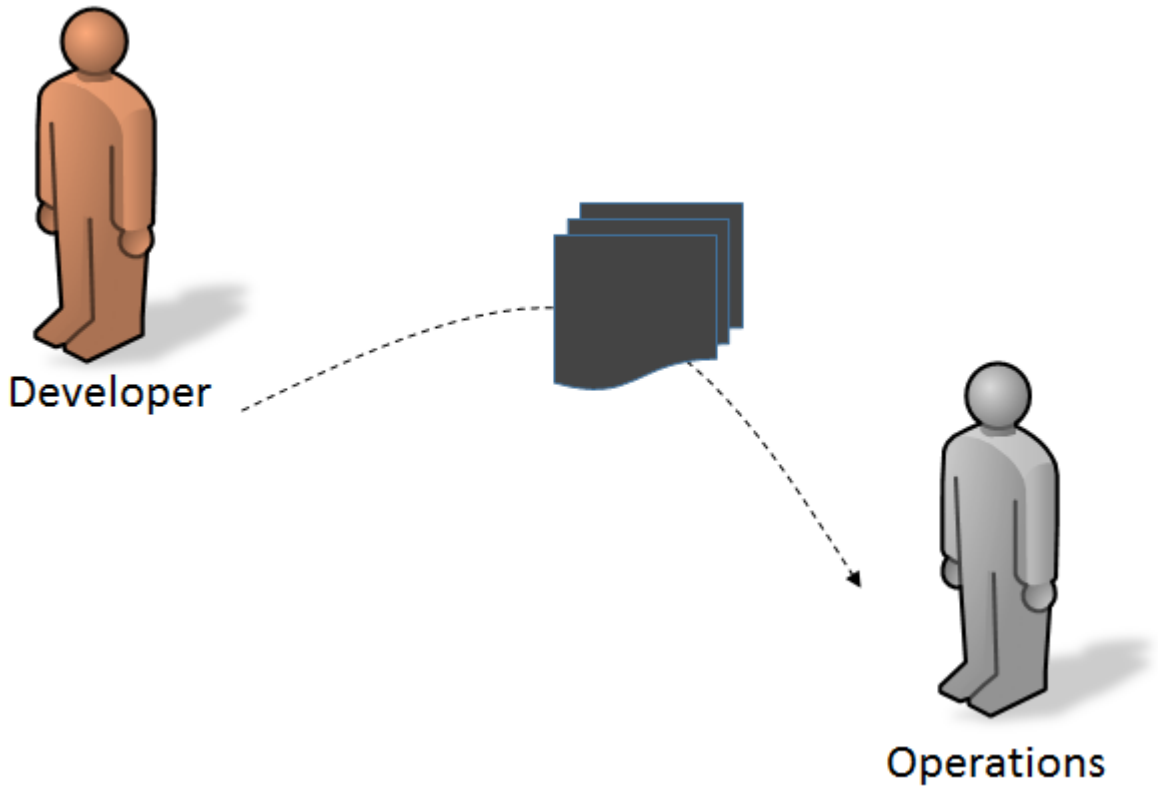
- **Circumstance equivalence:**

Strict **control on the application's ecosystem** whilst testing, facilitates resolving bugs. Testers need to control 3 aspects to have representative tests: **the environment** (e.g. used client application and user configuration), **the activities** (e.g. the sequence of steps and manipulations) and **the state** (e.g. the data and condition the application is in before and after the test). Activities are often well controlled, the state a bit less and there is limited or no control on the environment. When doing **exploratory testing** even activities might not be controlled.

Tools:

- Use tools to track the **test circumstances**. A good test suite will record not only activities but also the current condition of the environment and changes made to it; and state changes.
- Source control systems can support **impact analysis of changes**. Which requirements, tests and components are affected by a code change? For tests this is useful to identify which tests must be executed **to validate for regression**.

The Operations – Developer interaction



The interaction between system operators and developers should focus on **Deployable Software**. Often frustrations arise from a system operator – developer ping-pong:

- **There is no deployment script**

- **The deployment steps where not executed as requested**

Deployable code means the developer must leave his fortress and understands what it takes to **operate an application**. This means moving away from just delivering a bunch of binaries and thinking about long term application support. This what **DevOps** is all about. You can measure this by looking at the proportion of components that expose health information about themselves.

Two important aspects drive Deployable Software:

- **Operation readiness:**

When is an application ready for deployment? When the **deployment is automated**. Developers should make sure that deployments can be scripted. With a bit of effort there is no need for any manual activities. System operators do not know the internals of each and every application under their control. How can they **validate if everything is up and running**. This is where the **smoke tests** comes in to validate stability at deployment time. Smoke tests are a small set of validations provided by the development team, to check if all the system components are running. For post-deployment time, application must be **monitorable**. This can be achieved by enabling a **health check signal** (“is alive”) and **logging** for the different application components.

- **Controlled environments**

Operations responsibility is to control what happens in **environments**. Knowing what **versions and patches** were released in the environment with regards to the OS and frameworks. By creating scripts to check for required components (company owned or third party) by means of **pre-conditions for an application deployment**. This is becoming a lesser issue in the day and age of **containerized deployment** and **server-less computing**.

Tools:

- Use tools to **package the application and automate the deployment**. **Continuous integration** tools can even run tests to validate newly build and packaged applications.
- Deployments scripts, smoke tests and environment settings should be managed and controlled like source code i.e. by means of a **version management system**. Some organization go as far as to version control complete server images.

Towards a Formal Collaboration

When working in isolated or detached teams, collaboration is based on **task and responsibility segregation**.

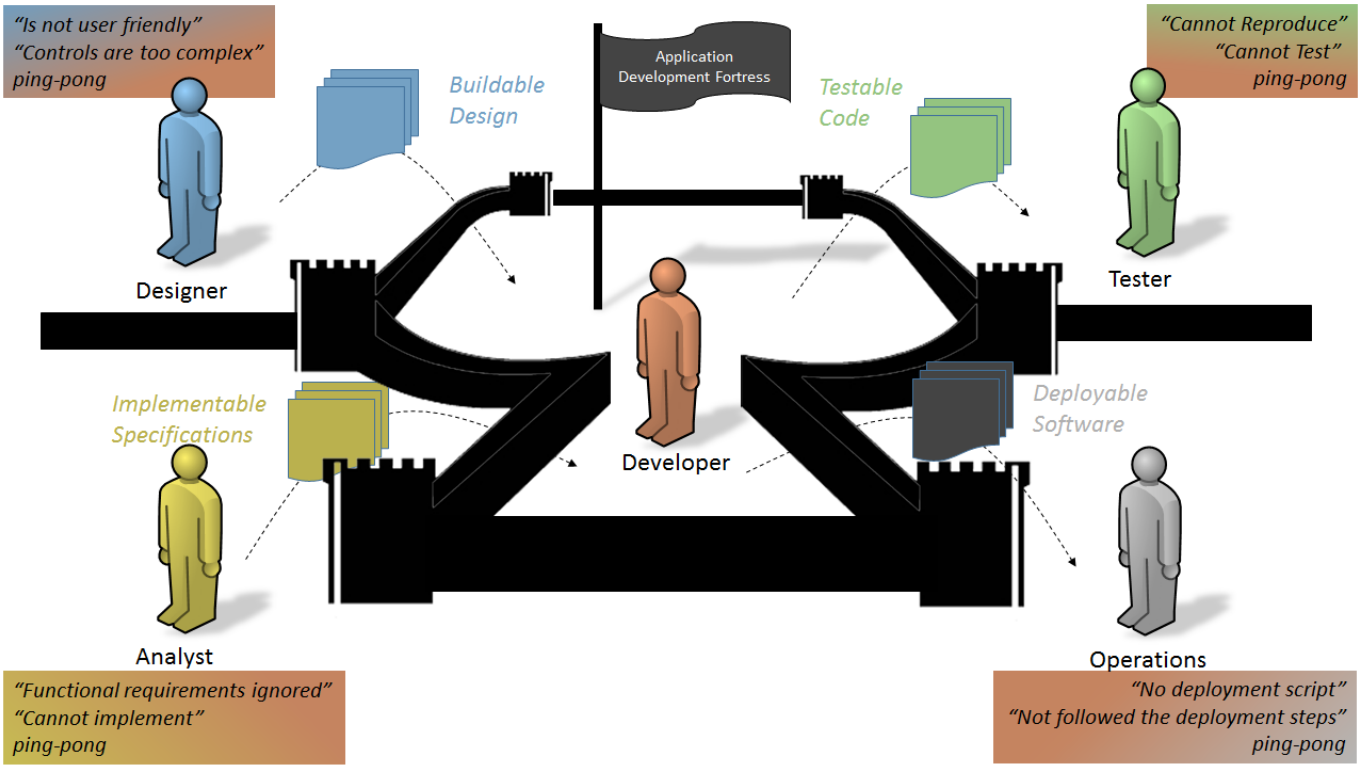
Under these conditions collaboration can be formal. **Artefacts are delivered and formally accepted**. To avoid the earlier mentioned Chinese wall between teams and the related frustrations, the usage of **acceptance criteria can enhance collaboration** as **expectations are clearly managed.**

Although organizing teams in an agile way will enhance the sense of co-owned responsibility on the end result, having mixed teams is not always possible. If the organization is not very mature it is a good idea to borrow and use some of the agile concepts (call it hybrid-agile if you must).

I would propose at least the use of following concepts:

- **The Definition of Done (DoD)**: this definition describes under which conditions an activity can be considered to be completed. The beauty of the concept is that this definition can evolve over time. So if you work without mixed teams, a DoD could be used for the artefacts thrown over the wall and **based on past experiences the DoD can evolve**.
- **The Sprint Retrospective**: this is looking back at a previous iteration of work to see **what went well at what can be improved**. Without mixed teams this could be inter-departmental meetings to optimize the software delivery process.

One Page Summary:



---

www.vanderbist.com